```python
from torch import nn
from torch.nn import functional as F
import torch
from torchvision import models
import torchvision


def conv3x3(in_, out):
    return nn.Conv2d(in_, out, 3, padding=1)


class ConvRelu(nn.Module):
    def __init__(self, in_, out):
        super().__init__()
        self.conv = conv3x3(in_, out)
        self.activation = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.conv(x)
        x = self.activation(x)
        return x


class DecoderBlock(nn.Module):
    def __init__(self, in_channels, middle_channels, out_channels):
        super().__init__()

        self.block = nn.Sequential(
            ConvRelu(in_channels, middle_channels),
            nn.ConvTranspose2d(middle_channels, out_channels, kernel_size=3,
            stride=2, padding=1, output_padding=1),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.block(x)


class UNet11(nn.Module):
    def __init__(self, num_filters=32, pretrained=False):
        """
        :param num_classes:
        :param num_filters:
        :param pretrained:
            False - no pre-trained network is used
            True  - encoder is pre-trained with VGG11
        """
        super().__init__()
        self.pool = nn.MaxPool2d(2, 2)

        self.encoder = models.vgg11(pretrained=pretrained).features

        self.relu = self.encoder[1]
        self.conv1 = self.encoder[0]
        self.conv2 = self.encoder[3]
        self.conv3s = self.encoder[6]
        self.conv3 = self.encoder[8]
        self.conv4s = self.encoder[11]
        self.conv4 = self.encoder[13]
        self.conv5s = self.encoder[16]
        self.conv5 = self.encoder[18]

        self.center = DecoderBlock(num_filters * 8 * 2, num_filters * 8 * 2,
        num_filters * 8)
        self.dec5 = DecoderBlock(num_filters * (16 + 8), num_filters * 8 * 2,
        num_filters * 8)
```

```
64          self.dec4 = DecoderBlock(num_filters * (16 + 8), num_filters * 8 * 2,
            num_filters * 4)
65          self.dec3 = DecoderBlock(num_filters * (8 + 4), num_filters * 4 * 2,
            num_filters * 2)
66          self.dec2 = DecoderBlock(num_filters * (4 + 2), num_filters * 2 * 2,
            num_filters)
67          self.dec1 = ConvRelu(num_filters * (2 + 1), num_filters)
68
69          self.final = nn.Conv2d(num_filters, 1, kernel_size=1)
70
71      def forward(self, x):
72          conv1 = self.relu(self.conv1(x))
73          conv2 = self.relu(self.conv2(self.pool(conv1)))
74          conv3s = self.relu(self.conv3s(self.pool(conv2)))
75          conv3 = self.relu(self.conv3(conv3s))
76          conv4s = self.relu(self.conv4s(self.pool(conv3)))
77          conv4 = self.relu(self.conv4(conv4s))
78          conv5s = self.relu(self.conv5s(self.pool(conv4)))
79          conv5 = self.relu(self.conv5(conv5s))
80
81          center = self.center(self.pool(conv5))
82
83          dec5 = self.dec5(torch.cat([center, conv5], 1))
84          dec4 = self.dec4(torch.cat([dec5, conv4], 1))
85          dec3 = self.dec3(torch.cat([dec4, conv3], 1))
86          dec2 = self.dec2(torch.cat([dec3, conv2], 1))
87          dec1 = self.dec1(torch.cat([dec2, conv1], 1))
88          return self.final(dec1)
89
90
91  def unet11(pretrained=False, **kwargs):
92      """
93      pretrained:
94              False - no pre-trained network is used
95              True  - encoder is pre-trained with VGG11
96              carvana - all weights are pre-trained on
97                  Kaggle: Carvana dataset
                    https://www.kaggle.com/c/carvana-image-masking-challenge
98      """
99      model = UNet11(pretrained=pretrained, **kwargs)
100
101     if pretrained == 'carvana':
102         state = torch.load('TernausNet.pt')
103         model.load_state_dict(state['model'])
104     return model
105
106
107 class DecoderBlockV2(nn.Module):
108     def __init__(self, in_channels, middle_channels, out_channels, is_deconv=True):
109         super(DecoderBlockV2, self).__init__()
110         self.in_channels = in_channels
111
112         if is_deconv:
