实验四、多层感知机

学生姓名：卢豪豪　　学号：202310310239　　指导老师：胡政伟

一、实验目的

手动实现全连接神经网络，分析多层感知机在 MNIST 与 CIFAR-10 数据集上的表现差异，并验证 CNN 的结构优势。

二、实验内容

1、从‘零’实现神经网络

此部分大量参考了 CS231N-Assignment1 的内容，主要是对其中 placeholder 进行补全，并依据本实验需求对相关函数重新封装，形成独立的 Python 模块。下面对完成的主要的 placefolder 进行介绍。

（1）Affine层（全连接层）的前向与反向传播

Affine 层为神经网络基础构建块，实现 **`y = xW + b`** 的线性变换。

- 前向传播：将输入数据展平为二维矩阵 **`x_row = x.reshape(N, -1)`** 后与权重矩阵 w 相乘，再加 b，得到输出结果 **`out = x_row.dot(w) + b`** 。
- 反向传播：主要就是输入、权重和偏置三种梯度的计算实现：
  - 权重梯度： **`dw = x_row.T.dot(dout)`** ；
  - 偏置梯度： **`db = np.sum(dout, axis=0)`** ；
  - 输入梯度： **`dx = dout.dot(w.T).reshape(x.shape)`** 。

（2）BatchNorm、LayerNorm实现

- BatchNorm、LayerNorm 都对进行特征归一化来稳定数据分布，并学习两个参数 γ、β 进行缩放、平移操作，保留网络对特征分布的自主调控能力，平衡训练稳定性与模型的表达能力。
  - BatchNorm 目的：同一层、同一个神经元的激活分布在整个batch上更加稳定；
  - LayerNorm 目的：单个样本的不同特征（维度）分布更加稳定。

- 因为LayerNorm是针对单个样本，能够适应不同“句长”的输入，因此 Transformer 中 Encoder & Decoder 使用 LayerNorm(x + sublayer(x)) 作为每个 sublayer 的输出。
- BatchNorm 代码实现：
  - **batchnorm_forward** ：训练阶段基于当前 batch 计算均值与方差完成归一化，测试阶段用滑动平均的全局统计量保持分布一致；
  - **batchnorm_backward** ：对 Batch 归一化链式求导，梯度依次回传到方差、均值和原始输入，计算 γ、β 的梯度。

```python
def batchnorm_forward(x, gamma, beta, bn_param):
    # Basic hyperparameters
    eps = bn_param.get('eps', 1e-5)
    momentum = bn_param.get('momentum', 0.9)
    mode = bn_param['mode']

    N, D = x.shape
    running_mean = bn_param.get('running_mean', np.zeros(D, dtype=x.dtype))
    running_var = bn_param.get('running_var', np.zeros(D, dtype=x.dtype))

    if mode == 'train':
        mean = np.mean(x, axis=0)
        var = np.var(x, axis=0)

        # 归一化
        x_hat = (x - mean) / np.sqrt(var + eps)
        out = gamma * x_hat + beta  # Restore learnable scale/shift

        cache = (x, x_hat, gamma, mean, var, eps) # 暂存

        # 维护滑动平均
        running_mean = momentum * running_mean + (1 - momentum) * mean
        running_var = momentum * running_var + (1 - momentum) * var
    else:
        # 使用滑动平均
        x_hat = (x - running_mean) / np.sqrt(running_var + eps)
        out = gamma * x_hat + beta
        cache = None

    bn_param['running_mean'] = running_mean
    bn_param['running_var'] = running_var
    return out, cache


def batchnorm_backward(dout, cache):
    x, x_hat, gamma, mean, var, eps = cache
    N, D = x.shape
```

```
    # Gradient w.r.t. normalized input
    dxhat = dout * gamma

    dvar = np.sum(dxhat * (x - mean) * -0.5 * (var + eps) ** -1.5, axis=0)
    dmean = np.sum(dxhat * -1 / np.sqrt(var + eps), axis=0) + dvar * np.sum(-2 * (x -

    # Final gradient wrt input
    dx = dxhat / np.sqrt(var + eps) + dvar * 2 * (x - mean) / N + dmean / N

    # 参数梯度
    dgamma = np.sum(dout * x_hat, axis=0)
    dbeta = np.sum(dout, axis=0)

    return dx, dgamma, dbeta
```

- LayerNorm 代码实现：
  - **layernorm_forward** ：在每个样本内部的特征维度上计算均值和方差，对激活归一化后再用 γ、β 重构可学习分布；
  - **layernorm_backward** ：按特征维度对其均值和方差反向求导，梯度回传给每个样本，并求得 γ、β 的梯度。

```python
def layernorm_forward(x, gamma, beta, ln_param):
    eps = ln_param.get('eps',1e-5)

    # 每个样本做特征归一化
    mean = np.mean(x, axis=1, keepdims=True)      # 行均值
    var = np.var(x, axis=1, keepdims=True)        # 行方差

    x_hat = (x - mean) / np.sqrt(var + eps)       # 标准化
    out = gamma * x_hat + beta                    # 缩放和平移

    cache = (x, x_hat, gamma, mean, var, eps)
    return out, cache


def layernorm_backward(dout, cache):
    x, x_hat, gamma, mean, var, eps = cache
    N, D = x.shape

    dxhat = dout * gamma
    dvar = np.sum(dxhat * (x-mean) * -0.5 * (var+eps)**-1.5,
                  axis=1, keepdims=True)          # 方差梯度
    dmean = (np.sum(dxhat * -1/np.sqrt(var+eps), axis=1, keepdims=True)
             + dvar * np.sum(-2*(x-mean), axis=1, keepdims=True) / D)  # 对均值的梯度

    dx = (dxhat / np.sqrt(var+eps)  + dvar * 2*(x-mean)/D + dmean / D)

    dgamma = np.sum(dout * x_hat, axis=0)         # gamma 梯度
    dbeta = np.sum(dout, axis=0)                  # beta 梯度

    return dx, dgamma, dbeta
```

- 我认为有必要特意说明的一点是：BatchNorm 相比 LayerNorm ，BatchNorm 需要额外维护滑动平均的全局均值 μ_running 和 全局方差 σ^2_running！
  - BatchNorm：推理时可以看作 **batch = 1** 的情形，如果直接利用单个样本的均值、方差，输入分布会极其不稳定，因此需要借助 滑动平均 （训练数据整体“记忆分布”）；

- LayerNorm：针对单个 sample ，自然与其它 sample 无关，无需！

（3）损失函数实现

实现了 **softmax_loss** ，主要思路就是：减去最大值避免指数运算溢出，将“得分”转换为概率，用交叉熵计算平均损失，同时对输入得分求梯度进行反向传播。代码如下：

```
def softmax_loss(x, y):
    # 稳定处理
    shifted_logits = x - np.max(x, axis=1, keepdims=True)
    # 计算每行的归一化因子 Z
    Z = np.sum(np.exp(shifted_logits), axis=1, keepdims=True)
    # log-softmax
    log_probs = shifted_logits - np.log(Z)
    probs = np.exp(log_probs)
    N = x.shape[0]
    # 交叉熵损失
    loss = -np.sum(log_probs[np.arange(N), y])/N
    # 对输入求梯度
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N
    return loss, dx
```

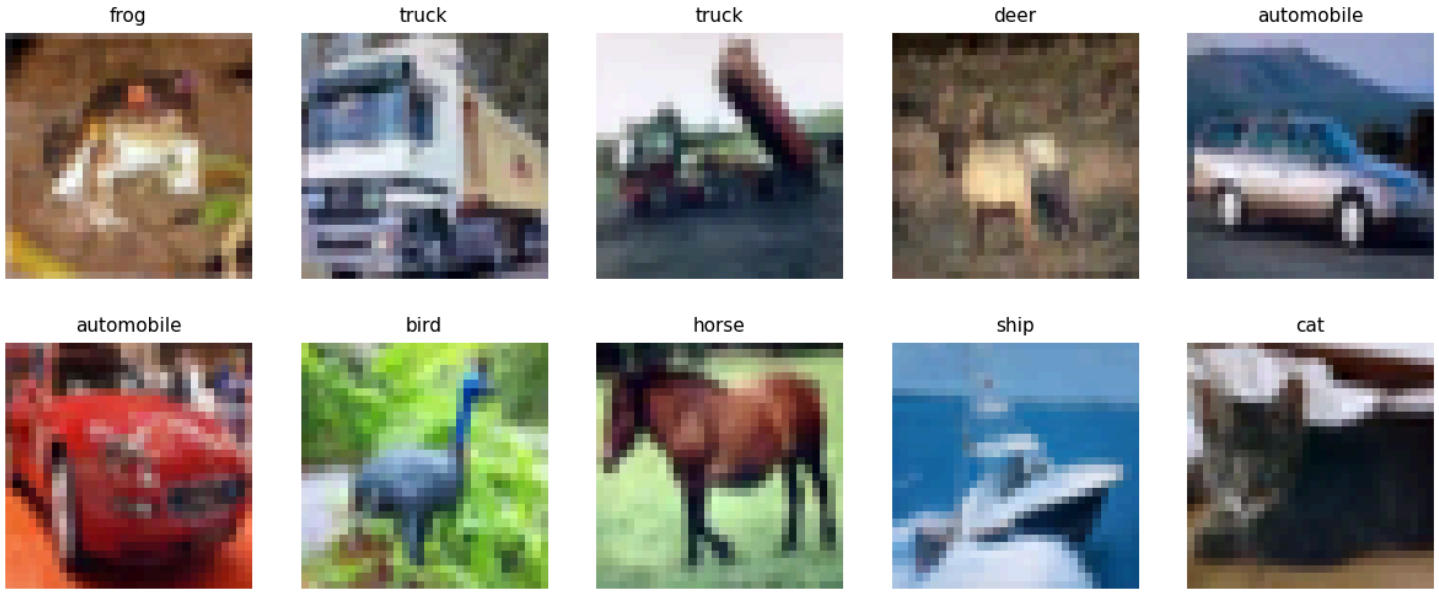
（4）Else

SGD 、SGD with Momentum 、RMSprop 、Adam 四种优化器实现，全连接层 **class FullyConnectedNet()** 、训练框架 **class Solver()** 的补全，由于内容较为繁杂，这里不再阐述，实现详见代码文件。

