Image Classification Model [Srijan Gyawali]

This repository contains the implementation of an image classification model using PyTorch. The model utilizes separable convolutions and squeeze-and-excitation (SE) blocks to enhance performance.

Table of Contents

Data Preprocessing

This document describes the steps required to prepare your dataset for training and validation.

1. Dataset Structure

The Seen dataset was provided in the structure as below:\

```
Seen Datasets/\
├── train/\
|   ├── Common-Kingfisher/\
|   |   ├── Common-Kingfisher_2.jpg\
|   |   ├── Common-Kingfisher_3.jpg\
|   |   └── ...\
|   ├── Cattle-Egret/\
|   |   ├── Cattle-Egret_1.jpg\
|   |   ├── Cattle-Egret_3.jpg\
|   |   └── ...\
|   └── ...\
└── val/\
  ├── Common-Kingfisher/\
  |   ├── Common-Kingfisher_1.jpg\
  |   ├── Common-Kingfisher_4.jpg\
  |   └── ...\
  ├── Cattle-Egret/\
  |   ├── Common-Kingfisher_1.jpg\
  |   ├── Common-Kingfisher_16.jpg\
  |   └── ...\
  └── ...\
```

2. Providing path for Train and Validation set

```
train_dir = r'Seen Datasets\train'
val_dir = r'Seen Datasets\val'
```

3. Transformation of image

```
transformations_to_perform = transform=tt.Compose([
                                tt.Resize(image_size),
                                tt.ToTensor(),
                                tt.Normalize(mean, std)])

train_ds = ImageFolder(train_dir, transformations_to_perform)
valid_ds = ImageFolder(val_dir,transformations_to_perform)
```

`image_size` : The image size was provided which was 416*416

4. Mean and Standard Deviation Calculation

This section of the code prepares the datasets and calculates the mean and standard deviation of the pixel values in the dataset. These statistics are essential for normalizing the dataset, which helps in improving the performance and stability of machine learning models.

1. Load the Dataset

```
train_set = ImageFolder(train_dir, transform=tt.Compose([
    tt.ToTensor()
]))

val_set = ImageFolder(val_dir, transform=tt.Compose([
    tt.ToTensor()
]))
```

2. Concatenate Datasets

The training and validation datasets are concatenated into a single dataset using ConcatDataset. This combined dataset will be used to calculate the mean and standard deviation.

```
dataset = ConcatDataset([train_set, val_set])
```

`ConcatDataset`: A utility from torch.utils.data that concatenates multiple datasets.

3. Create a DataLoader

A DataLoader is created from the concatenated dataset to load the data in batches. This is useful for iterating over the dataset efficiently.

```
dataset_dl = DataLoader(dataset, batch_size, shuffle=True)
```

`batch_size`: Number of images to load in each batch.\
`shuffle`: Whether to shuffle the dataset every epoch.

4. Calculate Mean and Standard Deviation

The get_mean_and_std function calculates the mean and standard deviation of the pixel values in the dataset. It iterates through the dataset, computes the sum and squared sum of the pixel values for each channel, and then calculates the mean and standard deviation.

```
def get_mean_and_std(dataloader):
    channels_sum, channels_squared_sum, num_batches = 0, 0, 0
    for data, _ in tqdm(dataloader):
        channels_sum += torch.mean(data, dim=[0, 2, 3])
        channels_squared_sum += torch.mean(data**2, dim=[0, 2, 3])
        num_batches += 1

    mean = channels_sum / num_batches
    std = (channels_squared_sum / num_batches - mean ** 2) ** 0.5
    return mean, std
```

`dataloader`: DataLoader instance to load the dataset in batches.\
`channels_sum`: Sum of the pixel values for each channel.\
`channels_squared_sum`: Sum of the squared pixel values for each channel.\
`num_batches`: Total number of batches processed.\
`mean`: Calculated mean of the pixel values for each channel.\
`std`: Calculated standard deviation of the pixel values for each channel.

Model Building

The model architecture is defined using PyTorch, with a base class for common operations and a specific implementation for the image classification task.

