人工智能基础课程实验三

PB22151796-莫环欣

---

1. Kmeans

1.1 实验目的

理解`K-Means`算法的原理与流程，并学习如何将其应用于图像压缩任务中，同时理解不同`K`值对于压缩效果的影响

1.2 TODO 部分实现说明

若按方法顺序来说明，首先是分配方法，按要求列式即可，对每个店都求出最近的中心点然后分配

```
distances = np.sqrt(((points[:, np.newaxis] - centers) ** 2).sum(axis=2))
labels = np.argmin(distances, axis=1)
```

分配完毕后，需要求新的中心点信息，这里将所有点分配给对应的类，然后再对所有类逐一求均值

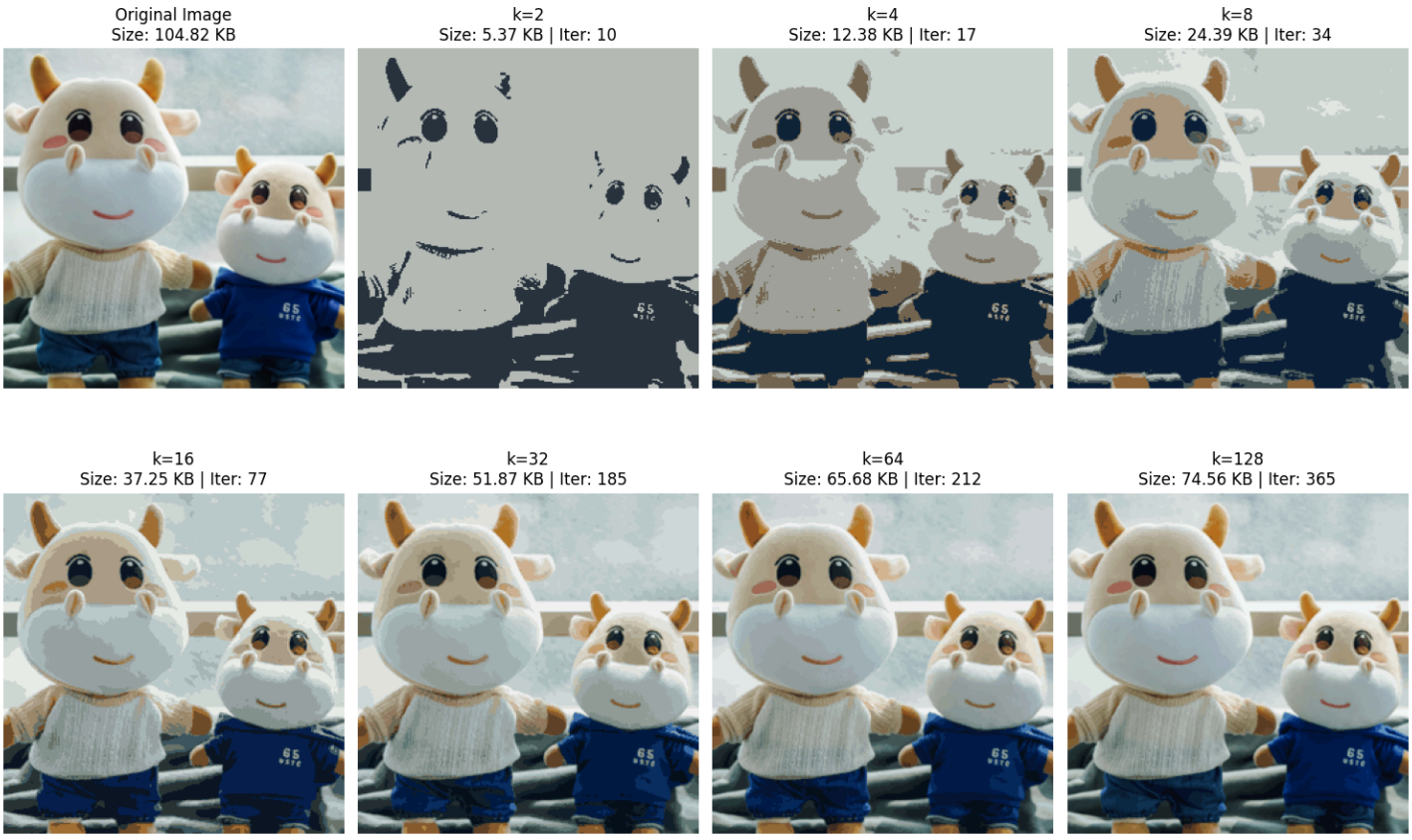
```
new_centers = np.zeros_like(centers)
for i in range(self.k):
    cluster = points[labels == i]
    if len(cluster) != 0:
        new_centers[i] = cluster.mean(axis=0)  # 对每个维度求均值 ---> (n_dims,)
return new_centers
```