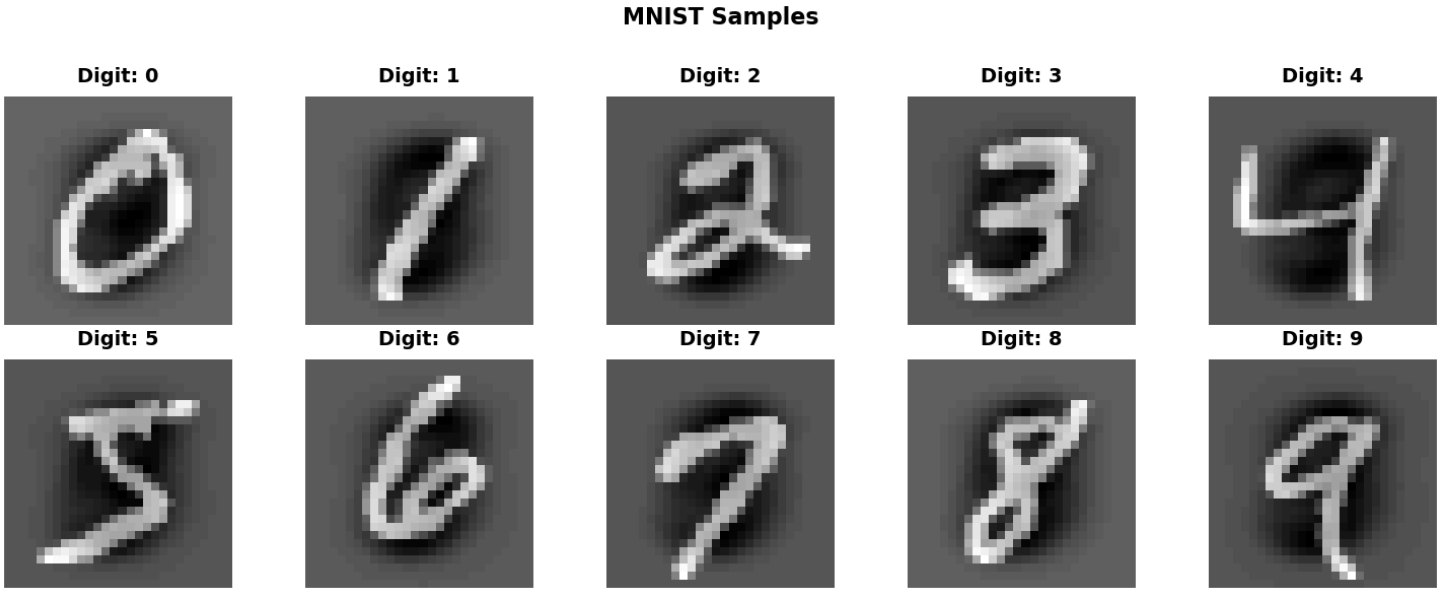
2、数据集

本实验用到了 MNIST 数据集（baseline）、CIFAR-10 两种数据集，利用 **get_datasets.sh** 脚本获取数据集，划分情况及数据集详情如下所示：

CIFAR-10 Dataset	Sample Count	Shape (C, H, W)	Label Shape
Training	45,000	(3, 32, 32)	(49000,)
Validation	5,000	(3, 32, 32)	(1000,)
Test	1,0000	(3, 32, 32)	(1000,)



MNIST Dataset	Sample Count	Shape (C, H, W)	Label Shape
Training	55,000	(1, 28, 28)	(55000,)
Validation	5,000	(1, 28, 28)	(5000,)
Test	10,000	(1, 28, 28)	(10000,)



3、梯度检查

这里用梯度检查来验证手动反向传播的正确性。

- 利用如下 中心差分公式 计算函数 $f(x)$ 在点 x 处梯度，不过其是一种近似思想，存在一定的误差。

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x_1, \ldots, \mathbf{x_i} + \mathbf{h}, \ldots, x_n) - f(x_1, \ldots, \mathbf{x_i} - \mathbf{h}, \ldots, x_n)}{2h}$$

- 定义函数 **`eval_numerical_gradient`**，用于对标量函数 $f : \mathbb{R}^n \to \mathbb{R}$ 在指定点 $x \in \mathbb{R}^n$ 进行数值梯度评估， h 为增量幅值。

```
def eval_numerical_gradient(f, x, verbose=True, h=0.00001):

    fx = f(x)  # evaluate function value at original point
    grad = np.zeros_like(x)
    # iterate over all indexes in x
    it = np.nditer(x, flags=["multi_index"], op_flags=["readwrite"])
    while not it.finished:

        # evaluate function at x+h
        ix = it.multi_index
        oldval = x[ix]
        x[ix] = oldval + h  # increment by h
        fxph = f(x)  # evalute f(x + h)
        x[ix] = oldval - h
        fxmh = f(x)  # evaluate f(x - h)
        x[ix] = oldval  # restore

        # compute the partial derivative with centered formula
        grad[ix] = (fxph - fxmh) / (2 * h)  # the slope
        if verbose:
            print(ix, grad[ix])
        it.iternext()  # step to next dimension

    return grad
```

- 在小规模随机数据下输出初始化网络在 $regulation = 0$、$regulation = 3.14$ 两种情况下解析梯度与中心差分梯度之间的相对误差，结果如下：

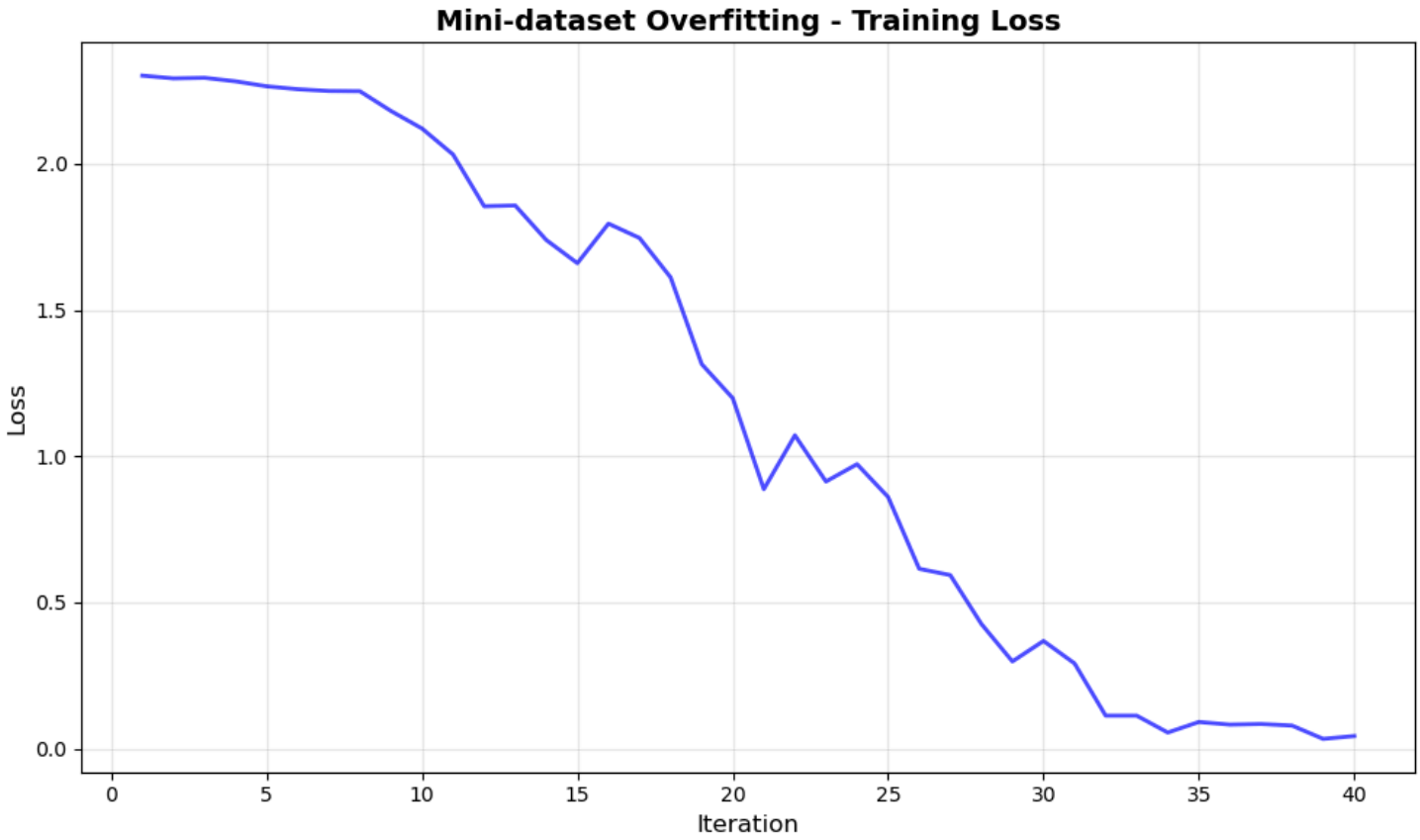
Parameter	reg = 0 (rel_error)	reg = 3.14 (rel_error)
W1	1.48e-7	1.14e-8
W2	2.21e-5	6.87e-8
W3	3.53e-7	3.48e-8
b1	5.38e-9	1.48e-8
b2	2.09e-9	1.72e-9

