# 目录

[目录 1](#_Toc18000)

[字符识别 2](#_Toc32328)

[1 实验背景 2](#_Toc8164)

[2 实验目标 2](#_Toc2870)

[3 理论介绍 2](#_Toc15589)

[4 实验过程与结果 3](#_Toc1644)

[5 实验结论 7](#_Toc16052)

[附录 8](#_Toc6258)

# 字符识别

## 1 实验背景

本次实验使用 U-Net 网络结构进行气球识别。实验提供 74 张图像，其中 61 张用于训练，13 张用于测试。实验通过 U-Net 先对图像进行编码，再通过解码过程生成预测的掩膜分类。最终，通过计算测试集的 IoU[[1]](#footnote-0)评估模型的表现。

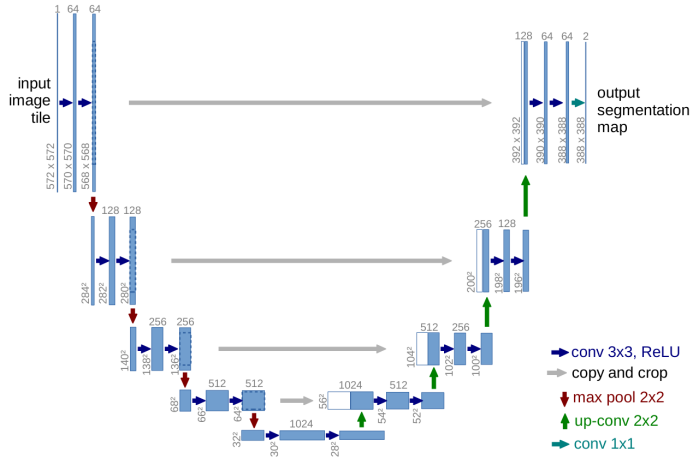
## 2 实验目标

### 1、掌握利用Unet网络的实现原理和方法

### 2、掌握基于pytorch的的网络搭建编程以及模型的训练

### 3 理论介绍

1. Net 结构呈现对称的 U 形，主要由编码器（Contracting Path）和解码器（Expanding Path）两部分组成。编码器：逐步降低输入图像的空间分辨率，同时提取重要的特征信息。解码器：逐步恢复空间分辨率，重建图像的细节部分。跳跃连接：在编码器和解码器之间建立直接连接，传递高分辨率的特征信息，从而保持细节，提高分割精度。U-Net网络的示意图如下[[2]](#footnote-1)：



### 4 实验过程与结果

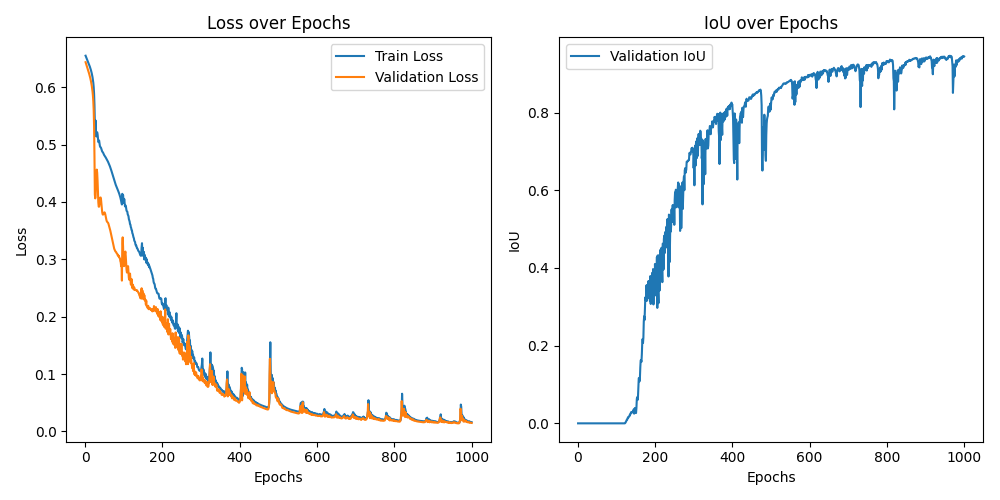
1、模型搭建

根据实验要求构建 U-Net 神经网络，该网络包含卷积层、池化层和上采样层等基本结构。由于实验中训练数据较少，为了避免增加网络的训练复杂度并减缓收敛速度，本次模型未使用批量归一化（BN）层和残差连接等高级结构。具体的模型架构如下，下文称为 UNetSmall：

