0.前言

文件里 model_paramsNnorYdropYregNcroYlay 中，nor 是归一化，drop 是 dropout，reg 是正则化，cro 是交叉验证，lay 是是否有复杂层。N 和 Y 是 no 和 yes，txt 是训练过程中的数据的意思。

1.Numpy 前向传播初步实现 CNN 网络

前言

本来想手写完成整个过程的，但是用 Numpy 手写到反向传播的时候发现自己的能力实在是不太够，所以反向传播交给了 pytorch，只用 Numpy 写了前向传播的过程。

而且 Numpy 无法用 gpu 加速，如果说真的用五万数据跑一次 epoch 的话，时间达到了恐怖的 8 个小时，到时候服务器都关了，所以这个 Numpy 写前向传播的代码只是一个尝试，调参和调用各种优化方法主要还是用 pytorch 实现。

前向传播

网络结构

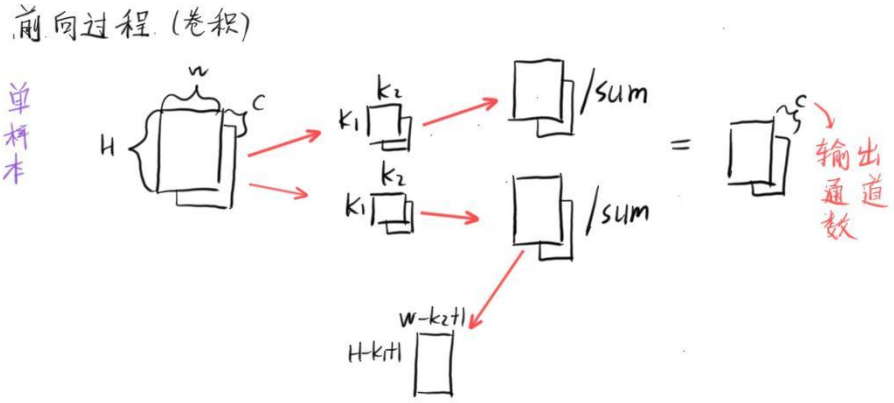
一层卷积一层池化一层全连接，卷积核的数量是 32，形状是 3x3，2x2 大小的池化层，全连接网络输入 32 * 16 * 16 = 8192 个特征，10 个类别的输出。

卷积

先来步长为 1 的卷积网络
为了简单起见，假设为样本数量为 1，步长为 1。
多通道卷积前向过程，我们需要下面的参数：

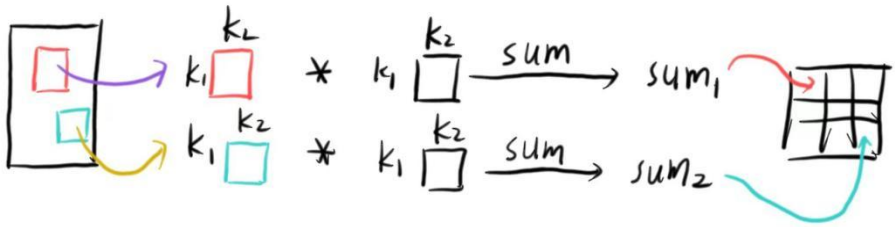
- z：卷积层矩阵,形状(N,C,H,W)，N 为 batch_size，C 为通道数
- K：卷积核,形状(C,D,k1,k2)，C 为输入通道数，D 为输出通道数
- b：偏置,形状(D,)

公式推导如下



前向传播过程 1（示意图）

上面是单样本，首先假设一个单通道的图片，一个卷积核，我们在卷积的时候，相当于把值慢慢的填入一个矩阵里，矩阵里的每一个值都是两个矩阵相乘（非点乘）后加起来的和，因此应该是这样的



前向传播过程 2（示意图）

写成公式就是

$$y[h,w]=\sum_{\{i=0\}}^{\{k_{\{1\}}-1\}}\sum_{\{j=0\}}^{\{k_{\{2\}}-1\}} z[h+i,w+j]\cdot K+b$$

如果我们想要用代码实现，那么就是

```
    for i in range(out_height):
        for j in range(out_width):
            input_slice = input_padded[i * stride:i * stride + filter_height, j * stride:j * stride
+ filter_width]
            output[i, j] = np.sum(input_slice * filter)
```

如果说我们现在有 C 个输入通道，其实就是在 sum 的时候顺便把其他通道一起 sum 了，然后把它们的 sum 结果继续 sum，说人话就是几层全 sum 一遍，代码上的改动也十分容易。

```
      for c in range(in_channels):  # For each input channel
          for i in range(out_height):
              for j in range(out_width):
                  input_slice = input_padded[b, c_in, i * stride:i * stride + filter_height, j
* stride:j * stride + filter_width]
                  output[b, c_out, i, j] += np.sum(input_slice * filters[c_out, c_in])
```

现在我们要拓展到 D 个输出通道，也就是 D 个卷积核。

其实也很简单，不像上面一个我们的处理，这里不需要把 sum 扩大，而是需要把输出的矩阵扩大到三维，并且因为矩阵变大了，我们需要多加一个循环。

```
        for d in range(out_channels):
            for c in range(in_channels):
                for i in range(out_height):
                    for j in range(out_width):
                        # Slice the input to the size of the filter and perform element-wise
multiplication
                        input_slice = input_padded[b, c_in, i * stride:i * stride + filter_height,
j * stride:j * stride + filter_width]
                        output[b, c_out, i, j] += np.sum(input_slice * filters[c_out, c_in])
```

因为多维矩阵的输出，所以偏置 b 也是多维的。

然后我们只是一次输入一个样本，做一件事让他输入 N 个样本。

像是 1 个样本变成 N 个样本总是很简单，因为我们只需要添加一个额外的维度即可，和上面的处理是一样的。

```
        for b in range(batch_size):
            for c_out in range(out_channels):
                for c_in in range(in_channels):
                    for i in range(out_height):
```

```
                    for j in range(out_width):
                        input_slice = input_padded[b, c_in, i * stride:i * stride + filter_height, j * stride:j * stride + filter_width]
                        output[b, c_out, i, j] += np.sum(input_slice * filters[c_out, c_in])
```

到这里除了步长只能为 1 外，前向传播基本完善完毕了，这里没有用公式去描述这个，因为代码好像反而更好理解一些，毕竟看到一堆西格玛容易头疼。

现在输出矩阵已经变成了一个四维巨无霸。

池化

然后是池化，池化我们用的是 maxpool 池化，思路和上面的类似，都是一个循环，然后用几乘几的规格去扫描矩阵，然后循环叠加的思路也和上面是一样的，具体不推导了，代码如下：

