```
! pip install kaggle

! mkdir ~/.kaggle
! cp kaggle.json ~/.kaggle/
! chmod 600 ~/.kaggle/kaggle.json

! kaggle datasets download romanleo2003/labtinkoff
! unzip -O utf8 "/content/labtinkoff.zip"
```

```
import torch
import torch.nn as nn
import numpy as np
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F
from torchvision import datasets, models
from PIL import Image
import os
import torch.utils.data as data_utils
import matplotlib.pyplot as plt
```

```
DATASET_PATH = "../content/CCPD2019-dl1"
dir = os.path.abspath(os.curdir)
data_dir=os.path.join(dir, "CCPD2019-dl1")
data_dir
```

{"type":"string"}

Создаем класс датасета как наследника torch.utils.data.Dataset

```
class GetData(torch.utils.data.Dataset):

    def __init__(self, path, transform = None, train = True):

        self.transform = transform
        if train:
            self.path = os.path.join(path, 'train')
        else:
            self.path = os.path.join(path, 'test')
        self.listdir = os.listdir(self.path)

    def __len__(self):
        return len(self.listdir)


    def __getitem__(self, x):
        if torch.is_tensor(x):
            x = x.tolist()
```

```
        img_name = os.path.join(self.path, self.listdir[x])
        image = Image.open(img_name)
        if self.transform:
            image = self.transform(image)
        label = img_name[-10:-4]
        data_item = {'image': image, 'label': label}
        return data_item

batch_size = 64
car_num_transform =
transforms.Compose([transforms.ToTensor(),transforms.Resize((85,256))]
)
train_data = GetData(data_dir, train = True, transform =
car_num_transform)
test_data = GetData(data_dir, train = False, transform =
car_num_transform)
train_loader = torch.utils.data.DataLoader(dataset = train_data,

                                            batch_size = batch_size)
test_loader = torch.utils.data.DataLoader(dataset = test_data,
                                            batch_size = batch_size)
```

Здесь реализованы основные функции, которые нам понадобятся: разбитие картинок и меток на символы, перевод буков в метки классов (у нас их будет 34: 10 цифр и (26 - 2) буквы, так как тут не будет символов 'O' и 'I' (я вроде узнал, что в китайских номерах их нет, что и логично: их трудно отличить от '0' и '1' соответсвенно, и буквы с цифрами стоят почти в разнобой)

```
mask = [35, 75, 115, 147, 180, 212]

def split_to_liters(x):
    liters = list(torch.tensor_split(x, mask, axis = x.dim() - 1))
    liters.pop(0)
    return liters

def split_labels(x):
    size = len(x)
    list_labels = [torch.zeros(size) for x in range(6)]
    for i in range(size):
        splitted_label = list(x[i])
        for j in range(6):
            list_labels[j][i] = class_to_number(splitted_label[j])
    return list_labels

def class_to_number(x):
    ind = ord(x)
    if(ind >= 48 and ind <= 57):
        return ind - ord('0')
    elif(ind < 73):
```

```
        return ind - ord('A') + 10
    elif(ind > 73 and ind < 79):
        return ind - ord('A') + 9
    else:
        return ind - ord('A') + 8

def number_to_class(x):
    if(x <= 9):
        return chr(ord('0') + x)
    elif(x <=17):
        return chr(ord('A') + x - 10)
    elif(x <= 22):
        return chr(ord('A') + x - 9)
    else:
        return chr(ord('A') + x - 8)
```

Смотрим на данные

```
plt.figure(figsize=(25, 25))
print('==================== TRAIN ====================')
for i in range(8):
    image = train_data[i]
    ax = plt.subplot(1, 8, i + 1)
    ax.set_title(image['label'])
    ax.axis('off')
    plt.imshow(image['image'].permute(1, 2, 0))

plt.show()
plt.figure(figsize=(25, 25))
print('==================== TEST ====================')
for i in range(8):
    image = test_data[i]
    ax = plt.subplot(1, 8, i + 1)
    ax.set_title(image['label'])
    ax.axis('off')
    plt.imshow(image['image'].permute(1, 2, 0))
plt.show()
```

```
==================== TRAIN ====================
```


```
==================== TEST ====================
```

Смотря на данные, можно понять:

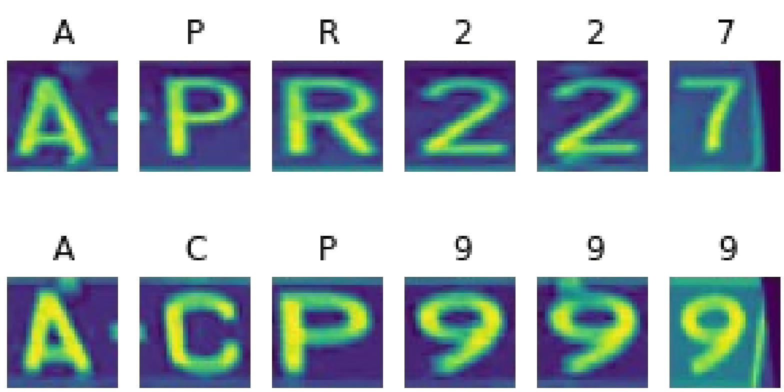
1. Число символов фиксированно
2. Номера хорошо просигментированы и одинаково спроецированы

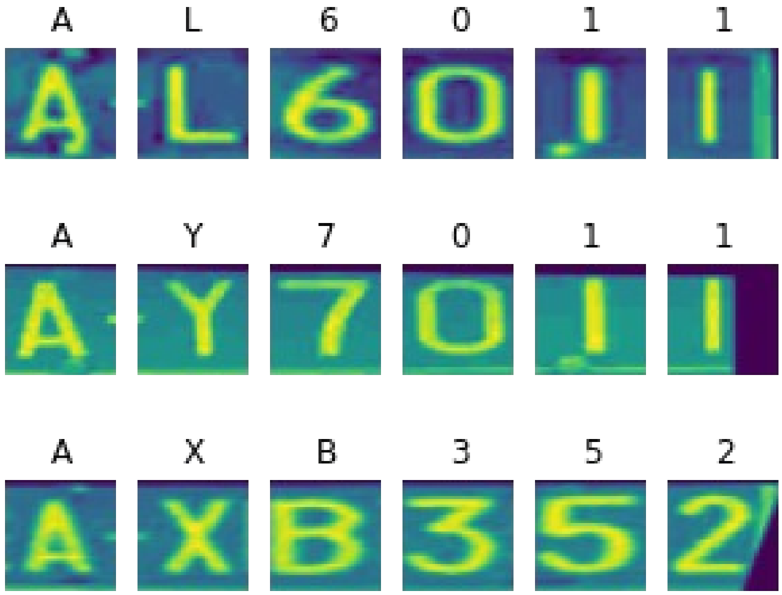
3. Нет особо никакой последовательной взаимосвязи между символами (то есть не как в российских - где-то буквы, где-то цифры (только буква вначале, но ее грех не распознать, тк она более удалена от остальных 5-ти))

Поэтому:

1. В рекурентных слоях нет необходимости из-за отсутствия последовательной связи
2. Одинаковая распределенность позволяет разбить номер на символы, регионы расположения одинаковы для всех номеров
3. Отдельные символы распознавать можно и с простой CNN + residual слой, просто нужно сделать эффективное по вычислениям разбиение на символы в процессе подгрузки данных с учетом батчей
4. Конец

```
transform_liters =
transforms.Compose([transforms.Resize((85,75)),transforms.CenterCrop((
75,75)),
transforms.Resize((28,28)),torchvision.transforms.Normalize((0.1307),
(0.3081)), transforms.Grayscale(num_output_channels=1)])

for i in range(5):
    K = train_data[i]
    liters = split_to_liters(K['image'])
    for i in range(len(liters)):
        ax = plt.subplot(1, len(liters) + 1 , i + 1)
        ax.set_title(K['label'][i])
        ax.axis('off')
        plt.imshow(transform_liters(liters[i]).squeeze(0))
    plt.show()
```