1. **class UNetSmall(UNet):**
2. **def \_\_init\_\_(self, IMG\_HEIGHT: int = 128, IMG\_WIDTH: int = 128, IMG\_CHANNELS: int = 3):**
3. **super(UNetSmall, self).\_\_init\_\_()**
4. **self.IMG\_HEIGHT = IMG\_HEIGHT**
5. **self.IMG\_WIDTH = IMG\_WIDTH**
6. **self.IMG\_CHANNELS = IMG\_CHANNELS**
7. **# Encoder (Contracting Path)**
8. **self.enc1 = nn.Sequential(**
9. **nn.Conv2d(IMG\_CHANNELS, 8, 3, padding=1),  # 3 x 128 x 128 -> 8 x 128 x 128**
10. **nn.ReLU(inplace=True),**
11. **nn.Conv2d(8, 8, 3, padding=1),  # 8 x 128 x 128 -> 8 x 128 x 128**
12. **nn.ReLU(inplace=True),**
13. **)**
14. **self.enc2 = nn.Sequential(**
15. **nn.MaxPool2d(2),**
16. **nn.Conv2d(8, 16, 3, padding=1),  # 8 x 64 x 64 -> 16 x 64 x 64**
17. **nn.ReLU(inplace=True),**
18. **nn.Conv2d(16, 16, 3, padding=1),  # 16 x 64 x 64 -> 16 x 64 x 64**
19. **nn.ReLU(inplace=True),**
20. **)**
21. **self.enc3 = nn.Sequential(**
22. **nn.MaxPool2d(2),**
23. **nn.Conv2d(16, 32, 3, padding=1),  # 16 x 32 x 32 -> 32 x 32 x 32**
24. **nn.ReLU(inplace=True),**
25. **nn.Conv2d(32, 32, 3, padding=1),  # 32 x 32 x 32 -> 32 x 32 x 32**
26. **nn.ReLU(inplace=True),**
27. **)**
28. **self.enc4 = nn.Sequential(**
29. **nn.MaxPool2d(2),**
30. **nn.Conv2d(32, 64, 3, padding=1),  # 32 x 16 x 16 -> 64 x 16 x 16**
31. **nn.ReLU(inplace=True),**
32. **nn.Conv2d(64, 64, 3, padding=1),  # 64 x 16 x 16 -> 64 x 16 x 16**
33. **nn.ReLU(inplace=True),**
34. **)**
35. **# Bridge (Bottle Neck) 64 x 16 x 16 -> 128 x 16 x 16**
36. **self.bridge = nn.Sequential(**
37. **nn.MaxPool2d(2),**
38. **nn.Conv2d(64, 128, 3, padding=1),  # 64 x 8 x 8 -> 128 x 8 x 8**
39. **nn.ReLU(inplace=True),**
40. **nn.Conv2d(128, 128, 3, padding=1),  # 128 x 8 x 8 -> 128 x 8 x 8**
41. **nn.ReLU(inplace=True),**
42. **)**
43. **# Decoder (Expanding Path)**
44. **self.dec4 = nn.Sequential(**
45. **nn.ConvTranspose2d(128, 128, 2, stride=2),  # 128 x 8 x 8 -> 128 x 16 x 16**
46. **nn.Conv2d(128, 64, 3, padding=1),  # 128 x 16 x 16 -> 64 x 16 x 16**
47. **nn.ReLU(inplace=True),**
48. **nn.Conv2d(64, 64, 3, padding=1),  # 64 x 16 x 16 -> 64 x 16 x 16**
49. **nn.ReLU(inplace=True),**
50. **)**
51. **self.dec3 = nn.Sequential(**
52. **nn.ConvTranspose2d(128, 64, 2, stride=2),  # 128 x 16 x 16 -> 64 x 32 x 32**
53. **nn.Conv2d(64, 32, 3, padding=1),  # 64 x 32 x 32 -> 32 x 32 x 32**
54. **nn.ReLU(inplace=True),**
55. **nn.Conv2d(32, 32, 3, padding=1),  # 32 x 32 x 32 -> 32 x 32 x 32**
56. **nn.ReLU(inplace=True),**
57. **)**
58. **self.dec2 = nn.Sequential(**
59. **nn.ConvTranspose2d(64, 32, 2, stride=2),  # 64 x 32 x 32 -> 32 x 64 x 64**
60. **nn.Conv2d(32, 16, 3, padding=1),  # 32 x 64 x 64 -> 16 x 64 x 64**
61. **nn.ReLU(inplace=True),**
62. **nn.Conv2d(16, 16, 3, padding=1),  # 16 x 64 x 64 -> 16 x 64 x 64**
63. **nn.ReLU(inplace=True),**
64. **)**
65. **self.dec1 = nn.Sequential(**
66. **nn.ConvTranspose2d(32, 16, 2, stride=2),  # 32 x 64 x 64 -> 16 x 128 x 128**
67. **nn.Conv2d(16, 8, 3, padding=1),  # 16 x 128 x 128 -> 8 x 128 x 128**
68. **nn.ReLU(inplace=True),**
69. **nn.Conv2d(8, 8, 3, padding=1),  # 8 x 128 x 128 -> 8 x 128 x 128**
70. **nn.ReLU(inplace=True),**
71. **)**
72. **# Final layer 16 x 128 x 128 -> 1 x 128 x 128**
73. **self.final = nn.Sequential(nn.Conv2d(16, 1, 1), nn.Sigmoid())**

2、训练细节：

训练时，选择优化器为 Adam[[3]](#footnote-2)，损失函数为 BCELoss[[4]](#footnote-3)，设置 batch\_size = 100（一个 batch 中训练完所有的训练数据）, 训练 1000 epochs，初始学习率为 lr=5e-4，每 100 个 epoch 保存一次模型以及相关 loss 数据方便后续分析，训练大致用时为6min（训练具体环境见附录），训练过程损失函数如下所示：



随着训练次数的增加，训练损失和测试损失逐渐下降，但仍然存在一定的波动。这种波动表明，模型在训练过程中可能并未稳定收敛，且训练过程中的损失值未能平稳地持续降低。分析其原因，可能是由于训练数据量不足，导致模型无法充分学习到任务中的所有特征和规律，从而出现了局部最优解的情况

