MSC 科研组 2025 招新题

Selen Su

2025 年 9 月 14 日

1　引言

许多学者认为，深度学习是一种压缩技术，主要解决的问题是将复杂的信息压缩为可以用少量参数表示的形式。Selen 老师非常喜欢自己的头像：图 1，因此也想把它通过神经网络压缩一下。这意味着，无论什么时候，Selen 都可以在找不到原图的时候通过这个网络来重建图像。

图 1: 这将是大家用于完成题目的图像，也就是代码中的 ye.png

具体来说，Selen 老师决定训练一个 MLP（多层感知机）来学习一个映射关系 $f:(x,y)\to(R,G,B)$。即，输入一个像素的坐标，网络输出该点的颜色值；输入一连串图像的每个像素点的坐标，网络就还原一个完整的图像。

Selen 老师很快写出来了代码 Listing 1. 然而，这效果实在是太差了!! 经过 8000 轮的训练，模型输出的人物还是糊糊的，如图 2。这个时候 Selen 想到，在深度学习的表示方式中，许多研究在处理图像的时候会采用位置编码的方式，即通过 sin-cos 编码图像不同位置的坐标。位置编码通过一系列不同频率的正弦和余弦函数，将低维的输入坐标 $\mathbf{p}$ 映射到一个更高维的特征向量 $\gamma(\mathbf{p})$。其公式如下：

$$\gamma(\mathbf{p})=\left(\ldots,\sin(2^k\pi\mathbf{p}),\cos(2^k\pi\mathbf{p}),\ldots\right)_{k=0}^{L-1}$$

Selen 觉得如果将原始的 x,y 坐标输入转换成这样编码以后的输入，模型就可以更好地了解不同坐标之间的位置关系，这样对于全局的理解是更好的。于是 Selen 为了控制变量，没有改变模型，

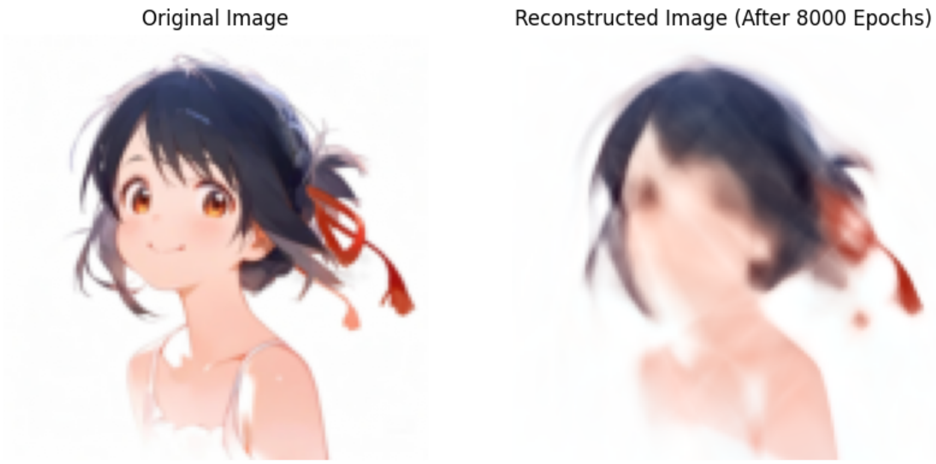

图 2: 糊糊的重建图像

也没有改变训练的轮数等任何设置，仅仅将输入的 x,y 替换为了 sin-cos 编码以后的表示，并写好了代码 Listing 2。

经过同样的模型同样程度的训练，重建结果却清晰了好多！如图 3.

