Aalto University
School of Electrical
Engineering

Vehicle Type Classification based on CNN

1. Introduction

Autonomous driving is a revolutionary technology designed to enable cars to drive safely without human intervention. Its realization requires the vehicle to be able to sense its surroundings, understand road conditions and the behavior of other traffic participants, make appropriate decisions, and control vehicle actions. Image categorization is a computer vision task whose goal is to classify input images into different categories. This task is usually realized using machine learning techniques, one of the most common approaches is Convolutional Neural Networks (CNN). In our study, we will classify the input images of different types of vehicles such as bus, car, motorcycle, and truck by CNN. This technique will help us to understand and explore autonomous driving initially and will be useful for practical applications.

2. Problem Formulation:

In our machine learning endeavor, we've leveraged the ResNet architecture within the framework of Convolutional Neural Networks (CNN). ResNet, proposed by Kaiming He et al. in 2015[1], is renowned for its introduction of Residual Blocks, a concept that effectively addresses the vanishing gradient problem commonly encountered when training deep networks. ResNet employs residual learning, enabling the network to focus on learning residuals, which represent differences between input and output data, rather than learning the entire mapping directly.

In our study, data points are images of individual vehicles. Each data point is a single image containing a vehicle, and these images are used as inputs to the machine learning model. Features are extracted from vehicle images such as color, texture, shape, and pattern. When using ResNet 34-based models, features are typically learned representations or embeddings extracted from specific layers of the ResNet model. These embeddings capture high-level information about the appearance and characteristics of the vehicle. Labels indicate the class or type that each vehicle image belongs to, categorized as bus, car, motorcycle, or truck. Each data point (image) is

associated with a label indicating the type of vehicle to be predicted by the model. So the type of machine learning task is supervised.

We selected a data set called Vehicle Type Recognition on Kaggle[2]. This data set contains pictures of four different types of vehicles, namely bus, car, motorcycle, and truck, with 100 pictures for each type. Below is a detailed description of each type:

- Car: Images of different car models and types, captured from various angles and under different lighting conditions.
- Truck: Images of different types of trucks, including pickup trucks, delivery trucks, and heavy-duty trucks.
- Bus: Images of buses used for public transportation, school buses, and other types of buses.
- Motorcycle: Images of motorcycles and motorbikes.

3. Methods

There are 400 images in the dataset, 100 in each category. In order to transform the images into data that can be used in a machine learning problem, we need to preprocess the original dataset by following steps: first, a data transformation is needed: a data transformation pipeline is created, which consists of three transformation steps: the input images are resized to 224*224 pixels in order to fit the model's input requirements. The image is then transformed into a PyTorch tensor. Finally, the tensor is normalized using predefined mean and standard deviation.

In ResNet, feature selection is done implicitly by Skip connections and deep design. Skip connections are when each residual block in ResNet includes two or more convolutional layers. The key idea is that instead of trying to learn the desired underlying mapping directly, ResNet tries to learn the residual mapping. The output of a block is the sum of its inputs and the learned residuals. Skip connections ensure that the features of the original input are preserved and passed directly to deeper layers. By retaining the input features, the network retains the ability to focus on both low-level and high-level features.

Mathematically, if $F(x)$ represents the output of the block, it is computed as

$$F(x) = H(x) + x$$

Where $H(x)$ represents the residual to be learned. The skip connection (x) simply passes the input directly to the output without any change.

In our machine learning approach, we created the ResNet-34 model and loaded it with pre-trained weights. The model is then trained in multiple rounds (30 rounds or more), with each round consisting of the following steps:

- Set up the model in training mode and initialize the losses.
- Traverse the training dataset, load the data and forward propagate, compute the loss, backpropagate, and update the weights.
- Output the training loss.
- Set the model to evaluation mode and calculate the accuracy of the validation set.
- Output the training loss and validation accuracy for each round and save the best-performing model weights.

For the loss function, we used the cross-entropy loss function, this is because the cross-entropy loss function is suitable for multi-category classification, for each sample, this loss function measures the difference between the output probability distribution of the model and the distribution of the real label. Meanwhile, the cross-entropy loss function outputs probability distributions, and comparing them with the probability distributions of the real labels measures the performance of the model. In addition, the cross-entropy loss function provides gradient information during training, which helps optimize the weights of the model so that it can better adapt to the training data. Not only that, the cross-entropy loss function has good mathematical properties, and properties such as logarithmic transformation make it more stable during optimization.