`K-Means`算法的核心内容不需要实现，接下来是对图片执行颜色聚类，将每个像素分配到对应的颜色簇，分配完成后将像素颜色替换为所在簇的中心颜色

```
centers = self.fit(points)
labels = self.assign_points(centers, points).astype(int)
compressed_points = centers[labels]
return compressed_points.reshape(img.shape)
```

1.3 不同`k`值压缩图像对比展示

图片的颜色种类较少时图片质量（大小、真实质感）受到影响是肯定的（为`2`时只有黑白色），压缩算法后在`k=64`处得到了较好的压缩比与迭代次数，这里为美观起见多测试了`k=128`。整体对比部分详见代码中的`compare_k`方法



1.4 思考题

1.4.1 K-Means 聚类算法的核心思想和步骤

这个算法小测就有写过，核心思想通俗来说就是把所有的数据分为`k`个类，每个类中各自的所有数据都离类中心点最近（相比其它类中心点），而算法步骤就是给数据分类→找出中心→比较当前中心是否与上一轮相同，如果相同则终止/如果不同则继续迭代，每轮迭代都会将数据点分给离它最近的中心点，然后对每个类重新求中心点

1.4.2 随机均值初始化相较全随机初始化的优点

这样做可以减少随机性对结果的影响，同时由于随机结果大概率是分散的，因此取均值更可能使得初始中心点更接近数据分布的中心区域，进而加快收敛速度并提高算法的稳定性（避免选到极端远点等）

2. PCA

2.1 实验目的

学习`PCA`的基本原理以及在数据降维中的应用，理解`K`近邻算法基本原理，并结合两种算法完成人脸识别任务，同时分析`PCA`中主成分数量对分类效率与分类性能的影响

2.2 TODO 部分实现说明

首先是实例化，除基本主成分数量传值外还需按说明要求使用随机`SVD`方法且不白化

```
pca = PCA(n_components=n_components, svd_solver="randomized", whiten=False)
```

接着是平均脸与特征脸的可视化，提取之后需要按展示要求调整形状

```
eigenfaces = pca.components_.reshape(-1, h, w)
mean_face = pca.mean_.reshape((h, w))
plot_gallery("Mean Face", [mean_face], h, w, n_row=1, n_col=1)
plot_gallery(
    f"Top Eigenfaces", eigenfaces[: min(n_components, 10)], h, w, n_row=2, n_col=5
)
```

投影部分直接调用现有`transform`即可

```
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
```

`KNN`的实例化也按要求进行即可，使用 `distance` 权重和 `euclidean` 距离

```
knn_classifier = KNeighborsClassifier(
    n_neighbors=1, weights="distance", metric="euclidean"
)
```

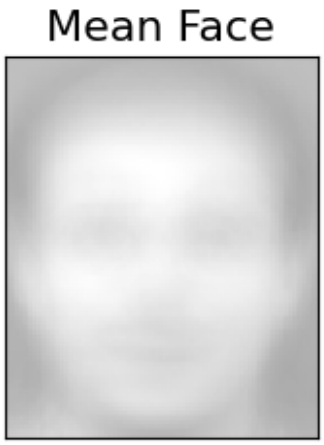
之后的评估与准确率计算也是直接调库

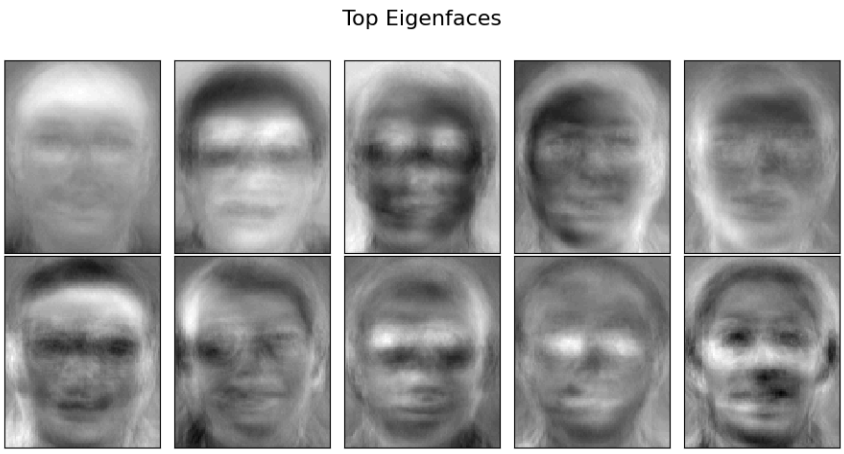
```
y_pred = knn_classifier.predict(X_test_pca)
```

```
accuracy = accuracy_score(y_test, y_pred)
```