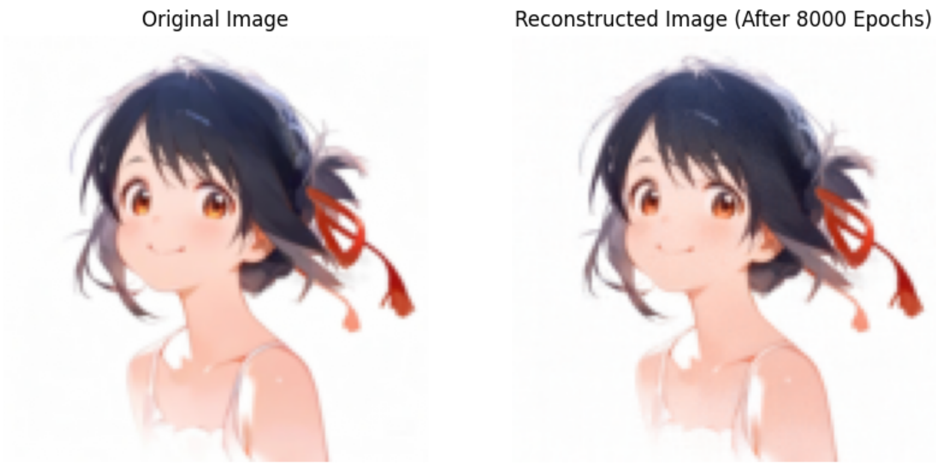

图 3: 清晰的重建图像

2 你需要解决的

- 为什么会出现这种差异呢?
- 有没有什么更好的重建方法呢?

- 你可以提交代码/文字报告到邮箱：s3702681@gmail.com，或通过 qq 联系 Selen
- 相关资料会上传到群文件中，请注意查收

3 代码

```python
import torch
import torch.nn as nn
from torchvision.transforms import functional as TF
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import time
import os

# --- 1. 超参数设置 ---
IMAGE_PATH = 'ye.png'
IMG_SIZE = 128
HIDDEN_DIM = 256
N_HIDDEN_LAYERS = 4
EPOCHS = 8000
LEARNING_RATE = 1e-4
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

print(f"Using device: {DEVICE}")

# 加载和预处理图像
img = Image.open(IMAGE_PATH).convert('RGB')
print(f"Successfully loaded image from: {IMAGE_PATH}")

img = TF.resize(img, (IMG_SIZE, IMG_SIZE))
img_tensor = TF.to_tensor(img).to(DEVICE) # shape: [3, H, W], 范围 [0, 1]

#     2. 对像素值进行标准化
# 计算每个通道的均值和标准差
# img_tensor shape is [C, H, W], so we calculate mean/std over H and W dimensions
mean = torch.mean(img_tensor, dim=[1, 2])
std = torch.std(img_tensor, dim=[1, 2])

# 为防止标准差为0（例如纯色通道）导致除零错误，增加一个极小值
std = torch.max(std, torch.tensor(1e-6).to(DEVICE))

print(f"\nImage stats (per channel):")
print(f"Mean: {mean.cpu().numpy()}")
print(f"Std:  {std.cpu().numpy()}")

# 应用标准化: (x - mean) / std
# 我们需要调整 mean 和 std 的形状以利用广播机制
img_tensor_standardized = (img_tensor - mean[:, None, None]) / std[:, None, None]

H, W = IMG_SIZE, IMG_SIZE
pixels = img_tensor_standardized.permute(1, 2, 0).view(-1, 3) # 使用标准化后的像素作为目标

# --- 3. 创建输入数据（坐标网格） ---
grid_y, grid_x = torch.meshgrid(torch.linspace(0, H - 1, H), torch.linspace(0, W - 1, W), indexing='ij')
coords = torch.stack([
    grid_x / (W - 1) * 2 - 1,
    grid_y / (H - 1) * 2 - 1
], dim=-1).to(DEVICE)
coords = coords.view(-1, 2)

print(f"\nImage Size: {H}x{W}")
```

```python
print(f"Input coordinates shape: {coords.shape}")
print(f"Target pixels shape: {pixels.shape}")


# --- 4. 构建 MLP 模型 ---
class MLPImageReconstructorLinear(nn.Module):
    def __init__(self, in_features, hidden_features, hidden_layers, out_features):
