学习报告（第一周）

25325186 刘金河

一、Introduction

学习了解了计算机视觉的发展历史以及深度学习和卷积神经网络的相关概念，掌握了这门课程作业的完成方式，进一步学习了使用 numpy 库进行矩阵运算。

二、Image Classification with Linear Classifiers

1、学习了机器学习中数据驱动的三个步骤：（如图 1 所示）

①收集带标签的图像数据集；

②使用机器学习算法训练分类器；

③在新图像上评估分类器。

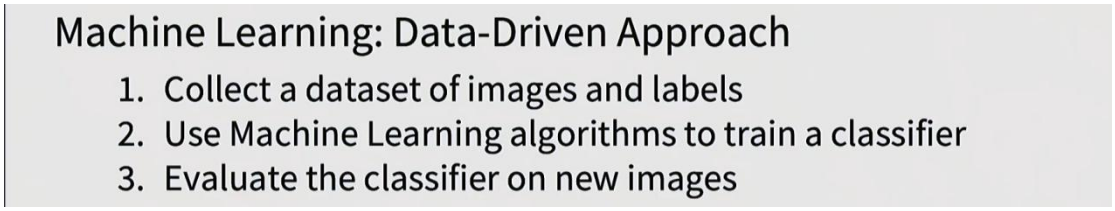

图 1 机器学习的步骤

2、学习了划分数据集（设置超参数）的策略以及交叉验证（**Cross-Validation**）的思想：

①设置超参数：

思路 1：选择在训练数据上表现最好的超参数（将所有数据用于训练）；

缺点：比如 K=1 在训练数据上总是表现完美（但泛化性差）。

思路 2：选择在测试数据上表现最好的超参数（将数据集划分为训练集和测试集）；

缺点：完全不知道算法在新数据上的表现。

思路 3：将数据拆分为训练集、验证集和测试集；

在验证集上选择超参数，在测试集上评估最终效果。

总结：思路 3 是最优的划分策略，可以有效避免训练的模型出现**过拟合**或者**欠拟合**的情况。

②交叉验证（**Cross-Validation**）

1）目的：重复利用数据，在有限样本下同时实现超参数选择和模型泛化能力评估。

2）关键步骤：

Ⅰ.数据拆分：将全部训练数据划分为 K 个子集（**Fold**）；

Ⅱ.循环训练与验证：

◦ 每次用 K-1 个 Fold 作为训练集训练模型，用剩下的 1 个 Fold 作为验证集评估模型的准确性；

◦ 重复 K 次，确保每个 Fold 都被用作一次验证集；

Ⅲ.结果聚合：将 K 次验证的准确率取平均值，作为模型在该超参数 K 下的 “泛化能力估计值”；

Ⅳ.超参数选择：对不同超参数 K，重复上述过程，选择平均验证性能最优

的 K；

Ⅴ.最终评估：用**最优超参数**在全部训练数据上**重新训练**模型，再用独立测试集**评估最终性能。**

3）弊端：仅适用于规模较小的数据集，当数据集较大时，所需的训练时间和计算开销剧增。

下图 2 是使用 Cross-Validation 进行训练得到的模型预测的准确度变化情况。

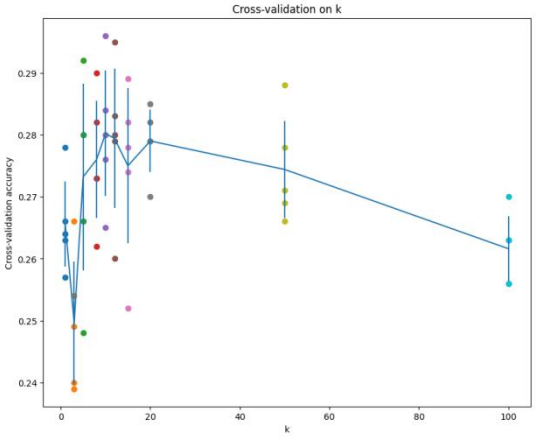

图 2 准确度与 K 的变化情况

3、学习了 **kNN（k-Nearest Neighbors）**算法：

①算法的核心思想：train 函数只负责储存图像数据，不做运算处理；predict 函数要将每一张图像数据与 train 中储存的数据进行一一比对，比对的方式常用 L1（曼哈顿）距离（$\boldsymbol{d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|}$）和 L2（欧氏）距离（$\boldsymbol{d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}}$），通过 K 值选择距离从小到大排序的前 K 个样例，再通过计算这些样例中出现最多的 label，进而判断输入的图像的类型。