“

```
    for b in range(batch_size):
        for c in range(channels):
            for i in range(out_height):
                for j in range(out_width):
                    pooled[b, c, i, j] = np.max(x[b, c, i*2:i*2+2, j*2:j*2+2])
```

运行方法和结果

1. 使用参数以及优化方法
   - 损失函数：交叉熵
   - 优化器：Adam
   - Normalization：除以 255 约束数据
   - dropout：无
   - 正则化方法：无
   - 学习率：0.001
   - Epoch：10
   - Batch_size:64
   - Steps:500/64
2. 运行结果

```
100%|██████████| 8/8 [05:20<00:00, 40.12s/batch, accuracy=11.8, loss=0.12]
Epoch [1/10], Loss: 0.1202, Accuracy: 11.80%
100%|██████████| 8/8 [05:25<00:00, 40.66s/batch, accuracy=12.6, loss=0.122]
Epoch [2/10], Loss: 0.1218, Accuracy: 12.60%
100%|██████████| 8/8 [05:23<00:00, 40.41s/batch, accuracy=17.4, loss=0.0901]
Epoch [3/10], Loss: 0.0901, Accuracy: 17.40%
100%|██████████| 8/8 [05:10<00:00, 38.82s/batch, accuracy=19.4, loss=0.0505]
Epoch [4/10], Loss: 0.0505, Accuracy: 19.40%
100%|██████████| 8/8 [05:24<00:00, 40.50s/batch, accuracy=22.6, loss=0.0446]
Epoch [5/10], Loss: 0.0446, Accuracy: 22.60%
100%|██████████| 8/8 [05:23<00:00, 40.39s/batch, accuracy=31.6, loss=0.0328]
Epoch [6/10], Loss: 0.0328, Accuracy: 31.60%
100%|██████████| 8/8 [05:21<00:00, 40.21s/batch, accuracy=35, loss=0.0324]
Epoch [7/10], Loss: 0.0324, Accuracy: 35.00%
100%|██████████| 8/8 [05:07<00:00, 38.45s/batch, accuracy=31, loss=0.0342]
Epoch [8/10], Loss: 0.0342, Accuracy: 31.00%
100%|██████████| 8/8 [05:06<00:00, 38.32s/batch, accuracy=30.6, loss=0.0366]
Epoch [9/10], Loss: 0.0366, Accuracy: 30.60%
100%|██████████| 8/8 [05:05<00:00, 38.25s/batch, accuracy=34.6, loss=0.0323]
Epoch [10/10], Loss: 0.0323, Accuracy: 34.60%
```

P.S：这里的准确率是训练集上的

看着不太像是 CNN 的水平，上面代码运行了大概一个小时，然后好像没有收敛？我就再加大学习率到 0.01，然后：

```
100%|██████████| 8/8 [05:43<00:00, 42.93s/batch, accuracy=9.2, loss=0.211]
Epoch [1/15], Loss: 0.2109, Accuracy: 9.20%
100%|██████████| 8/8 [05:49<00:00, 43.63s/batch, accuracy=12.8, loss=0.155]
Epoch [2/15], Loss: 0.1549, Accuracy: 12.80%
100%|██████████| 8/8 [05:36<00:00, 42.00s/batch, accuracy=17.2, loss=0.128]
Epoch [3/15], Loss: 0.1284, Accuracy: 17.20%
100%|██████████| 8/8 [05:12<00:00, 39.03s/batch, accuracy=19.4, loss=0.0738]
Epoch [4/15], Loss: 0.0738, Accuracy: 19.40%
100%|██████████| 8/8 [05:11<00:00, 38.92s/batch, accuracy=27.4, loss=0.0522]
Epoch [5/15], Loss: 0.0522, Accuracy: 27.40%
100%|██████████| 8/8 [05:07<00:00, 38.47s/batch, accuracy=26, loss=0.0437]
Epoch [6/15], Loss: 0.0437, Accuracy: 26.00%
100%|██████████| 8/8 [05:04<00:00, 38.08s/batch, accuracy=28.8, loss=0.0404]
Epoch [7/15], Loss: 0.0404, Accuracy: 28.80%
100%|██████████| 8/8 [05:05<00:00, 38.18s/batch, accuracy=29.2, loss=0.0385]
Epoch [8/15], Loss: 0.0385, Accuracy: 29.20%
100%|██████████| 8/8 [05:05<00:00, 38.15s/batch, accuracy=27.2, loss=0.0395]
Epoch [9/15], Loss: 0.0395, Accuracy: 27.20%
100%|██████████| 8/8 [05:06<00:00, 38.35s/batch, accuracy=27.2, loss=0.0394]
Epoch [10/15], Loss: 0.0394, Accuracy: 27.20%
100%|██████████| 8/8 [05:05<00:00, 38.14s/batch, accuracy=33, loss=0.0338]
Epoch [11/15], Loss: 0.0338, Accuracy: 33.00%
100%|██████████| 8/8 [05:08<00:00, 38.54s/batch, accuracy=35.2, loss=0.0308]
Epoch [12/15], Loss: 0.0308, Accuracy: 35.20%
100%|██████████| 8/8 [05:07<00:00, 38.41s/batch, accuracy=30.8, loss=0.0341]
Epoch [13/15], Loss: 0.0341, Accuracy: 30.80%
100%|██████████| 8/8 [05:11<00:00, 38.89s/batch, accuracy=29, loss=0.0361]
Epoch [14/15], Loss: 0.0361, Accuracy: 29.00%
100%|██████████| 8/8 [05:11<00:00, 38.88s/batch, accuracy=33.2, loss=0.0354]
Epoch [15/15], Loss: 0.0354, Accuracy: 33.20%
```

效果好像不太好，由于这只是一个尝试，我并没有让他画图。

2.Pytorch-CNN 网络代码解释

➢ 随机种子

“

```
def set_seed(seed):
    random.seed(seed)

    np.random.seed(seed)

    torch.manual_seed(seed)

    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
set_seed(42)
```

PY 的随机种子，Np 的随机种子，torch 的随机种子，GPU 上的随机种子，cuDNN 的随机种子全部设置一遍，最后禁用 cuDNN 的自动调优。这样比较方便比较。