113             """
114                 Paramaters for Deconvolution were chosen to avoid artifacts, following
                    link https://distill.pub/2016/deconv-checkerboard/
115             """
116
117             self.block = nn.Sequential(
118                 ConvRelu(in_channels, middle_channels),
119                 nn.ConvTranspose2d(middle_channels, out_channels, kernel_size=4,
                    stride=2,
120                                    padding=1),
121                 nn.ReLU(inplace=True)
122             )
123         else:
124
```

```
125             self.block = nn.Sequential(
126                 nn.Upsample(scale_factor=2, mode='bilinear'),
127                 ConvRelu(in_channels, middle_channels),
128                 ConvRelu(middle_channels, out_channels),
129             )
130
131     def forward(self, x):
132         return self.block(x)
133
134
135 class AlbuNet(nn.Module):
136     """
137         UNet (https://arxiv.org/abs/1505.04597) with
            Resnet34(https://arxiv.org/abs/1512.03385) encoder
138
139         Proposed by Alexander Buslaev: https://www.linkedin.com/in/al-buslaev/
140
141         """
142
143     def __init__(self, num_classes=1, num_filters=32, pretrained=False,
        is_deconv=False):
144         """
145         :param num_classes:
146         :param num_filters:
147         :param pretrained:
148             False - no pre-trained network is used
149             True  - encoder is pre-trained with resnet34
150         :is_deconv:
151             False: bilinear interpolation is used in decoder
152             True: deconvolution is used in decoder
153         """
154         super().__init__()
155         self.num_classes = num_classes
156
157         self.pool = nn.MaxPool2d(2, 2)
158
159         self.encoder = torchvision.models.resnet34(pretrained=pretrained)
160
161         self.relu = nn.ReLU(inplace=True)
162
163         self.conv1 = nn.Sequential(self.encoder.conv1,
164                                    self.encoder.bn1,
165                                    self.encoder.relu,
166                                    self.pool)
167
168         self.conv2 = self.encoder.layer1
169
170         self.conv3 = self.encoder.layer2
171
172         self.conv4 = self.encoder.layer3
173
174         self.conv5 = self.encoder.layer4
175
176         self.center = DecoderBlockV2(512, num_filters * 8 * 2, num_filters * 8,
            is_deconv)
177
178         self.dec5 = DecoderBlockV2(512 + num_filters * 8, num_filters * 8 * 2,
            num_filters * 8, is_deconv)
179         self.dec4 = DecoderBlockV2(256 + num_filters * 8, num_filters * 8 * 2,
            num_filters * 8, is_deconv)
180         self.dec3 = DecoderBlockV2(128 + num_filters * 8, num_filters * 4 * 2,
            num_filters * 2, is_deconv)
181         self.dec2 = DecoderBlockV2(64 + num_filters * 2, num_filters * 2 * 2,
            num_filters * 2 * 2, is_deconv)
182         self.dec1 = DecoderBlockV2(num_filters * 2 * 2, num_filters * 2 * 2,
            num_filters, is_deconv)
```

```
183             self.dec0 = ConvRelu(num_filters, num_filters)
184             self.final = nn.Conv2d(num_filters, num_classes, kernel_size=1)
185
186         def forward(self, x):
187             conv1 = self.conv1(x)
188             conv2 = self.conv2(conv1)
189             conv3 = self.conv3(conv2)
190             conv4 = self.conv4(conv3)
191             conv5 = self.conv5(conv4)
192
193             center = self.center(self.pool(conv5))
194
195             dec5 = self.dec5(torch.cat([center, conv5], 1))
196
197             dec4 = self.dec4(torch.cat([dec5, conv4], 1))
198             dec3 = self.dec3(torch.cat([dec4, conv3], 1))
199             dec2 = self.dec2(torch.cat([dec3, conv2], 1))
200             dec1 = self.dec1(dec2)
201             dec0 = self.dec0(dec1)
202
203             if self.num_classes > 1:
204                 x_out = F.log_softmax(self.final(dec0), dim=1)
205             else:
206                 x_out = self.final(dec0)
207
208             return x_out