1. Base Class for Image Classification

The `ImageClassificationBase` class contains methods for training and validation steps, as well as for calculating accuracy and logging epoch results.

```
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))

class ImageClassificationBase(nn.Module):
    def training_step(self, batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out, labels)
        acc = accuracy(out, labels)
        return loss, acc

    def validation_step(self, batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out, labels)
        acc = accuracy(out, labels)
        return {'val_loss': loss.detach(), 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], train_loss: {:.4f},train_acc: {:.4f}, val_loss: {:.4f}, val_acc: {:.4f}".format(
            epoch, result['train_loss'],result['train_acc'], result['val_loss'], result['val_acc']))
```

2. Squeeze-and-Excitation (SE) Block

The `SEBlock` class implements the Squeeze-and-Excitation mechanism to recalibrate channel-wise feature responses.

```
class SEBlock(nn.Module):
    def __init__(self, in_channels, reduction=16):
        super(SEBlock, self).__init__()
        self.fc1 = nn.Linear(in_channels, in_channels // reduction, bias=False)
        self.fc2 = nn.Linear(in_channels // reduction, in_channels, bias=False)

    def forward(self, x):
        batch_size, num_channels, _, _ = x.size()
        y = F.adaptive_avg_pool2d(x, 1).view(batch_size, num_channels)
        y = F.relu(self.fc1(y))
        y = torch.sigmoid(self.fc2(y)).view(batch_size, num_channels, 1, 1)
        return x * y.expand_as(x)
```

Initialization (init method):

`in_channels`: The number of input channels.\
`reduction`: The reduction ratio, which controls the bottleneck in the block. The default value is 16.\
`fc1`: A fully connected layer that reduces the number of channels by the reduction ratio.\
`fc2`: A fully connected layer that restores the number of channels to the original number.\

Forward Pass (forward method):

`x`: The input tensor with shape (batch_size, num_channels, height, width).\
`F.adaptive_avg_pool2d(x, 1)`: Applies adaptive average pooling to reduce the spatial dimensions to 1x1, resulting in a tensor of shape (batch_size, num_channels, 1, 1).\
`view(batch_size, num_channels)`: Reshapes the tensor to (batch_size, num_channels) for the fully connected layers.\
`F.relu(self.fc1(y))`: Applies the first fully connected layer followed by a ReLU activation function. This reduces the number of channels by the reduction ratio.\
`torch.sigmoid(self.fc2(y))`: Applies the second fully connected layer followed by a sigmoid activation function. This restores the number of channels to the original number and squashes the output to the range [0, 1].\
`view(batch_size, num_channels, 1, 1)`: Reshapes the tensor back to (batch_size, num_channels, 1, 1).\
`x * y.expand_as(x)`: Multiplies the original input tensor x by the recalibrated weights y, which are broadcasted to match the spatial dimensions of x.

Purpose

The purpose of the SE block is to enhance the representational power of a network by explicitly modeling channel interdependencies. It does this by recalibrating the feature maps (channels) through a process of squeezing (global information embedding) and excitation (adaptive recalibration).

3. Convolutional and Separable Convolutional Blocks

Create standard and separable convolutional blocks.

```
 def conv(in_channels, out_channels, kernel_size=3, stride=1, padding=1, groups=1, use_se_block=False):
    layers = [
        nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding, groups=groups),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True)
    ]
    if use_se_block:
        layers.append(SEBlock(out_channels))
    return nn.Sequential(*layers)

def SeparableConv(in_channels, out_channels, use_se_block=False):
    layers = [
        nn.Conv2d(in_channels, in_channels, kernel_size=3, padding=1, groups=in_channels),
        nn.Conv2d(in_channels, out_channels, kernel_size=1),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True)
    ]
    if use_se_block:
        layers.append(SEBlock(out_channels))
    return nn.Sequential(*layers)
```

4. Fully Connected Layer with Dropout

A function to create a fully connected layer with dropout.

```
def linear(in_features, out_features, dropout_rate=0.3):
    return nn.Sequential(
        nn.Dropout(dropout_rate),
        nn.Linear(in_features, out_features),
        nn.BatchNorm1d(out_features),
        nn.ReLU(inplace=True)
    )
```