In our ResNet 34 model, the training set, validation set, and test set were set up separately. The 4*100 images in the Vehicle Type Recognition dataset were randomly divided into a training set (4*80) and a validation set (4*20) in the ratio of 8:2. The validation set was used to verify the training results immediately after each training was completed. In addition, we set up 80 separate images of different types of vehicles for testing, 20 of each type.

Aalto University
School of Electrical
Engineering

Reference

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun; Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-778

[2] Vehicle Type Recognition, https://www.kaggle.com/datasets/kaggleashwin/vehicle-type-recognition

Aalto University
School of Electrical
Engineering

Appendices

1. train

```
import torchvision.models.resnet
import json
import os
import torch
import torch.nn as nn
import torchvision.transforms
from torch.utils.data import DataLoader
from model import *
import torch.optim
from tqdm import tqdm
from torch.utils.tensorboard import SummaryWriter

device = torch.device("cuda:0" if torch.cuda.is_available() else
'cpu')
print("use {} to train".format(device))

data_transform = {
    "train":
torchvision.transforms.Compose([torchvision.transforms.RandomResize
dCrop(224),

torchvision.transforms.RandomHorizontalFlip(),

torchvision.transforms.ToTensor(),

torchvision.transforms.Normalize((0.485, 0.456, 0.406),(0.229,
0.224, 0.225))]),
    "val":
torchvision.transforms.Compose([torchvision.transforms.Resize(256),

torchvision.transforms.CenterCrop(224),

torchvision.transforms.ToTensor(),

torchvision.transforms.Normalize((0.485, 0.456, 0.406), (0.229,
0.224, 0.225))])}
```

```
train_image_path = r"D:\python\car_classifier\train"
val_image_path = r"D:\python\car_classifier\val"

writer = SummaryWriter('logs')

train_dataset = torchvision.datasets.ImageFolder(root =
train_image_path, transform=data_transform["train"])
train_num = len(train_dataset)

flower_list = train_dataset.class_to_idx
cla_dict = dict((val, key) for key, val in flower_list.items())
json_str = json.dumps(cla_dict, indent=4)
with open("class_indices.json", 'w') as json_file:
    json_file.write(json_str)

batch_size = 16

train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True, num_workers=0)

validate_dataset =
torchvision.datasets.ImageFolder(root=val_image_path,
transform=data_transform["val"])
val_num = len(validate_dataset)
validate_loader = DataLoader(validate_dataset, shuffle=True,
num_workers=0,batch_size=batch_size)

print("using {} images for training, {} images for
validation.".format(train_num, val_num))

net = resnet34()
model_weight_path = "./resnet34-pre.pth"
missing_keys, unexpected_keys =
net.load_state_dict(torch.load(model_weight_path), strict=False)
inchannel = net.fc.in_features
net.fc = nn.Linear(inchannel, 4)
net.to(device)

loss_function = torch.nn.CrossEntropyLoss()
```

```
loss_function = loss_function.to(device)

optimizer = torch.optim.Adam(net.parameters(), lr=0.0003)

epochs = 30
best_acc = 0.0
save_path = "./resnet34.pth"

train_steps = len(train_loader)

for epoch in range(epochs):
    net.train()
    running_loss = 0.0
    train_bar = tqdm(train_loader)
    for step, data in enumerate(train_bar):
        images, labels =data
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        output = net(images)
        loss = loss_function(output, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        train_bar.desc = "train epoch[{}/{}]
loss:{:.3f}".format(epoch+1, epochs, running_loss)

    net.eval()
    acc = 0.0
    with torch.no_grad():
        val_var = tqdm(validate_loader)
        for val_data in val_var:
            val_images, val_labels = val_data
            val_images = val_images.to(device)
            val_labels = val_labels.to(device)
            outputs = net(val_images)
            predict_y = torch.max(outputs, dim=1)[1]
            acc += torch.eq(predict_y, val_labels).sum().item()
```

```
            val_var.desc = "val epoch[{}/{}]".format(epoch + 1,
epochs)

    val_accurate = acc / val_num
    print("[epoch %d] train_loss: %.3f  val_accuracy: %.3f" %
          (epoch + 1 ,running_loss / train_steps, val_accurate))

    if val_accurate > best_acc:
        best_acc = val_accurate
        torch.save(net.state_dict(), save_path)

    writer.add_scalar("accuracy", val_accurate, epoch+1)

writer.close()
print("Finished Training")
```

