ECE371 Neural Networks and Deep Learning Assignment 1

Zhenrui Pan 22308141
School of Electronics and Communication Engineering
Sun Yat-sen University, Shenzhen Campus
panzhr3@mail2.sysu.edu.cn

Abstract: This report details the completion of ECE371 Assignment 1, which focuses on flower image classification using deep learning methods. The assignment is divided into two main exercises. Exercise 1 involved fine-tuning a pre-trained classification model using the MMClassification framework to adapt it to the flower dataset. Exercise 2 required completing a provided PyTorch script to implement a custom training and validation pipeline for a classification model. This document outlines the dataset preparation, the specific implementation methodologies and configurations employed in both exercises, and presents the experimental results obtained. Insights gained regarding the distinct workflows of framework-based fine-tuning versus script-based implementation, as well as challenges encountered during debugging, are discussed.

Keywords: Image Classification, Fine-tuning, MMClassification, PyTorch, Convolutional Neural Networks

1 Introduction

Image classification, the task of categorizing images based on their content, is a fundamental problem in computer vision with broad applications. This assignment explores the application of deep neural networks to classify images of various flower species. We investigate two distinct methodologies: leveraging a high-level framework for fine-tuning and implementing a core training pipeline using a popular deep learning library. This report details the dataset preparation, the specific techniques applied in each exercise, and presents the experimental outcomes. Through these exercises, we successfully developed and trained classification models for the flower dataset.Specifically, fine-tuning with MMClassification achieved a validation accuracy of up to 95.98%, while the implemented PyTorch script reached a validation accuracy of up to 92.81%. These results demonstrate the effectiveness of applying transfer learning techniques for accurate image classification on this specific dataset.

2 Related Work

Image classification has progressed from traditional methods using handcrafted features to deep learning-based approaches. The introduction of CNNs, starting with AlexNet [1], significantly improved performance by enabling end-to-end learning. Later architectures such as VGG [2] and ResNet [3] further enhanced accuracy and training stability.

Transfer learning has become a popular strategy for improving classification performance on limited datasets [4]. By fine-tuning pre-trained models, high accuracy can be achieved with fewer resources. Frameworks like MMClassification [5] support efficient fine-tuning, while libraries such as PyTorch [6] allow for flexible implementation of training pipelines.

These advancements have made deep learning-based classification both more accessible and more effective across diverse tasks.

3 Method

This section provides a detailed description of the implementation methodology used in both exercises, along with the rationale behind key design decisions and configurations.

3.1 Dataset Preparation

As described in Section 2, the flower dataset was prepared by splitting the original images into 80% training and 20% validation sets and organizing them into an ImageNet-compatible directory structure with annotation files (‘train.txt‘, ‘val.txt‘, ‘classes.txt‘). Data augmentation and normalization pipelines were applied during training.

3.2 Exercise 1: MMClassification Fine-tuning Implementation

For Exercise 1, fine-tuning was performed using the MMClassification framework. A custom configuration file was created by inheriting from a standard ImageNet training setup with ResNet50 as the base model. The following core modifications were made:

- **Model Head Adaptation:** The number of output classes in the classification head (`model.head.num_classes`) was changed to 5 to match the number of flower categories. The top-k evaluation metric was set to top-1 only via `model.head.topk = (1,)`.
- **Dataset Configuration:** The dataset was changed to a custom flower dataset by setting `data_root = 'data/flower_dataset'`.

  Both training and validation dataloaders were configured with `data_prefix = ''`, and annotation files were specified as `train.txt` and `val.txt`, respectively. Class names were provided in `classes.txt`.
- **Pre-trained Model Loading:** The pre-trained weights from ImageNet were loaded via the `load_from` key, pointing to `resnet50_8xb32_in1k_20210831-ea4938fc.pth`, enabling transfer learning.
- **Evaluation Metric:** The validation evaluator was defined with type `Accuracy` and `topk = (1,)`, aligning with the task's requirement to evaluate only top-1 classification accuracy.
- **Learning Strategy Adjustment:** The optimizer was set to SGD with a lower learning rate (`lr = 0.005`), momentum of 0.9, and weight decay of 0.0001.

  The training schedule was shortened to `max_epochs = 20` with validation every epoch. Although a learning rate scheduler was not explicitly defined, the reduced learning rate and epochs are standard strategies for efficient fine-tuning.

Training was initiated using the `tools/train.py` script with the customized configuration and a specified working directory.

3.3 Exercise 2: PyTorch Script Implementation Details

Exercise 2 involved completing the provided `main.py` script to implement a PyTorch-based training pipeline from a lower-level perspective. A pre-trained model from `torchvision.models` (e.g., `resnet18`) was used as the backbone by setting `pretrained=True`. The following key components were implemented:

- **Data Transformations:** The `data_transforms` pipeline included several data augmentation operations: `RandomResizedCrop`, `RandomHorizontalFlip`, `ColorJitter`, `RandomRotation`, and `RandomAffine`, followed by `ToTensor` and `Normalize` using ImageNet mean and standard deviation. These augmentations increased the diversity of the training data and helped improve model generalization.
- **Model Adaptation:** The final fully connected layer (`model.fc`) of the pre-trained `ResNet18` was replaced with a new `nn.Linear` layer.