➢ GPU 使用

“

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

这样会开局打印

```
Using device: cuda
```

千万别用 CPU。

➢ 加载数据

用加载数据用的是和 A1 一个函数，这个函数就不说了，但是因为有 Pytorch 的存在，我们需要对读取的数据进行一点处理。

```
train_x = torch.tensor(train_x, dtype=torch.float32).view(-1, 3, 32, 32)  # Reshape to (N, C, H, W)
train_y = torch.tensor(train_y, dtype=torch.long)

test_x = torch.tensor(test_x, dtype=torch.float32).view(-1, 3, 32, 32)
test_y = torch.tensor(test_y, dtype=torch.long)
```

把 numpy 转化为 PyTorch Tensors

➢ 网络结构与代码

网络结构

卷积层:

输入通道数： 3（对应 RGB 图像的 3 个颜色通道）。
输出通道数： 32（输出 32 个卷积核，也就是 32 个特征图）。
卷积核大小： 3x3 的卷积核，用于从输入图像中提取特征。
填充：为了保持图像的空间维度不变，使用了 1 的填充。输入的 32x32 图像通过 3x3 卷积后，仍然是 32x32。

池化层:

池化大小： 2x2 的最大池化层。池化操作将每个 2x2 区域替换为该区域中的最大值，从而减少图像的空间维度（宽度和高度都会减半）。
输入图像： 32x32 变为 16x16。

全连接层:

输入尺寸：32 * 16 * 16（从卷积和池化后得到的特征图，32 个通道，16x16 的空间尺寸）。
输出尺寸：10（10 个类别，用于 CIFAR-10 数据集的 10 类分类任务）。

前向传播

代码如下：

```
def forward(self, x):
    x = self.conv1(x)

    if self.normalization_method == 'batchnorm':
        x = self.bn1(x)
    elif self.normalization_method == 'layernorm':
        x = self.ln1(x)

    x = torch.relu(x)
    x = self.pool(x)

    if self.use_dropout:
        x = self.dropout(x)

    x = x.view(-1, 32 * 16 * 16)  # Flatten for fully connected layer
    x = self.fc1(x)

    return x
```

没什么好看的，主要是用了 relu 来激活需要注意一下。

模型训练

输入参数

model：要训练的模型（例如卷积神经网络）。
train_loader：训练数据的 DataLoader，用于按批次加载训练数据。
criterion：损失函数，用于计算模型预测值与真实标签之间的误差。
optimizer：优化器，负责调整模型参数以最小化损失。
device：训练设备（例如 CPU 或 GPU）。
num_epochs：训练的总周期数。
test_dataset：测试数据集，用于评估模型的性能。

训练模型

主要是两部分，
正向传播：

```
outputs = model(inputs)
loss = criterion(outputs, labels)
```

反向传播：

```
loss.backward()
optimizer.step()
```

Tqdm

```
for inputs, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs}", leave=False):
```

通过 tqdm 提供进度条，实时显示训练进度。

保存文件

```
        if epoch+1 == 15:
            torch.save(model.state_dict(), 'basemodel_paramsYnorYdropYregYcroYlay.pth')
```

如果是第 x 个周期，则将当前模型的参数保存到文件中。保存的文件名包含了当前的配置（例如正则化方法、是否使用 Dropout 等）。

评估模型（无交叉验证）

分为两个部分，一个是不用 test 函数，一个会用到。
首先是不用 test 函数，主要是计算在训练集上的准确率和误差。

```
            for i in range(labels.size(0)):
                label = labels[i]
                class_total[label] += 1
                class_correct[label] += (predicted[i] == label).item()

        avg_loss = running_loss / len(train_loader)
        accuracy = 100 * correct / total
        train_loss.append(avg_loss)
        train_accuracy.append(accuracy)
```

在每个训练周期结束后，调用 test 函数来计算模型在测试集上的准确率，并将该值记录在 true_accuracy 列表中。

```
true_accuracy.append(test(model, test_dataset, epoch,batch_size=64, device=device))
```

返回值

返回三个列表：train_loss、train_accuracy 和 true_accuracy，分别记录了每个训练周期的训练损失、训练准确率以及在测试集上的准确率。

模型测试

没啥好看的，他的输出是什么接下来会讲。

画图

没啥好说的，画图很简单。

```
plt.subplot(1, 2, 1)
plt.plot(epochs, true_accuracy, label='True Accuracy', color='red')
plt.title('true_accuracy vs Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.grid(True)
```

交叉验证

我们 5-fold 交叉验证。
核心代码是：

```
kf = KFold(n_splits=num_folds)
```

KFold 是一种交叉验证技术，它将数据集划分成 num_folds（默认为 5）个子集。在每一轮交叉验证中，KFold 会将其中一个子集作为验证集，其他 num_folds-1 个子集作为训练集。。

在 kf.split(train_x) 中，train_x 是输入特征数据，kf.split() 会返回数据的训练集索引（train_idx）和验证集索引（val_idx），以便在每一轮交叉验证中根据这些索引切分数据。