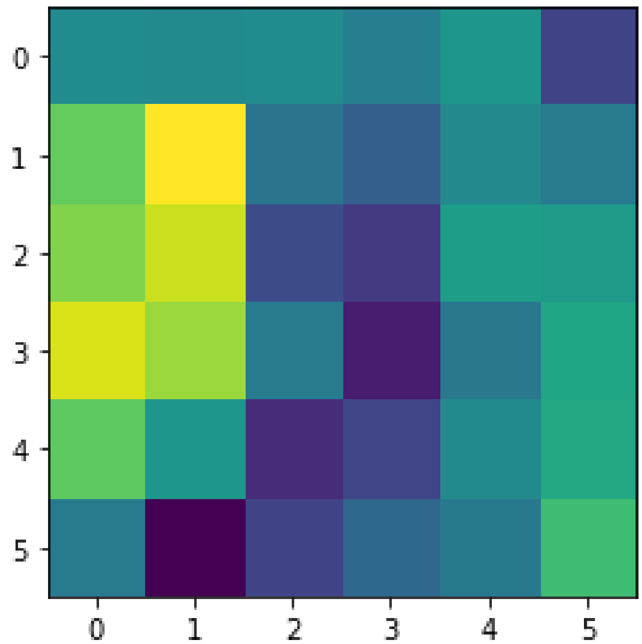


Как видно все норм разбивается

```
class Res(nn.Module):
    def __init__(
        self,
        in_channels: int,
        out_channels: int
    ):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.ds = torch.nn.Conv2d(
            in_channels,
            out_channels,
            kernel_size=1
        )
        self.conv1 = torch.nn.Conv2d(
            in_channels,
            out_channels,
            kernel_size=3,
            padding=1
        )
        self.conv2 = torch.nn.Conv2d(
            out_channels,
            out_channels,
            kernel_size=3,
            padding=1
        )
        self.relu = torch.nn.ReLU(inplace=True)
    def forward(self, input: torch.Tensor) -> torch.Tensor:
        buff = input
```

```python
        x = self.conv1(input)
        x = self.relu(x)
        x = self.conv2(x)

        if (self.in_channels != self.out_channels):
            buff = self.ds(buff)

        x += buff
        x = self.relu(x)

        return x

j = cnn_block = torch.nn.Sequential(
            torch.nn.Conv2d(1, 16, 7),
            torch.nn.ReLU(),
            torch.nn.Conv2d(16, 32, 5),
            torch.nn.ReLU(),
            torch.nn.Conv2d(32, 64, 5),
            torch.nn.ReLU(),
            torch.nn.Conv2d(64, 64, 3),
            torch.nn.ReLU(),
            Res(64,64),
            torch.nn.MaxPool2d(2)
        )
plt.figure()
K = train_data[0]
liters = split_to_liters(K['image'])
J = j(transform_liters(liters[0]))
plt.imshow(J[7].detach())
print(J.shape)
plt.show()
```

```
torch.Size([64, 6, 6])
```




```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.cnn_block = torch.nn.Sequential(
            torch.nn.Conv2d(1, 16, 7),
            torch.nn.ReLU(),
            torch.nn.Conv2d(16, 32, 5),
            torch.nn.ReLU(),
            Res(32,32),
            torch.nn.Conv2d(32, 64, 5),
            torch.nn.ReLU(),
            torch.nn.Conv2d(64, 64, 3),
            torch.nn.ReLU(),
            Res(64,64),
            torch.nn.MaxPool2d(2)
        )
        self.clf = torch.nn.Linear(6 * 6 * 64, 24 + 10)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        feature_map = self.cnn_block(x)
        feature_vector = torch.flatten(feature_map, x.dim() - 3)
        return self.clf(feature_vector)

net = Net()
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
optimizer = torch.optim.SGD(net.parameters(), lr=0.05)
criterion = torch.nn.CrossEntropyLoss()
net.to(device);
```

```
from tqdm import tqdm
from typing import Callable, NamedTuple, List
from collections import namedtuple

def train_epoch(
    model: torch.nn.Module,
    optimizer: torch.optim.Optimizer,
    loader: torch.utils.data.DataLoader,
    criterion: torch.nn.modules.loss._Loss,
    device: torch.device,
    verbose: bool = False,
) -> float:
    model.train(True)
    model.to(device)
    optimizer.zero_grad()
    acc_loss = 0
    total = len(loader.dataset)
    if verbose:
        loader = tqdm(loader, desc="Training", total=len(loader),
leave=True)
    for input_data in loader:
        liters = split_to_liters(input_data['image'])
        labels = split_labels(input_data['label'])
        for i in range(6):
            input = transform_liters(liters[i]).to(device)
            target = labels[i].type(torch.LongTensor)
            target = target.to(device)

            predicted = model(input)

            loss = criterion(predicted, target)

            # calculate gradient
            loss.backward()
            # update weights
            optimizer.step()
            # flush gradients
            optimizer.zero_grad()
            acc_loss += loss.item()

    return acc_loss / total/ 6


EvalOutput = namedtuple("EvalOutput", ["loss", "accuracy"])


def eval_epoch(
    model: torch.nn.Module,
    loader: torch.utils.data.DataLoader,
    criterion: torch.nn.modules.loss._Loss,
```

```
    device: torch.device,
    verbose: bool = False,
) -> EvalOutput:
    model.train(False)
    model.to(device)
    acc_loss = 0
    acc = 0
    total = len(loader.dataset)
    # no grad for context manager to accelerate evaluation
    with torch.no_grad():
        if verbose:
            loader = tqdm(loader, desc="Evaluation",
total=len(loader), leave=True)
        for input_data in loader:
            liters = split_to_liters(input_data['image'])
            labels = split_labels(input_data['label'])
            for i in range(6):
                input = transform_liters(liters[i]).to(device)
                target = labels[i].type(torch.LongTensor)
                target = target.to(device)
                predicted = model(input)

                loss = criterion(predicted, target)
                acc_loss += loss.item()

                acc += torch.sum(
                    torch.argmax(predicted, 1) == target
                ).item()

    return EvalOutput(
        loss=acc_loss / total /6,
        accuracy=acc / total /6
    )


class TrainOutput(NamedTuple):
    train_loss: List[float]
    val_loss: List[float]
    val_accuracy: List[float]


def train(
    num_epochs: int,
    model: torch.nn.Module,
    optimizer: torch.optim.Optimizer,
    train_loader: torch.utils.data.DataLoader,
    test_loader: torch.utils.data.DataLoader,
    criterion: torch.nn.modules.loss._Loss,
    device: torch.device
) -> TrainOutput:
```

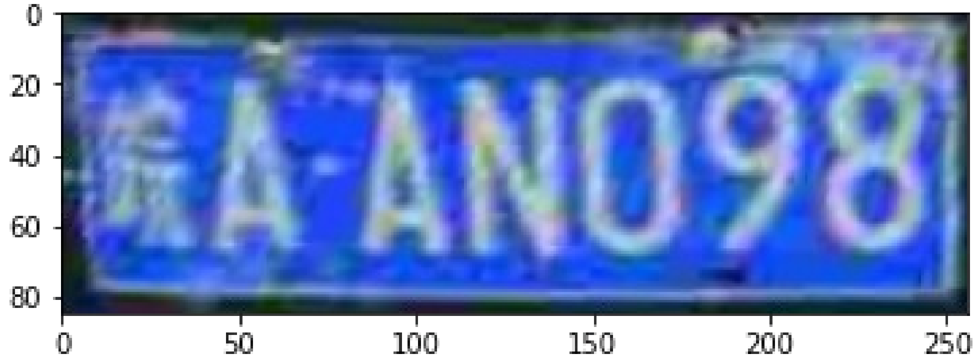
```
    train_loss = []
    val_loss = []
    val_acc = []
    for epoch in range(num_epochs):
        loss = train_epoch(
            model, optimizer, train_loader, criterion, device,
verbose=True
        )
        train_loss.append(loss)
        eval_out = eval_epoch(
            model, test_loader, criterion, device, verbose=True
        )
        val_loss.append(eval_out.loss)
        val_acc.append(eval_out.accuracy)

        print(f"Epoch #{epoch}:")
        print(f"Training Loss: {loss}")
        print(f"Evaluation Loss: {eval_out.loss}")
        print(f"Accuracy: {eval_out.accuracy}")
    return TrainOutput(
        train_loss=train_loss,
        val_loss=val_loss,
        val_accuracy=val_acc
    )

num_epochs = 5

training_results = train(
    num_epochs,
    net,
    optimizer,
    train_loader,
    test_loader,
    criterion,
    device
)
```

```
Training: 100%|██████████| 3125/3125 [09:43<00:00,  5.36it/s]
Evaluation: 100%|██████████| 157/157 [00:26<00:00,  6.03it/s]

Epoch #0:
Training Loss: 0.0034646165605452216
Evaluation Loss: 0.0007442258928709243
Accuracy: 0.9888322165549889

Training: 100%|██████████| 3125/3125 [09:24<00:00,  5.53it/s]
Evaluation: 100%|██████████| 157/157 [00:21<00:00,  7.31it/s]

Epoch #1:
Training Loss: 0.00010849459648283088
```

```
Evaluation Loss: 0.0005389873720008297
Accuracy: 0.9912324565789912

Training: 100%|██████████| 3125/3125 [09:01<00:00,  5.78it/s]
Evaluation: 100%|██████████| 157/157 [00:21<00:00,  7.28it/s]

Epoch #2:
Training Loss: 7.139462926044459e-05
Evaluation Loss: 0.0005036425654328657
Accuracy: 0.9920658732539921

Training: 100%|██████████| 3125/3125 [08:52<00:00,  5.87it/s]
Evaluation: 100%|██████████| 157/157 [00:20<00:00,  7.58it/s]

Epoch #3:
Training Loss: 5.460775743324978e-05
Evaluation Loss: 0.00058047389253935
Accuracy: 0.9912324565789912

Training: 100%|██████████| 3125/3125 [08:44<00:00,  5.96it/s]
Evaluation: 100%|██████████| 157/157 [00:20<00:00,  7.53it/s]

Epoch #4:
Training Loss: 4.320976056281075e-05
Evaluation Loss: 0.0004951160394569068
Accuracy: 0.9923158982564924
```

```python
def predict(img):
    liters_list = split_to_liters(img)
    word = ''
    for i in range(6):
        pred = net(transform_liters(liters_list[i]).to(device))
        x = torch.argmax(pred)
        word = word + number_to_class(x)
    return word

with torch.no_grad():
    for i in range(100):
        K = test_data[i]
        plt.figure()
        plt.imshow(K['image'].permute(1,2,0))
        plt.title(predict(K['image']))
        plt.show()
```