②L1 距离与 L2 距离的主要区别：

L1 距离：旋转时距离会改变，通常用在图形特征非常具体且有意义，并且希望保留其信息的时候；

L2 距离：旋转时距离不会改变，通常用在图形特征比较不突出（casually）的时候。

③kNN 算法的实现方式：

1）Two loops

2）One loop

3）No loops

*三者的时间复杂度均为 $O(num\_test \times num\_train \times D)$ ，而运行时间的差异原因如下：

Two Loop：显式嵌套循环，Python 解释器的循环开销大，跑得最慢；

One Loop：利用 numpy 中的函数减少一层显式循环，开销变小，跑得更快；

No loops：完全无显式循环，numpy 底层用 C 语言实现并行计算，跑得最快。

*上述优化均通过将 for 循环替换成 numpy 库中的函数实现（No loops 为最优解，三种实现方式所需时间成本如图 3 所示）

```
Two loop version took 20.865758 seconds
One loop version took 41.265743 seconds
No loop version took 0.131032 seconds
```

图 3 不同的 kNN 算法的实现方式所需的时间成本

④kNN 算法的弊端：

kNN 算法的弊端亦是线性分类器（Linear Classifiers）的弊端，根本原因在于 L1 距离和 L2 距离只能在区分线性的特征数据时有效，在遇到非线性的数据分布时较难进行分类，具体如图 4 所示。

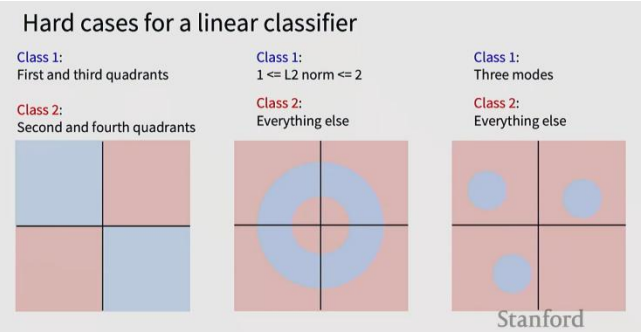

图 4 线性分类器难以解决的几种情况

4、学习了 Softmax 算法

①实现步骤：

对于输入样本 x_i（维度 D），权重矩阵 W（维度 $(D \times C)$，C 为类别数），分数计算为：$(s_i = x_i \cdot W = [s_{i1}, s_{i2}, \ldots, s_{iC}])$其中 (s_{ic}) 表示样本 (x_i) 属于第 c 类的原始分数。

1）数值处理（由于数值可能很大，为避免溢出，做此处理）

操作：对每个样本的分数减去该样本的最大分数：$(\hat{s}_{ic} = s_{ic} - \max(s_i))$

原理：指数函数的性质$(e^{a-b} = \frac{e^a}{e^b})$，减去最大值不改变概率分布（后续归一化会抵消），但能将分数压缩到 $((-\infty, 0])$ 区间，$(e^{\hat{s}_{ic}})$ 范围为 $((0,1])$，避免溢出。

2）计算指数分数：

对处理后的分数取指数，将分数映射到正实数域（概率必须为正）：$\exp(\hat{s}_{ic}) = e^{\hat{s}_{ic}}$

3）归一化（计算概率）：

将每个样本的指数分数除以所有类别指数分数之和，确保概率之和为 1，公式如下：$p_{ic} = \frac{\exp(\hat{s}_{ic})}{\sum_{k=1}^{C} \exp(\hat{s}_{ik})}$.