Parameter	reg = 0 (rel_error)	reg = 3.14 (rel_error)
b3	5.80e-11	1.80e-10
Initial Loss	2.3005	7.0521
Mean rel_error	3.71e-6	1.41e-8

- 结合表格可知：
  - relative error 都非常小：
    大部分参数的误差在 (10^{-7} \sim 10^{-9}) 量级，检验了正确性；
  - 对于正则化系数（ **reg=0** 和 **reg=3.14** ），随着正则化增强，初始损失增加（2.3 -> 7.05），相对误差整体变得更小，也验证了正则化提升了梯度计算数值稳定性。

4、 **3** 层网络训练 VS **5** 层网络训练

作为另一种合理性检查，就是确保构建的网络能够在50张图像的小数据集上过拟合，CS231N-Assignment-1-q5-Traning-a-fcnet 中分别选择 3 层、5 层网络（每层100神经元）进行过拟合训练，通过调整 **learning_rate** 、 **weight_scale** 两个超参数使得模型快速过拟合，以下为示例图像：



通过手动调整这两个超参数，体会不同层数神经网络训练特点，这里我选择对原 assignment 中的问题进行作答：

> ### *InlineQuestion* 1:
>
> Did you notice anything about the comparative difficulty of training the three-layer network vs. training the five-layer network? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

> ### *MyAnswer*:
>
> - 训练 5 层网络明显难于 3 层网络，主要表现为 5 层网络受到 **`weight_scale`** 尺度的影响很大，很容易造成梯度爆炸、梯度消失，具体表现为 loss 下降非常缓慢、loss 剧烈震荡。
> - 深层网络对初始化尺度更敏感的本质原因是深度带来的累积效应——无论是前向传播的信号衰减/爆炸,还是反向传播的梯度消失/爆炸,都会随着层数增加而加剧。这也是为什么现代深度学习中发展出了Batch Normalization、Layer Normalization、残差连接等技术来缓解这个问题。
>
> 参考-vanishing-and-exploding-gradients-problem

5、优化策略

- SGD（随机梯度下降）：最基础，直接利用学习率 * 梯度来更新权重。
- SGD with Momentum（带动量的SGD)：引入速度变量累积历史梯度信息，使参数更新具有惯性（我认为其是从物理学中的动量受到了启发，表现为当前梯度方向与历史方向一致时会加速，方向冲突时会减速缓冲），能够加速收敛并减少震荡，更容易越过鞍点！二者实现代码如下所示：

```
def sgd(w, dw, config=None):
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-2)

    w -= config["learning_rate"] * dw # 更新参数
    return w, config

def sgd_momentum(w, dw, config=None):
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-2)
    config.setdefault("momentum", 0.9)

    v = config.get("velocity", np.zeros_like(w))

    # Momentum update rule
    v = config["momentum"] * v - config["learning_rate"] * dw
    next_w = w + v

    config["velocity"] = v

    return next_w, config
```

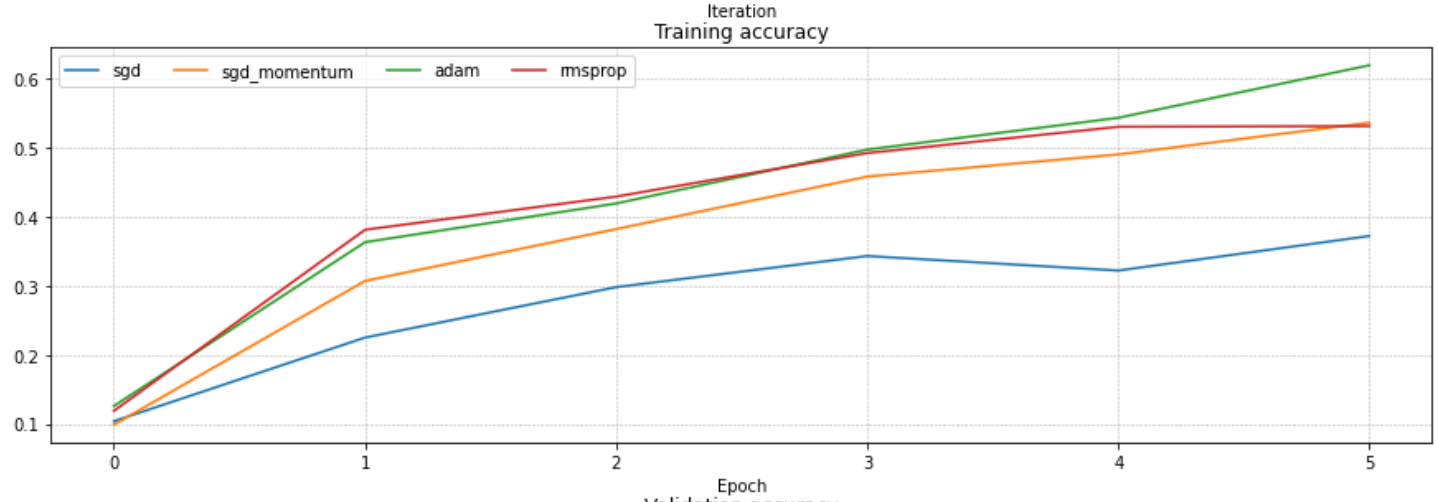
- RMSProp：主要就是为每个参数维护一个“梯度平方的指数滑动平均值”，这个平均值越大就对其应用更小的更新幅度，核心为

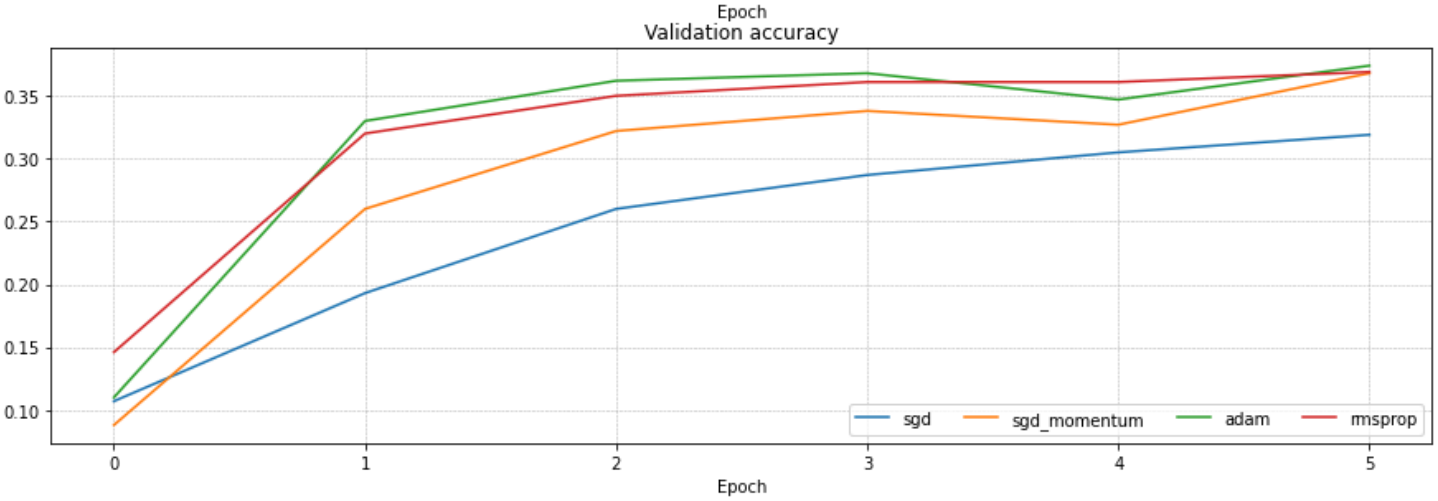
  `next_w = w - learning_rate * dw / (sqrt(cache) + epsilon)` ；

- Adam：它同时维护梯度的一阶矩和二阶矩的滑动平均，并进行校准，这里的一阶距、二阶距其实就是 **SGD with Momentun** 、 **RMSProp** 的思想，我将其看作二者的折中方案😀。二者实现代码如下：

```
def rmsprop(w, dw, config=None):
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-2)
    config.setdefault("decay_rate", 0.99)
    config.setdefault("epsilon", 1e-8)
    config.setdefault("cache", np.zeros_like(w)) #避免除零的作用

    # 更新滑动二阶距
    config["cache"] = config["decay_rate"] * config["cache"] + (1 - config["decay_rat

    next_w = w - config["learning_rate"] * dw / (np.sqrt(config["cache"]) + config["e

    return next_w, config

def adam(w, dw, config=None):
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-3)
    config.setdefault("beta1", 0.9)
    config.setdefault("beta2", 0.999)
    config.setdefault("epsilon", 1e-8)
    config.setdefault("m", np.zeros_like(w))
    config.setdefault("v", np.zeros_like(w))
    config.setdefault("t", 0)
    config["t"] += 1

    # 一阶距 & 校正
    m_hat = config["m"] / (1 - config["beta1"] ** config["t"])

    # 二阶距 & 矫正
    v_hat = config["v"] / (1 - config["beta2"] ** config["t"])

    # 更新滑动average
    config["m"] = config["beta1"] * config["m"] + (1 - config["beta1"]) * dw
    config["v"] = config["beta2"] * config["v"] + (1 - config["beta2"]) * dw**2

    next_w = w - config["learning_rate"] * m_hat / (np.sqrt(v_hat) + config["epsilon"
```

```
    return next_w, config
```