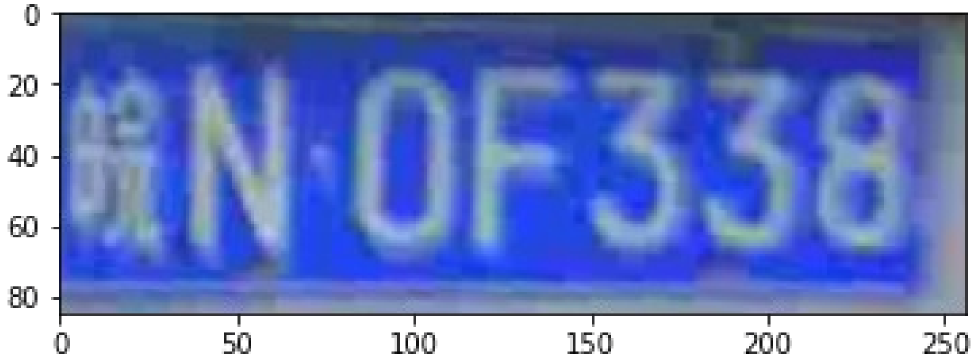
AAN098



AMC799

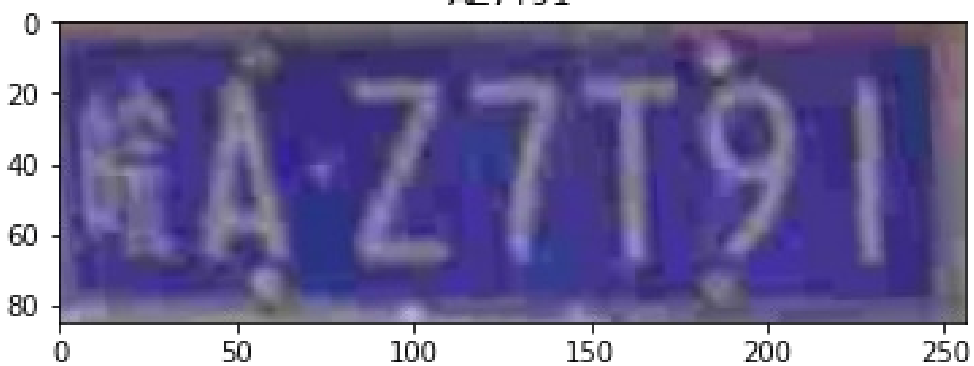
AM136W

N0F338



AXA253

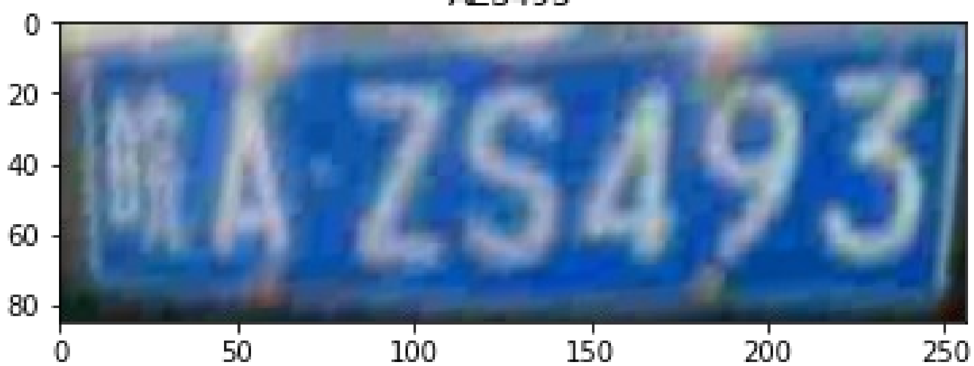
AZ7T91



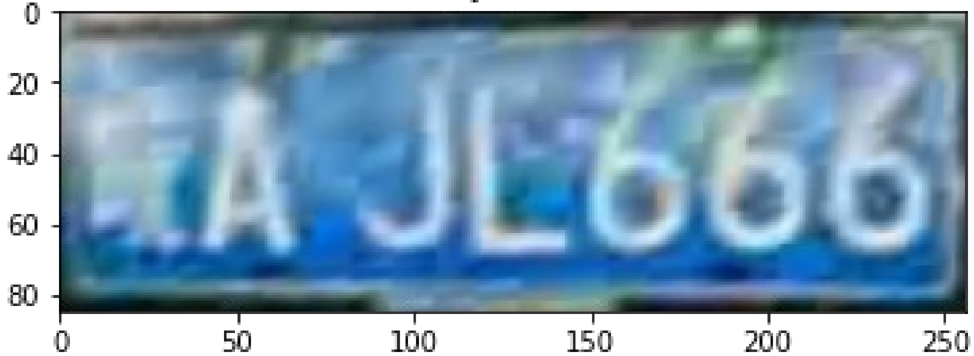
A0U120

AW2S13

AZS493

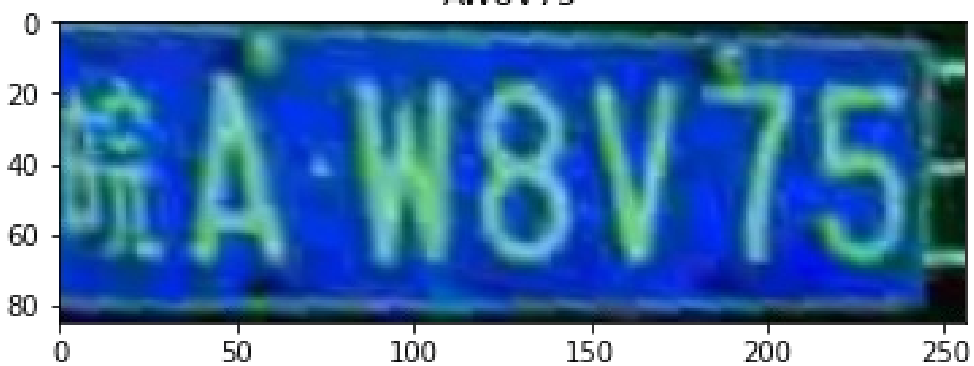


AJE666



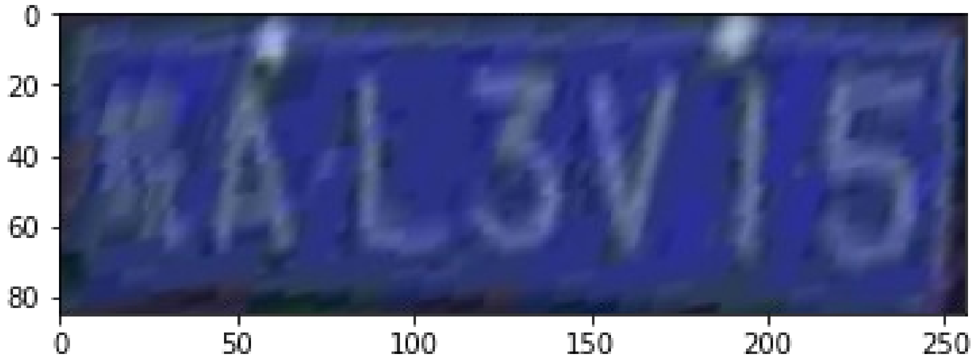
AS4982

AW8V75

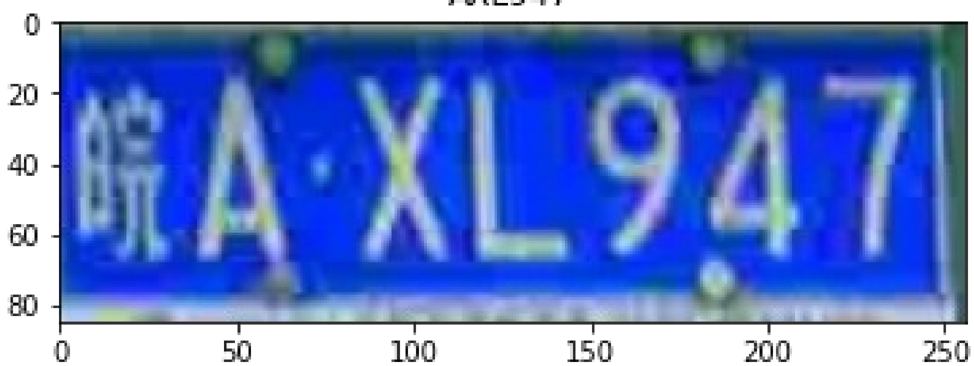


AL7900

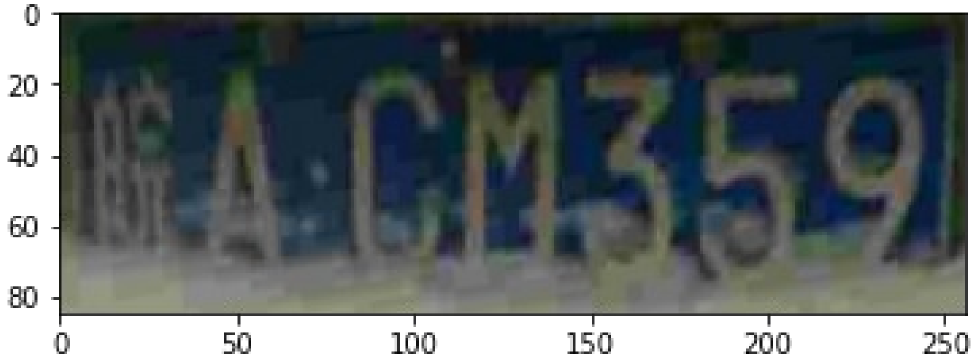
AL3V15