* p_{ic} 越大，模型越认为样本 x_i 属于第 c 类。

②损失函数（交叉熵损失）

在模型预测中，模型不可避免地会出现误判的情况，为了衡量一个模型的误判概率，亦或是评估一个模型的性能，在此引入损失函数，用于计算模型预测概率与真实标签的差距，进而通过梯度下降优化权重 W，从而提高模型的适用范围。

交叉熵损失：

Ⅰ、样本真实的标签：

对于样本 (x_i) 的真实标签 (y_i)（如 $(y_i = 2)$ 表示属于第 3 类），用指示向量 (I_i) 表示：$(I_i = [0,0,\dots,1,\dots,0])$，其中仅第 (y_i) 位为 1，其余为 0（如 $(y_i = 2)$ 时，$(I_i[2] = 1)$）。

Ⅱ、交叉熵损失公式：

单个样本 (x_i) 的损失为：$L_i = -\sum_{c=1}^{C} I_{ic} \cdot \log(p_{ic})$.

由于 I_{ic} 仅在 $c = y_i$ 时为 1，原公式可进一步简化为：$L_i = -\log(p_{i,y_i})$.

含义：真实类别的预测概率 (p_{i,y_i}) 越大，损失 L_i 越小。

极端情况：

若 $p_{i,y_i} = 1$（预测完全正确），损失 $L_i = 0$；

若 $p_{i,y_i} = 0$（预测完全错误），损失 $L_i = \infty$。

Ⅲ、交叉熵计算的优化：

为了减少训练时的损失计算的计算成本，在此使用计算批量平均损失的方式来计算交叉熵损失，同时引入 L2 正则化以避免模型出现过拟合的现象。

以下是最终使用的交叉熵损失计算公式：

$$L = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i}_{\text{批量平均交叉熵损失}} + \underbrace{\lambda \cdot \|W\|_2^2}_{L2\text{ 正则化项}}$$

三、Regularization and Optimization

1、进一步学习了正则化的相关知识：

在上一次课程中初步了解了正则化在交叉熵损失计算中的应用，在这次课中进一步了解到正则化的种类及其计算方式。

种类：（其中，λ是正则化强度，是一个人为设定的超参数）

①L1 正则化：

公式：$L_1 = \lambda \sum |W_{dc}|$

②L2 正则化：

公式：$L_2 = \lambda \cdot \sum_{d=1}^{D} \sum_{c=1}^{C} W_{dc}^2$

应用：

L1 和 L2 正则化均可应用于 Softmax 算法，得到的交叉熵损失有所不同，适用于不同的情况。

L1 的损失函数：$L = \frac{1}{N}\sum_{i=1}^{N}\left[-\log\left(\frac{\exp(s_{i,y_i} - \max(s_i))}{\sum_{k=1}^{C}\exp(s_{ik} - \max(s_i))}\right)\right] + \lambda \sum_{d=1}^{D}\sum_{c=1}^{C}|W_{dc}|$

L2 的损失函数：$L = \frac{1}{N}\sum_{i=1}^{N}\left[-\log\left(\frac{\exp(s_{i,y_i} - \max(s_i))}{\sum_{k=1}^{C}\exp(s_{ik} - \max(s_i))}\right)\right] + \lambda \sum_{d=1}^{D}\sum_{c=1}^{C} W_{dc}^2$

其中，L1 函数由于使用的是计算绝对值的加和，因此对数据的波动远没有 L2 函数计算平方那么大，因此 L2 函数容易被误差值干扰；然而，L1 函数在使用传统优化算法时，在接近最优解时容易出现震荡的情况，因此难以收敛到足够精确的数值，所以，在后续的优化过程中，我们的优化函数使用 L2 正则化。

2、学习了如何优化函数以找到最合适的权重矩阵 W

主要优化函数类型：

①SGD（Stochastic Gradient Descent，随机梯度下降）

核心代码如图 5 所示：

```
while True:
    data_batch = sample_training_data(data, batch_size)
    grad = compute_gradient(loss, data_batch, weights)
    weights -= learning_rate * grad
```