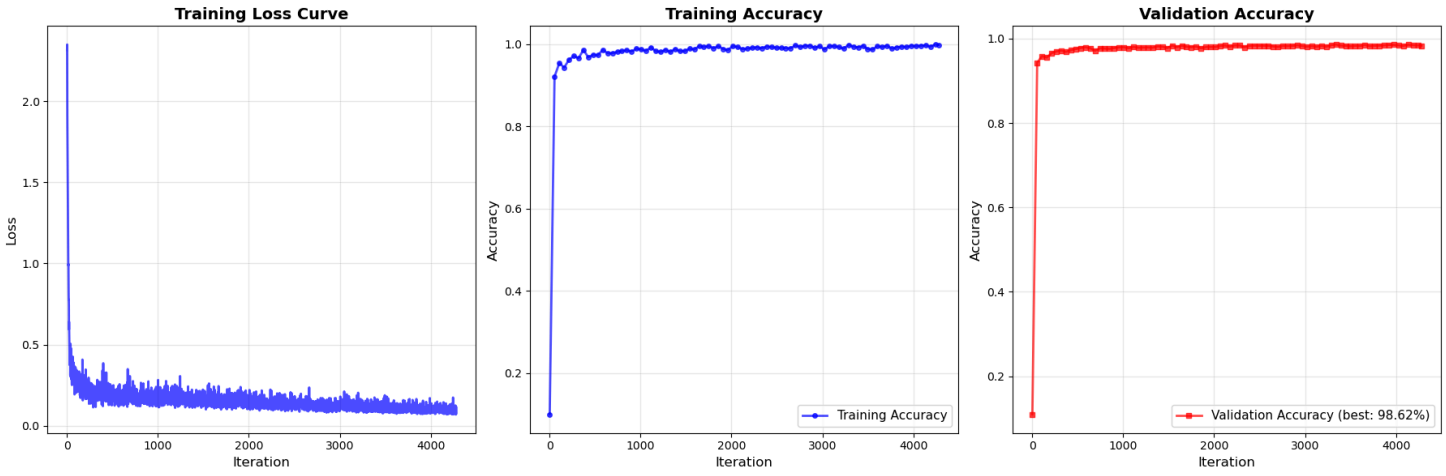
- 结合图像不难看出：
  - Adam Loss下降最快最稳定，准确率最高的也是它！
  - SGD 三个指标上都是最差的💔！
  - 两种自适应优化器（Adam、RMSProp）收敛最快。

6、MLP Baseline on MNIST

- 这部分主要是作为基准实验，验证 MLP 在MNIST数据集上的能力。
- 设置 MLP 架构为如下：

Layer	Layer Name	Neurons	Activation	Input Dim	Output Dim	Parameters
0	Input Layer	784	-	(28×28)	784	0
1	Hidden Layer 1	256	ReLU	784	256	200,960
2	Hidden Layer 2	128	ReLU	256	128	32,896
3	Hidden Layer 3	64	ReLU	128	64	8,256
4	Output Layer	10	Softmax	64	10	650
Total						242,762

- 训练设置就不再进行阐述，值得说明的一点就是我先在小规模数据集（SAMPLES = 1000）上进行超参数搜索，将最佳验证参数（包括 **lr**、**reg**、**ws**、**kp** 这四个超参数）直接应用于其在完整数据集上的训练，简化调参。训练完整模型过程如下所示：



- 可知，在 MNIST 数据集上，该 MLP 模型能够迅速收敛并达到较高的验证准确率。通过 Adam 优化器、Batch Normalization 和 Dropout 技术的组合使用，模型最终在测试集上实现了超过 98.05% 的分类准确率，表明了对于 MNIST 这类特征简单、模式清晰的灰度图像分类任务，单纯简单的 MLP 架构已能够取得很好性能。

7、 MLP vs CNN on CIFAR-10

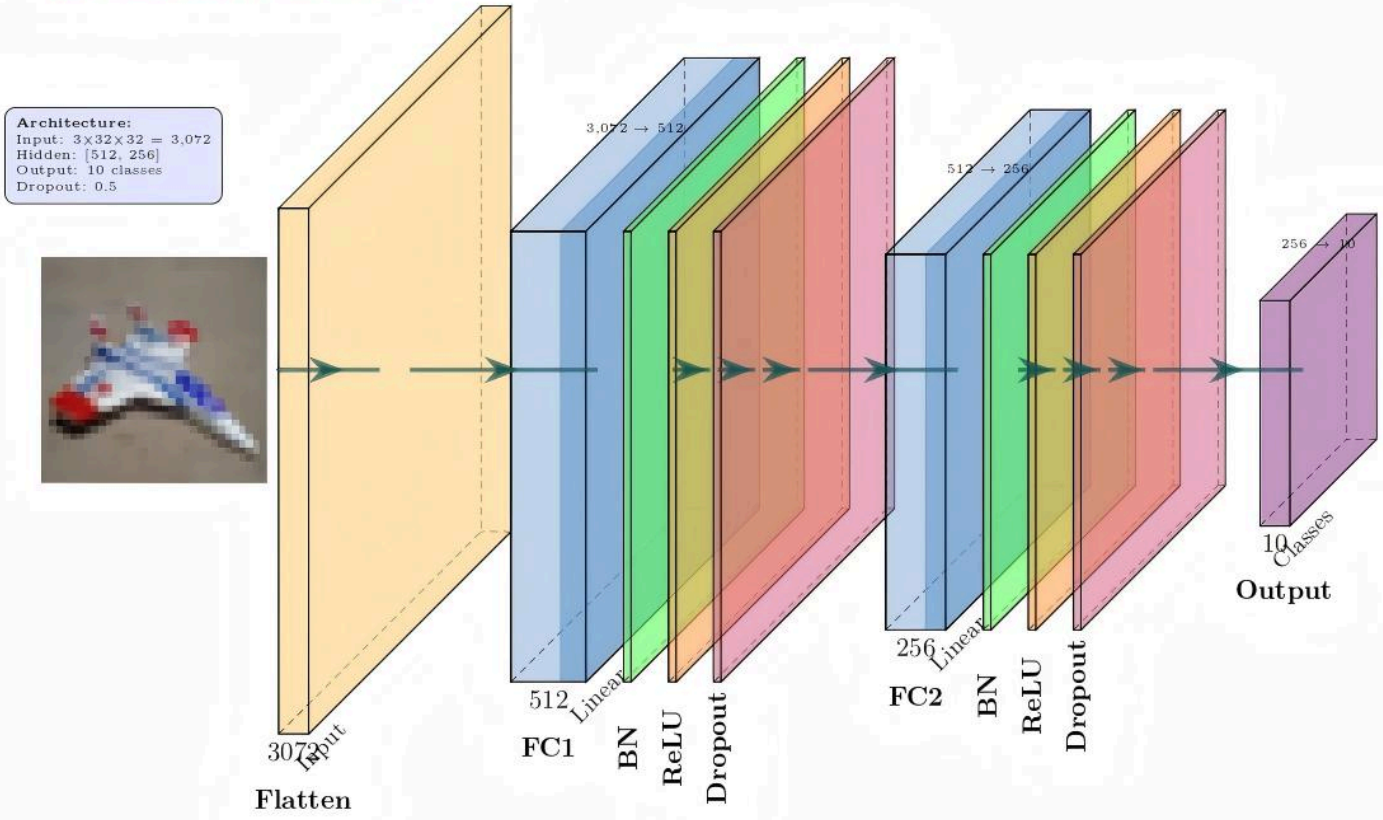
在 3-layer vs 5-layer network training 这一部分我注意到：虽然是过拟合训练（CIFAR mini测试集合准确率 100%），但是 MLP 在验证集上的准确率只有约为 14% 左右！为了探究这原因异即为了比较 MLP 与 CNN 在相对复杂（相比 MNIST 数据集）分类任务上的差异，设计了此部分实验。