测试代码如下：

1. **def predict(model: UNet, images: torch.Tensor, masks, device) -> np.ndarray:**
2. **model.eval()**
3. **images = images.to(device)**
4. **masks = masks.to(device).bool().float()**
5. **criterion = nn.BCELoss().to(device)**
6. **iou\_metric = JaccardIndex(task="binary", num\_classes=2).to(device)**
7. **with torch.inference\_mode():**
8. **outputs = model(images)**
9. **preds = (outputs > 0.5).cpu().numpy().astype(int)**
10. **iou = iou\_metric(outputs, masks)**
11. **loss = criterion(outputs, masks)**
12. **print(f"Test Loss: {loss.item():.4f}. Test IoU: {iou.item():.4f}")**
13. **return preds**

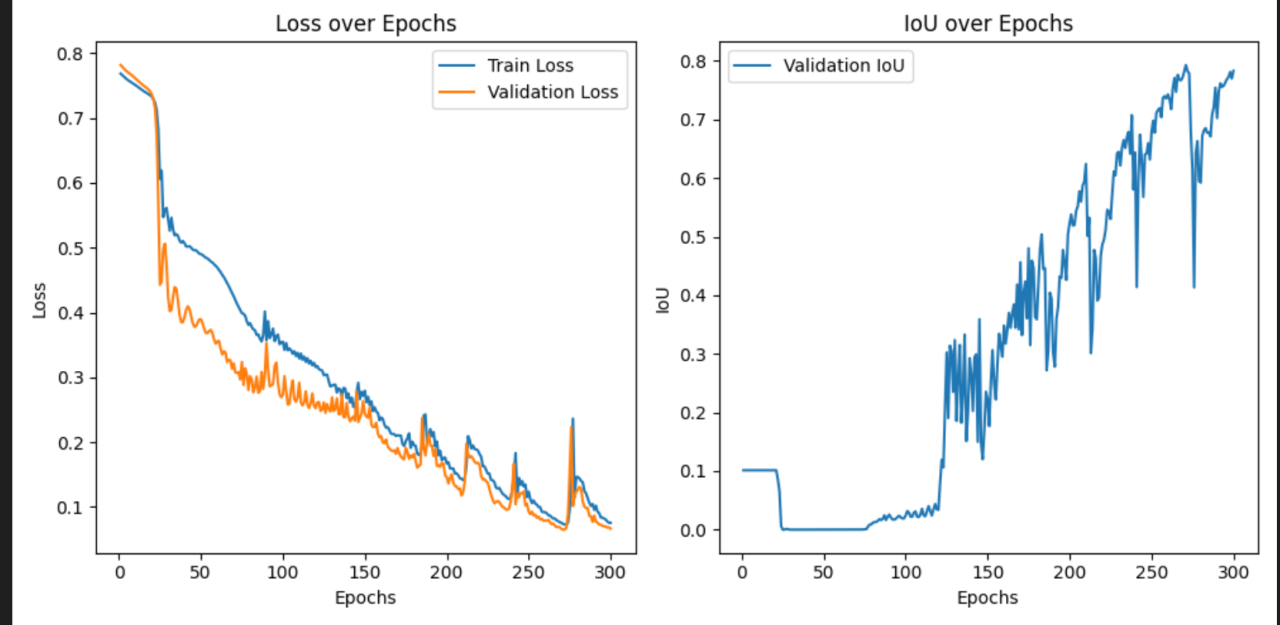
选取最后一个回合的模型进行测试，结果为：

1. **Test Loss: 0.0147. Test IoU: 0.9446**

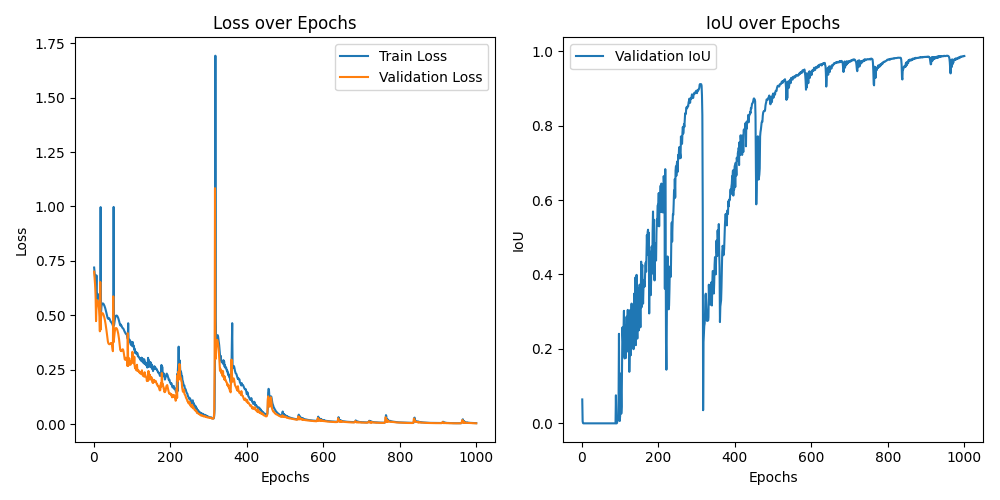
测试集分割效果展示见附录，可见U-Net模型可以较好地实现气球识别功能。

3、训练分析

3.1 尝试不同的学习率。发现较大的学习率如lr=1e-3会导致更加抖动的训练效果（训练时 loss 记录如下图），在训练初期（300epoch 以内）即产生更大幅度的抖动，同时并没有显著提高模型收敛的速度（对比下图和上图的IoU指标）；而采用过小的学习率如lr=1e-4会导致训练收敛速度极慢，显著降低了训练效率，故最终采用学习率 lr=5e-4 进行训练。