5. Image Classification Model

The `ImgClassifier` class defines the complete model architecture, using convolutional and separable convolutional blocks, as well as fully connected layers.

```
class ImgClassifier(ImageClassificationBase):
    def __init__(self, output_dim):
        super(ImgClassifier, self).__init__()

        self.features = nn.Sequential(
            conv(3, 64, stride=2, use_se_block=True),
            nn.MaxPool2d(2),

            SeparableConv(64, 128, use_se_block=True),
            nn.MaxPool2d(2),

            SeparableConv(128, 256, use_se_block=True),
            nn.MaxPool2d(2),

            SeparableConv(256, 512, use_se_block=True),
            nn.MaxPool2d(2),

            SeparableConv(512, 512, use_se_block=True),
            nn.MaxPool2d(2),

            conv(512, 512, use_se_block=True)
        )

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))

        self.classifier = nn.Sequential(
            linear(512, 1024),
            nn.Linear(1024, output_dim)
        )
```

Model Architecture

Input Layer

- **Input**: Image (3 x 416 x 416)

Sequential Block 1

- **Conv2d**: (3, 64, 3, 2, 1)
    - **Params**: 1,728
- **BatchNorm2d**: (64)
    - **Params**: 128
- **ReLU**
- **SEBlock**: (64)
    - **fc1**: (64, 4)
        - **Params**: 256
    - **fc2**: (4, 64)
        - **Params**: 256
- **MaxPool2d**: (2)

Sequential Block 2

- **Conv2d**: (64, 128, 3, 1, 1)
- **Depthwise Conv2d**: (128, 128, 3, padding=1, groups=128)
  - **Params**: 1,152
- **Pointwise Conv2d**: (128, 128, 1)
  - **Params**: 16,384
- **BatchNorm2d**: (128)
  - **Params**: 128
- **ReLU**
- **SEBlock**: (128)
  - **fc1**: (128, 8)
    - **Params**: 1,024
  - **fc2**: (8, 128)
    - **Params**: 1,024
- **MaxPool2d**: (2)

Sequential Block 3

- **Conv2d**: (128, 256, 3, 1, 1)
- **Depthwise Conv2d**: (256, 256, 3, padding=1, groups=256)
  - **Params**: 2,304
- **Pointwise Conv2d**: (256, 256, 1)
  - **Params**: 65,536
- **BatchNorm2d**: (256)
  - **Params**: 256
- **ReLU**
- **SEBlock**: (256)
  - **fc1**: (256, 16)
    - **Params**: 4,096
  - **fc2**: (16, 256)
    - **Params**: 4,096
- **MaxPool2d**: (2)

Sequential Block 4

- **Conv2d**: (256, 512, 3, 1, 1)
- **Depthwise Conv2d**: (512, 512, 3, padding=1, groups=512)
  - **Params**: 4,608
- **Pointwise Conv2d**: (512, 512, 1)
  - **Params**: 262,144
- **BatchNorm2d**: (512)
  - **Params**: 512
- **ReLU**
- **SEBlock**: (512)
  - **fc1**: (512, 32)
    - **Params**: 16,384
  - **fc2**: (32, 512)
    - **Params**: 16,384
- **MaxPool2d**: (2)

Sequential Block 5

- **Conv2d**: (512, 512, 3, 1, 1)
- **Depthwise Conv2d**: (512, 512, 3, padding=1, groups=512)
  - **Params**: 4,608
- **Pointwise Conv2d**: (512, 512, 1)
  - **Params**: 262,144
- **BatchNorm2d**: (512)
  - **Params**: 512
- **ReLU**

- **SEBlock**: (512)
    - **fc1**: (512, 32)
        - **Params**: 16,384
    - **fc2**: (32, 512)
        - **Params**: 16,384

Classifier

- **Linear (512, 512)**
    - **Params**: 262,656
- **BatchNorm1d (512)**
    - **Params**: 1,024
- **Linear (512, 1024)**
    - **Params**: 525,312
- **BatchNorm1d (1024)**
    - **Params**: 2,048
- **Linear (1024, 25)**
    - **Params**: 25,625

