DSA5203 Assignment 3 Project Report

Lin Xueshun

Student ID: A0274618W

The purpose of this report is to complete an image classification task using the Convolutional Neural Network method. The overall training dataset contain 1500 images, belongs to 10 separated classes with the following categories and corresponding labels:

category	label	category	label	category	label
bedroom	1	Coast	2	Forest	3
Highway	4	industrial	5	Insidecity	6
kitchen	7	livingroom	8	Mountain	9
Office	10	OpenCountry	11	store	12
Street	13	Suburb	14	TallBuilding	15

This report records all walkthroughs, separated into several parts of Data preparation and processing, Model Design and Implementation, Training and Testing process, and finally the Hyperparameter tuning section.

Data Preparation and Preprocessing

A **load_data** function was defined for the image classification pipeline, designed to prepare and manage data for both training and validation phases. This function operates by loading images from a specified path, applying transformations, and splitting the dataset into training and validation sets.
Below is the function signature of this custom function:

```
def load_data(mean, std, path, batch_size, train_size = 0.8):
```

- **mean**: The mean value of the dataset, used for normalization.
- **std**: The standard deviation of the dataset, also used for normalization.
- **path**: Path to the dataset directory.
- **batch_size**: Number of images in each batch of data.
- **train_size**: Fraction of the dataset to be used for training; the remainder is used for validation.

This function contains few points of details below:

1. Data Transformations

This function defines two sets of transformations using PyTorch's **transforms.Compose**:

- **Training Transformations**: Aimed at artificially increasing the dataset's diversity and robustness against overfitting through data augmentation techniques. These include:
    - **GaussianBlur**: Adds blur to the images, simulating variations in focus.
    - **RandomPerspective**: Modifies the perspective of images, adding variance in the viewing angle.
    - **RandomHorizontalFlip**: Mirrors images horizontally, beneficial for rotational invariance.
    - **RandomRotation**: Rotates the images randomly within a given degree range.
    - **RandomAutocontrast**: Automatically adjusts the image contrast.
    - **RandomErasing**: Randomly erases a rectangle region in the image to reduce overfitting and enhance model robustness.
    - **Resizing:** Images were resize to 256*256 for making all images uniform in size
    - **Grayscaling:** As all images are grey images, reduce the channels of the image to one by performing the **Greyscale** transform
    - **To Tensor:** Convert images to Pytorch tensors
    - **Normalization:** normalize the image using the mean and standard deviation of the dataset. Normalization helps in reducing discrepancies in lighting and color variations across different images, thus stabilizing the learning process.
- **Validation Transformations**: Applied to the validation data, these transformations are simpler as they exclude augmentation to provide a consistent basis for evaluating model performance. They include resizing, grayscaling, tensor conversion, and normalization.
- **Train-Validation split:** The dataset is splitted in to train and validation set by utilizing the **random_split** function **from torch.utils.data** library before the applying of the respective transformation, the training and validation sets were separated by 80-20 ratio by default. When the **train_size** argument is set to 0, meaning the test mode would be activated and only validation set would be processed and returned.

An helper function named **get_mean_std** was defined here to get the mean and std by all images from the dataset, adding up the value and calcualting the mean and std separely. As the parameters for the normalization transformation.

Another two helper functions and one class called **get_device**, **to_device**, and **ToDeviceLoader** was also definded previously to move the model and dataset to the corresponding device. An cuda and mps device's availability were checked here. CUDA (Compute Unified Device Architecture), which allows developers to use Nvidia GPUs for general purpose processing, and MPS (Metal Performance Shaders) enables the Apple's GPU acceleration. Since I'm using an Macbook at the time of doing this project, so the MPS check also included here as well. These functions and classe was referenced from: https://www.kaggle.com/code/toygarr/resnet-implementation-for-image-classification