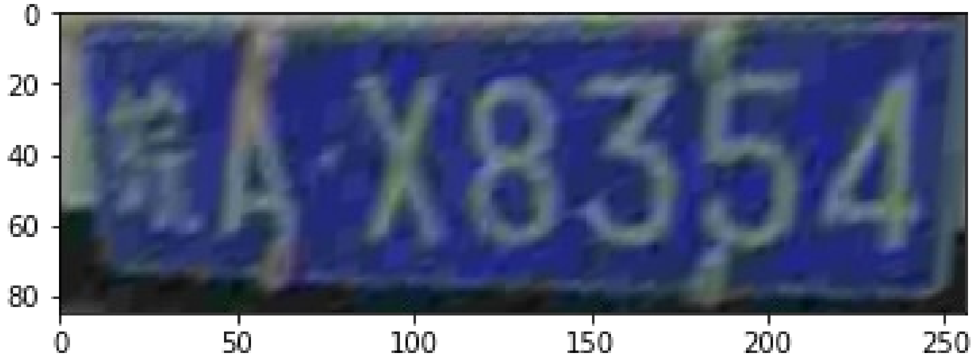
AXL947



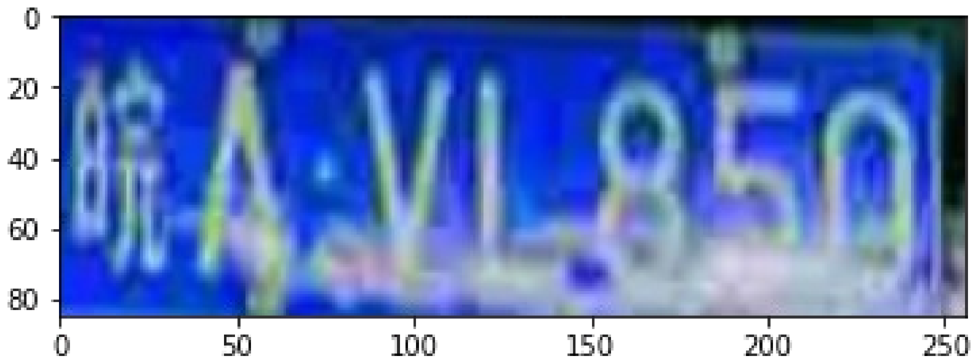
ACM3F9



AX8354



AVL8E0



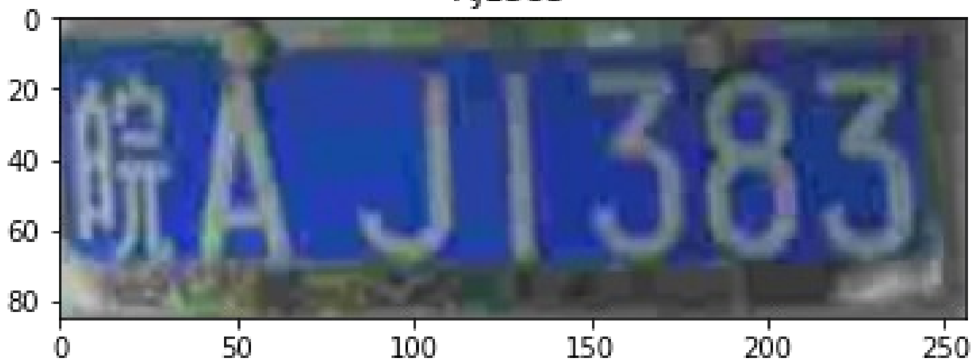
A18RQ0

L57733

DLN338

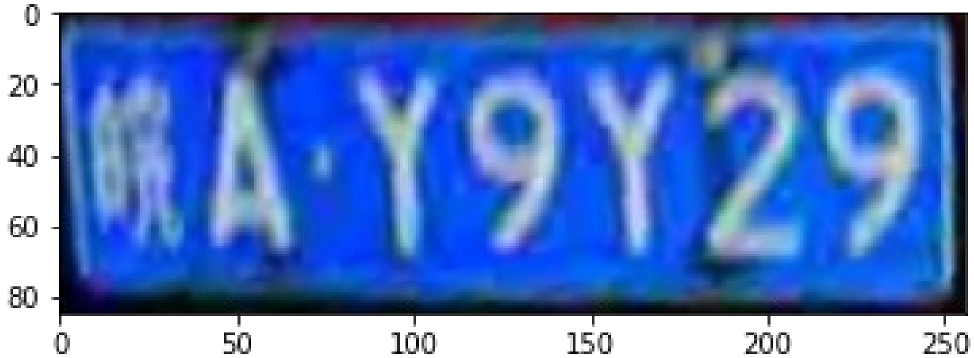
BW66Q9

AJ1383



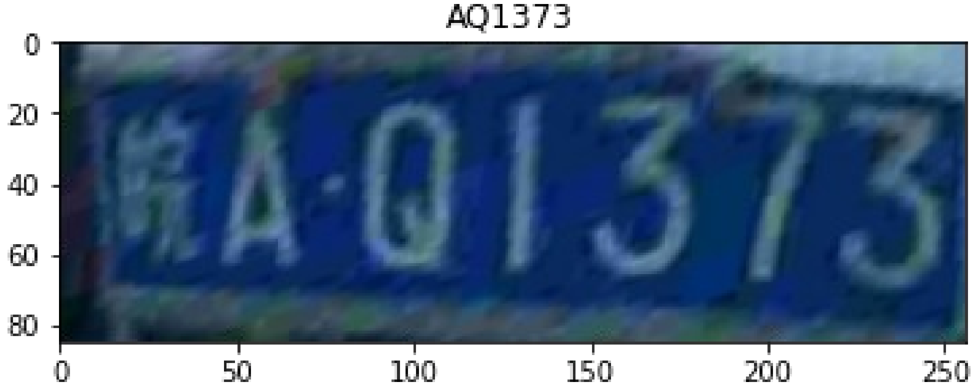
A379Y0

AY9Y29

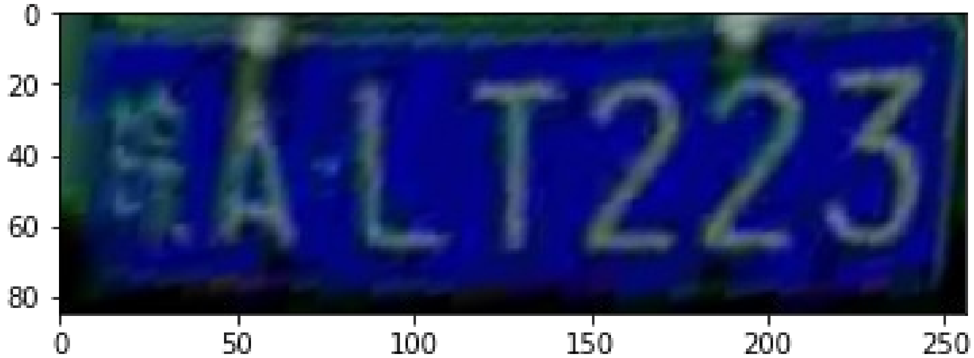


A474B7

AQ1373



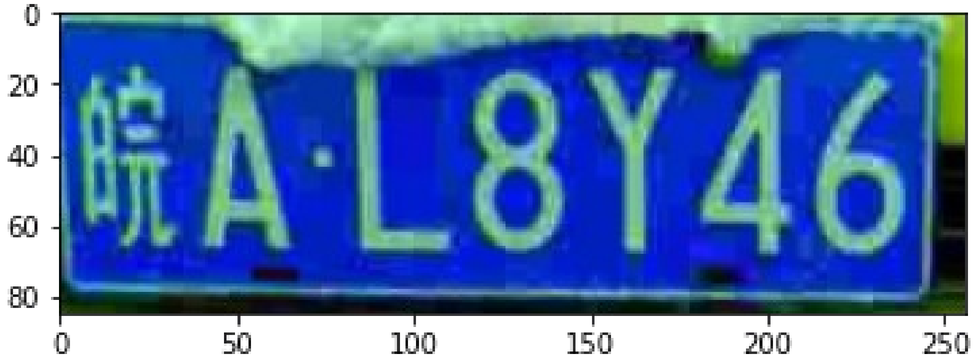
ALT223



AZ7Z57

AYR537

AL8Y46



AT278K

AJ300F

AWN293

AEW537