Total Parameters

- **Total Params**: 3,472,793

```
Input Layer(3x416x416)
  |
Sequential Block 1
  |-- Conv2d (3, 64, 3, 2, 1) [1,728]
  |-- BatchNorm2d (64) [128]
  |-- ReLU
  |-- SEBlock (64)
  |   |-- fc1 (64, 4) [256]
  |   |-- fc2 (4, 64) [256]
  |-- MaxPool2d (2)
  |
Sequential Block 2
  |-- Conv2d (64, 128, 3, 1, 1)
  |   |-- Depthwise (128, 128, 3) [1,152]
  |   |-- Pointwise (128, 128, 1) [16,384]
  |-- BatchNorm2d (128) [128]
  |-- ReLU
  |-- SEBlock (128)
  |   |-- fc1 (128, 8) [1,024]
  |   |-- fc2 (8, 128) [1,024]
  |-- MaxPool2d (2)
  |
Sequential Block 3
  |-- Conv2d (128, 256, 3, 1, 1)
  |   |-- Depthwise (256, 256, 3) [2,304]
  |   |-- Pointwise (256, 256, 1) [65,536]
  |-- BatchNorm2d (256) [256]
  |-- ReLU
  |-- SEBlock (256)
  |   |-- fc1 (256, 16) [4,096]
  |   |-- fc2 (16, 256) [4,096]
  |-- MaxPool2d (2)
  |
Sequential Block 4
  |-- Conv2d (256, 512, 3, 1, 1)
  |   |-- Depthwise (512, 512, 3) [4,608]
  |   |-- Pointwise (512, 512, 1) [262,144]
  |-- BatchNorm2d (512) [512]
  |-- ReLU
  |-- SEBlock (512)
  |   |-- fc1 (512, 32) [16,384]
  |   |-- fc2 (32, 512) [16,384]
  |-- MaxPool2d (2)
  |
Sequential Block 5
  |-- Conv2d (512, 512, 3, 1, 1)
  |   |-- Depthwise (512, 512, 3) [4,608]
  |   |-- Pointwise (512, 512, 1) [262,144]
  |-- BatchNorm2d (512) [512]
  |-- ReLU
  |-- SEBlock (512)
  |   |-- fc1 (512, 32) [16,384]
  |   |-- fc2 (32, 512) [16,384]
  |
Classifier
  |-- Linear (512, 512) [262,656]
  |-- BatchNorm1d (512) [1,024]
  |-- Linear (512, 1024) [525,312]
  |-- BatchNorm1d (1024) [2,048]
  |-- Linear (1024, 25) [25,625]

Total params: 3,472,793
Trainable params: 3,472,793
Non-trainable params: 0
```

Training Pipeline

The training pipeline involves setting up the model training process, including early stopping to prevent overfitting, evaluating the model, and training it over multiple epochs. Below are the components and steps for the training process:

1. Early Stopping

Early stopping helps to stop the training process when the model's performance on the validation set stops improving, preventing overfitting.

```
class EarlyStopping:
    def __init__(self, patience=5, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_score = None
        self.early_stop = False

    def __call__(self, val_loss, model):
        score = -val_loss

        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
        elif score < self.best_score + self.min_delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
            self.counter = 0

    def save_checkpoint(self, val_loss, model):
        torch.save(model.state_dict(), 'checkpoint.pt')
```

2. Evaluation Function

The evaluation function runs the model on the validation data and computes the validation loss and accuracy.

```
@torch.no_grad()
def evaluate(model, val_loader):
    model.eval()
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)
```

3. Training Loop

The training loop involves iterating over the dataset for a specified number of epochs. During each epoch, the model performs forward and backward passes, and the optimizer updates the model's weights. The loop includes gradient clipping to avoid exploding gradients and uses a learning rate scheduler.

```
def modeltrain(epochs, lr, model, train_loader, val_loader,
                  weight_decay=0, grad_clip=None, opt_func=torch.optim.Adam):
    history = []
    optimizer = opt_func(model.parameters(), lr, weight_decay=weight_decay)
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=5, factor=0.5)
    early_stopping = EarlyStopping(patience=7, min_delta=0.001)
    torch.cuda.empty_cache()

    for epoch in range(epochs):
        model.train()
        train_losses = []
        train_accs = []

        for batch in tqdm(train_loader):
            batch = to_device(batch, device)
            loss, acc = model.training_step(batch)
            train_losses.append(loss)
            train_accs.append(acc)
            loss.backward()

            if grad_clip:
                nn.utils.clip_grad_value_(model.parameters(), grad_clip)

            optimizer.step()
            optimizer.zero_grad()

        result = evaluate(model, val_loader)
        result['train_loss'] = torch.stack(train_losses).mean().item()
        result['train_acc'] = torch.stack(train_accs).mean().item()
        scheduler.step(result['val_loss'])
        early_stopping(result['val_loss'], model)

        if early_stopping.early_stop:
            print("Early stopping")
            break

        model.epoch_end(epoch, result)
        history.append(result)

    model.load_state_dict(torch.load('checkpoint.pt'))

    return history
```

This pipeline handles the complete training process including setting up early stopping, evaluating the model performance, and managing the training loop. Adjust the parameters and hyperparameters as necessary for your specific use case.