The **CustomImageFolder** is the custom class enhances the functionality of PyTorch's datasets: **ImageFolder** by using a predefined dictionary of labels for image classification, rather than inferring them from directory names. The predefined label is given by the project' requirement as shown in the first page. The class overrides the **find_classes** method to implement this by directly mapping class names to specific numerical labels. This approach simplifies class management and reduces potential errors in label handling, making it particularly useful for projects with fixed category labels. This implement is referenced from: https://discuss.pytorch.org/t/what-is-the-simplest-way-to-change-class-to-idx-attribute/86778/4

The **TrDataset** class is a another custom class used to help apply specific transformations respectively. It takes a base dataset and a set of transformations as inputs, applying these transformations dynamically to each item as it's retrieved. This allows for tailored data preprocessing in the scenario where different transformations are needed during different training phases, such as more aggressive data augmentation during training versus more standardized preprocessing during validation. The class methods include **__len__** for returning the dataset size and **__getitem__** for fetching and transforming individual data items. This design keeps the dataset flexible and efficient for various machine learning workflows. The implement of this class is referenced from https://stackoverflow.com/questions/74920920/pytorch-apply-data-augmentation-on-training-data-after-random-split

Model Design and Implementation

A **ResNet** model was designed and implemented in this section and is inspired and referenced from https://www.kaggle.com/code/toygarr/resnet-implementation-for-image-classification

ResNet Architecture Overview

The ResNet model designed for this project employs several layers and blocks that collectively aim to capture an extensive range of features from the input images, crucial for accurate classification. The key components of this ResNet implementation include:

- **Initial Convolutional Stage**:
    - This stage starts with a convolutional layer that has a kernel size of 3x3, a stride of 1, and padding to maintain the size of the output relative to the input. It is followed by batch normalization and a ReLU activation function to introduce non-linearity. A MaxPooling layer with a size of 2x2 is applied at the end to reduce the spatial dimensions of the output, helping to reduce computation for subsequent layers.
- **Residual Blocks**:
    - **Shortcut Connections**: Each stage includes shortcut connections that employ **conv_shortcut** function, which adds a convolutional layer with a 1x1 kernel to adjust the number of channels and align the dimensions for addition with the main path outputs. This technique is essential for enabling training of deeper networks by alleviating the vanishing gradient problem.

    - **Main Residual Blocks**: Defined by the **block** function, these blocks consist of two layers of convolutional operations. The first layer uses a 1x1 convolution to reduce the dimensionality before applying a 3x3 convolution (the main feature extractor), followed by batch normalization and ReLU activation.

- **Classification Stage**:
    - The output from the last residual stage passes through an average pooling layer to reduce its dimensionality further, followed by dropout layers to prevent overfitting. The flattened output is then fed into a **LazyLinear** layer, which automatically infers the correct input size and produces the final classification across the predefined number of classes.

Below is the signature of the constructor of the model:

```
def __init__(self, in_channels, num_classes, stage2_loop, stage3_loop, dropout1, dropout2, out_channels):
```

- **in_channels**: Specifies the number of input channels in the data. For grayscale images in this project, this would be 1. This parameter adjusts the model to accommodate different types of image data.
- **num_classes**: The number of classes in the output layer of the network. This parameter should be set to the number of target classifications in the dataset. Here, it's set to 15 because we have 15 different classes of images
- **stage2_loop** and **stage3_loop**: These parameters control the number of times the residual blocks are repeated in stages 2 and 3 of the network, respectively. Increasing the number of loops allows the network to learn more complex features at the expense of increased computational complexity.
- **dropout1** and **dropout2**: These represent the dropout rates applied after the average pooling layer and just before the final classification layer, respectively. Dropout is a regularization technique used to prevent overfitting by randomly setting a fraction of input units to 0 at each update during training time.
- **out_channels**: Defines the number of output channels for the first convolutional layer, which sets the stage for the complexity of subsequent layers. Typically, a higher number of channels can capture more detailed features but also increases the computational load.