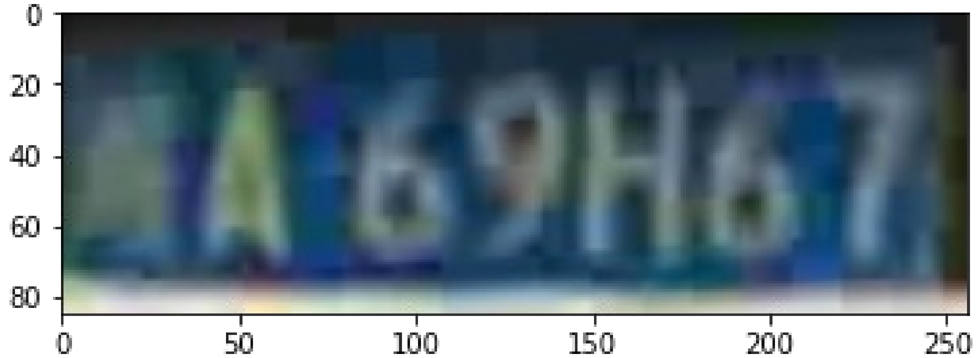
AUA010

AHS512

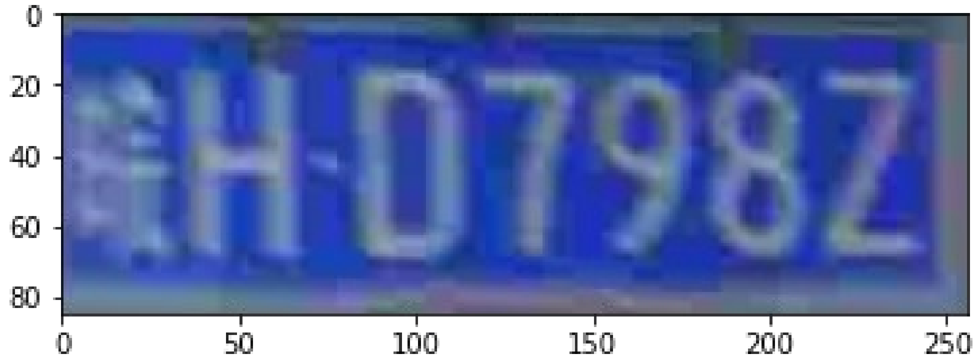
ACQ091

AY5T93

A69H67

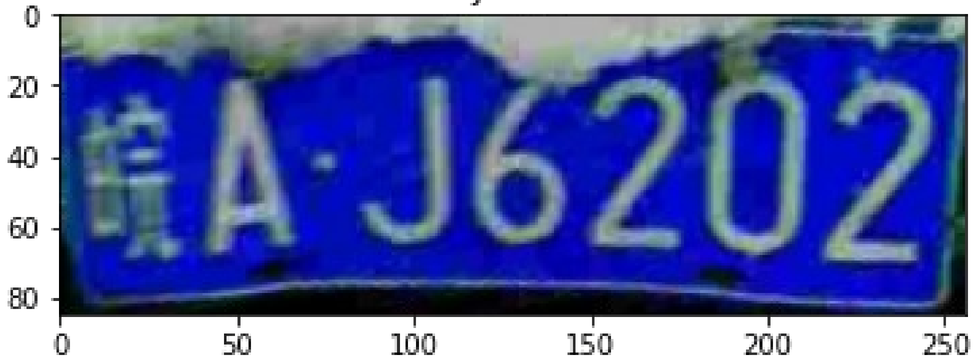


HD798Z

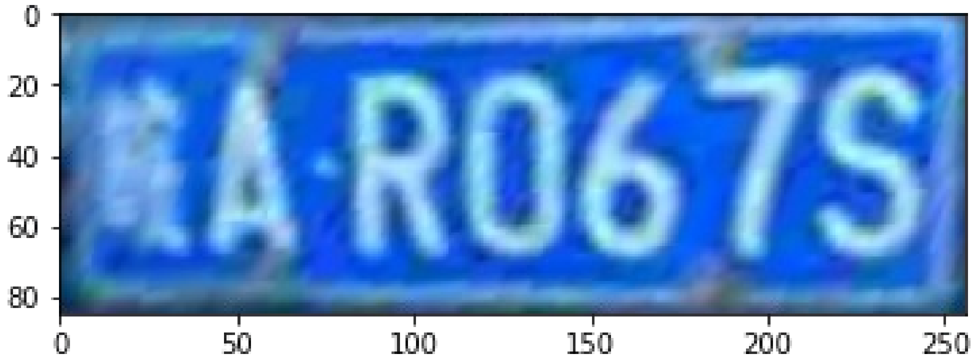


A10R21

AJ6202

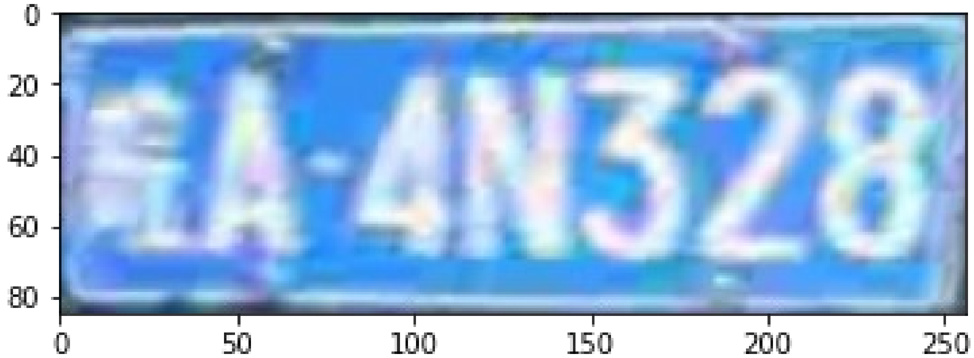


AR067S



A58G48

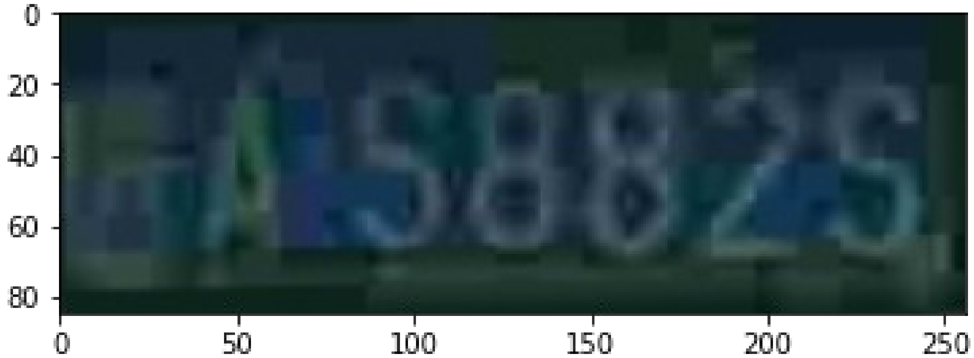
A4N328



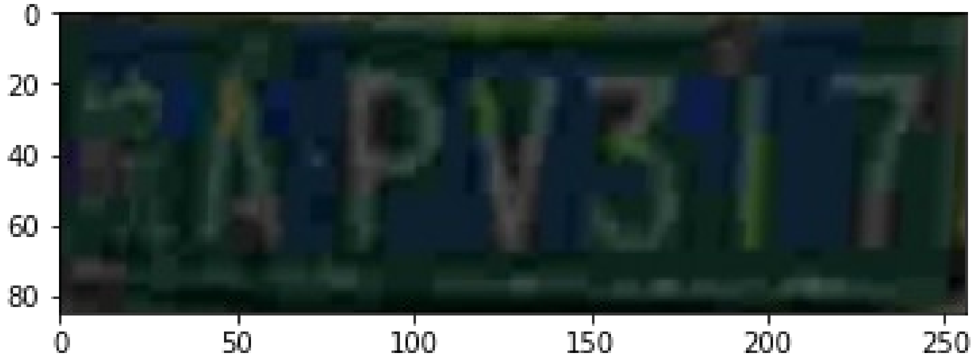
AAC819

ABB303

AS882S

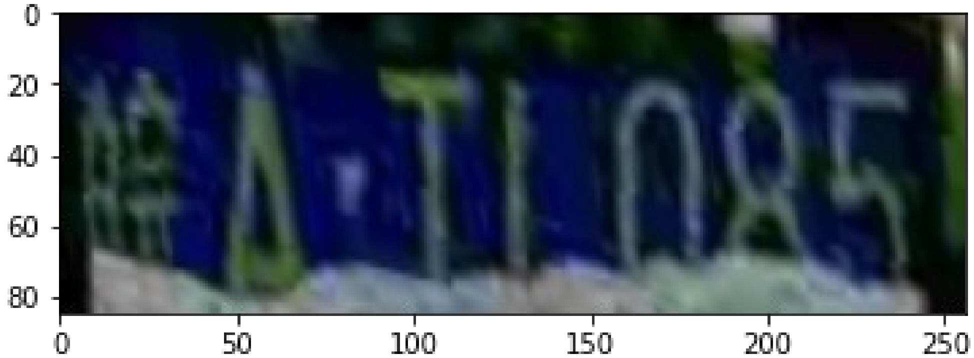


APV317