3.2 尝试更大的U-Net网络结构。在本次实验中，我采用了一个更大的 U-Net 网络结构（具体设置见附录），命名为 UNetLarge。实验结果表明，在相同的实验环境和设置下，UNetLarge 模型不仅能够顺利收敛，还能够实现更好的识别效果。然而，尽管该模型在训练过程中并未显著增加训练的计算开销或影响训练效率，但其损失（loss）曲线的波动较大（见下图），表现出较强的抖动。这表明，在当前的训练设置下，该模型可能需要更多的训练数据或更多的训练 epoch 才能完全收敛并发挥出其优势，或者采用更好的模型参数初始化方法，从而更快达到对目标的收敛。



使用上述的测试代码，结果如下：

1. **Test Loss: 0.0038. Test IoU: 0.9869**

4、创新点：

由于本次实验采用的是少量数据和多次 epoch 的训练，每个 epoch 都需要对训练数据进行相同的处理。具体来说，在数据集类的 \_\_getitem\_\_ 方法中，图像需要经过编码处理并生成对应的掩码，这一过程相对复杂。如果每个 epoch 都重新对数据进行这一系列处理，将导致训练总耗时显著增加，从而大大影响训练的效率。为了避免这种情况，我采用了缓存机制，将每个 epoch 处理过的数据保存到本地文件中，这样后续的训练过程中可以直接读取已处理的数据，而无需重复计算。通过这种方式，显著提升了训练的效率，减少了不必要的计算开销。实验中，这一优化将训练时间大幅缩短，将 1000 epoch 的训练时间从原本的 3 小时减少到了仅 10 分钟。相关代码实现如下：

1. **def \_\_getitem\_\_(self, idx: int) -> tuple[torch.Tensor, torch.Tensor]:**
2. **tid: str = list(self.annotations.keys())[idx]**
3. **cache\_path = self.\_get\_cache\_path(tid)**
4. **if os.path.exists(cache\_path):**
5. **with np.load(cache\_path) as data:  # 从缓存加载数据**
6. **image = data["image"]**
7. **mask = data["mask"]**
8. **else:**
9. **anno: dict[str] = self.annotations[tid]**
10. **mask, image, \_, \_, \_, \_ = self.\_get\_mask(anno, self.dataset\_dir)**
11. **mask = resize(mask, self.img\_size, mode="constant", preserve\_range=True).astype(np.float32)**
12. **image = resize(image, self.img\_size, mode="constant", preserve\_range=True).astype(np.float32) / 255.0**
13. **if self.transform is not None:**
14. **image = self.transform(image).to(torch.float32)**
15. **mask: torch.Tensor = torch.tensor(mask, dtype=torch.float32).unsqueeze(0)**
16. **np.savez\_compressed(cache\_path, image=image, mask=mask)**
17. **return image, mask**