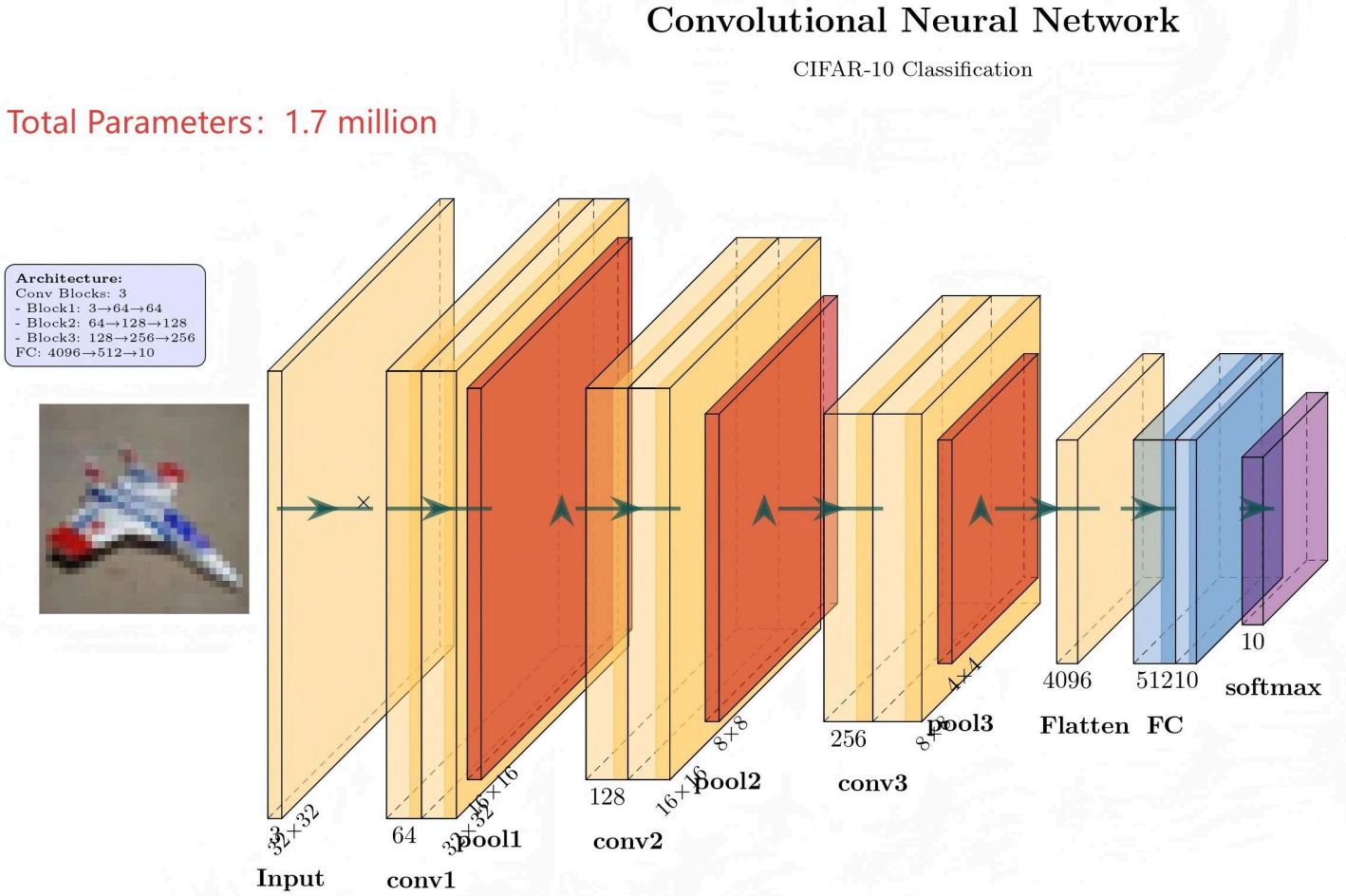
- 因为数据规模较大，为了使用GPU加速训练，这部分我选择利用 PyTorch 实现。
- 在设计 CNN 架构时，我参考了 VGG 架构中常用的一点：$KernelSize = 3$ & $Stride = 1$ ，该 CNN 架构所用的卷积全部使用了这样的设定。我想为什么更偏向于使用 small kernel size 主要有以下原因：
  - 每层卷积感受野范围 +2 pixel，每个点会“看”到其上下左右方位 8个邻居 ，更充分捕捉空间特征；
  - 感受野逐层累积较慢，需要连续堆叠卷积层，但同时带来关键好处：更好的非线性表达能力！
  - 更少的参数（我认为这个最次要）。
- 两种网络架构详见下图：

Fully Connected Neural Network

CIFAR-10 Classification

Total Parameters: 3.9 Million





- 训练策略方面，值得说明的一点是，在权重初始化时，BatchNorm参数、线性层权重采取了和手动实现时一样的策略（γ 初始化为 1、β 初始化为 0； 高斯初始化-`np.random.randn() * weight_scale`），而卷积核采用了He 初始化（`weights = np.random.randn(*shape) * np.sqrt(2 / fan_in)`）。He 初始化解决了 Xavier 初始化由于 ReLU 输出非对称、一般神经元输出为0（正、反向方差均减半）带来的问题，核心解决思想就是扩大权重方差来补偿这个方差损失。

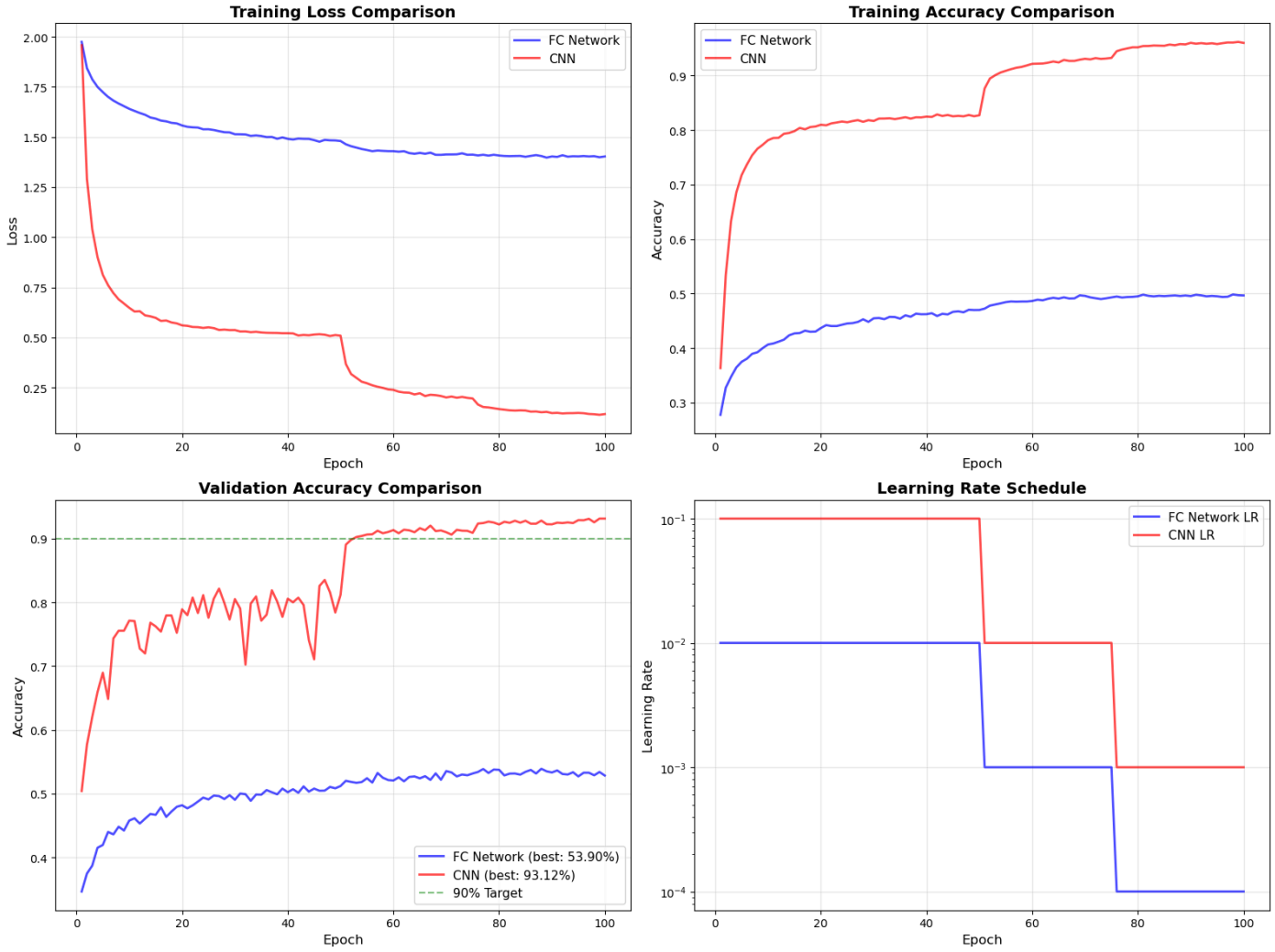
```
def _initialize_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d): # He 初始化
            nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            if m.bias is not None:
                nn.init.constant_(m.bias, 0)

        elif isinstance(m, nn.BatchNorm2d) or isinstance(m, nn.BatchNorm1d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)

        elif isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, 0, 0.01) # 对应weight_scale = 0.01
            nn.init.constant_(m.bias, 0)
```

- 关于为什么卷积核一般不采用简单的高斯初始化，我查阅到如下信息：

> As He et al. note, prior to the publication of their method, network weights were initialized using a Gaussian distribution with a fixed variance. With this approach, “deep” networks (networks with >8 layers) had difficulty converging. He et al. noted that to avoid the vanishing/exploding gradient problem, the standard deviation of the weights must be a function of the filter dimensions and provided a theoretically sound mathematical framework (based on earlier work by Bengio and others)
> 参考-Initializing Weights for the Convolutional and Fully Connected Layers