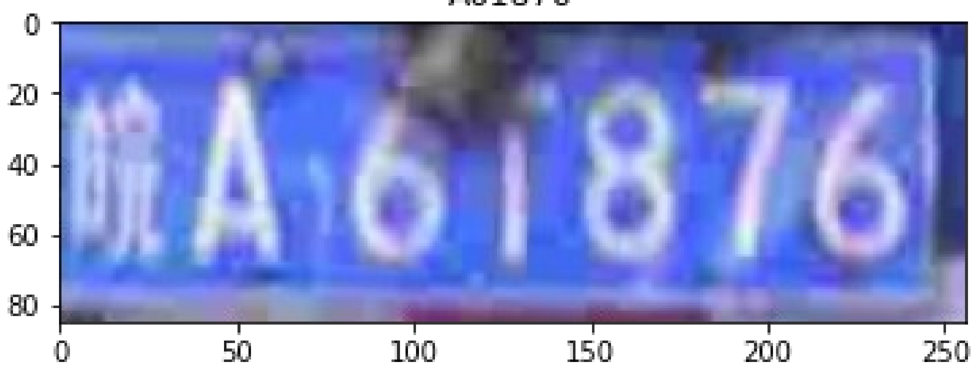
AN7K97

A7109F



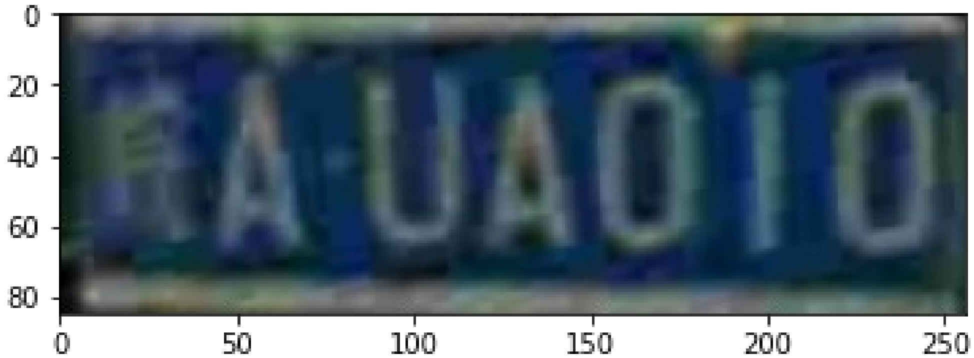
A668L2

A61876



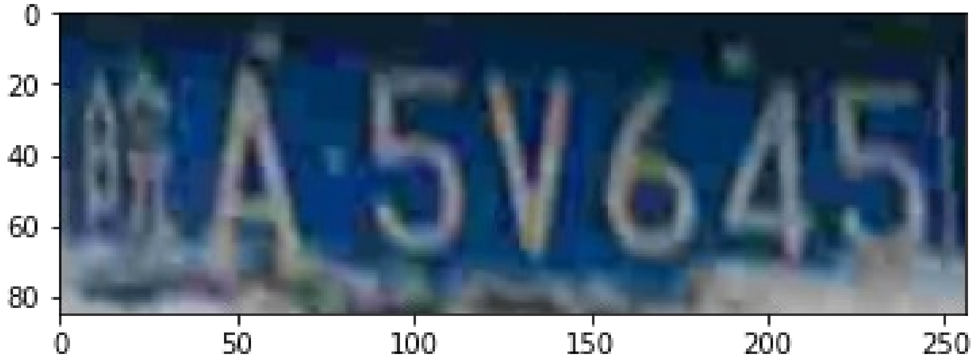
ACE819

AUA010

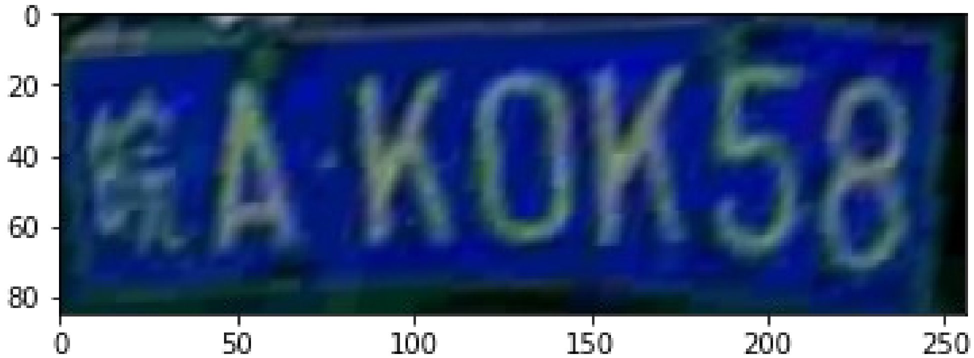


J92E33

A5V645



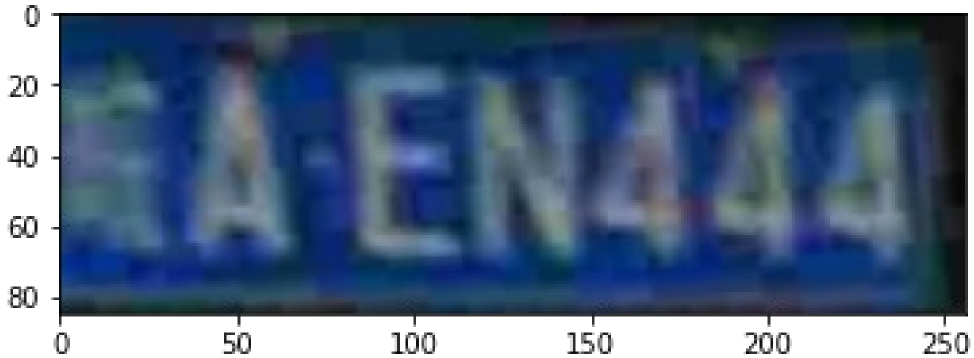
AK0K58



AF8X03

ANA567

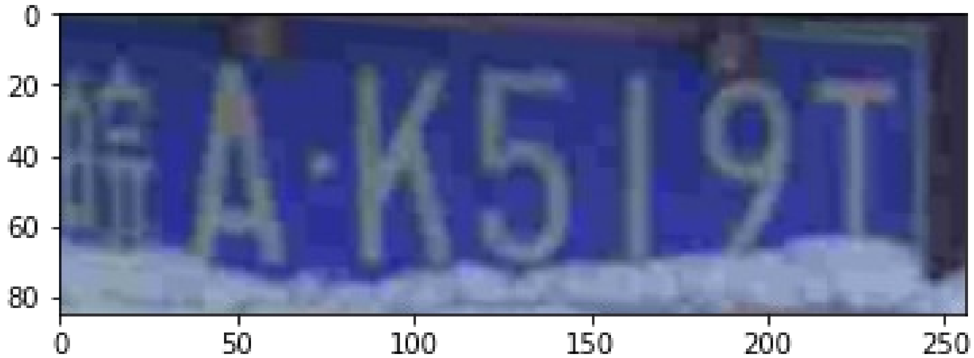
AEN444



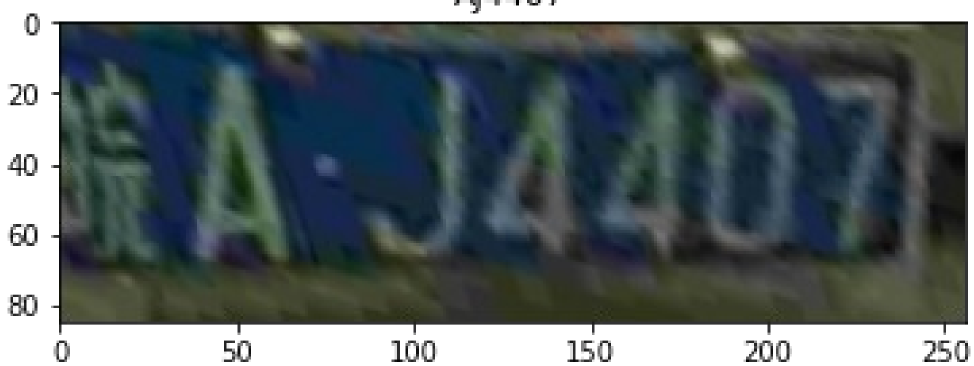
A1B521

AZF762

AK519D



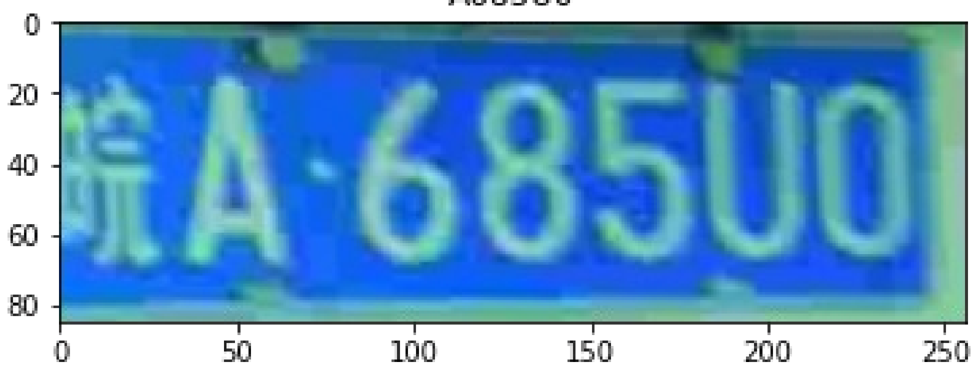