然后用 DataLoader 循环加载 用于加载训练集和验证集数据。五次训练是分开的，对于每一折的训练只进行一次，但我会使用不同的训练集和验证集。

3.Pytorch-CNN 网络结果分析

无 dropout，无 normalization，无正则化，无交叉验证

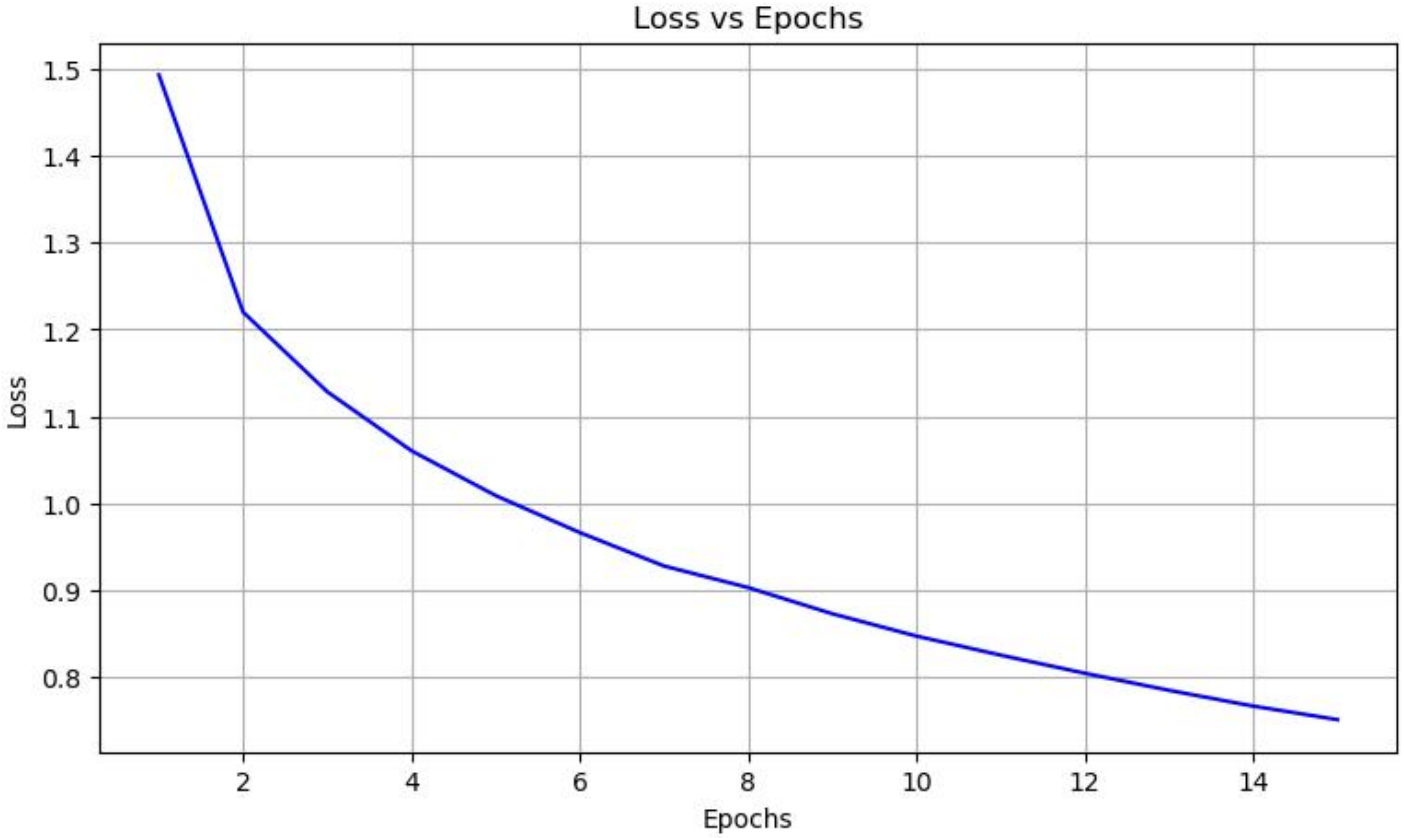
使用参数以及优化方法

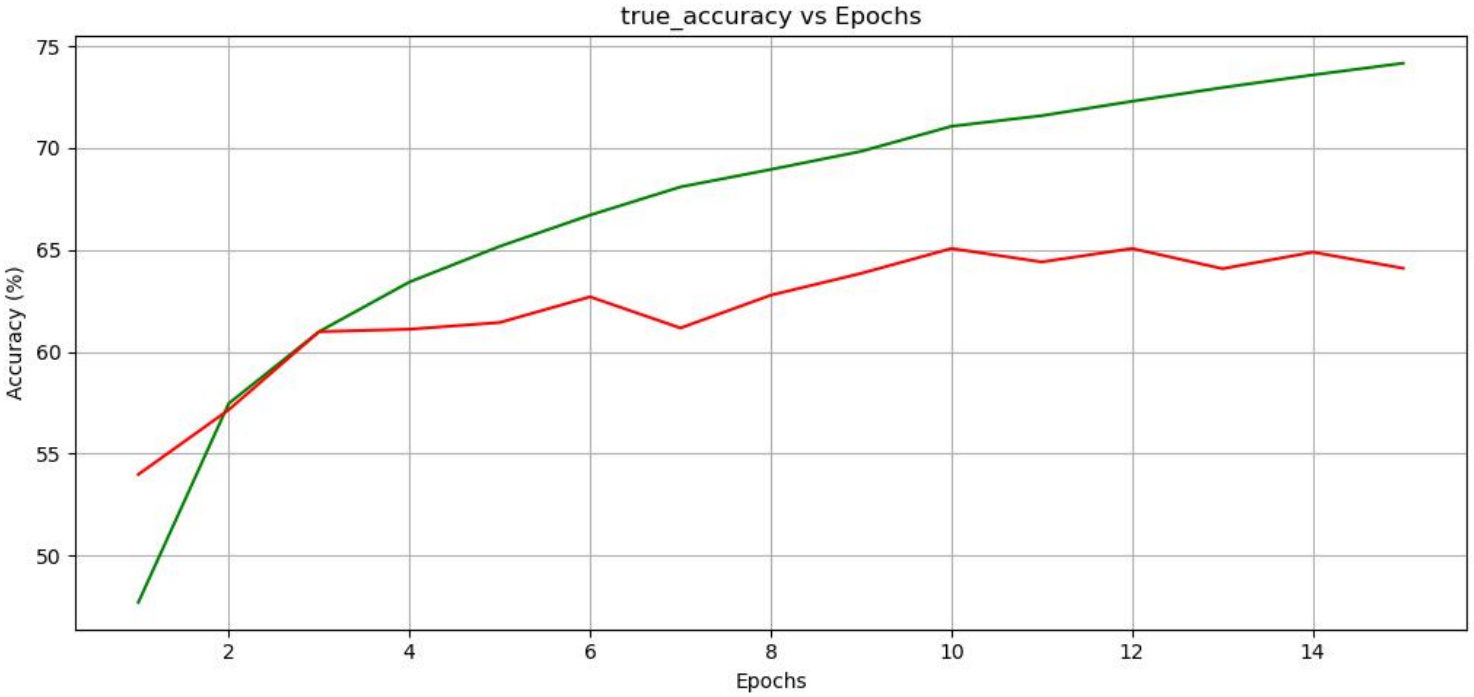
- 损失函数：交叉熵
- 优化器：Adam
- dropout：0.5
- 学习率：0.001
- Epoch：15
- Batch_size:64