The **stage2_loop**, **stage3_loop, dropout1**, **dropout2** and **out_channels** are definded as adjustable hyperparameters for the model, this is helpful conveniently to control the model's complexity, and also futher help to implement the hyperparameter tuning process which would be introduced later in the report

Training and testing Process

The training process is mainly defined in the **fit** function, below is the signature of the function:

```
def fit(model, train_loader, val_loader, optimizer, n_epochs, early_stopping, criterion, grad_clip, scheduler, accuracy, model_dir):
```

- **model** (**nn.Module**): The neural network model that will be trained. In this project, we'll use our pre-defined ResNet model as introduced before.
- **train_loader** (**DataLoader**): The DataLoader that provides batches of training data. This loader yield batches of input data and corresponding target labels, and is created from the **Data Preparation and Processing** section before.
- **val_loader** (**DataLoader**): The DataLoader used for validating the model at the end of each epoch. Like **train_loader**, it provides batches of data that are used to evaluate the model's performance on unseen data. Is also created in the previous **Data Preparation and Processing** section.
- **optimizer** (**torch.optim.Optimizer**): The optimization algorithm used to update the model's weights based on the gradients computed during backpropagation. Examples include SGD, Adam, RMSprop, etc. And is set to a adjustable hyperparameter for hyperparameter tuning in this project
- **n_epochs** (**int**): The number of times the training algorithm will iterate over the entire training dataset. Each epoch represents a complete pass through all batches in the **train_loader**.
- **early_stopping** (A Custom Class Instance): A mechanism to halt the training process early if the validation loss does not improve for a predefined number of epochs. This helps prevent overfitting and saves computational resources.
- **criterion** (torch.nn.modules.loss._Loss): The loss function used to compute the difference between the model predictions and the actual targets. The choise in this project is **nn.CrossEntropyLoss**.
- **grad_clip** (float): The maximum value to which gradients are clipped during training. Clipping gradients can prevent exploding gradient problems in deep networks.
- **scheduler** (torch.optim.lr_scheduler): A scheduler object that adjusts the learning rate according to a predefined policy. It can be used to lower the learning rate during training to fine-tune the model as training progresses. The choice in this project is **OneCycleLR,** which works by cycling the learning rate between a lower and an upper bound within a single training run, and is designed to train neural networks both faster and with better final performance.
- **accuracy** (torchmetrics.Metric): A metric instance used to compute the model's performance metric. For the classification task in this project, the choice is accuracy.
- **model_dir** (**str**): The directory path where the trained model should be saved if it achieves a new best accuracy according to the validation results.

The main training is processed in the loop as below:

```
for data, target in tqdm(train_loader, desc=f'Epoch {epoch+1} Training'):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target - 1) # Subtract 1 from the target since the labels start from 1
        train_loss.append(loss)
        loss.backward()
```

```
            # Gradient clipping
            if grad_clip:
                nn.utils.clip_grad_value_(model.parameters(), grad_clip)
            # Update the weights
            optimizer.step()
            scheduler.step()
```

Each iteration of this loop starts by zeroing the optimizer's gradients to ensure clean gradient calculations for each batch. The model then performs a forward pass to generate predictions, followed by a loss calculation, adjusting for label indexing starting from 1 by subtracting 1. After computing the loss, a backward pass calculates gradients, which are potentially clipped to prevent exploding gradients if specified. Subsequently, the optimizer updates the model's weights based on these gradients, and a learning rate scheduler adjusts the learning rate to optimize convergence, closing the loop until the next batch is processed.