Saving the Model

After training the model, it's important to save it so that you can reload it later for inference or further training. The following steps outline how to save both the model's state and its scripted version.

1. Save the Model State

You can save the model's state dictionary, which contains all the parameters learned during training. This allows you to load the model later without retraining.

```
torch.save(model, 'model.pth')
```

2. Save a Scripted Version

```
model_scripted = torch.jit.script(model)
model_scripted.save('modelscripted.pt')
```

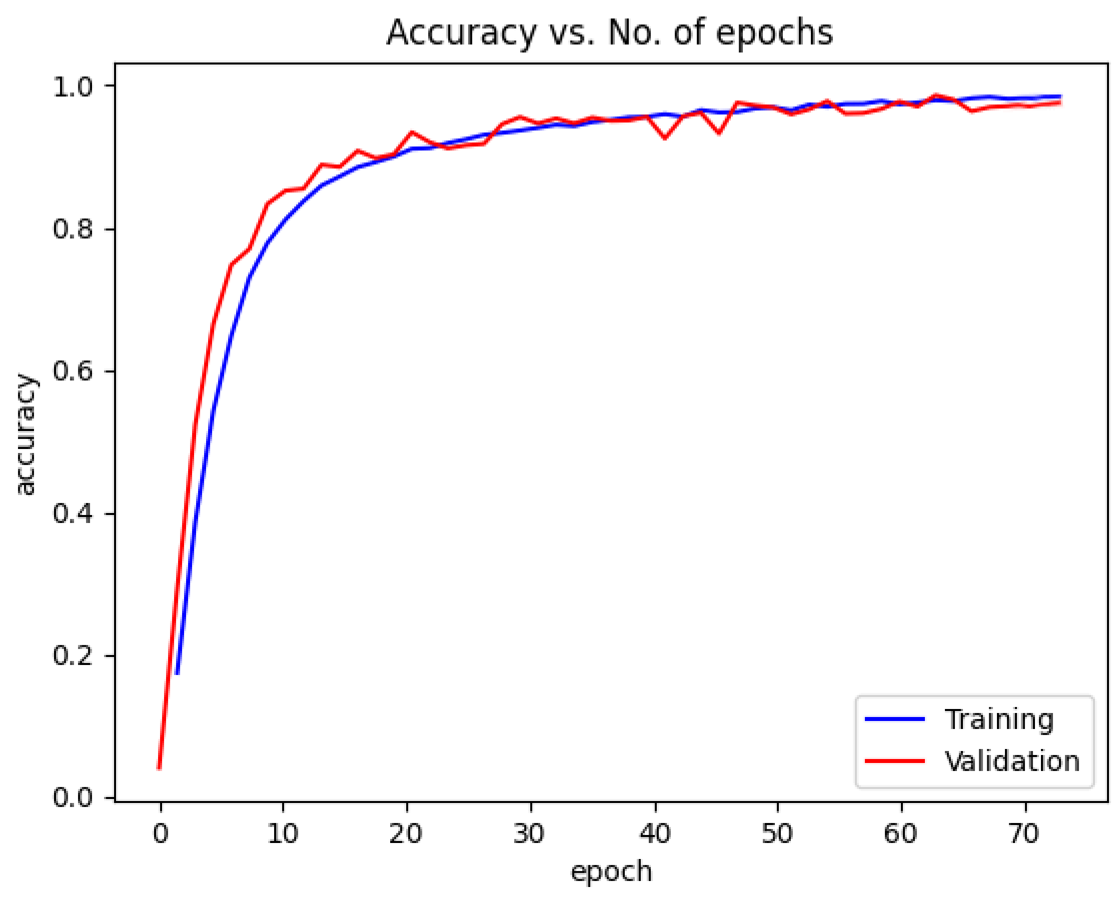
Plotting Metrics

Visualizing training metrics such as accuracy and loss can help you understand the model's performance over epochs.