- Steps:50000/64

➢ 运行结果

蓝色的线是损失函数与 epoch 的关系，红色的线是在测试集上的效果，绿色的线是训练集上的效果。下面给图的时候不会再赘述。





下面是运行准确率最高的一个 epoch 的各个参数。注意这个每个类的准确率是预测正确个数比上这个类的总的正确个数（也就是 1000）

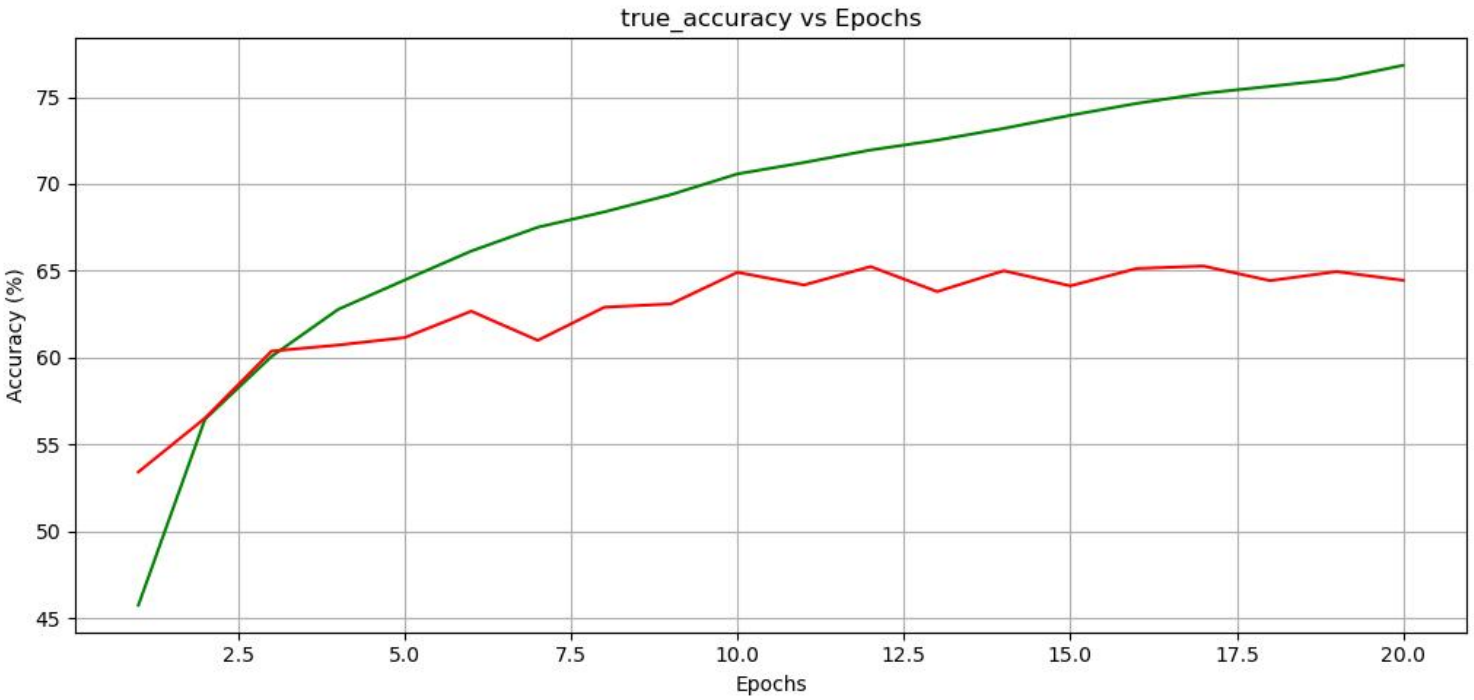
```
Epoch 10/15: Loss: 0.8476, Accuracy: 71.06%
Test Accuracy: 65.07%
Class 0 Accuracy: 64.60%
Class 1 Accuracy: 76.90%
Class 2 Accuracy: 48.00%
Class 3 Accuracy: 45.20%
Class 4 Accuracy: 51.60%
Class 5 Accuracy: 61.70%
Class 6 Accuracy: 84.10%
Class 7 Accuracy: 71.50%
Class 8 Accuracy: 74.70%
Class 9 Accuracy: 72.40%
```

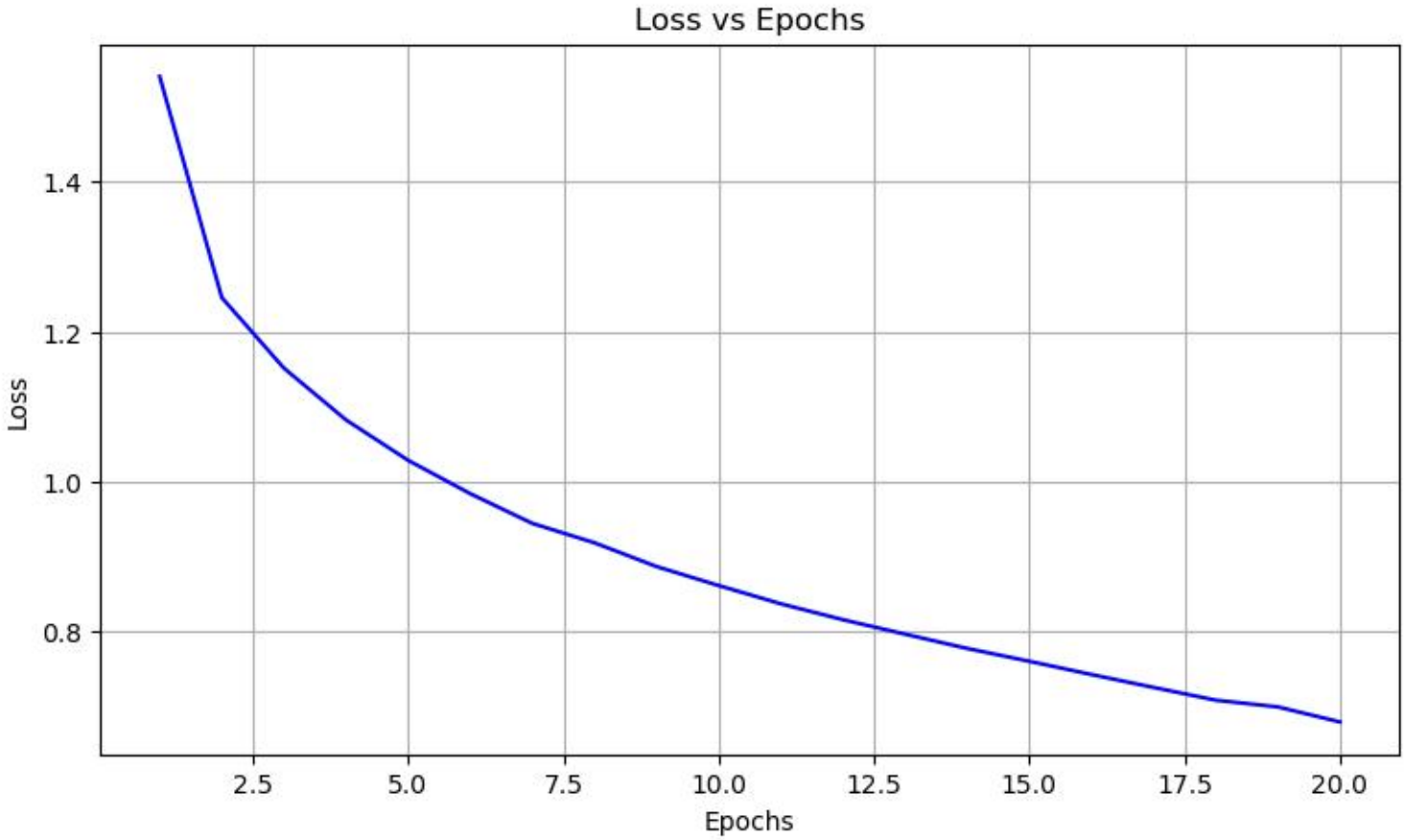
- 结果分析

比全连接网络优秀多了。

有 dropout，无 normalization，无正则化，无交叉验证

- 使用参数以及优化方法
  - 损失函数：交叉熵
  - 优化器：Adam
  - dropout：0.5
  - 学习率：0.001
  - Epoch：20
  - Batch_size:64
  - Steps:50000/64
- 运行结果