        super().__init__()
        layers = []
        layers.append(nn.Linear(in_features, hidden_features))
        layers.append(nn.ReLU())
        for _ in range(hidden_layers):
            layers.append(nn.Linear(hidden_features, hidden_features))
            layers.append(nn.ReLU())

        layers.append(nn.Linear(hidden_features, out_features))

        self.net = nn.Sequential(*layers)

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

model = MLPImageReconstructorLinear(
    in_features=2,
    hidden_features=HIDDEN_DIM,
    hidden_layers=N_HIDDEN_LAYERS,
    out_features=3
).to(DEVICE)

print("\nModel Architecture:")
print(model)

# --- 5. 定义损失函数和优化器 ---
loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)

# --- 6. 训练模型 ---
print("\nStarting training...")
start_time = time.time()
for epoch in range(EPOCHS):
    predicted_pixels = model(coords)
    loss = loss_fn(predicted_pixels, pixels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if (epoch + 1) % 100 == 0:
        print(f"Epoch [{epoch+1}/{EPOCHS}], Loss: {loss.item():.6f}")

end_time = time.time()
print(f"\nTraining finished in {end_time - start_time:.2f} seconds.")

# --- 7. 重建与可视化 ---
model.eval()
with torch.no_grad():
    reconstructed_pixels_standardized = model(coords)

#     将标准化的输出逆向转换回 [0, 1] 范围以便显示
# 逆向操作: x_norm = x_std * std + mean
# 调整 mean 和 std 的形状以匹配 [N, C] 的像素列表
```

```python
reconstructed_pixels = reconstructed_pixels_standardized * std.view(1, -1) + mean.view(1, -1)

# 重要: 逆向转换后, 数值可能略微超出[0,1]范围, 需要裁剪
reconstructed_pixels.clamp_(0.0, 1.0)

reconstructed_img_tensor = reconstructed_pixels.view(H, W, 3)
reconstructed_img_np = reconstructed_img_tensor.cpu().numpy()
original_img_np = img_tensor.permute(1, 2, 0).cpu().numpy() # 原始图像依然使用[0,1]范围的张量

# 显示
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(original_img_np)
plt.axis('off')
plt.subplot(1, 2, 2)
plt.title(f"Reconstructed Image (After {EPOCHS} Epochs)")
plt.imshow(reconstructed_img_np)
plt.axis('off')
plt.show()
```