图 5 SGD 算法关键代码

SGD 算法搜寻最小损失的大致图像：

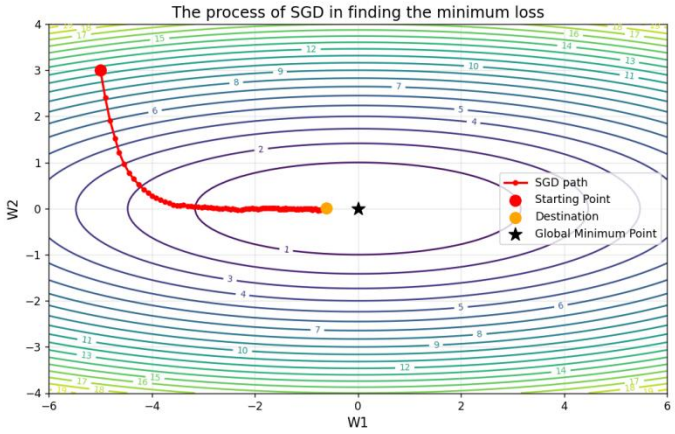

图 6 SGD 算法搜寻最小损失

缺陷：

1）收敛速度慢：由于每次更新只使用一个小批量的数据，梯度估计带有一定的随机性，导致参数更新路径震荡比较剧烈，需要很长时间才能收敛到最优解；

2）学习率难以选择：

Ⅰ、学习率太大：在接近最优解时，参数会在最小值附近来回震荡，无法稳定收敛；

Ⅱ、学习率太小：收敛速度会变得极其缓慢，并且在训练过程中容易陷入局部最小值或鞍点。

②SGD+Momentum（随机梯度下降+动量法）

由于 SGD 算法可能陷入某个鞍点，将其误判为最小值，因此在 SGD 算法原有的基础上，加上一个 Momentum，使得 SGD 算法处于鞍点时仍然能够继续搜寻更小梯度。

核心代码如图 7 所示：

```
vx = 0
while True:
    grad = compute_gradient(loss, data_batch, weights)
    vx = rho * vx + grad  # rho 通常为 0.9
    weights -= learning_rate * vx
```

图 7 SGD+Momentum 算法关键代码

SGD 算法搜寻最小损失的大致图像：

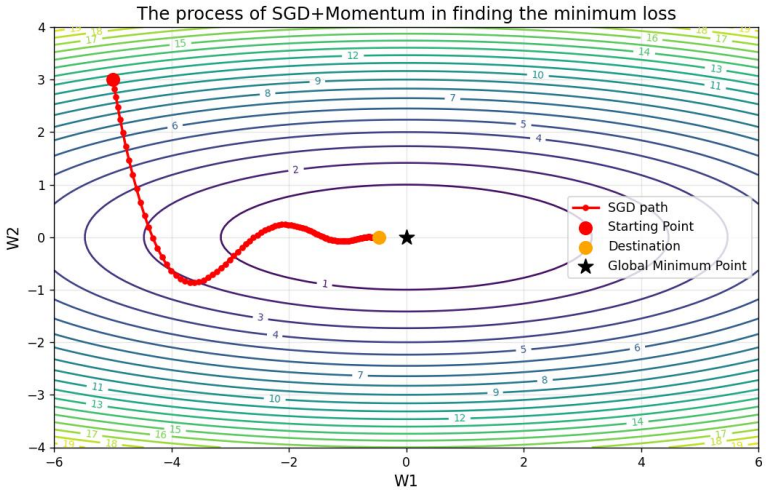

图 8 SGD+Momentum 算法搜寻最小损失

缺陷：从图 8 中不难看出，由于 Momentum 模拟物理中的惯性，使得参数更新具有 “冲量”，这种特性可能导致参数在接近收敛时 “冲过”最优点，让参数在最小值附近来回震荡。

③RMSProp（Root Mean Square Propagation，均方根传播优化器）

提出背景：

SGD+Momentum 未能解决 “学习率自适应” 问题：

Ⅰ、在稀疏数据场景下，部分参数仅在少数样本中出现梯度，需较大学习率才能更新；

Ⅱ、在稠密数据场景下，部分参数梯度持续较大，需较小学习率避免更新幅度过大；

因此引入梯度平方的指数加权平均，为每个参数自适应调整学习率。核心代码如图 9 所示：

```
grad_squared = 0
while True:
    grad = compute_gradient(loss, data_batch, weights)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * grad**2
    weights -= learning_rate * grad / (np.sqrt(grad_squared) + 1e-7)
```