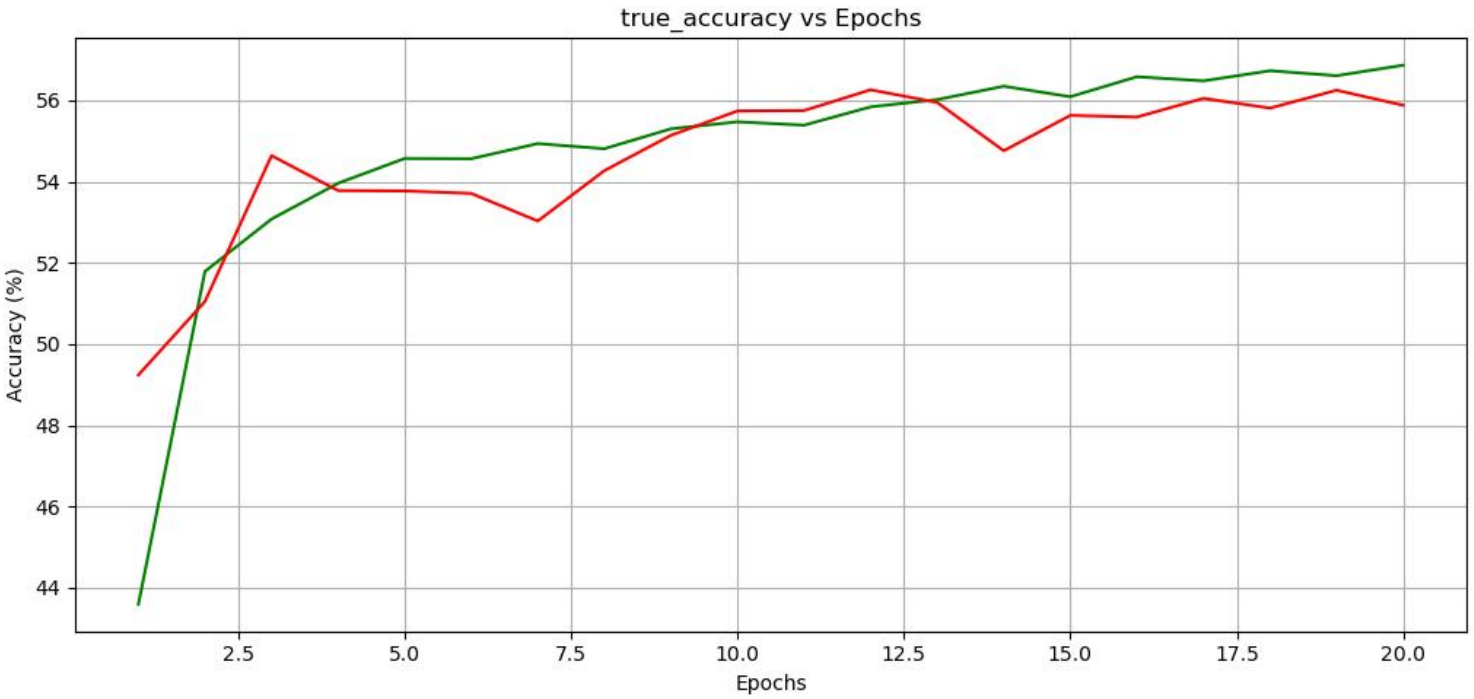


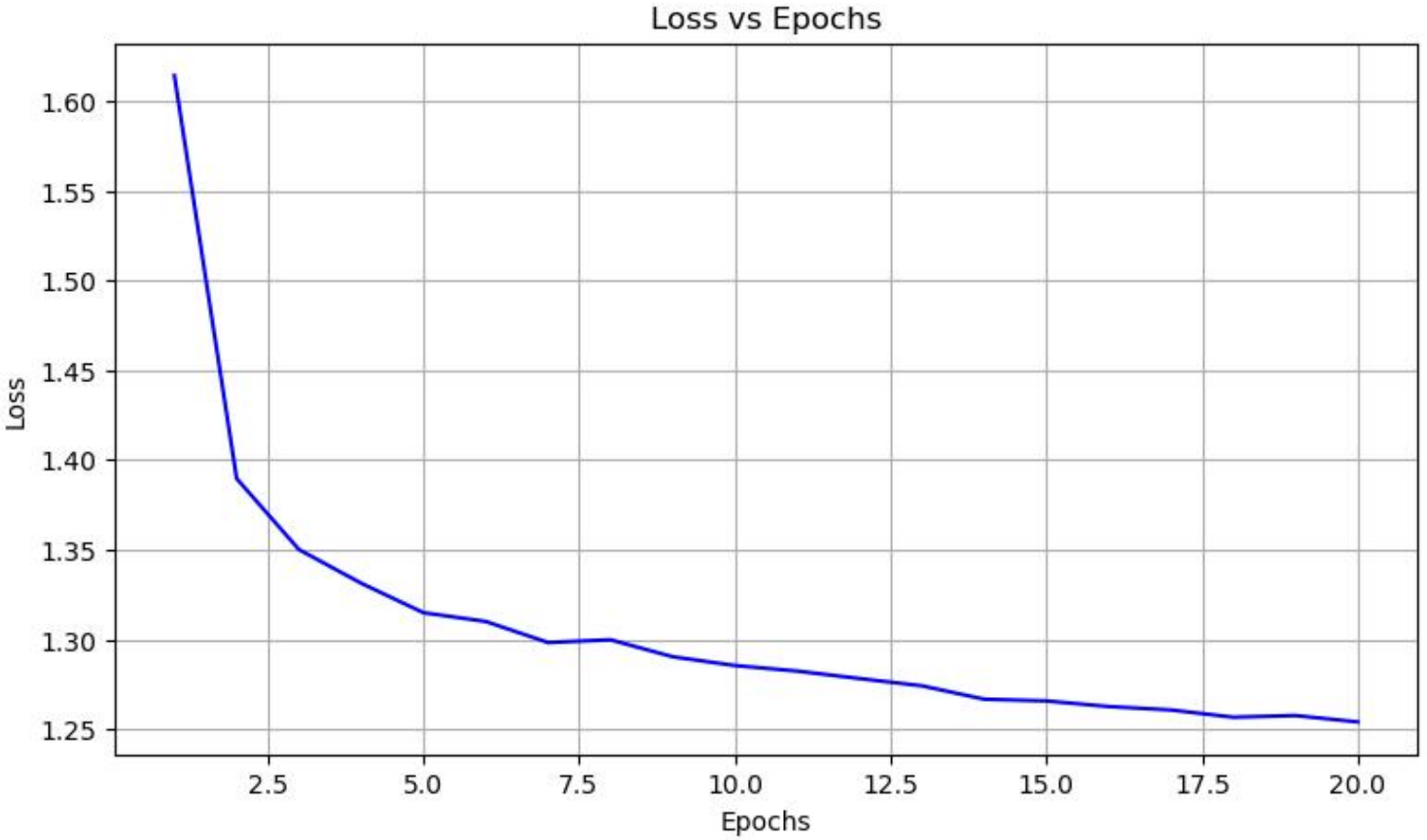
- 结果分析