The input feature size was matched to the original model's output, and the output dimension was set to 5 to support 5-class flower classification.

- **Loss Function:** The loss criterion was defined as `nn.CrossEntropyLoss`, which is commonly used in multi-class classification tasks and internally applies softmax.
- **Optimizer Definition:** An optimizer (e.g., `optim.SGD`) was initialized with the model's parameters and a learning rate (e.g., 0.001) to handle gradient-based updates during training.
- **Backward Pass and Optimization:** In the training phase of each epoch, standard backpropagation was implemented using `loss.backward()`, followed by `optimizer.step()` to update model parameters.
- **Model Saving:** The model's weights (`state_dict()`) were saved to `Ex2/work_dir/best_model.pth` whenever the validation accuracy exceeded the current best.Accuracy history was also saved to EX2.txt

4 Experiments

This section reports the results obtained from training the models in Exercise 1 and Exercise 2, supported by quantitative metrics and visualizations. It also provides a comprehensive analysis that connects classroom theory to practical implementation and highlights the lessons learned from solving real-world issues.

4.1 Results

The training processes for both exercises were executed as described in the methodology section. The validation accuracy was tracked over epochs, and the best model checkpoint was saved based on the highest validation accuracy achieved. Training logs recorded the loss, learning rate, and validation accuracy at each epoch.

Table 1 summarizes the best validation accuracy obtained from the two exercises.

Table 1: Comparison of Best Validation Accuracy

Method	Model Backbone	Best Validation Accuracy (%)
Exercise 1 (MMClassification)	ResNet50	95.98
Exercise 2 (PyTorch Script)	ResNet18	92.81

4.2 Analysis and Discussion

Comparing the results in Table 1, it is evident that the fine-tuned ResNet50 model (Exercise 1) achieved a slightly higher validation accuracy (95.98%) compared to the ResNet18 model (Exercise 2) at 92.81%. This is consistent with expectations, as ResNet50 has a deeper architecture and higher representational capacity than ResNet18, making it more suitable for complex classification tasks.

The training logs and accuracy curves show steady convergence in both experiments. The validation accuracy increased consistently with training, confirming the effectiveness of both training pipelines.

Notable technical and methodological observations include:

- **Learning Rate Scheduling:** In Exercise 1, the learning rate remained constant at 0.005 throughout the 20 epochs, due to the MultiStepLR scheduler not being triggered within the training window. In contrast, Exercise 2 employed a StepLR scheduler, which effectively decayed the learning rate at epochs 7, 14, and 21. This decay strategy likely contributed to better convergence in later epochs.
- **Dataset Path Configuration Errors:** Initially, incorrect dataset path settings caused `FileNotFoundError` exceptions. The issue was resolved by aligning the dataset directory paths in both the MMClassification config file and the PyTorch script, ensuring paths were relative to the correct working directory.

- **PyTorch Multiprocessing on Windows:** When using `num_workers > 0` on Windows, the PyTorch script encountered `BrokenPipeError` and `RuntimeError` exceptions due to improper multiprocessing setup. Adding the standard Python guard `if __name__ == '__main__'` resolved this issue, preventing the training loop from being unintentionally executed in child processes. This is a common best practice in Python when using multiprocessing.
- **Validation Accuracy Fluctuations:** Minor fluctuations in validation accuracy across epochs were observed. These variations are expected in deep learning training due to random data augmentation, mini-batch stochasticity, and dynamic Batch Normalization behavior. Overall trends, however, were upward and stable.
- **Achieving Target Accuracy and Further Improvements:** The assignment specified that surpassing 90% accuracy should be achievable, which both experiments successfully accomplished. While satisfactory, further gains could be made via hyperparameter tuning (e.g., learning rate, batch size, optimizer type), more aggressive data augmentation, or using deeper models. Experimentation with regularization (e.g., dropout, weight decay) could also help reduce overfitting and stabilize accuracy.

4.3 Implementation Comparison

From a practical standpoint, the MMClassification framework offers a high-level, configuration-centric workflow, which significantly simplifies model training and evaluation. Users can define the entire training pipeline—including dataset paths, model architecture, optimizer, scheduler, and evaluation metrics—via a single .py or .yaml configuration file. This approach is highly modular and reproducible, enabling rapid prototyping and easier debugging. However, this abstraction also limits flexibility when implementing custom data processing, model modifications, or experimental logic beyond what the framework natively supports.

In contrast, the PyTorch script-based implementation demands manual construction of all pipeline components, such as dataset preparation, data augmentation, model modification, training loop, learning rate scheduling, loss computation, and checkpoint saving. While this results in increased development overhead, it provides maximum flexibility and transparency, which is particularly beneficial for learning and fine-grained experimentation. This low-level approach allows developers to understand and control every step in the training process, such as when and how gradients are computed and applied, how the learning rate evolves across epochs, and how model performance is monitored and saved.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 2012.

[2] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations*, 2015.

[3] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CVPR*, 2016.

[4] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems*, 2014.

[5] OpenMMLab. Mmclassification. https://github.com/open-mmlab/mmclassification, 2020.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 2019.