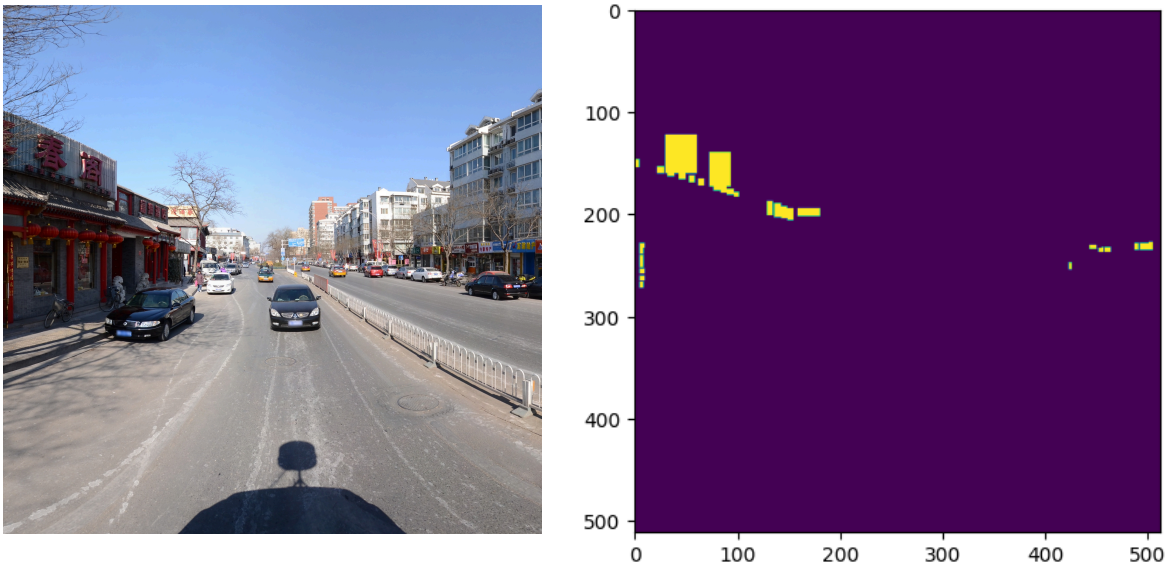
Chinese character detection

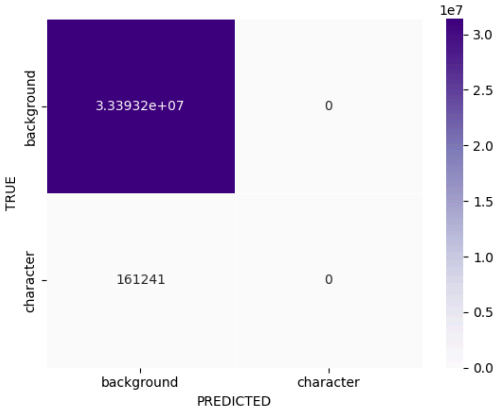
In my code I first go through all the files that are in the folder “images”. I’m making a list with names of all these files. Later on I’m comparing the content of this list with data we can find in the “train” J-son file. 845 images can be found both in the folder “images” and in “train” J-son file. This is the date we are going to use.

Working on 2048x2048 files was too aggravating for our GPUs. That’s why I decided to divide a side of an image by 4. This will become a variable called "divider" and will be used when creating golden labels. Golden labels were created as 0-1 matrix, where 1 symbolizes an area of box with a chinese character. When comparing my matrixes with the original photos it’s visible that they’re correct. Even though the image was resized.