2.3 平均脸和前几个特征脸的截图

平均脸简单说就是所有脸的均值；特征脸简单说就是主成分，每个脸是一个方向（数据中最大的方差信息）。另外经过降维后的图片基本上只剩下了大致轮廓

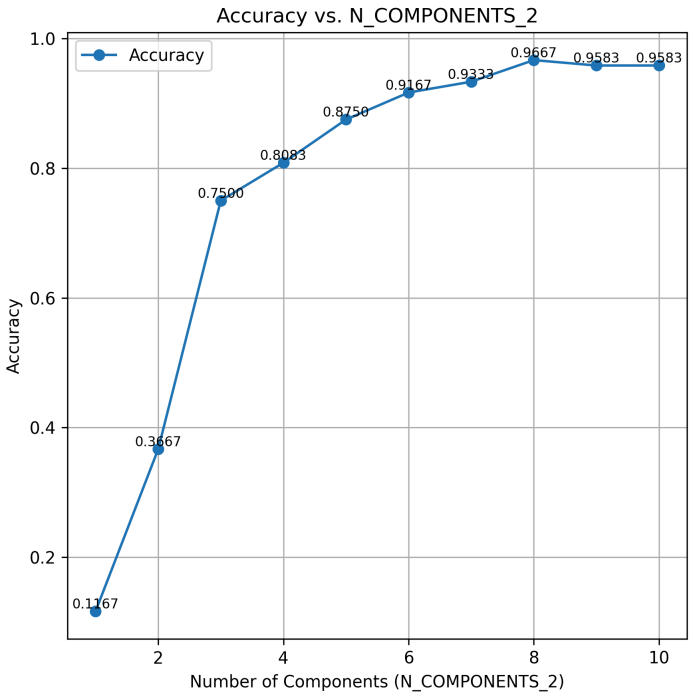


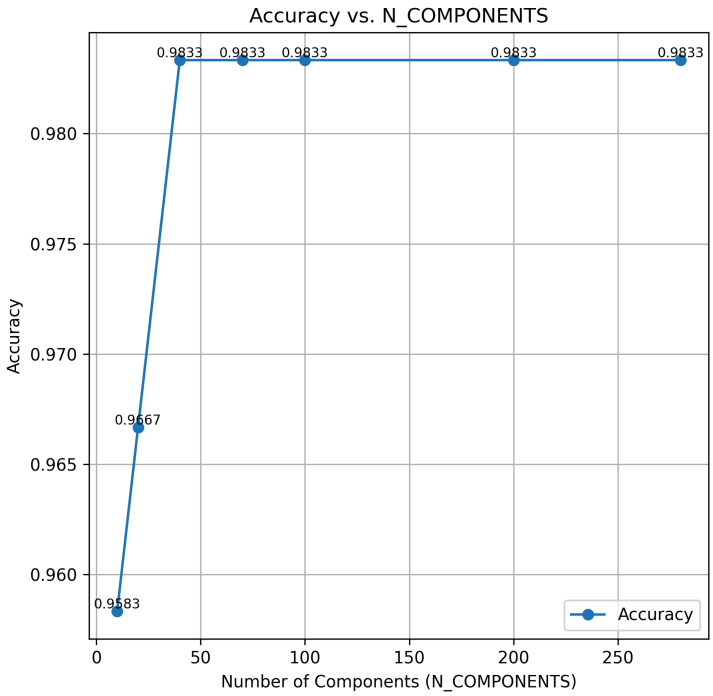


2.4 不同 `N_COMPONENTS` 值下的模型准确率对比图

这里需要调整参数配置，根据准确率结果选取了几个不会太密集的点，可以看到对于本任务，选取主成分数为`50`是比较好的选择，同时兼顾效率与效果（虽然说怎么都不会太慢）

```
N_COMPONENTS = [
    10,
    20,
    40,
    70,
    100,
    200,
    (int)(
        min(400 * (1 - TEST_SIZE_RATIO), IMAGE_HEIGHT * IMAGE_WIDTH)
    ),  # PCA的主成分数量由协方差矩阵的秩（受限于行数与列数）决定
]
N_COMPONENTS_2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```





2.5 思考题

2.5.1 PCA 算法的基本原理与目标

`PCA`借助数据的方差寻找一组正交的主成分并使数据在该方向上的投影方差最大，进而可以通过线性变换将高维数据映射到低维上，实现数据降维，进而在关键信息不丢失的前提下提高其它任务的数据处理效率