AJ4407



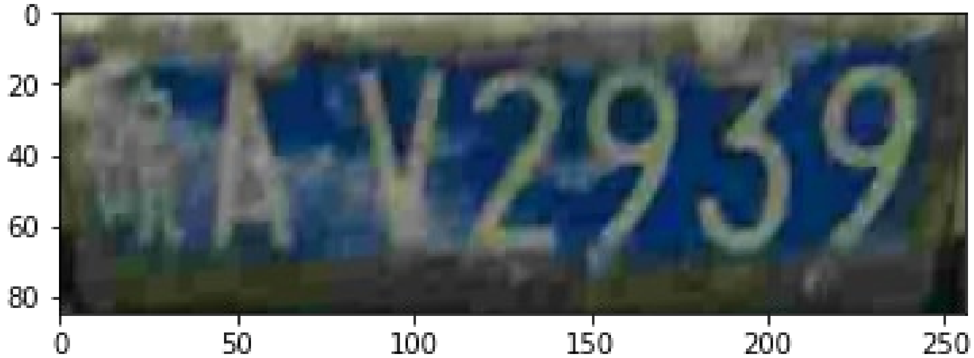
A781F7

A871M5

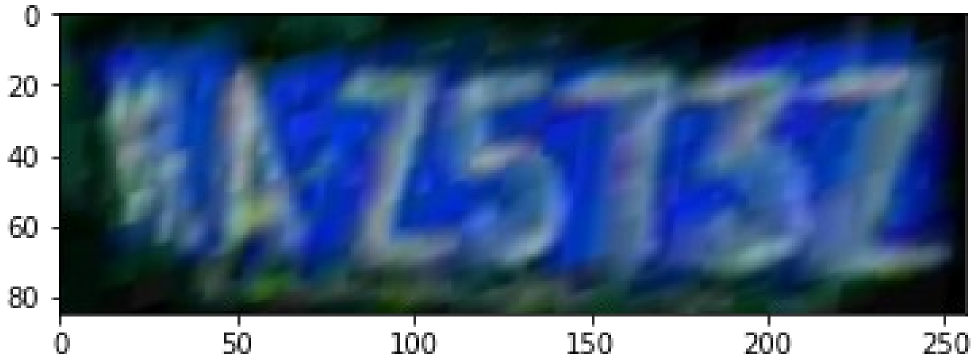
A685U0



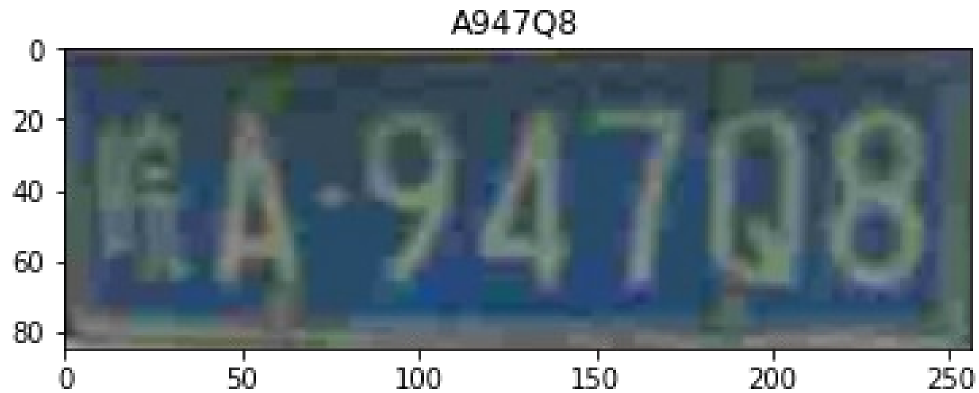
AV2939



AZ573Z



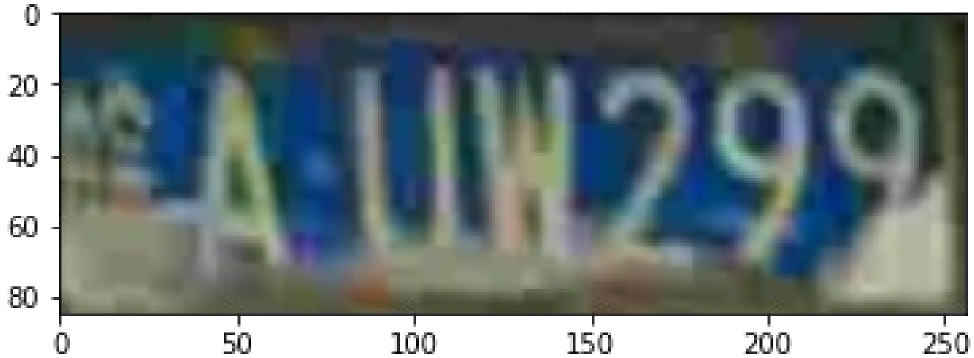
ACZ906


A947Q8


AD019T

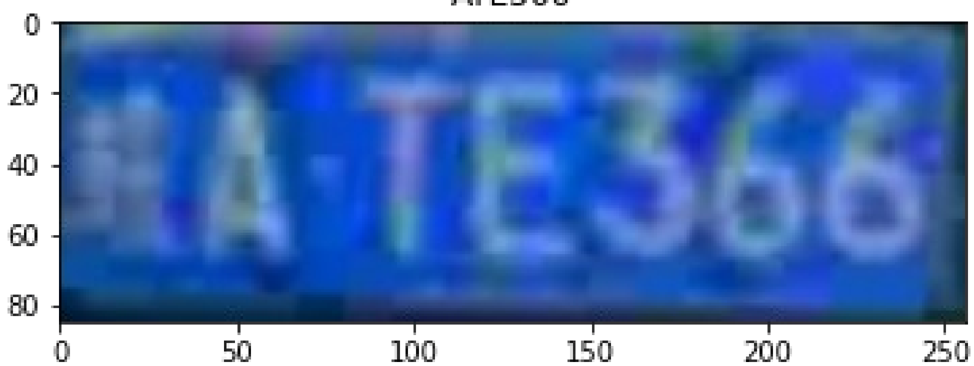

E5692K

AUW299



A3Y735

ATE366



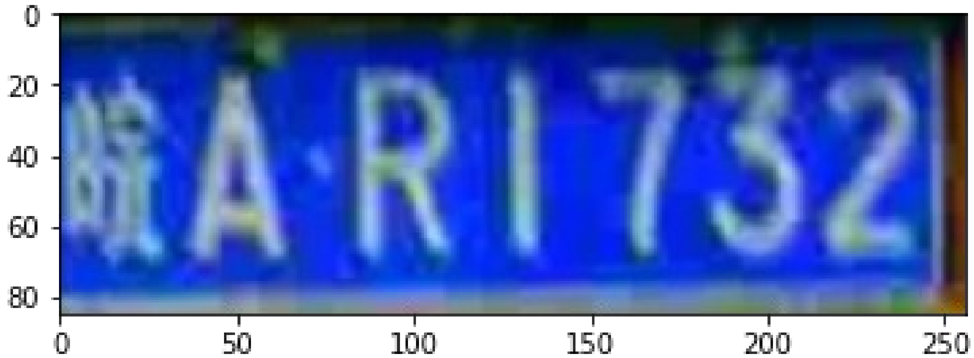
AK003H

HS1H77

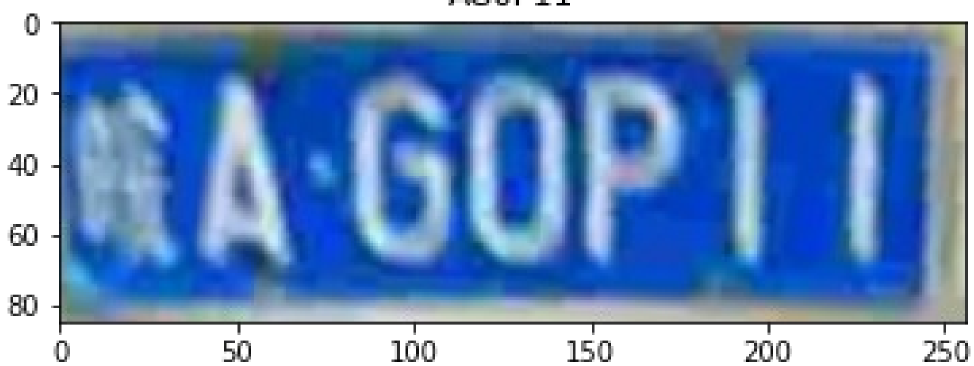
E51QG7

AC1V23

AR1732