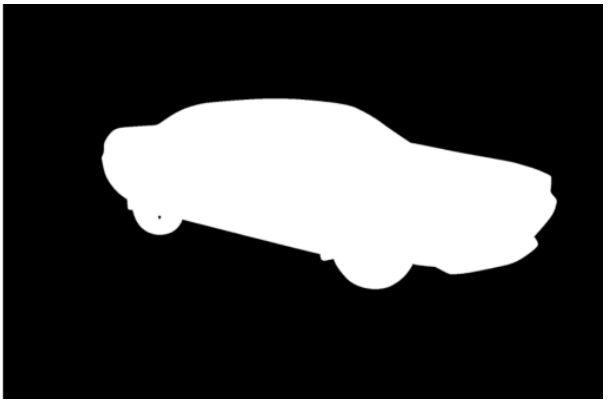


I created two models. The first model, called My_Model**,** was designed by me from scratch. The model is a small convolutional neural network. It follows an encoder–decoder structure, which means it first reduces the image size to extract important features, and then upsamples it back to the original size to produce an output. The first block (down1) takes an RGB image (3 channels) and applies a convolution layer with 16 filters, followed by a ReLU activation and a max-pooling layer that reduces the image size by half. The second block (down2) does the same but increases the number of filters to 32. The first upsampling layer (up1) increases the image size back using a transposed convolution (from 32 to 16 channels). Then a convolution block (conv_up1) refines these features with another convolution and ReLU activation. And repeat again with doubling image size from 8 to 16. A final 1×1 convolution reduces the number of channels to 1, producing a single-channel output.

```python
class My_Model(nn.Module):
    def __init__(self, ):
        super().__init__()

        self.down1 = nn.Sequential(nn.Conv2d(3, 16, kernel_size=3, padding=1),
                                   nn.ReLU(inplace=True),
                                   nn.MaxPool2d(2))

        self.down2 = nn.Sequential(nn.Conv2d(16, 32, kernel_size=3, padding=1),
                                   nn.ReLU(inplace=True),
                                   nn.MaxPool2d(2))

        self.up1 = nn.ConvTranspose2d(32, 16, kernel_size=2, stride=2)
        self.conv_up1 = nn.Sequential(nn.Conv2d(16, 16, 3, padding=1),
                                      nn.ReLU(inplace=True))

        self.up2 = nn.ConvTranspose2d(16, 8, kernel_size=2, stride=2)
        self.conv_up2 = nn.Sequential(nn.Conv2d(8, 8, 3, padding=1),
                                      nn.ReLU(inplace=True))

        self.output = nn.Conv2d(8, 1, kernel_size=1)

    def forward(self, x):
        x1 = self.down1(x)
        x2 = self.down2(x1)
        x3 = self.up1(x2)
        x4 = self.conv_up1(x3)
        x5 = self.up2(x4)
        out = self.output(x5)
        return out
```



My first model performed extremely poorly. As visible on the confusion matrix it assumed that everything is a background. This gives us a very high Accuracy and F1 score (0.9927), but it's not what we are looking for. Because of this struggle I looked for some more advanced methods in kaggle. I found a competition called "Carvana Image Masking Challenge" which was also based on semantic segmentation. Competitors were supposed to detect cars from the background.



My second model was mostly inspired by a competitor called Alan Yu. I liked how he dealt with the problem of background. Just like Alan I created a function called dice_coefficient.

```
def dice_coefficient(prediction, target, epsilon=1e-07):
    prediction_copy = prediction.clone()

    prediction_copy[prediction_copy >= 0.5] = 1
    prediction_copy[prediction_copy < 0.5 ] = 0

    intersection = abs(torch.sum(prediction_copy * target))
    union = abs(torch.sum(prediction_copy) + torch.sum(target
    dice = (2. * intersection + epsilon) / (union + epsilon)

    return dice
```

This function is used to measure how well the predicted image matches the target. First, the prediction values are rounded to 0 or 1 using a threshold of 0.5, this makes the prediction a binary mask. Then, the function calculates how much the prediction and target overlap. Intersection is the number of pixels where both prediction and target are 1. Union stands for total number of pixels that are 1 in either prediction or target. Finally, the Dice coefficient is calculated as:

$$\text{Dice} = \frac{2 \times intersection + \varepsilon}{union + \varepsilon}$$

Where $\varepsilon = 10^{-7}$ is a small number added to prevent division by zero. As a next step I adapted the neural network model to my code. It is much more complex and is designed as a U-net model. Even though it's much more complex than my own creation it still does very poorly. Thanks to the dice feature the character detection is not 0, but still it's irrelevant. The dc test which is defined as test_dc = test_running_dc/(idx + 1) has a value around 2.635816573142401e-10. The full model is on the following page.

As a summary, both models performed very poorly. Although the second model is of course better.

```python
class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv_op = nn.Sequential(
                            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
                            nn.ReLU(inplace=True),
                            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
                            nn.ReLU(inplace=True)
                            )
    def forward(self, x):
        return self.conv_op(x)

class DownSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = DoubleConv(in_channels, out_channels)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
    def forward(self, x):
        down = self.conv(x)
        p = self.pool(down)
        return down, p
class DownSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = DoubleConv(in_channels, out_channels)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
    def forward(self, x):
        down = self.conv(x)
        p = self.pool(down)
        return down, p

class UpSample(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.up = nn.ConvTranspose2d(in_channels, in_channels//2, kernel_size=2, stride=2)
        self.conv = DoubleConv(in_channels, out_channels)
    def forward(self, x1, x2):
        x1 = self.up(x1)
        x = torch.cat([x1, x2], 1)
        return self.conv(x)
class UNet(nn.Module):

    def __init__(self, in_channels, num_classes):
        super().__init__()
        self.down_convolution_1 = DownSample(in_channels, 64)
        self.down_convolution_2 = DownSample(64, 128)
        self.down_convolution_3 = DownSample(128, 256)

        self.bottle_neck = DoubleConv(256, 512)

        self.up_convolution_1 = UpSample(512, 256)
        self.up_convolution_2 = UpSample(256, 128)
        self.up_convolution_3 = UpSample(128, 64)

        self.out = nn.Conv2d(in_channels=64, out_channels=num_classes, kernel_size=1)

    def forward(self, x):
        down_1, p1 = self.down_convolution_1(x)
        down_2, p2 = self.down_convolution_2(p1)
        down_3, p3 = self.down_convolution_3(p2)

        b = self.bottle_neck(p3)

        up_1 = self.up_convolution_1(b, down_3)
        up_2 = self.up_convolution_2(up_1, down_2)
        up_3 = self.up_convolution_3(up_2, down_1)

        out = self.out(up_3)
        return out
```