# **5 实验结论**

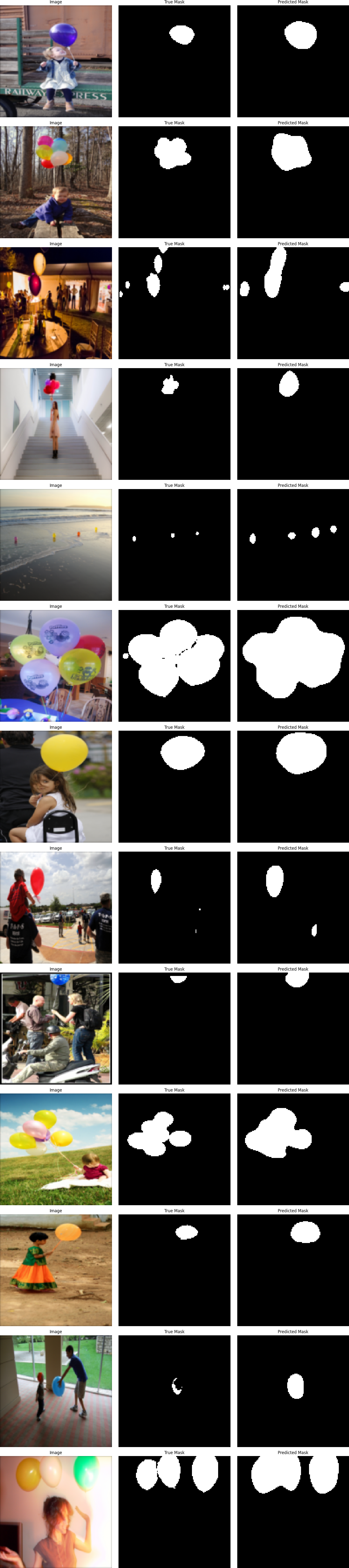
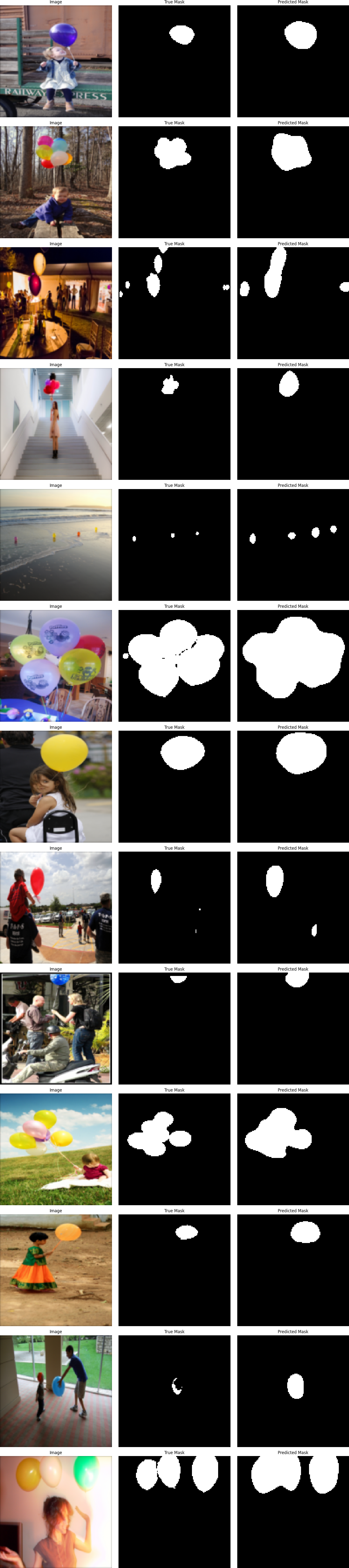
实验结果表明，对于气球识别任务，使用 U-Net 网络结构能够在较少的训练数据条件下实现令人满意的效果。U-Net 通过其独特的编码器-解码器结构和跳跃连接，能够有效提取图像中的重要特征，并在保持空间细节的同时实现精准的分割，特别适合用于小样本的图像分割任务。即使在数据量相对较少的情况下，U-Net 也能迅速收敛，并获得较高的分割精度。

此外，实验还表明，通过采用缓存机制等方法，可以显著减少训练过程中的数据预处理时间，从而有效降低训练的总耗时。尤其在训练数据量有限时，每个 epoch 的数据处理往往成为瓶颈。通过将处理后的数据保存到本地并在后续训练中直接调用，避免了重复的计算和数据处理，极大提高了训练的效率。通过这些优化，实验中的训练时间得到了显著缩短，进一步提升了模型训练的速度和效率。

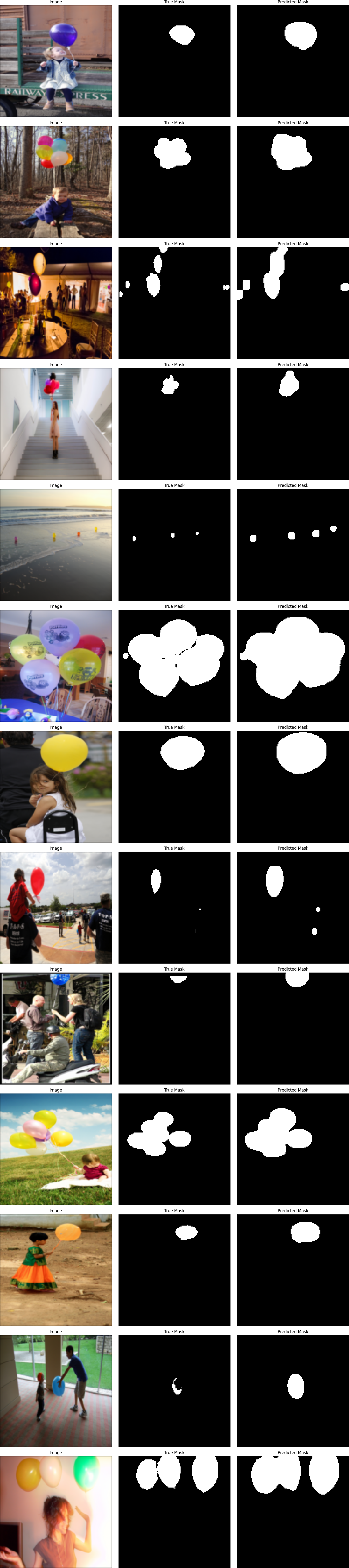
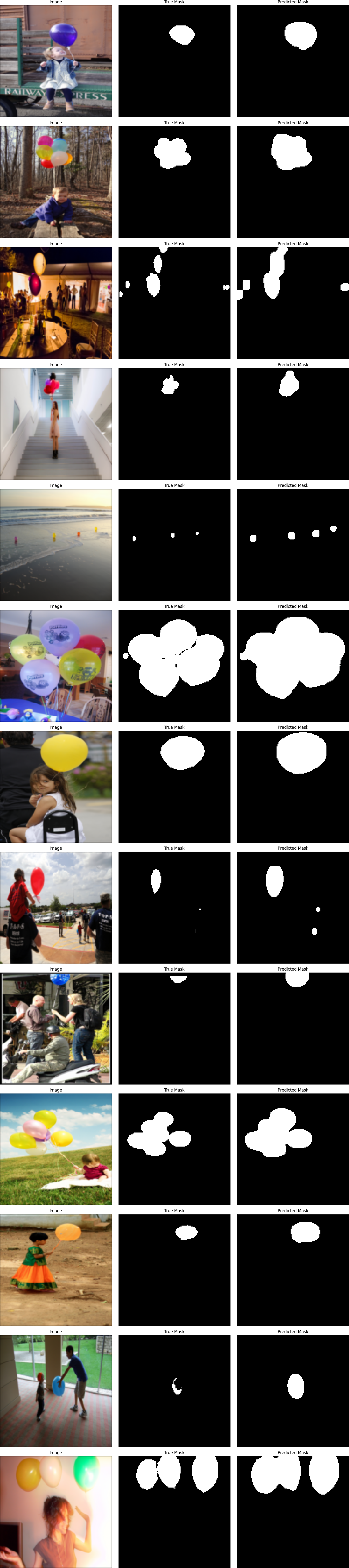
### 附录