2.5.2 特征脸的说明

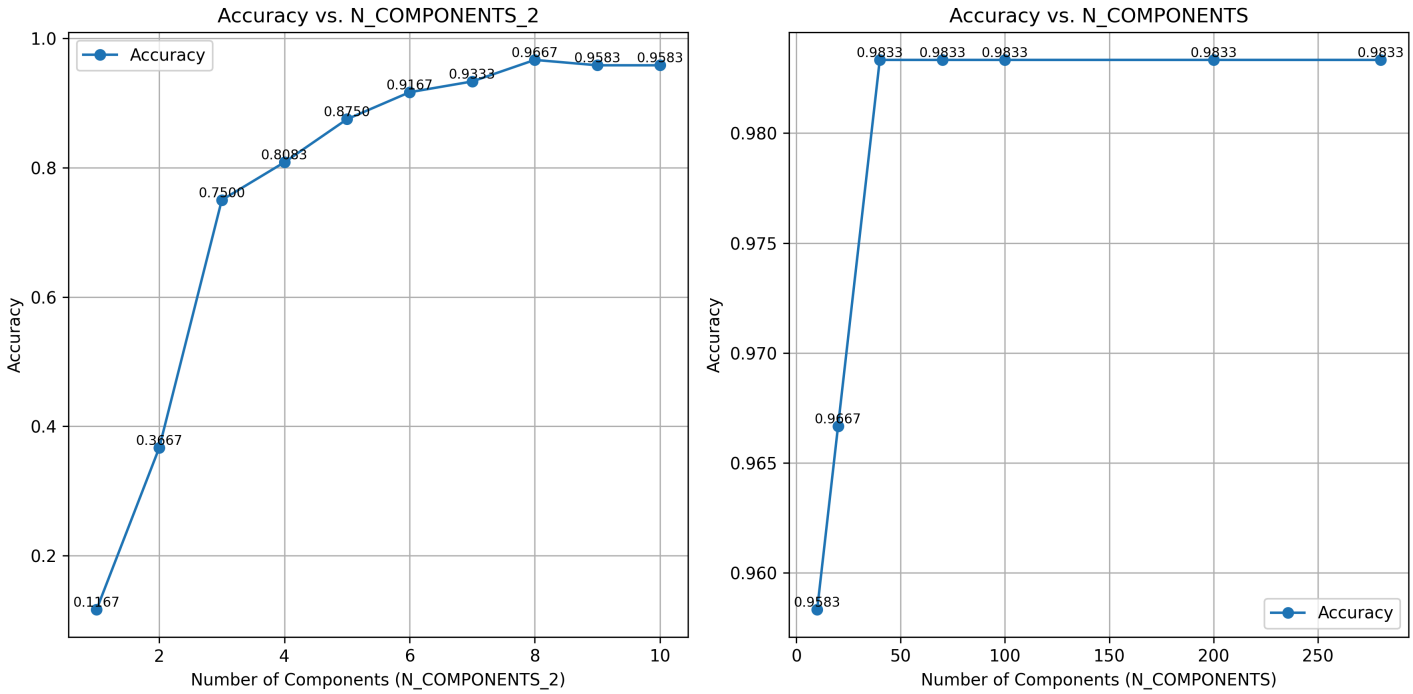
如上文所述，特征脸的本质就是一组主成分（或者说一种特征），其中表示的就是人脸数据的变化形式

在计算时，需要先将人脸灰度图展平为一维向量后构成矩阵，并进行标准化（减均值，除标准差），随后计算协方差矩阵，这个矩阵分解出来的特征值就是特征脸（当然一般还会排降序）

特征脸是`PCA`的结果，自然也与`PCA`有着相同的作用——数据降维，保留关键信息并剔除次要信息，进而减少计算复杂度

2.5.3 参数`N_COMPONENTS`的影响

当提高参数时，准确率也逐渐上升，但由于我们是对主成分排过降序的，提高参数值之后拿到的会是越来越不重要的内容，也因此参数提高到一定值后准确率不再有变化



可以通过多次随机取少量数据进行实验以选择合适的参数值，并取靠前的变化点。例如本次实验中取值为`50`可兼顾处理效率与最终性能（实际上测试发现`40`时也可以，但是不太稳定，取`45`可能也行）

3. GAN

3.1 实验目的

本部分实验任务较为简单，主要目的是学习`GAN`的基本原理、工作流程与训练方法，并尝试构建简单的生成器与判别器与观察训练过程

3.2 TODO 部分实现说明

本部分需要补全生成器、判别器、判别器真假样本的损失计算以及生成器的损失计算，整体上没有什么需要自行设计的地方，完全照着提示补全即可（`TODO`的说明也已经非常清晰了）

生成器：