图 9 RMSProp 算法关键代码

RMSProp 算法搜寻最小损失的大致图像：

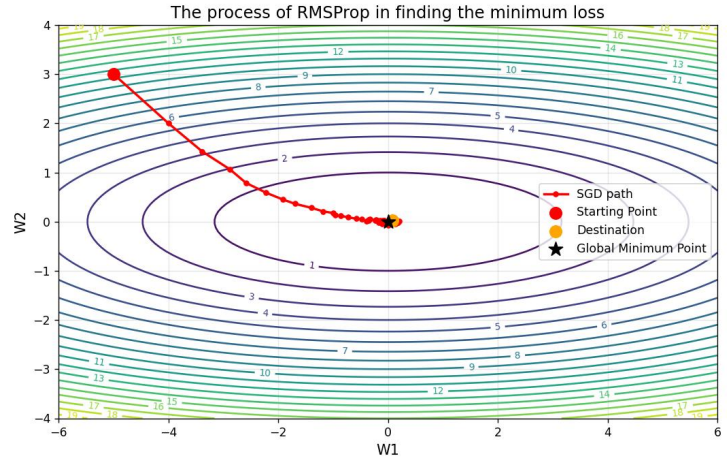

图 10 RMSProp 算法搜寻最小损失

缺陷：由于 RMSProp 仅优化了学习率自适应，未利用历史梯度的方向信息，收敛速度可能慢于 Momentum，且若某参数梯度极小，$(E[g^2]_t)$（即代码中的 np.sqrt(grad_squared)）会趋近于 0，导致学习率过大。

④Adam（Adaptive Moment Estimation，自适应矩估计优化器）

提出背景：

为了综合 SGD+Momentum 通过动量积累历史梯度方向，减少震荡以及 RMSProp 通过梯度平方累积，实现参数级学习率调整这两个优点，Diederik P. Kingma 和 Jimmy Ba 在 2014 年提出了 Adam 算法，将上面两种算法的优点进行结合。

核心代码如图 11 所示：

```
first_moment = 0  # Momentum
second_moment = 0 # RMSProp
beta1 = 0.9
beta2 = 0.999
epsilon = 1e-7

while True:
    grad = compute_gradient(loss, data_batch, weights)

    first_moment = beta1 * first_moment + (1 - beta1) * grad
    second_moment = beta2 * second_moment + (1 - beta2) * grad ** 2

    weights -= learning_rate * first_moment / (np.sqrt(second_moment) + epsilon)
```

图 11 Adam 算法关键代码

Adam 算法搜寻最小损失的大致图像：

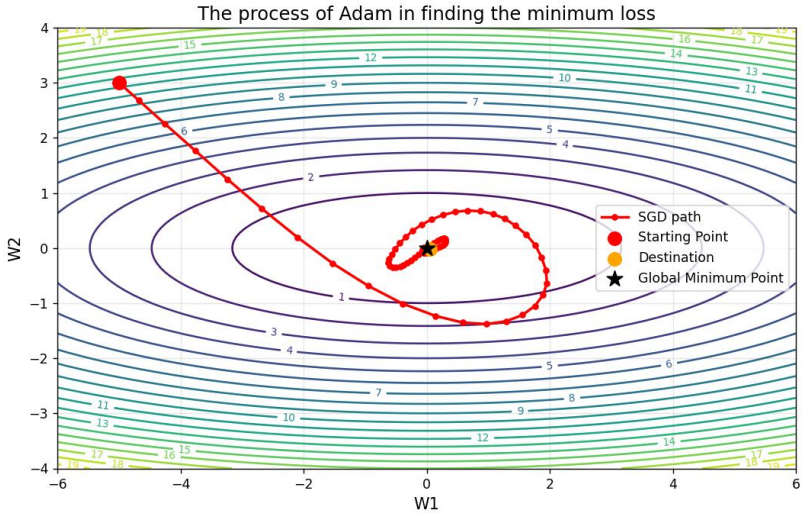

图 12 Adam 算法搜寻最小损失

缺陷：

Adam 的一阶矩（动量项，对应代码中的 first_moment）和二阶矩（自适应学习率项，对应对应代码中的 second_moment）初始值为 0。在迭代早期，由于β_1（如 0.9）和β_2（如 0.999）接近 1，β_1^t和β_2^t衰减缓慢，导致矩估计严重偏低。