1. 实验环境与配置
2. **- Platform: Linux-5.4.0-150-generic-x86\_64-with-glibc2.27**
3. **- Python version: 3.10.16**
4. **- Huggingface\_hub version: 0.27.0**
5. **- Safetensors version: 0.4.5**
6. **- Accelerate version: not installed**
7. **- Accelerate config: not found**
8. **- PyTorch version (GPU?): 2.5.1+cu124 (True)**
9. **- Tensorflow version (GPU?): not installed (NA)**
10. **- Flax version (CPU?/GPU?/TPU?): not installed (NA)**
11. **- Jax version: not installed**
12. **- JaxLib version: not installed**
13. 模型训练具体代码
14. **def train\_model(**
15. **model,**
16. **criterion,**
17. **optimizer,**
18. **train\_loader: DataLoader,**
19. **val\_loader: DataLoader,**
20. **num\_epochs: int,**
21. **need\_show: bool = False,**
22. **):**
23. **best\_model\_wts = model.state\_dict()**
24. **best\_iou, best\_epoch = 0.0, 0**
25. **iou\_metric = JaccardIndex(task="binary", num\_classes=2).to(device)**
26. **metrics = {"train\_loss": [], "val\_loss": [], "val\_iou": []}**
27. **for idx, epoch in enumerate(tqdm(range(num\_epochs))):**
28. **train\_loss = 0.0**
29. **model.train()**
30. **for images, masks in train\_loader:**
31. **images, masks = images.to(device), masks.to(device)**
32. **masks = masks.bool().float()**
33. **optimizer.zero\_grad()**
34. **outputs = model(images)**
35. **loss = criterion(outputs, masks)**
36. **loss.backward()**
37. **optimizer.step()**
38. **train\_loss += loss.item() \* images.size(0)**
39. **train\_loss /= len(train\_loader.dataset)**
40. **metrics["train\_loss"].append(train\_loss)**
41. **val\_loss, val\_iou = 0.0, 0.0**
42. **model.eval()**
43. **for images, masks in val\_loader:**
44. **with torch.inference\_mode():**
45. **images, masks = images.to(device), masks.to(device)**
46. **masks = masks.bool().float()**
47. **outputs = model(images)**
48. **# 计算测试集的loss和ioU**
49. **loss = criterion(outputs, masks)**
50. **val\_loss += loss.item() \* images.size(0)**
51. **iou = iou\_metric(outputs, masks)**
52. **val\_iou += iou.item() \* images.size(0)**
53. **if need\_show:**
54. **balloon = outputs[0] > 0.5**
55. **balloon = balloon.float() \* 255**
56. **balloon = balloon.permute(1, 2, 0)  # (1, 128, 128) -> (128, 128, 1)**
57. **image = masks[0].permute(1, 2, 0) \* 255**
58. **print(balloon.shape)**
59. **cv2.imwrite(f"balloon\_{idx}.jpg", balloon.cpu().numpy())**
60. **cv2.imwrite(f"image\_{idx}.jpg", image.cpu().numpy())**
61. **val\_loss /= len(val\_loader.dataset)**
62. **val\_iou /= len(val\_loader.dataset)**
63. **metrics["val\_loss"].append(val\_loss)**
64. **metrics["val\_iou"].append(val\_iou)**
65. **print(**
66. **"Epoch {}/{} - Train Loss: {:.4f} - Val Loss: {:.4f} - Val IoU: {:.4f}".format(**
67. **epoch + 1, num\_epochs, train\_loss / len(train\_loader.dataset), val\_loss, val\_iou**
68. **)**
69. **)**
70. **# 每 100 个 epoch 保存一次模型**
71. **if (epoch + 1) % 100 == 0:**
72. **os.makedirs(os.path.join(CUR\_DIR, "model"), exist\_ok=True)**
73. **torch.save(model.state\_dict(), os.path.join(CUR\_DIR, "model", f"model\_epoch\_{epoch + 1}.pth"))**
74. **save\_metrics(metrics, f"training\_metrics\_{epoch + 1}")**
75. **# 保存最佳模型**
76. **if val\_iou > best\_iou:**
77. **best\_iou = val\_iou**
78. **best\_model\_wts = model.state\_dict()**
79. **best\_epoch = epoch**
80. **model.load\_state\_dict(best\_model\_wts)**
81. **os.makedirs(os.path.join(CUR\_DIR, "model"), exist\_ok=True)**