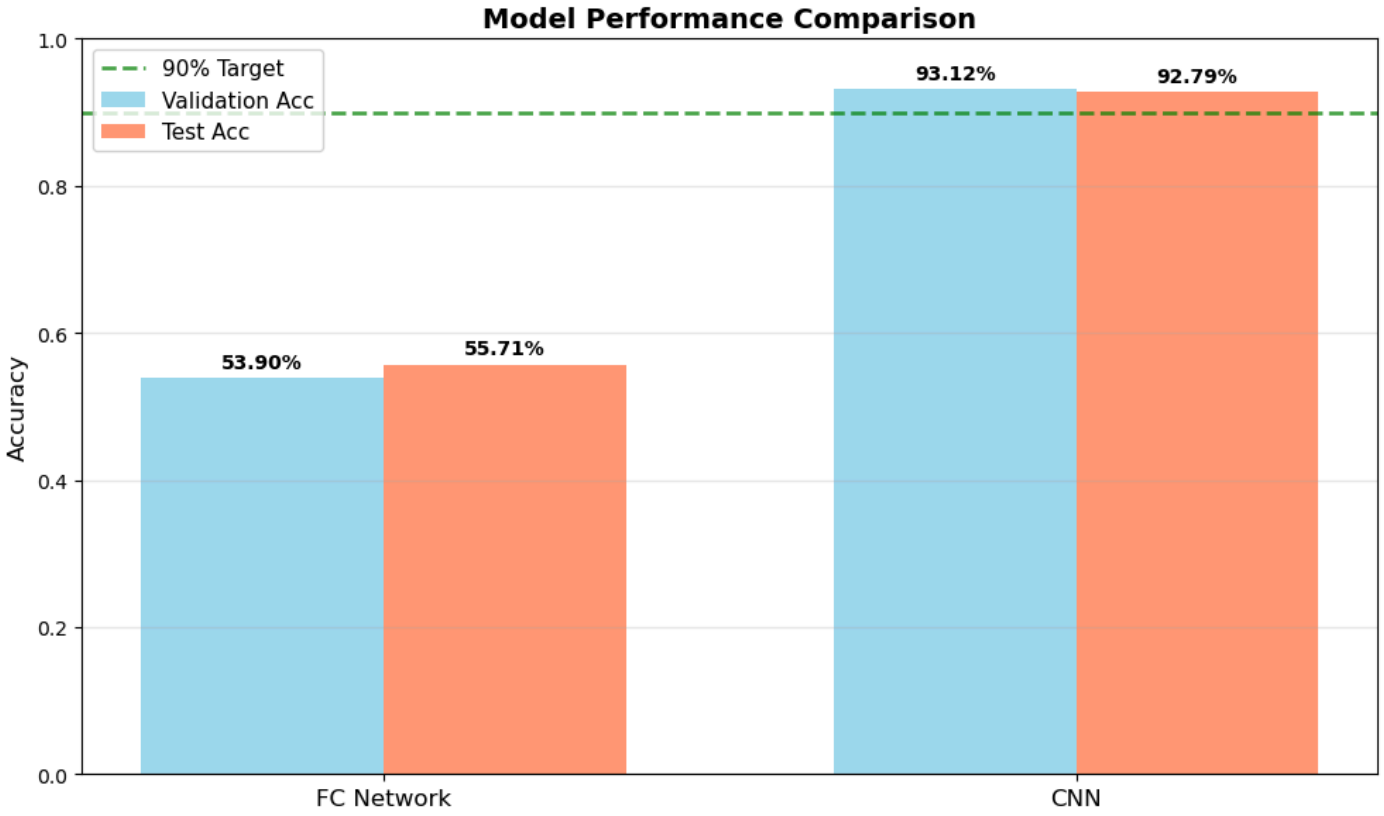
- 简而言之就是为了避免梯度消失或爆炸问题，权重的标准差必须是滤波器维度的函数！我简单理解为可能与卷积核对 **weight_scale** 敏感有关，数学论证则尚未探究。
- 学习率调度策略采用的为多步衰减（50、75 epoch 时衰减），有关训练策略其他部分不在进行阐述，二者运用了一样的训练策略。以下为可视化评估：