改进方法：加入偏差修正项，使得训练早期的数据不会因为训练次数小而出现偏差衰减缓慢的情况，具体实现代码如下图 13 所示。

```
first_moment = 0
second_moment = 0
beta1=0.9
beta2=0.999
t = 0
while True:
    t += 1
    grad = compute_gradient(loss, data_batch, weights)

    first_moment = beta1 * first_moment + (1 - beta1) * grad
    second_moment = beta2 * second_moment + (1 - beta2) * grad**2

    # 偏差修正
    first_unbias = first_moment / (1 - beta1**t)
    second_unbias = second_moment / (1 - beta2**t)

    weights -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)
```

图 11 改进后的 Adam 算法关键代码

改进后的效果如图 12 所示。

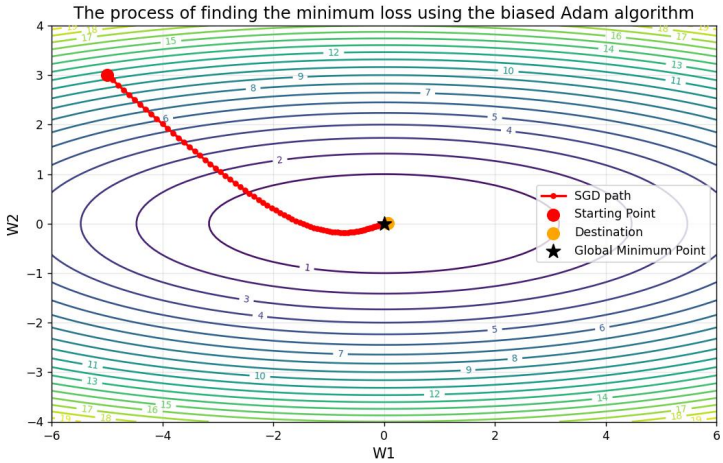

图 12 改进后的 Adam 算法搜寻最小损失

最后，我们把以上所有算法的实现过程都放到一张图片中，如图 13 所示，不难看出，带偏差修正的 Adam 算法是这里面所有算法的最优解。

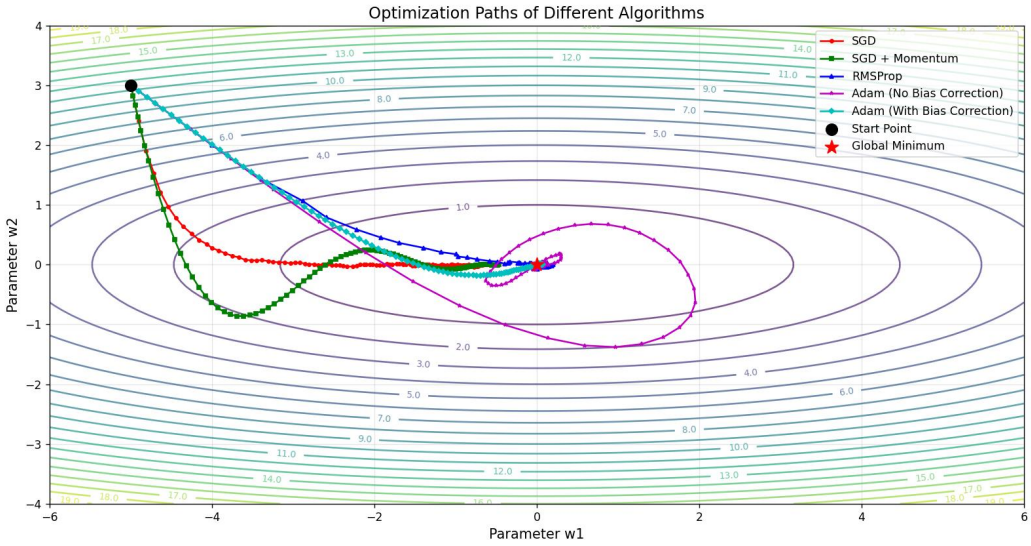

图 13 五种不同的算法搜寻最小损失

四、总结与体会

第一周我围绕着计算机视觉线性分类器及优化技术进行了学习，构建了从数据处理、算法原理到模型优化的知识框架，掌握了数据集划分、交叉验证、kNN/Softmax 算法及 SGD 到 Adam 的优化演进方法。

在学习中，我深刻体会到算法与数学的紧密关系，理论需结合工具实践（如 numpy），优化处理（如数值稳定、正则化）和与所提供的场景进行适配是选择算法的关键。此次学习为我夯实了计算机视觉的基础，为后续复杂模型的学习做铺垫。