82. **torch.save(model.state\_dict(), os.path.join(CUR\_DIR, "model", f"best\_model\_{num\_epochs}.pth"))**
83. **print(f"Best model found at epoch {best\_epoch} with IoU: {best\_iou}")**
84. **save\_metrics(metrics, "training\_metrics")**
85. **return model**
86. 大参数模型UNetLarge的具体配置与前向过程
87. **class UNetLarge(UNet):**
88. **def \_\_init\_\_(self, IMG\_HEIGHT: int = 128, IMG\_WIDTH: int = 128, IMG\_CHANNELS: int = 3):**
89. **super(UNetLarge, self).\_\_init\_\_()**
90. **self.IMG\_HEIGHT = IMG\_HEIGHT**
91. **self.IMG\_WIDTH = IMG\_WIDTH**
92. **self.IMG\_CHANNELS = IMG\_CHANNELS**
93. **# Encoder (Contracting Path)**
94. **self.enc1 = nn.Sequential(**
95. **nn.Conv2d(IMG\_CHANNELS, 64, 3, padding=1),  # 3 x 128 x 128 -> 64 x 128 x 128**
96. **nn.ReLU(inplace=True),**
97. **nn.Conv2d(64, 64, 3, padding=1),  # 64 x 128 x 128 -> 64 x 128 x 128**
98. **nn.ReLU(inplace=True),**
99. **)**
100. **self.enc2 = nn.Sequential(**
101. **nn.MaxPool2d(2),**
102. **nn.Conv2d(64, 128, 3, padding=1),  # 64 x 64 x 64 -> 128 x 64 x 64**
103. **nn.ReLU(inplace=True),**
104. **nn.Conv2d(128, 128, 3, padding=1),  # 128 x 64 x 64 -> 128 x 64 x 64**
105. **nn.ReLU(inplace=True),**
106. **)**
107. **self.enc3 = nn.Sequential(**
108. **nn.MaxPool2d(2),**
109. **nn.Conv2d(128, 256, 3, padding=1),  # 128 x 32 x 32 -> 256 x 32 x 32**
110. **nn.ReLU(inplace=True),**
111. **nn.Conv2d(256, 256, 3, padding=1),  # 256 x 32 x 32 -> 256 x 32 x 32**
112. **nn.ReLU(inplace=True),**
113. **)**
114. **self.enc4 = nn.Sequential(**
115. **nn.MaxPool2d(2),**
116. **nn.Conv2d(256, 512, 3, padding=1),  # 256 x 16 x 16 -> 512 x 16 x 16**
117. **nn.ReLU(inplace=True),**
118. **nn.Conv2d(512, 512, 3, padding=1),  # 512 x 16 x 16 -> 512 x 16 x 16**
119. **nn.ReLU(inplace=True),**
120. **)**
121. **# Bridge (Bottle Neck) 512 x 16 x 16 -> 1024 x 16 x 16**
122. **self.bridge = nn.Sequential(**
123. **nn.MaxPool2d(2),**
124. **nn.Conv2d(512, 1024, 3, padding=1),  # 512 x 8 x 8 -> 1024 x 8 x 8**
125. **nn.ReLU(inplace=True),**
126. **nn.Conv2d(1024, 1024, 3, padding=1),  # 1024 x 8 x 8 -> 1024 x 8 x 8**
127. **nn.ReLU(inplace=True),**
128. **)**
129. **# Decoder (Expanding Path)**
130. **self.dec4 = nn.Sequential(**
131. **nn.ConvTranspose2d(1024, 1024, 2, stride=2),  # 1024 x 8 x 8 -> 1024 x 16 x 16**
132. **nn.Conv2d(1024, 512, 3, padding=1),  # 1024 x 16 x 16 -> 512 x 16 x 16**
133. **nn.ReLU(inplace=True),**
134. **nn.Conv2d(512, 512, 3, padding=1),  # 512 x 16 x 16 -> 512 x 16 x 16**
135. **nn.ReLU(inplace=True),**
136. **)**
137. **self.dec3 = nn.Sequential(**
138. **nn.ConvTranspose2d(1024, 512, 2, stride=2),  # 1024 x 16 x 16 -> 512 x 32 x 32**
139. **nn.Conv2d(512, 256, 3, padding=1),  # 512 x 32 x 32 -> 256 x 32 x 32**
140. **nn.ReLU(inplace=True),**
141. **nn.Conv2d(256, 256, 3, padding=1),  # 256 x 32 x 32 -> 256 x 32 x 32**
142. **nn.ReLU(inplace=True),**
143. **)**
144. **self.dec2 = nn.Sequential(**
145. **nn.ConvTranspose2d(512, 256, 2, stride=2),  # 512 x 32 x 32 -> 256 x 64 x 64**
146. **nn.Conv2d(256, 128, 3, padding=1),  # 256 x 64 x 64 -> 128 x 64 x 64**
147. **nn.ReLU(inplace=True),**
148. **nn.Conv2d(128, 128, 3, padding=1),  # 128 x 64 x 64 -> 128 x 64 x 64**
149. **nn.ReLU(inplace=True),**
150. **)**
151. **self.dec1 = nn.Sequential(**
152. **nn.ConvTranspose2d(256, 128, 2, stride=2),  # 256 x 64 x 64 -> 128 x 128 x 128**
153. **nn.Conv2d(128, 64, 3, padding=1),  # 128 x 128 x 128 -> 64 x 128 x 128**