An **EarlyStopping** helper class was defined in this section to enhance training efficiency by preventing unnecessary computations when the model ceases to show improvement. The **EarlyStopping** class is designed with a **patience** parameter, which dictates the number of epochs to continue training without any improvement in validation loss before halting the training process. This is set to 20 for a longer training progress in this project. The class monitors the validation loss during training: if the loss decreases, it resets an internal counter; if the loss fails to decrease, the counter increments. If the counter reaches the specified **patience** threshold, the **EarlyStopping** triggers a halt in training, signaling through an **early_stop** flag. This approach helps prevent overfitting and ensures computational resources are utilized only until beneficial learning occurs, making the training process both efficient and effective. The implemented is referenced from: https://stackoverflow.com/questions/71998978/early-stopping-in-pytorch

The testing or evalution process is implemented by the function below:

```
# evalueate the model on the validation set
def evaluate(model, accuracy, criterion, val_loader):
    model.eval()
    val_loss = 0
    validation_loss = []

    with torch.no_grad():
        for data, target in tqdm(val_loader, desc=f'Validation'):
            output = model(data)
            val_loss = criterion(output, target - 1).detach() #target - 1 because the labels start from 1
            validation_loss.append(val_loss)
            preds = torch.argmax(output, dim=1)  # Get the predicted class labels
```

```
        accuracy(preds , target-1)  # Update the running accuracy


    # Compute the overall accuracy and loss
    val_acc = accuracy.compute()
    accuracy.reset()
    overall_loss = torch.stack(validation_loss).mean().item()
```

This function switches the model to evaluation mode with **model.eval()**, ensuring that operations like dropout are suspended during the assessment. Using **torch.no_grad()**, it iterates over the validation dataset without calculating gradients, which saves memory and computations. For each batch, the function computes the loss after adjusting for the label indexing starting from 1, appends this loss to a list, and updates the accuracy metric based on the predictions. After processing all batches, the overall validation accuracy and the average loss are computed, providing a comprehensive measure of the model's performance on unseen data. This systematic evaluation helps in tuning the model and deciding when the training should be stopped to prevent overfitting, facilitated by mechanisms like early stopping.

Hyperparameter tuning

This section explored an integral part of the machine learning workflow—hyperparameter tuning using Optuna, a hyperparameter optimization framework. The objective function **objective(trial, train_data_dir, model_dir)** is meticulously designed to facilitate this process by evaluating various configurations of a ResNet model to identify the optimal set of parameters.

Hyperparameter Optimization Setup

1. **Initialization**: The function initializes the number of epochs, number of classes, and other fixed parameters such as patience for early stopping, image channels, and the directory path for training data. It also calculates the mean and standard deviation of the dataset for normalization purposes during data loading.
2. **Hyperparameter Space Definition**: Using Optuna, the function defines a search space for several hyperparameters:
    - **batch_size**: The size of each batch of data during training. Choice between 16, 32 and 64 because the limit of the computing resource.
    - **lr** (learning rate): A crucial parameter affecting the step size at each iteration while moving toward a minimum of the loss function. Was tested between 1e-4 adn 1e-1in log scale.
    - **weight_decay**: Regularization parameter to prevent overfitting by penalizing large weights. Tested between 1e-5 and 1e-1 in log scale.
    - **grad_clip**: Limits the values of gradients to a specific range to prevent the exploding gradient problem. Tested between 0.1 to 5.0 as float.
    - **state1_loop** and **state2_loop**: Specific to the ResNet model, defining how many times certain blocks are repeated. Was tested between 1 to 5 as integers.

- **dropout1** and **dropout2**: Dropout rates to prevent overfitting by randomly turning off a proportion of neurons during training. Was tested between 0 to 0.5 as float represent the proportionalaility of the dropout.
- **out_channels**: The number of output channels for the first convolutional layer as well as defining the futher layers, impacting the complexity of the model. Was tested between choices of 8, 16, 32, 64, 128
- **optimizer_name**: Type of optimizer to use, which can significantly affect the convergence and stability of training. Tested between 'Adam', 'SGD', and 'RMSprop'