```
class Generator(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            # TODO: 使用线性层将随机噪声映射到第一个隐藏层
            nn.Linear(input_dim, hidden_dim),
            # TODO: 使用 ReLU 作为激活函数，帮助模型学习非线性特征
            nn.ReLU(),
            # TODO: 使用线性层将第一个隐藏层映射到第二个隐藏层
            nn.Linear(hidden_dim, hidden_dim),
            # TODO: 再次使用 ReLU 激活函数
            nn.ReLU(),
            # TODO: 使用线性层将第二个隐藏层映射到输出层，输出为图像的像素大小
            nn.Linear(hidden_dim, output_dim),
            # TODO:使用 Tanh 将输出归一化到 [-1, 1]，适用于图像生成
            nn.Tanh(),
        )

    def forward(self, x):
        # TODO: 前向传播：将输入 x 通过模型进行计算，得到生成的图像
        return self.model(x)
```

判别器

```
class Discriminator(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            # TODO: 输入层到第一个隐藏层，使用线性层
            nn.Linear(input_dim, hidden_dim),
            # TODO:  使用 LeakyReLU 激活函数，避免梯度消失问题，negative_slope参数设置为0.1
            nn.LeakyReLU(0.1),
            # TODO:  第一个隐藏层到第二个隐藏层，使用线性层
            nn.Linear(hidden_dim, hidden_dim),
            # TODO:   再次使用 LeakyReLU 激活函数，negative_slope参数设置为0.1
            nn.LeakyReLU(0.1),
            # TODO:  第二个隐藏层到输出层，使用线性层
            nn.Linear(hidden_dim, 1),
            # TODO:  使用 Sigmoid 激活函数，将输出范围限制在 [0, 1]
            nn.Sigmoid(),
        )

    def forward(self, x):
        # TODO:  前向传播：将输入 x 通过模型进行计算，得到判别结果
        return self.model(x)
```

（其实还可以加上`Kaiming`初始化）

随后是根据上面定义好的生成器与判别器创建实例，这里在创建时直接移动到设备上

```
G = Generator(input_dim, hidden_dim, output_dim).to(device)
D = Discriminator(output_dim, hidden_dim).to(device)
```

并且按要求选用`Adam`优化器与`0.0002`学习率

```
optim_G = torch.optim.Adam(G.parameters(), lr=0.0002)
optim_D = torch.optim.Adam(D.parameters(), lr=0.0002)
```

对于判别器的损失计算，首先将图像数据展平并使用判别器进行预测，随后使用`torch.ones`或`torch.zeros`生成全`1`（真）或全`0`（假）张量（真假样本设置不同是因为我们需要判别器在训练过程中调整参数以区分这两种样本，不然标签一样就没意义了），并用于计算即可，于是这里就有