154. **nn.ReLU(inplace=True),**
155. **nn.Conv2d(64, 64, 3, padding=1),  # 64 x 128 x 128 -> 64 x 128 x 128**
156. **nn.ReLU(inplace=True),**
157. **)**
158. **# Final layer 128 x 128 x 128 -> 1 x 128 x 128**
159. **self.final = nn.Sequential(nn.Conv2d(128, 1, 1), nn.Sigmoid())**
160. **def forward(self, x: torch.Tensor) -> torch.Tensor:**
161. **# x: 1 x 3 x 128 x 128**
162. **# Encoder**
163. **e1 = self.enc1(x)  # e1: 1 x 64 x 128 x 128**
164. **e2 = self.enc2(e1)  # e2: 1 x 128 x 64 x 64**
165. **e3 = self.enc3(e2)  # e3: 1 x 256 x 32 x 32**
166. **e4 = self.enc4(e3)  # e4: 1 x 512 x 16 x 16**
167. **# Bridge**
168. **b = self.bridge(e4)  # b: 1 x 1024 x 8 x 8**
169. **# Decoder**
170. **d4 = self.dec4(b)  # d4: 1 x 512 x 16 x 16**
171. **d4 = torch.cat((d4, e4), dim=1)  # d4: 1 x 1024 x 16 x 16**
172. **d3 = self.dec3(d4)  # d3: 1 x 256 x 32 x 32**
173. **d3 = torch.cat((d3, e3), dim=1)  # d3: 1 x 512 x 32 x 32**
174. **d2 = self.dec2(d3)  # d2: 1 x 128 x 64 x 64**
175. **d2 = torch.cat((d2, e2), dim=1)  # d2: 1 x 256 x 64 x 64**
176. **d1 = self.dec1(d2)  # d1: 1 x 64 x 128 x 128**
177. **d1 = torch.cat((d1, e1), dim=1)  # d1: 1 x 128 x 128 x 128**
178. **# Final layer**
179. **out = self.final(d1)  # out: 1 x 1 x 128 x 128**
180. **return out**
181. 展示分割效果部分代码
182. **def plot\_predictions(**
183. **images: np.ndarray, true\_masks: np.ndarray, pred\_masks: np.ndarray,**
184. **need\_show: bool = True**
185. **) -> None:**
186. **"""**
187. **画出预测结果，包括真实掩模和预测掩模**
188. **Args:**
189. **images: np.ndarray, shape=(N, 3, 128, 128)**
190. **true\_masks: np.ndarray, shape=(N, 1, 128, 128)**
191. **pred\_masks: np.ndarray, shape=(N, 1, 128, 128)**
192. **"""**
193. **num\_samples: int = len(images)**
194. **fig, axes = plt.subplots(num\_samples, 3, figsize=(15, 5 \* num\_samples))**
195. **for idx in range(num\_samples):**
196. **image: np.ndarray = images[idx].transpose(1, 2, 0) \* 255**
197. **true\_mask = true\_masks[idx]  # (1, 128, 128)**
198. **pred\_mask = pred\_masks[idx][0]  # (128, 128)**
199. **pred\_mask = np.expand\_dims(pred\_mask, axis=0)  # (1, 128, 128)**
200. **if need\_show:**
201. **axes[idx, 0].imshow(image.astype(np.uint8))**
202. **axes[idx, 0].set\_title("Image")**
203. **axes[idx, 0].axis("off")**
204. **axes[idx, 1].imshow((true\_mask > 0.5).astype(int).squeeze())**
205. **axes[idx, 1].set\_title("True Mask")**
206. **axes[idx, 1].axis("off")**
207. **axes[idx, 2].imshow(pred\_mask.squeeze(), cmap="gray")**
208. **axes[idx, 2].set\_title("Predicted Mask")**
209. **axes[idx, 2].axis("off")**
210. **plt.tight\_layout()**
211. **os.makedirs(os.path.join(CUR\_DIR, "prediction"), exist\_ok=True)**
212. **plt.savefig(os.path.join(CUR\_DIR, "prediction", "all\_predictions.png"), bbox\_inches="tight", pad\_inches=0)**
213. **plt.close()**
214. 分割效果展示

1、UNetSmall 模型效果：



1. UNetLarge 模型效果：



1. IoU（Intersection over Union）是一个常用的评估指标，特别在目标检测和图像分割任务中，用于衡量预测结果与真实标签之间的重叠程度。它计算的是预测区域与实际区域交集的面积与它们并集的面积之比。 [↑](#footnote-ref-0)
2. Ronneberger, O., Fischer, P., Brox, T.: U-net: Convolutional networks for biomed ical image segmentation [↑](#footnote-ref-1)
3. Adam 是一种自适应优化算法，结合了动量法（Momentum）和自适应学习率的优势。它通过计算梯度的第一矩（均值）和第二矩（方差）来调整每个参数的学习率，从而在不同的参数上采用不同的更新步长，进而提高了训练的效率和稳定性。 [↑](#footnote-ref-2)
4. BCELoss 是二分类问题中常用的损失函数，用于衡量真实标签和预测概率之间的差异。它通过计算真实标签和预测标签概率之间的交叉熵，来评估模型的输出与真实结果的接近程度。 [↑](#footnote-ref-3)