1. Plot Accuracy

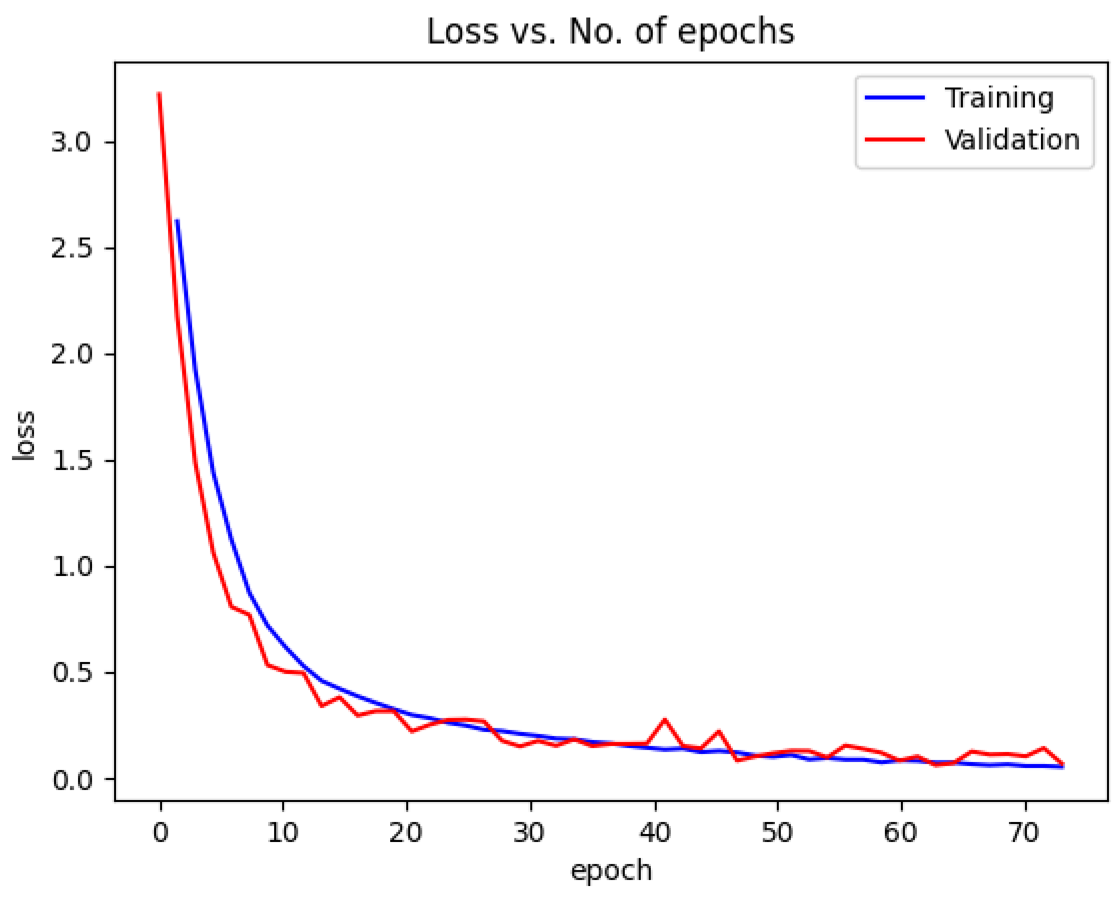
To plot the accuracy over epochs, use the following code:

```
def plot_accuracies(history):
    train_accs = [x.get('train_acc') for x in history]
    val_accs = [x['val_acc'] for x in history]
    plt.plot(train_accs, '-b')
    plt.plot(val_accs, '-r')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend(['Training', 'Validation'])
    plt.title('Accuracy vs. Number of Epochs')
    plt.savefig('AccuracyVsEpoch.png')
```



2. Plot Loss

```
def plot_losses(history):
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x['val_loss'] for x in history]
    plt.plot(train_losses, '-b')
    plt.plot(val_losses, '-r')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend(['Training', 'Validation'])
    plt.title('Loss vs. Number of Epochs')
    plt.savefig('LossVsEpoch.png')
```



Author

Srijan Gyawali

srijangyawali0@gmail.com