收敛变慢了，在 Test 上的效果更好。

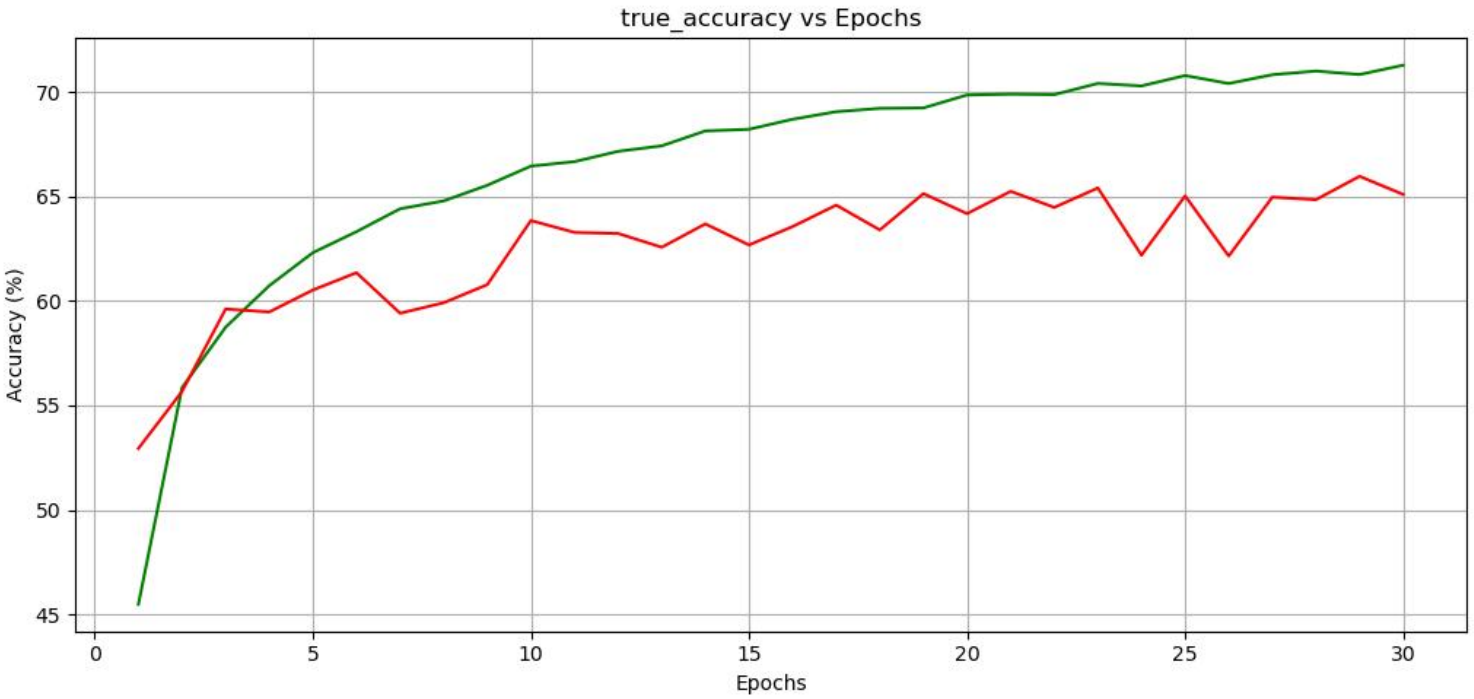
有 dropout，无 normalization，有正则化（L2），无交叉验证（附加题）