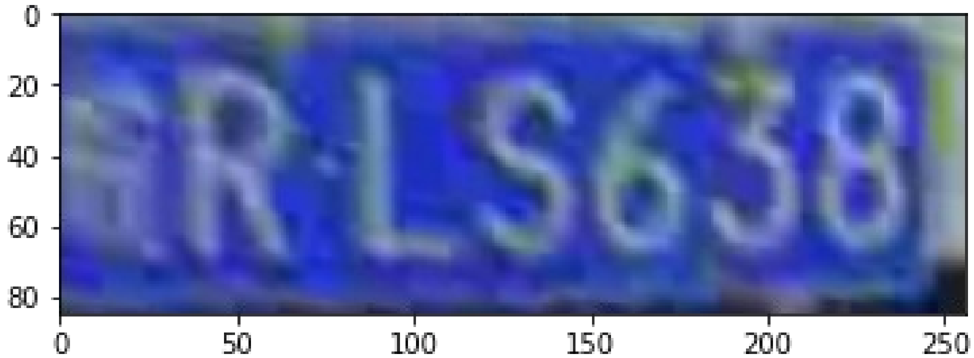


AG0P11



AJH019

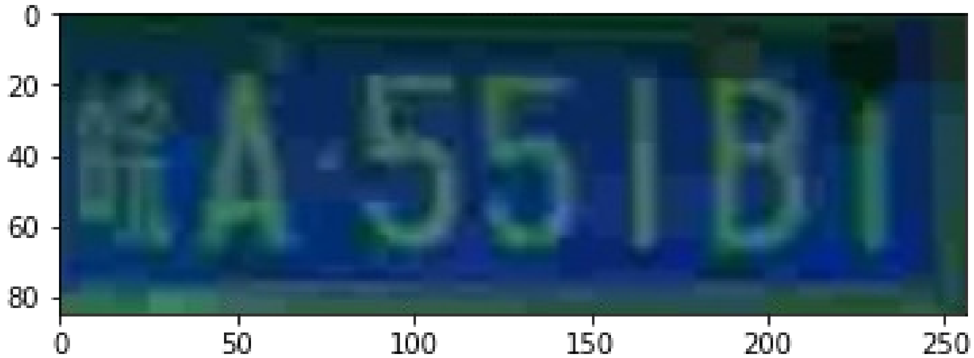
RLS638



A0K195

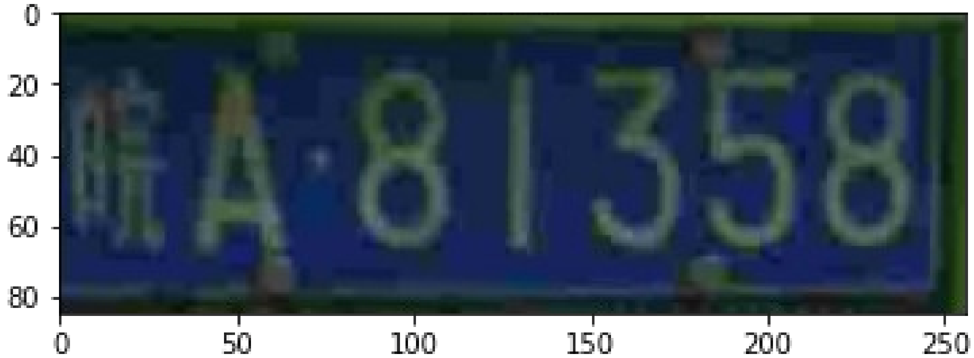
AYB400

A551B1

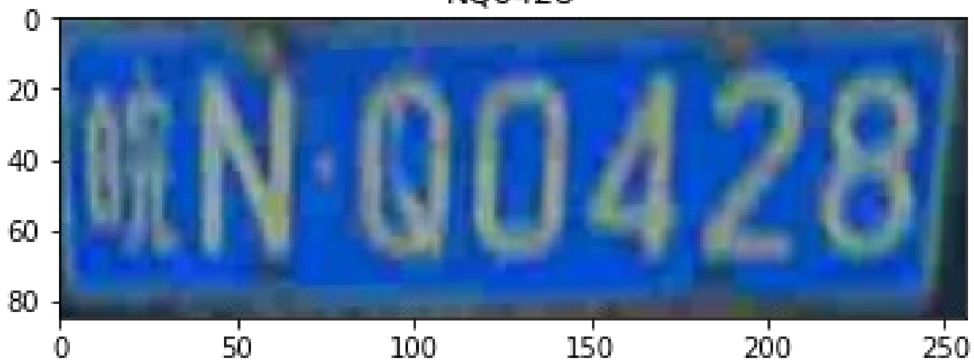


A156L6

A81358



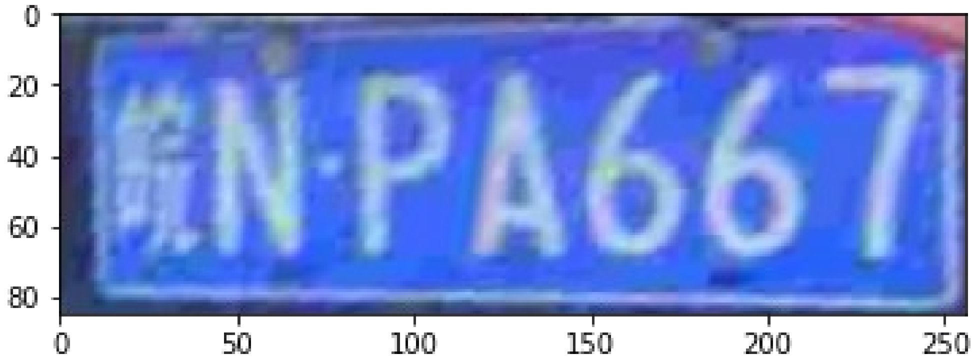
NQ0428



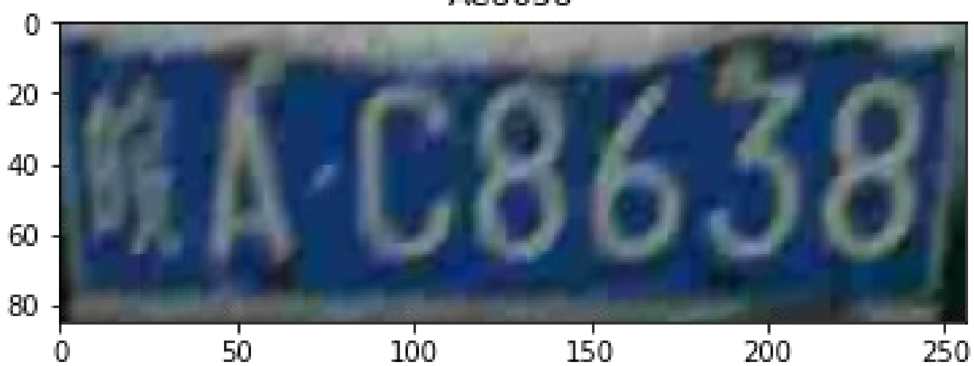
A48126

B54700

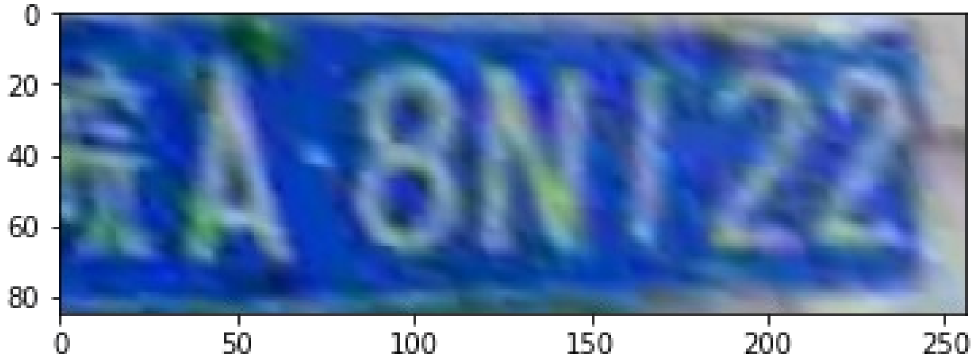
NPA667



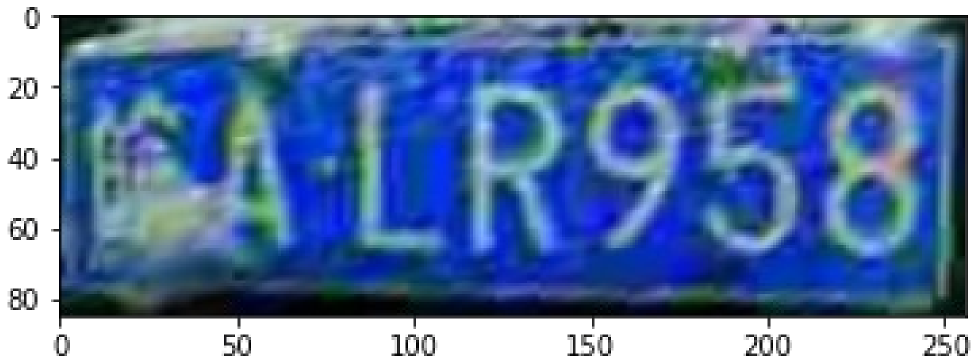
AC8638



A8N122



ALR958



AUE118

AUZ088