2. model

```
import torch
import torch.nn as nn

class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_channel, out_channel, stride=1,
downsample = None):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=in_channel,
out_channels=out_channel, stride=stride,
                               kernel_size=3, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channel)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=out_channel,
out_channels=out_channel,
                               kernel_size=3, stride=1, padding=1,
bias=False)
        self.bn2 = nn.BatchNorm2d(out_channel)
        self.downsample = downsample

    def forward(self, x):
        identity = x
```

```
        if self.downsample is not None:
            identity = self.downsample(x)

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        out += identity
        out = self.relu(out)

        return out

class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, in_channel, out_channel, stride=1,
downsample=None):
        super(Bottleneck, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=in_channel,
out_channels=out_channel,
                               stride=1, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channel)

        self.conv2 = nn.Conv2d(in_channels=out_channel,
out_channels=out_channel,
                               kernel_size=3, stride=stride, bias=False,
padding=1)
        self.bn2 = nn.BatchNorm2d(out_channel)

        self.conv3 = nn.Conv2d(in_channels=out_channel,
out_channels=out_channel * self.expansion,
                               kernel_size=1, stride=1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_channel * self.expansion)

        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
```

```
    def forward(self, x):
        identity = x
        if self.downsample is not None:
            identity = self.downsample(x)

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        out += identity
        out = self.relu(out)

        return out

class ResNet(nn.Module):
      def __init__(self, block, blocks_num, num_classes=1000,
include_top=True):
        super(ResNet, self).__init__()
        self.include_top = include_top
        self.in_channel = 64

        self.conv1 = nn.Conv2d(3, self.in_channel, kernel_size=7,
stride=2,
                               padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(self.in_channel)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2,
padding=1)

        self.layer1 = self._make_layer(block, 64, blocks_num[0])
        self.layer2 = self._make_layer(block, 128, blocks_num[1],
stride=2)
```

```
        self.layer3 = self._make_layer(block, 256, blocks_num[2],
stride=2)
        self.layer4 = self._make_layer(block, 512, blocks_num[3],
stride=2)

        if self.include_top:
            self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
            self.fc = nn.Linear(512 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode="fan_out",
nonlinearity="relu")

    def _make_layer(self, block, channel, block_num, stride=1):
        downsample = None

        if stride != 1 or self.in_channel != channel *
block.expansion:
                downsample = nn.Sequential(
                nn.Conv2d(self.in_channel, channel * block.expansion,
kernel_size=1,
                          stride=stride, bias=False),
                nn.BatchNorm2d(channel * block.expansion))

        layers = []
        layers.append(block(self.in_channel, channel, stride=stride,
downsample=downsample))
        self.in_channel = channel * block.expansion

        for _ in range(1, block_num):
            layers.append(block(self.in_channel, channel))

        return nn.Sequential(*layers)
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
```

```
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        if self.include_top:
            x = self.avgpool(x)
            x = torch.flatten(x, 1)
            x = self.fc(x)

        return x

def resnet34(num_classes=1000, include_top=True):
    return ResNet(BasicBlock, [3, 4, 6, 3],
num_classes=num_classes, include_top=include_top)

def resnet101(num_classes=1000, include_top=True):
    return ResNet(Bottleneck, [3, 4, 23, 3],
num_classes=num_classes, include_top=include_top)
```

3. detect

```
import json
import torch
from torchvision import transforms
import matplotlib.pyplot as plt
import os
from PIL import Image
from model import *
from torch.utils.tensorboard import SummaryWriter

device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")

data_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224,
0.225))])

writer = SummaryWriter('logs')
```

```
img_path = r"D:\python\car_classifier\test/1.jpg"
assert os.path.exists(img_path), "file: '{}' does not
exist.".format(img_path)
img = Image.open(img_path)
plt.imshow(img)
img = data_transform(img)
img = torch.unsqueeze(img, dim=0)

json_path = "./class_indices.json"
assert os.path.exists(json_path), "file: '{}' does not
exist.".format(json_path)

with open(json_path, 'r') as f:
    class_dict = json.load(f)

model = resnet34(num_classes=4).to(device)
weights_path = "./resnet34.pth"
assert os.path.exists(weights_path), "file: '{}' does not
exist.".format(weights_path)
model.load_state_dict(torch.load(weights_path))


model.eval()
with torch.no_grad():
    img = img.to(device)
    output = torch.squeeze(model(img)).cpu()
    predict = torch.softmax(output, dim=0)
    predict_cla = torch.argmax(predict).numpy()

print_res = "class:{}
prob:{:.3}".format(class_dict[str(predict_cla)],
predict[predict_cla].numpy())
plt.title(print_res)

len = len(predict)
print("{} classes".format(len))
for i in range(len):
    print("class:{} prob:{:.6f}".format(class_dict[str(i)],
predict[i].numpy()))
```

```
    writer.add_scalar("prediction probility", predict[i].numpy(), i)

writer.close()
plt.show()
```