209
210
211     class UNet16(nn.Module):
212         def __init__(self, num_classes=1, num_filters=32, pretrained=False,
        is_deconv=False):
213             """
214             :param num_classes:
215             :param num_filters:
216             :param pretrained:
217                 False - no pre-trained network used
218                 True - encoder pre-trained with VGG16
219             :is_deconv:
220                 False: bilinear interpolation is used in decoder
221                 True: deconvolution is used in decoder
222             """
223             super().__init__()
224             self.num_classes = num_classes
225
226             self.pool = nn.MaxPool2d(2, 2)
227
228             self.encoder = torchvision.models.vgg16(pretrained=pretrained).features
229
230             self.relu = nn.ReLU(inplace=True)
231
232             self.conv1 = nn.Sequential(self.encoder[0],
233                                        self.relu,
234                                        self.encoder[2],
235                                        self.relu)
236
237             self.conv2 = nn.Sequential(self.encoder[5],
238                                        self.relu,
239                                        self.encoder[7],
240                                        self.relu)
241
242             self.conv3 = nn.Sequential(self.encoder[10],
243                                        self.relu,
244                                        self.encoder[12],
245                                        self.relu,
246                                        self.encoder[14],
247                                        self.relu)
```

```
248
249             self.conv4 = nn.Sequential(self.encoder[17],
250                                        self.relu,
251                                        self.encoder[19],
252                                        self.relu,
253                                        self.encoder[21],
254                                        self.relu)
255
256             self.conv5 = nn.Sequential(self.encoder[24],
257                                        self.relu,
258                                        self.encoder[26],
259                                        self.relu,
260                                        self.encoder[28],
261                                        self.relu)
262
263             self.center = DecoderBlockV2(512, num_filters * 8 * 2, num_filters * 8,
                is_deconv)
264
265             self.dec5 = DecoderBlockV2(512 + num_filters * 8, num_filters * 8 * 2,
                num_filters * 8, is_deconv)
266             self.dec4 = DecoderBlockV2(512 + num_filters * 8, num_filters * 8 * 2,
                num_filters * 8, is_deconv)
267             self.dec3 = DecoderBlockV2(256 + num_filters * 8, num_filters * 4 * 2,
                num_filters * 2, is_deconv)
268             self.dec2 = DecoderBlockV2(128 + num_filters * 2, num_filters * 2 * 2,
                num_filters, is_deconv)
269             self.dec1 = ConvRelu(64 + num_filters, num_filters)
270             self.final = nn.Conv2d(num_filters, num_classes, kernel_size=1)
271
272         def forward(self, x):
273             conv1 = self.conv1(x)
274             conv2 = self.conv2(self.pool(conv1))
275             conv3 = self.conv3(self.pool(conv2))
276             conv4 = self.conv4(self.pool(conv3))
277             conv5 = self.conv5(self.pool(conv4))
278
279             center = self.center(self.pool(conv5))
280
281             dec5 = self.dec5(torch.cat([center, conv5], 1))
282
283             dec4 = self.dec4(torch.cat([dec5, conv4], 1))
284             dec3 = self.dec3(torch.cat([dec4, conv3], 1))
285             dec2 = self.dec2(torch.cat([dec3, conv2], 1))
286             dec1 = self.dec1(torch.cat([dec2, conv1], 1))
287
288             if self.num_classes > 1:
289                 x_out = F.log_softmax(self.final(dec1), dim=1)
290             else:
291                 x_out = self.final(dec1)
292
293             return x_out
294
```