```
real_labels = torch.ones(real_output.size(), device=device)
real_loss = loss_func(real_output, real_labels)
```

其中假样本是使用生成器生成的

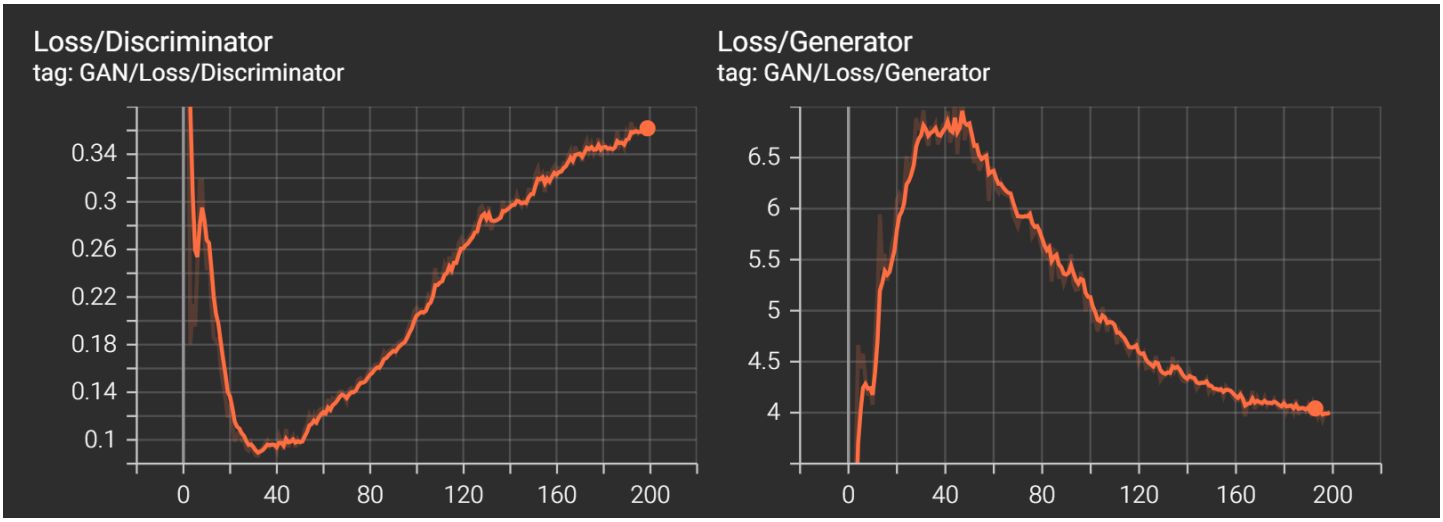
```
fake_labels = torch.zeros(fake_output.size(), device=device)
fake_loss = loss_func(fake_output, fake_labels)
```

那么对于生成器，基于我们希望生成的图片被认定为真的目的，在求损失时就应该生成全`1`张量

```
real_labels = torch.ones(fake_output.size(), device=device)
loss_G = loss_func(fake_output, real_labels)
```

3.3 训练过程损失曲线截图

由于代码中定义了`writer = SummaryWriter(log_dir="./logs/experiment_gan")`，因此可以通过`tensorboard --logdir=./logs/experiment_gan  --samples_per_plugin=images=1000`命令在指定端口中（默认`6006`）查看训练过程



3.4 不同训练 epoch 下生成的示例图像

可以看到，在一开始时生成的图像上基本全是噪点，经过几十轮的训练之后开始有了一些扭曲的数字，而在经过一百多轮后已经勉强可以分辨出数字的形状（即使依然不算太清晰，不过根据生成器与判别器的损失变化趋势看此时是没有收敛的）

3.5 思考题

3.5.1 GAN 与生成/判别器

`GAN`的基本原理正如其名，包含着“生成”与“对抗”，通俗来说就是一方（生成器）负责生成内容，另一方（判别器）负责分辨内容，两方需要进行对抗，最终目的是让生成器生成的内容能够骗过判别器，从而得到高质量生成样本

在每轮博弈中，生成器都会生成一批新的图片，而判别器会将其用于判断，两者互相给予反馈，最终生成器的生成能力越来越强、判别器的判别能力也越来越强——当然，我们最需要的实际上是生成器的生成能力

3.5.2 损失分析

对于两方损失，生成器的损失应该逐渐降低（说明生成的质量越来越好），判别器的损失应该逐渐增高（说明越来越难以区分真假样本），另外对于判别器，在本任务中最后的理想情况应该是在`0.5`左右波动

在实际训练中未必总能达到理想状态，一个原因可能是训练器与生成器的表现失衡，比如说如果判别器太强了、生成器会出现梯度消失问题——这意味着我们不能直接使用预训练模型进行微调，或者至少要让两者拥有相似的起点与模型结构。并且由于这是一个对抗过程，损失的震荡幅度相较于单一模型训练会更大一些

至于思考题中提到的“Mode Collapse”，指的是生成器仅生成有限种类的样本，多样性较低（或者说“泛化能力”不强，可以简单理解为陷入了能欺骗判别器的局部最优）

> 感觉可以拿预训练模型来试试，一个微调分类头，另一个微调“生成头”