$Training Procedure$



$Performance Comparison$

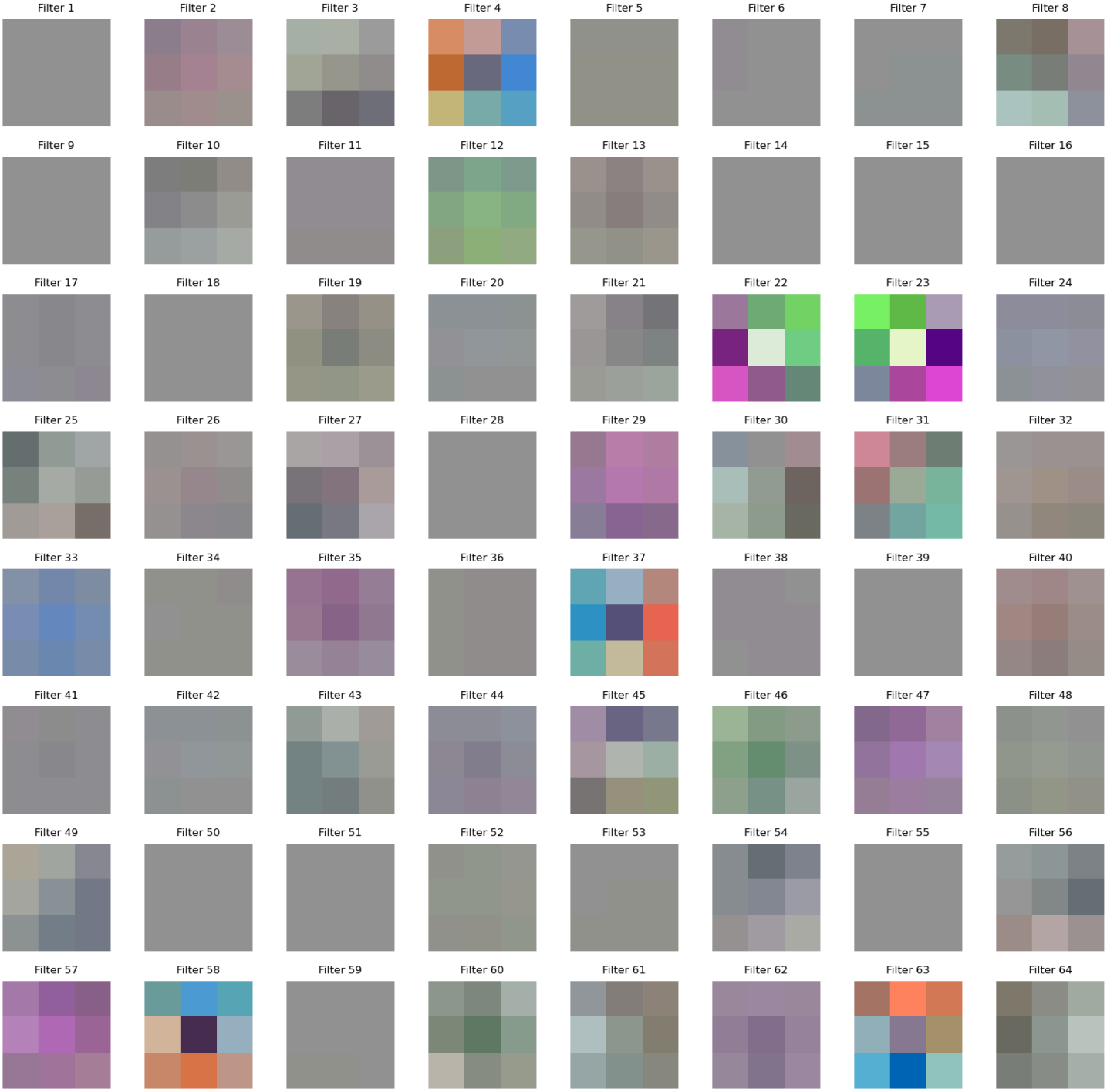
TrainingProcedure

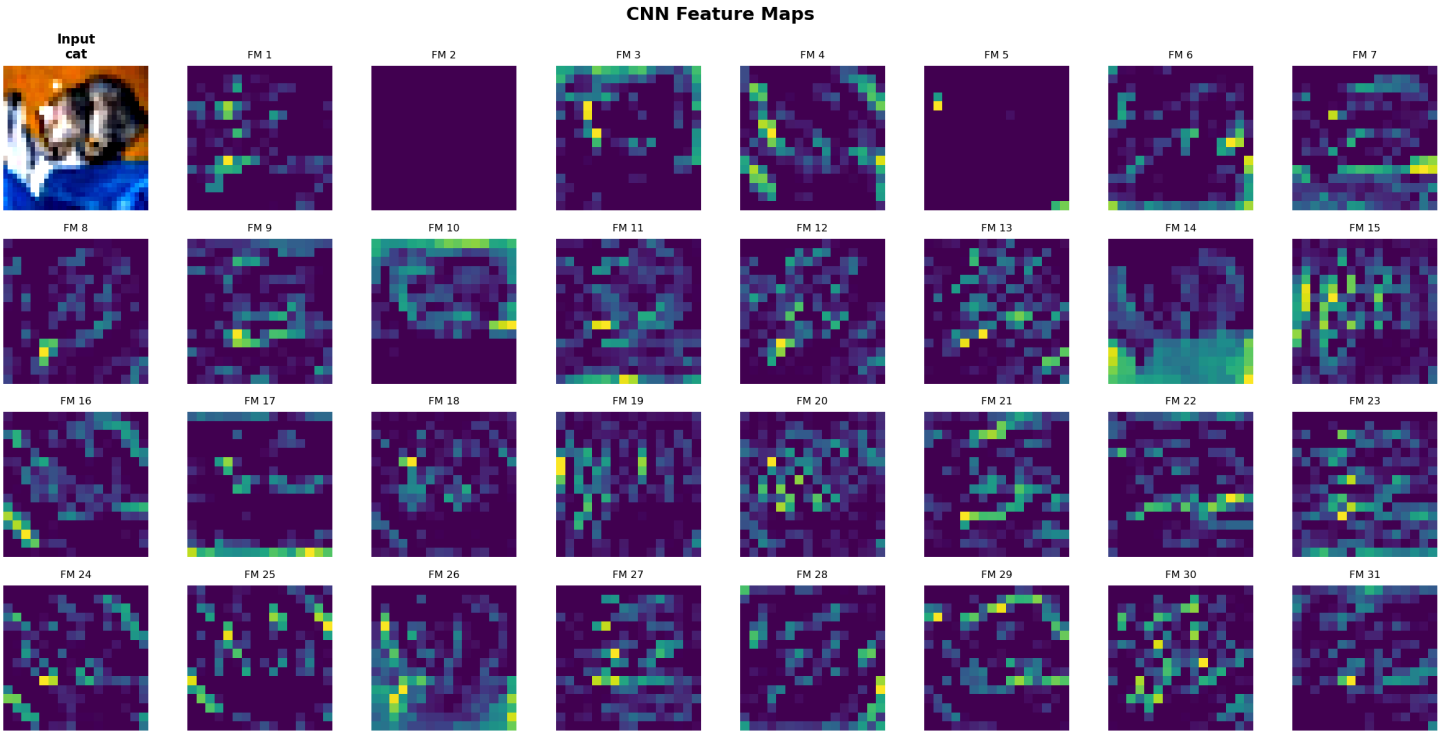


- 似乎 MLP 在 CIFAR-10 真实表现上限也就在 54% 左右，增加层数、更换优化策略仍然不能突破这一瓶颈；
- CNN 性能则远胜于 MLP ，在测试集上达到的 92.79% 的准确率；
- 训练过程中值得关注的一点就是在两次学习率衰减

  epoch = 50 ： lr 0.1 -> 0.01 & epoch = 75 ： lr 0.01 -> 0.001 后准确率快速提升（第一次： **80.09% -> 89.78%** ） ，说明这时切换更小的学习率在“低谷”附近实现了很好的优化效果！
- 为了进一步了解 CNN 特性，可视化了训练后 CNN 第一卷积层的 64 个 3×3 卷积核及其（部分）应的特征图响应，详情如下：

CNN first layer learned filters





- 结合图像不难看出，一些卷积核（F4、F22、F23等等）学习到了边缘检测器，一些卷积核（F29、F33等等）学习到了颜色选择器（对特定颜色通道敏感），等等这些其实都是低级特征；
- 不同特征图展现了捕捉图像的不同特征（边缘、颜色等）！
- 不过某些卷积核（F2等等）似乎什么都没有学到（难道是因为图像尺寸小而卷积核数量很多？）

8、MLP vs CNN on Gray-CIFAR-10

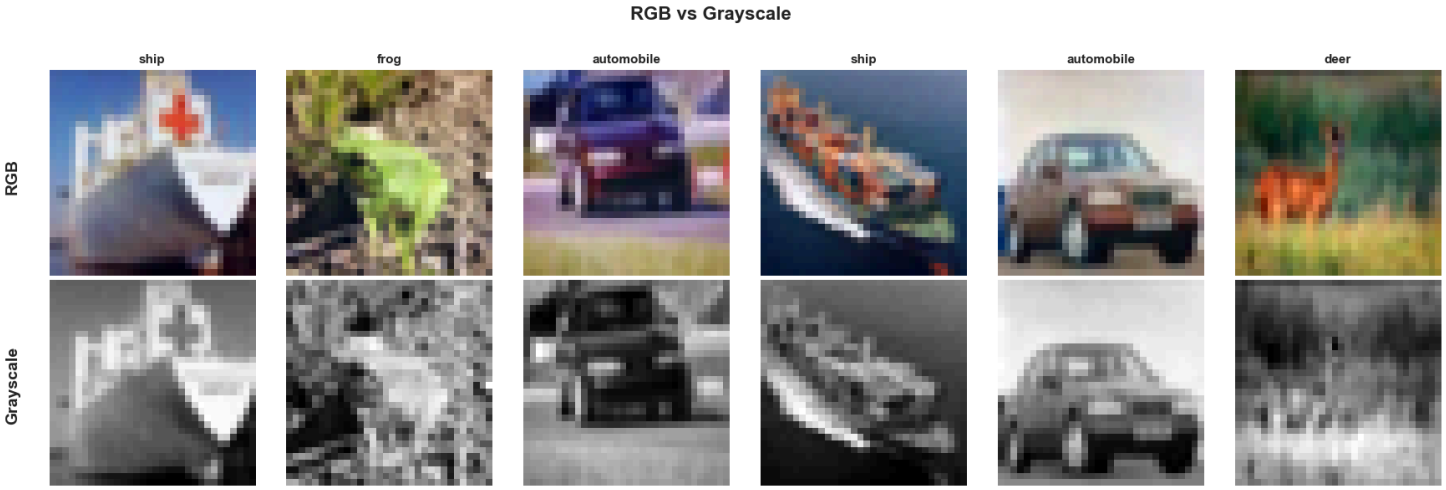
MNIST 为灰度图像，图片尺寸为 28 × 28；CIFAR-10 为 RGB 图像，图像尺寸为 32 × 32。二者图像尺寸差别不大，但是 MLP 的性能差异确实天差地别。为此我设置了 RGB → Grayscale 的消融实验，主要有如下目的：

