Aalto University
School of Electrical
Engineering

Vehicle Type Classification based on CNN and KNN

1. Introduction

Autonomous driving is a groundbreaking technology that allows cars to operate safely without human control. To achieve this, vehicles must sense their surroundings and make decisions independently. Image categorization, a key aspect of autonomous driving, often relies on machine learning techniques like Convolutional Neural Networks (CNNs). In our study, we will use CNNs to classify vehicle types, such as buses, cars, motorcycles, and trucks, from input images. This approach will enhance our understanding of autonomous driving and have practical applications.

In section 2, we outline our problem model. Moving on to section 3, we delve into the principles and specific steps of two distinct machine learning methods: CNN (ResNet) and k-nearest neighbor (KNN). Section 4 presents our analysis of the results obtained using these two methods, with CNN achieving a significantly higher accuracy rate in image classification compared to KNN. Finally, we wrap up our findings with a summary in section 5.

2. Problem Formulation

In our problem formulation, we've utilized the ResNet architecture within the CNN framework. ResNet, introduced by Kaiming He et al. in 2015[1], is renowned for its Residual Blocks, which address the vanishing gradient issue when training deep networks. ResNet adopts residual learning, focusing on learning the differences between input and output instead of attempting to learn the entire mapping directly.

In our study, we use individual vehicle images as data points for the model. These images serve as inputs, and we extract features like color, texture, shape, and pattern from them. When employing ResNet 34-based models, these features are typically learned representations or embeddings from specific ResNet layers, capturing high-level information about the vehicle's appearance. Labels indicate the vehicle's class, such as bus, car, motorcycle, or truck. Each image is associated with a label specifying the type of vehicle for prediction, making our machine learning task supervised.

3. Methods

• Data Set

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.

In our models, 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 8:2 ratio is a widely adopted practice because it performs well in many machine learning problems. This ratio is usually valid for small to medium-sized datasets. This provides a sufficient number of samples for training while allowing reliable evaluation of the model's performance. The validation set was used to verify the training results immediately after each training was completed.

- CNN (ResNet)

In ResNet, feature selection is achieved through skip connections and deep design. Skip connections involve multiple convolutional layers within each residual block. Rather than learning the desired mapping directly, ResNet focuses on learning the residual mapping. Each block's output is the sum of its inputs and the learned residuals. These skip connections preserve and pass the original input features to deeper layers, allowing the network to maintain a focus on both low-level and high-level features.

To prepare these images for machine learning, we undertake a preprocessing process. This involves creating a data transformation pipeline with three steps. First, resizing the input images to 224*224 pixels to match the requirements. Second, converting the image into a PyTorch tensor. Third, normalizing the tensor using predefined mean and standard deviation values.

In our study, we utilized the pre-trained ResNet-34 model. The training process involved multiple rounds (typically 30 or more), each comprising the following steps:

a. Prepare the model for training and initialize losses.
b. Iterate through the training data, perform forward and backward passes, and update model weights.
c. Track and display the training loss.
d. Evaluate the model on the validation set to compute accuracy.
e. Record training loss and validation accuracy, saving the best model weights in each round.

We employed the cross-entropy loss function for our multi-category classification task. This loss measures the disparity between the model's output probability distribution and the actual label distribution for each sample. It facilitates model performance assessment by comparing output and actual probability distributions. Moreover, the cross-entropy loss function yields gradient information during training, aiding in weight optimization for improved adaptation to the training data.

- KNN

K-Nearest Neighbors (KNN) is an intuitive machine learning algorithm for classification and regression tasks. Given a new data point, KNN identifies the K nearest data points in the training data set. "Nearest" is usually determined using a distance metric. For classification tasks, the KNN performs majority voting on the K nearest neighbors to assign class labels to the new data point.

The process of feature selection is common. First, we converted the image into a grayscale map. Then we resized it, converted the image into an array, and added labels to the different categories.

KNN uses the concept of a zero-one loss function, if the model's predicted category matches the true category exactly, the loss is 0. If the model's predicted category does not match the true category, the loss is 1. The 0-1 loss function is very intuitive and easy to understand. It transforms the performance measure of a classification problem into a simple metric.

In KNN method, first, the image is read to prepare the data and labels for classification, and the dataset is divided into a training set and a test set. Then the value of k (number of neighbors) is defined and KNN classifier. The model is trained on the training data and predicted on the test data. Finally, the performance of KNN classifier was evaluated. Accuracy scores and classification reports were calculated and printed, which included precision, recall, F1 score, and support for each category on the test data.

4. Results

In order to make the final test results representative, we set up 80 separate images of different types of vehicles for testing, 20 of each type. These images were chosen completely randomly, partly from the Kaggle dataset and partly from the internet.