Listing 1: MLP 重建图像

```python
import torch
import torch.nn as nn
from torchvision.transforms import functional as TF
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import time
import os

# --- 1. 参数设置 ---
IMAGE_PATH = 'ye.png'
IMG_SIZE = 128
HIDDEN_DIM = 256
N_HIDDEN_LAYERS = 4
EPOCHS = 8000
LEARNING_RATE = 1e-4
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

#     新增: 位置编码参数
# L: 决定了编码后维度的参数。维度 = 2 * 输入维度 * L
# 越高的 L 能表示越高的频率
N_ENCODING_FUNCTIONS = 10

print(f"Using device: {DEVICE}")


#     新增: 位置编码器模块
class PositionalEncoder(nn.Module):
    def __init__(self, input_dims, num_functions):
        super().__init__()
        self.input_dims = input_dims
        self.num_functions = num_functions

        # 创建频率列表 [1, 2, 4, 8, ..., 2^(L-1)]
        self.freq_bands = 2.0 ** torch.arange(num_functions)

        # 计算编码后的输出维度
        # 对于每个输入维度(x, y), 每个频率都产生sin和cos两个值
        self.output_dims = input_dims * num_functions * 2

    def forward(self, x):
        # x shape: [N, input_dims]
        # unsqueeze(dim=-1) -> [N, input_dims, 1]
        # self.freq_bands -> [num_functions]
        # x * self.freq_bands -> [N, input_dims, num_functions]
        scaled_inputs = x.unsqueeze(-1) * self.freq_bands.to(x.device)

        # 将 sin 和 cos 的结果拼接起来
        # torch.sin(scaled_inputs) -> [N, input_dims, num_functions]
        # torch.cos(scaled_inputs) -> [N, input_dims, num_functions]
        # shape -> [N, input_dims, num_functions * 2]
        encoded = torch.cat([torch.sin(scaled_inputs), torch.cos(scaled_inputs)], dim=-1)

        # 展平为 [N, output_dims]
        return torch.flatten(encoded, start_dim=1)


# --- 2. 加载和预处理图像 ---
img = Image.open(IMAGE_PATH).convert('RGB')
```

```python
print(f"Successfully loaded image from: {IMAGE_PATH}")

img = TF.resize(img, (IMG_SIZE, IMG_SIZE))
img_tensor = TF.to_tensor(img).to(DEVICE)

# 标准化流程保持不变
mean = torch.mean(img_tensor, dim=[1, 2])
std = torch.std(img_tensor, dim=[1, 2])
std = torch.max(std, torch.tensor(1e-6).to(DEVICE))
img_tensor_standardized = (img_tensor - mean[:, None, None]) / std[:, None, None]
H, W = IMG_SIZE, IMG_SIZE
pixels = img_tensor_standardized.permute(1, 2, 0).view(-1, 3)

# --- 3. 创建输入数据 (坐标网格) ---
grid_y, grid_x = torch.meshgrid(torch.linspace(0, H - 1, H), torch.linspace(0, W - 1, W), indexing='ij')
coords = torch.stack([
    grid_x / (W - 1) * 2 - 1,
    grid_y / (H - 1) * 2 - 1
], dim=-1).to(DEVICE)
coords = coords.view(-1, 2)

#     修改点 1: 对坐标进行位置编码
encoder = PositionalEncoder(input_dims=2, num_functions=N_ENCODING_FUNCTIONS)
coords_encoded = encoder(coords) # 新的、高维的坐标输入

print(f"\nOriginal coords shape: {coords.shape}")
print(f"Encoded coords shape: {coords_encoded.shape}") # 维度显著增加
print(f"Target pixels shape: {pixels.shape}")


# --- 4. 构建 MLP 模型 ---
class MLPImageReconstructorLinear(nn.Module):
    def __init__(self, in_features, hidden_features, hidden_layers, out_features):
        super().__init__()
        layers = []
        layers.append(nn.Linear(in_features, hidden_features))
        layers.append(nn.ReLU())
        for _ in range(hidden_layers):
            layers.append(nn.Linear(hidden_features, hidden_features))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(hidden_features, out_features))
        self.net = nn.Sequential(*layers)

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

#     修改点 2: 使用编码后的维度作为模型输入
model = MLPImageReconstructorLinear(
    in_features=encoder.output_dims, # <-- 使用编码后的维度
    hidden_features=HIDDEN_DIM,
    hidden_layers=N_HIDDEN_LAYERS,
    out_features=3
).to(DEVICE)

print("\nModel Architecture:")
print(model)

# --- 5. 定义损失函数和优化器 ---
```

```python
loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)

# --- 6. 训练模型 ---
print("\nStarting training...")
start_time = time.time()
for epoch in range(EPOCHS):
    #     修改点 3: 使用编码后的坐标进行训练
    predicted_pixels = model(coords_encoded) # <-- 使用编码后的坐标

    loss = loss_fn(predicted_pixels, pixels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if (epoch + 1) % 100 == 0:
        print(f"Epoch [{epoch+1}/{EPOCHS}], Loss: {loss.item():.6f}")

end_time = time.time()
print(f"\nTraining finished in {end_time - start_time:.2f} seconds.")

# --- 7. 重建与可视化 ---
model.eval()
with torch.no_grad():
    # 重建时同样需要对坐标进行编码
    reconstructed_pixels_standardized = model(coords_encoded)

# 逆向标准化流程保持不变
reconstructed_pixels = reconstructed_pixels_standardized * std.view(1, -1) + mean.view(1, -1)
reconstructed_pixels.clamp_(0.0, 1.0)
reconstructed_img_tensor = reconstructed_pixels.view(H, W, 3)
reconstructed_img_np = reconstructed_img_tensor.cpu().numpy()
original_img_np = img_tensor.permute(1, 2, 0).cpu().numpy()

# 显示
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(original_img_np)
plt.axis('off')
plt.subplot(1, 2, 2)
plt.title(f"Reconstructed Image (After {EPOCHS} Epochs)")
plt.imshow(reconstructed_img_np)
plt.axis('off')
plt.show()
```

Listing 2: MLP 位置编码后重建图像