1、量化颜色信息对模型性能的影响；

2、比较模型在彩色与灰度图像上的表现，验证卷积结构提取的空间特征给模型性能带来的优势；

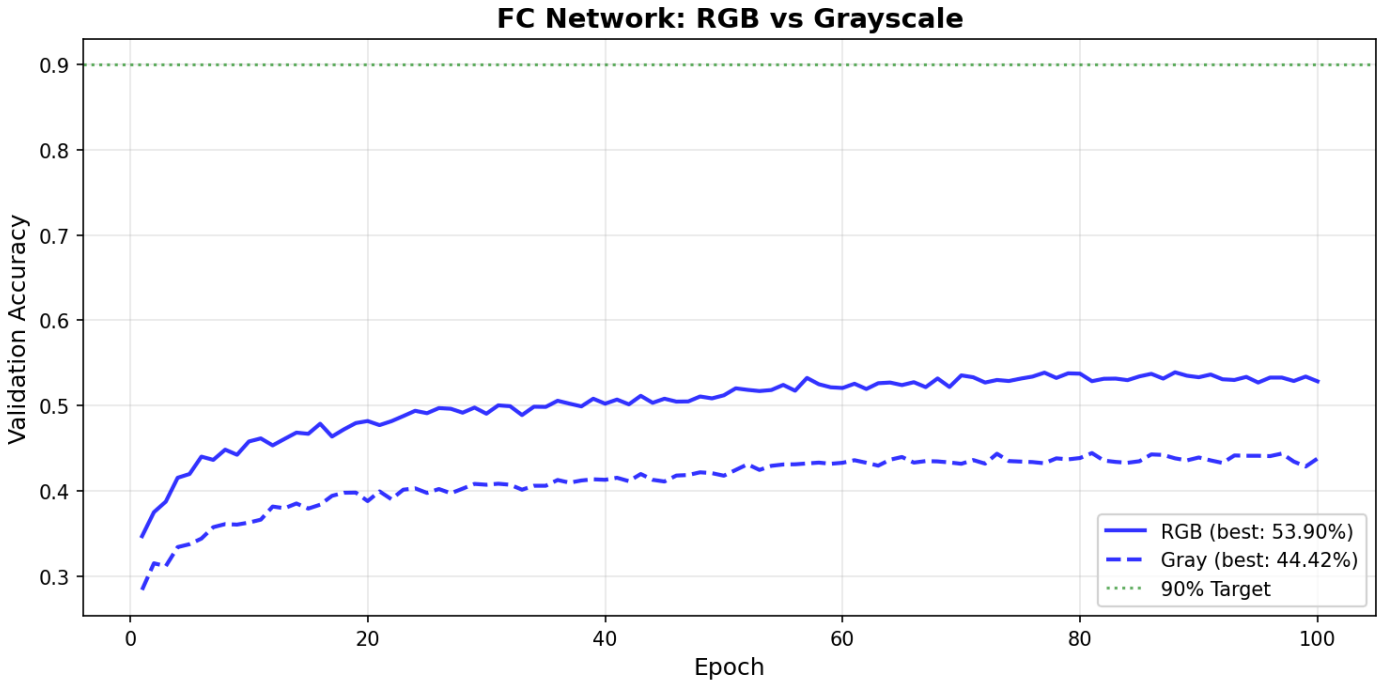
3、探究颜色信息复杂度是否为 MLP 性能差异的主要原因。

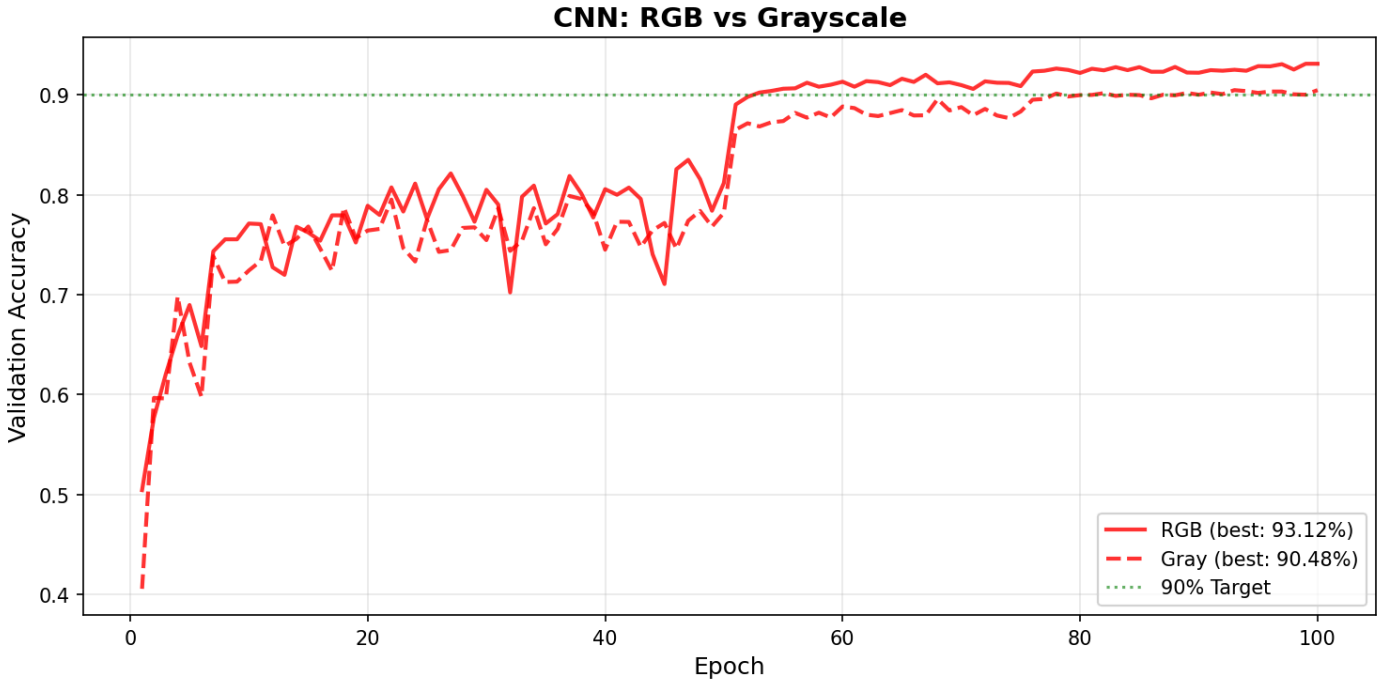
- 主要思路就是利用 PyTorch 的 torchvision 库中 **transforms.functional.to_grayscale()** API 函数将 RGB 图像转化为灰度图像。



- 数据处理、训练策略几乎与 [RGB MLP CNN](#MNIST & CNN & CIFAR-10) 差别不大，这里不再进行赘述！

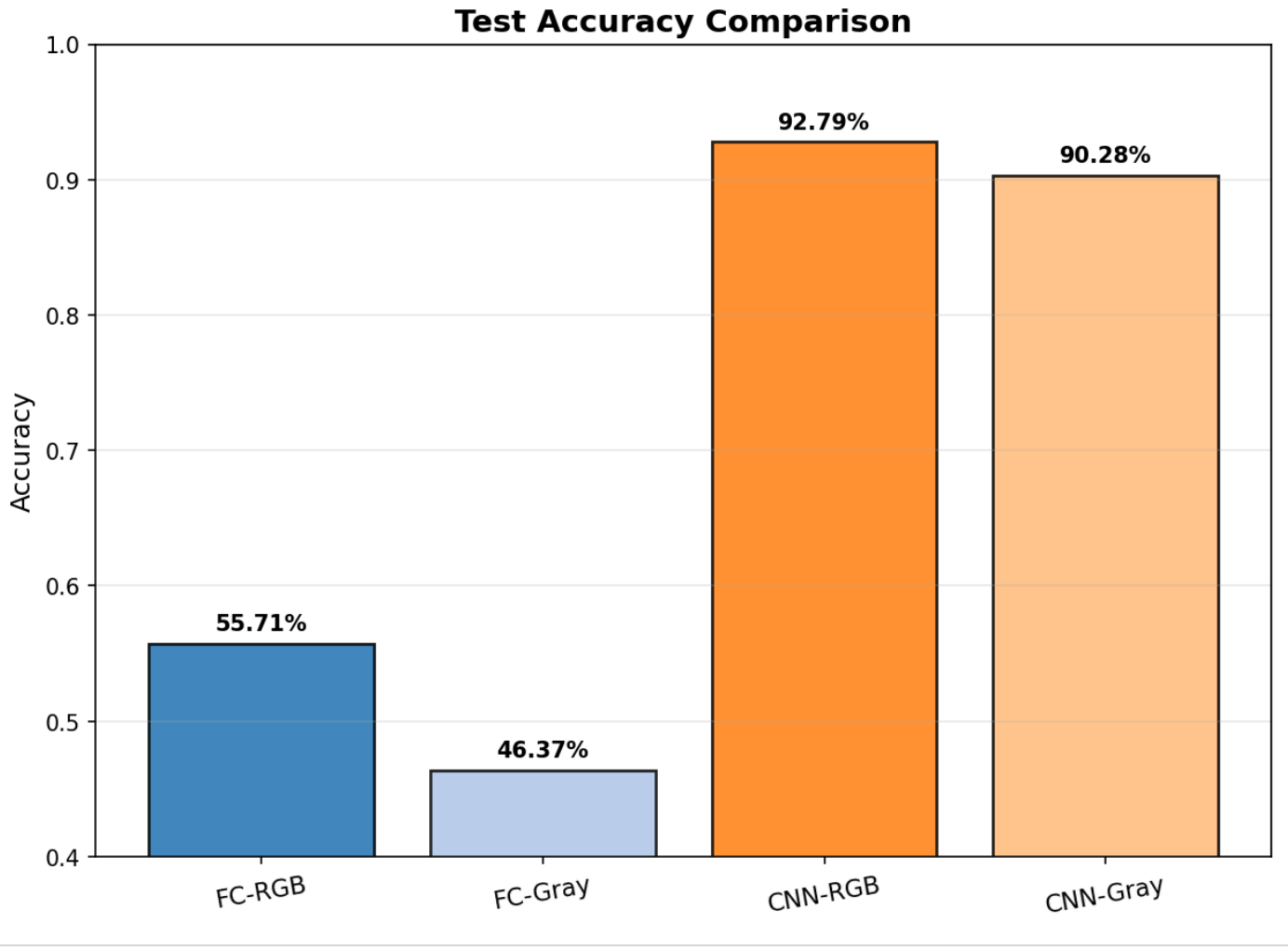
$Training Procedure$





$Performance Comparison$

TrainingProcedure



- 颜色信息对于 MLP 影响显著（约 9% 性能下降），灰度化输入维度减少 67%，严重限制其表达能力；而对于 CNN 来说，对颜色信息的依赖相对较低（仅 2.5% 下降）；
- 灰度化后 CNN 性能仍远超 MLP，我认为足以说明提取空间特征的能力是关键。 MLP 将二维空间信息"拉伸"为一维向量，完全破坏了像素间的拓扑关系，导致无法学习具有平移不变性的特征表示。

三、总结与探索

1、MLP 性能差异分析：

- 输入图像直接被展平，直接失去了空间局部性，MLP学到的“模板”其实更像是像素排列下的统计模式而已（似乎挺像模版匹配的）！
- CIFAR-10 相比 MNIST 图像的变化太多了：位置、角度、颜色等等，这就变化对于 MLP 来说“超纲了😖”！

- 似乎 MLP 在 CIFAR-10 真是上限也就 55% 左右（尝试不同层次、不同训练策略似乎都变化不大）。

2、CNN 表现分析：

- CNN 的卷积操作天然具有局部连接性和权重共享，使其能够以平移不变的方式学习特征。
- 至于为什么丢失颜色信息其性能下降幅度很小（约2.5%），通过查阅 blog 得到如下信息：

> 通道信息与空间信息的区别
>
> 通道信息和空间信息是图像中两种不同的信息类型。
>
> - 通道信息是指图像中每个像素点的不同颜色通道（例如RGB图像中的红、绿、蓝通道）之间的信息差异。通道信息通常表示图像的全局特征，例如图像的颜色、明暗等。
> - 空间信息则是指图像中每个像素点的位置和周围像素点之间的空间关系。空间信息通常表示图像的局部特征，例如图像中的纹理、边缘等。
>
> 参考：关于神经网络中的信息——通道信息、空间信息

- 结合如上信息，可以作出一定解释：
  - 卷积操作能够有效捕捉像素之间的空间关系（边缘、纹理和形状等局部特征）；
  - 相比之下，通道信息主要提供颜色等全局特征，而对于大多数视觉识别任务，空间结构才是判别的关键（细粒度分类任务中可能影响显著）。因此，即使去除颜色信息，CNN 依然能够通过空间特征实现高效的图像分类！

3、在实验中对第一层卷积核可视化观察到某些卷积核似乎并未学习到任何信息（”死亡”），查阅后了解到似乎与学习率过高导致 dead ReLU 有关：

> 当权重参数变为负值时，输入网络的正值会和权重相乘后也会变为负值，负值通过relu后就会输出0；如果w在后期有机会被更新为正值也不会出现大问题，但是当relu函数输出值为0时，relu的导数也为0，因此会导致后边Δw一直为0，进而导致w一直不会被更新，因此会导致这个神经元永久性死亡（一直输出0）
> 参考：对Relu激活函数导致[神经元死亡]的理解

也就是，训练初期学习率过高导致权重更新幅度过大，使得某些神经元的激活值长期为负，经 ReLU 激活后恒为 0，梯度消失，权重永久停止更新，卷积核陷入"死亡"状态。

4、其他内容这里不再进行赘述，实验内容部分已有较详细说明！

参考：

[1] CS231N-Assignment1
[2] Initializing Weights for the Convolutional and Fully Connected Layers
[3] 关于神经网络中的信息——通道信息、空间信息
[4] 对Relu激活函数导致[神经元死亡]的理解