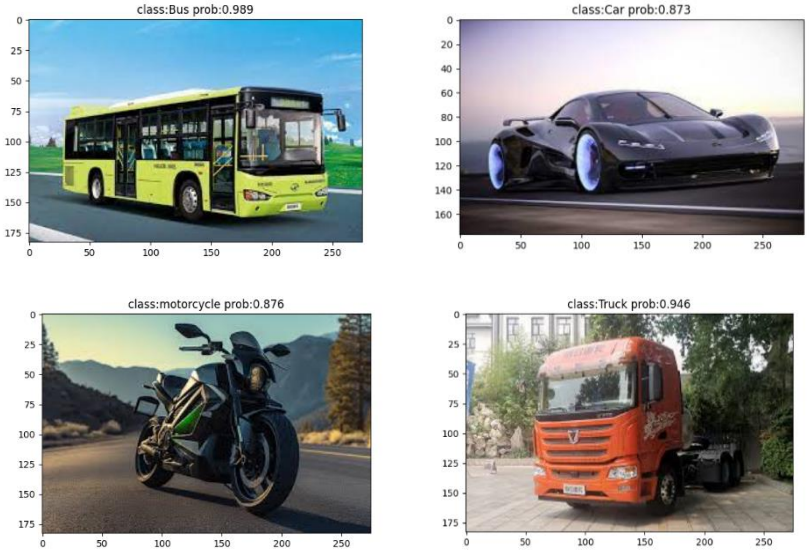

Fig 1. Results of ResNet-based vehicle type classification

In CNN (ResNet), we get the results shown in Fig 1, which shows the probability of determining that the vehicle belongs to a certain type, which can be considered as the accuracy rate. From the results of the test, we got a high percentage of accuracy, for large vehicles (buses and trucks), the rate of correctness is close to or over 95%, while for small vehicles (cars and motorcycles), the rate is slightly lower but above 87%.

In KNN, it stays around 50% correct for four different vehicle types showed in fig 2. It also shows the precision, recall, F1-score and support of our KNN model. It has classification ability, but the effect is not as good as CNN. The model needs to be further improved to enhance its performance. Methods such as tuning hyperparameters and adding more training data may need to be considered to improve the performance of the model.

	precision	recall	f1-score	support
Truck	0.50	0.47	0.49	19
bus	0.50	0.69	0.58	16
car	0.53	0.53	0.53	19
motorcycle	0.50	0.37	0.42	19

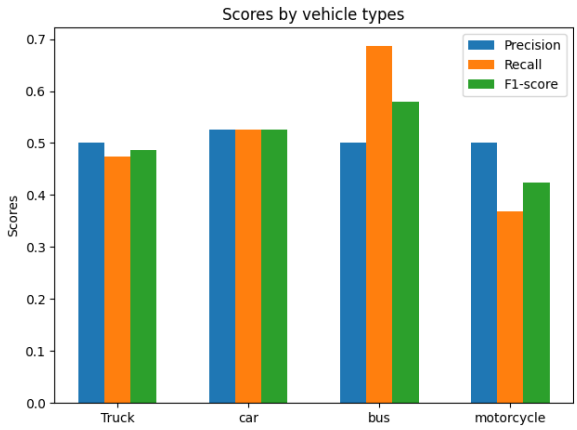

Fig 2. Results of KNN-based vehicle type classification

5. Conclusion

In our report, we apply two methods, CNN and KNN, to the machine learning problem of vehicle type classification. We present the principles of the two methods and our implementation, and we describe the selection and processing of the datasets and how to train, validate, and test the sets. Finally, we evaluate the two methods.

Given CNN's 90% accuracy, significantly higher than KNN's 50%, we find CNN more suitable for our classification task. CNN's advanced learning abilities, including deep learning and convolutional operations, excel at capturing intricate image patterns, making it a superior choice for image classification tasks.

We believe that our model can be optimized with more training such as larger datasets, and this is something we intend to continue to work on in the future.

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

CNN (ResNet)

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.RandomResizedCrop(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()
```

KNN

```
import os
import numpy as np
from PIL import Image
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_fscore_support

vehicle_database = 'D:/PyCharmProjects/MLprj/test2/Dataset'
vehicletypes = ['Truck', 'car', 'bus', 'motorcycle']
vector_data = [] ; labels = []
```

```
for vehicletype in vehicletypes:
    folder_path = os.path.join(vehicle_database, vehicletype)
    for img_name in os.listdir(folder_path):
        if img_name.endswith('.jpg') or img_name.endswith('.png'):
            img_path = os.path.join(folder_path, img_name)
            img = Image.open(img_path).convert('L')
            img = img.resize((32, 32))
            vector_data.append(np.array(img))
            labels.append(vehicletype)

vector_data = np.array(vector_data).reshape(len(vector_data), -1)
labels = np.array(labels)
#data groups
X_train, X_test, y_train, y_test = train_test_split(vector_data, labels,
test_size=0.2, random_state=43)
#classify
k = 20
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
# assess
print(classification_report(y_test, y_pred))
#get score
precision, recall, f1, _ = precision_recall_fscore_support(y_test, y_pred,
labels=vehicletypes)
#plot draw
x = np.arange(len(vehicletypes)) ; width = 0.2
fig, ax = plt.subplots()
rects1 = ax.bar(x - width, precision, width, label='Precision')
rects2 = ax.bar(x, recall, width, label='Recall')
rects3 = ax.bar(x + width, f1, width, label='F1-score')
ax.set_ylabel('Scores')
ax.set_title('Scores by vehicle types')
ax.set_xticks(x)
ax.set_xticklabels(vehicletypes)
ax.legend()
fig.tight_layout()
plt.show()
```