3. **Model and Optimizer Initialization**: The function constructs the ResNet model with the specified hyperparameters and selects the optimizer based on the trial's suggestion.
4. **Data Loading**: It loads the training and validation data, applying transformations including normalization using the previously computed mean and standard deviation.
5. **Training Environment Setup**: Ensures the model and data are moved to the appropriate device (CPU or GPU) and initializes the criterion for loss calculation, accuracy metric for performance evaluation, and early stopping mechanism.
6. **Learning Rate Scheduler**: Implements the **OneCycleLR** scheduler to dynamically adjust the learning rate during training, promoting faster convergence and reducing the likelihood of training stagnation.
7. **Model Training**: Calls the **fit** function to train the model on the dataset, capturing training history and the best achieved accuracy.
8. **Results Collection**: The best accuracy from the training process is returned, allowing Optuna to gauge the effectiveness of the current set of hyperparameters.

After 30 trial of the tuning process, the hyperparameters of the model and training with the best performance are listed below:

```
Best trial:
  Accuracy: 0.7633333206176758
  Params:
    batch_size: 32
    lr: 0.004304870060658268
    weight_decay: 0.0007918823690228014
    grad_clip: 1.572382720992464
    state1_loop: 1
    state2_loop: 1
    dropout1: 0.212104912726848
    dropout2: 0.33715478476437205
    out_channels = 64
    optimizer: Adam
```

The Training and validation result curve of the model with best performance is plotted below:

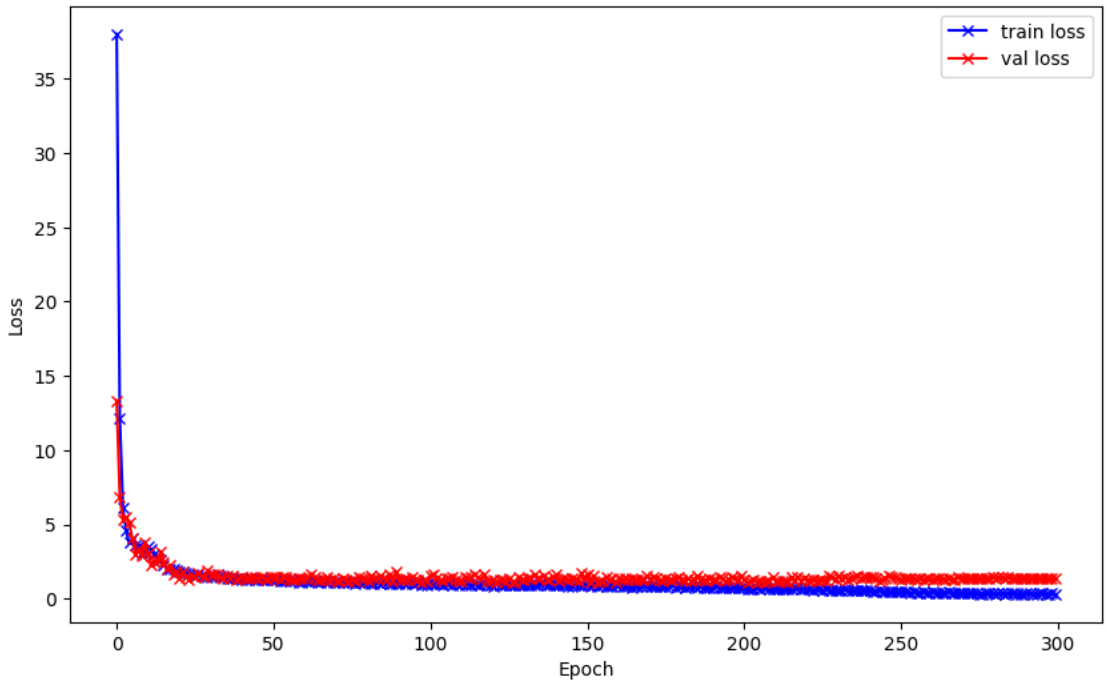

Figure 1: Training and Validation loss against epoch numbers

The variation of accuracy of the model in testing the validation dataset is plotted as below:

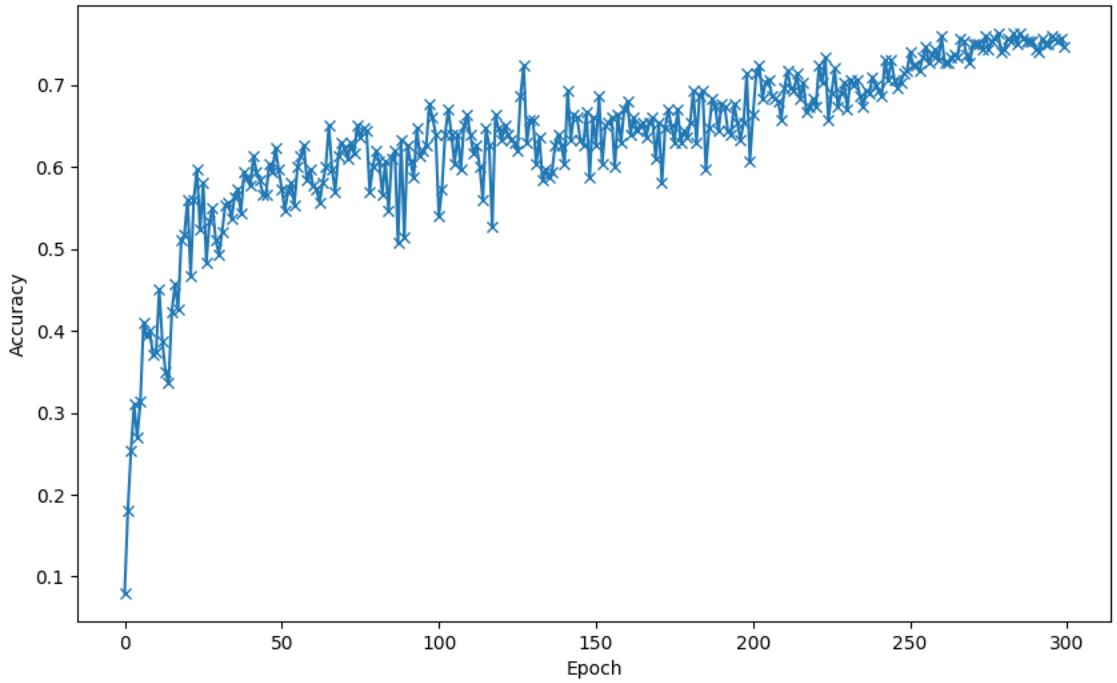

Figure 2: Accuracy against epoch numbers

In Figure 1, we can observe that the typical convergence behavior of a machine learning model, with both training and validation losses rapidly decreasing in the initial epochs before stabilizing to a relatively steady low value, indicating that the model is learning from the data with no severe overfitting problem as both validation and train loss are decreasing. Figure 2 shows the

model's accuracy on the validation dataset across epochs, starting with a swift increase and then fluctuating around an upward trend, suggesting that the model's predictive performance is improving, although with some variance between epochs.

The whole project is finally packaged into the **train** and **test** function:

The **train** function calls the **optuna.create_study** function to start the overall training and hyperparameter tuning progress in the direction of maximizing the validation accuracy. It passes the **train_data_dir**, **model_dir** to the specific function for the data loading process and model saving process. After all trials done, it returns the best training accuracy by vising the **trial_.value** parameter.

The **test** function will check if the **model_dir** exist first, if not, meaning there is no trained model in the directory, an warning message that need to train the model first will popup, if the model exist, the model would be loaded directly from **model_dir** using **torch.load()**. The validation dataset was loaded by setting the **train_size** argument of the **load_data** to 0. And diretly tested using the **evaluate()** function. The validation accuracy would be returned as the result of the test.