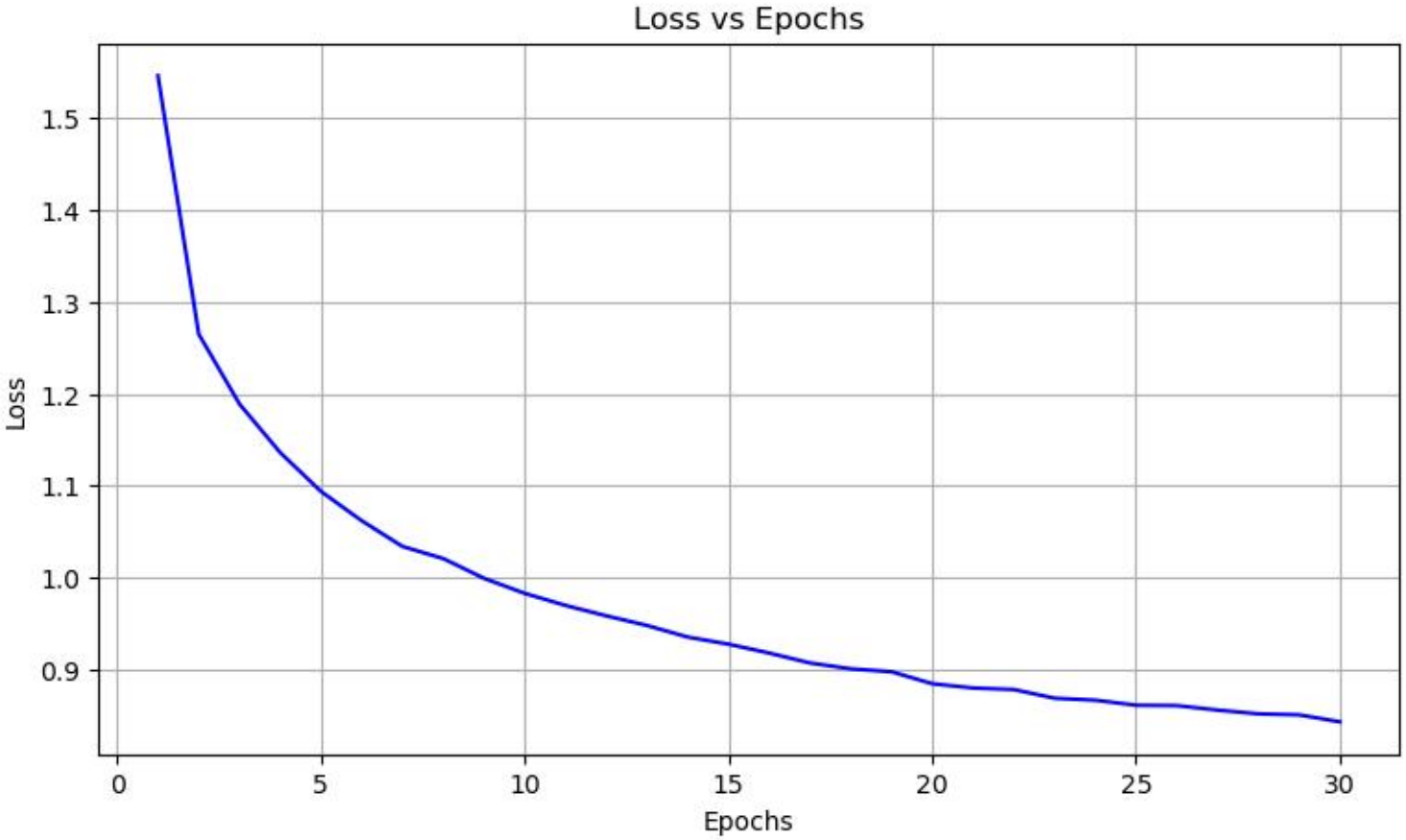
- 使用参数以及优化方法
  - 损失函数：交叉熵
  - 优化器：Adam
  - dropout：0.5
  - 学习率：0.001
  - Epoch：30
  - Batch_size:64
  - Steps:50000/64
  - weight_decay=0.01 *（这个是 L2 的参数）*
- 运行结果





无论是训练集还是测试集，都比起上面的来说有点差，可能是我 L2 设置大了，优化一下 `weight_decay=0.001`





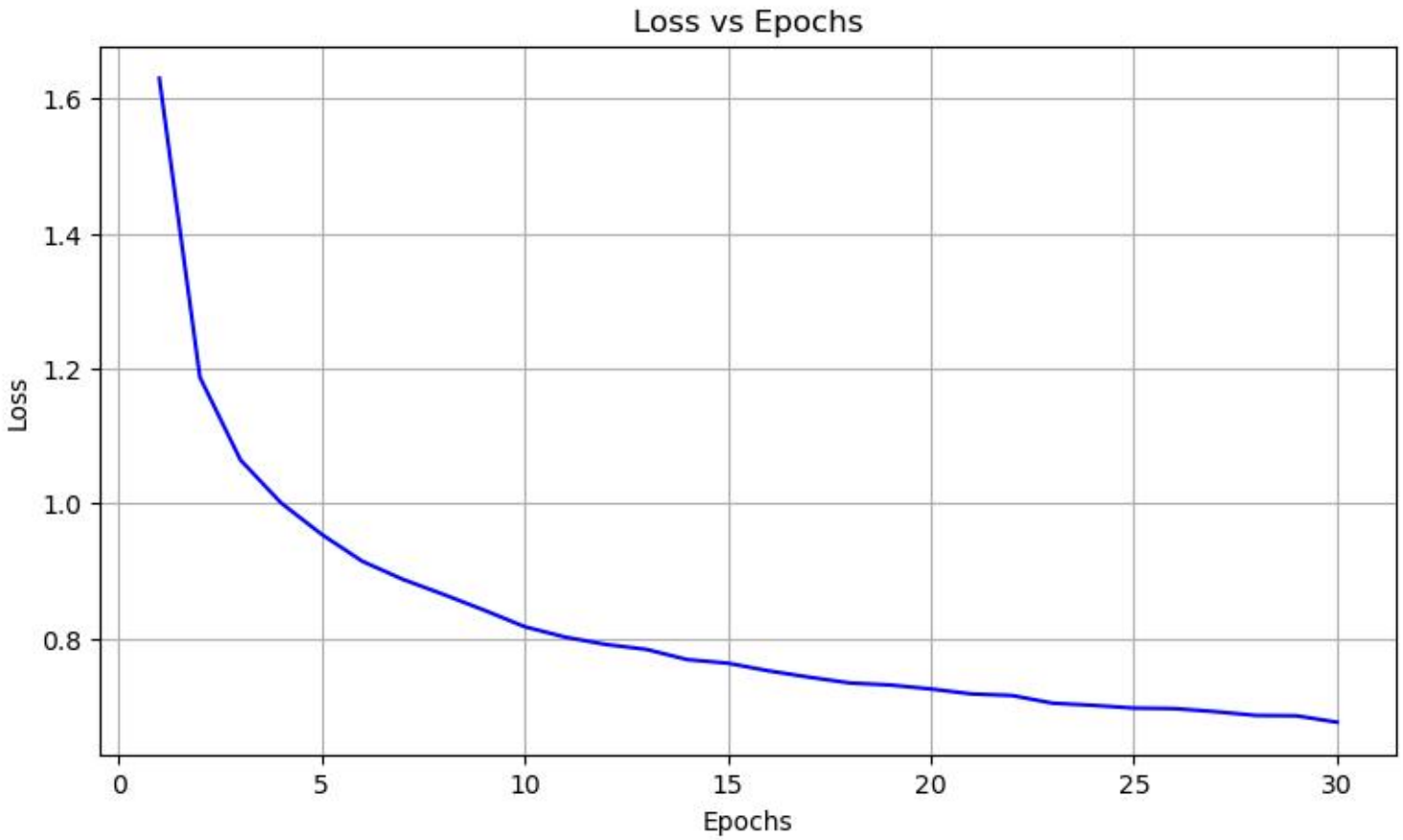
- 结果分析

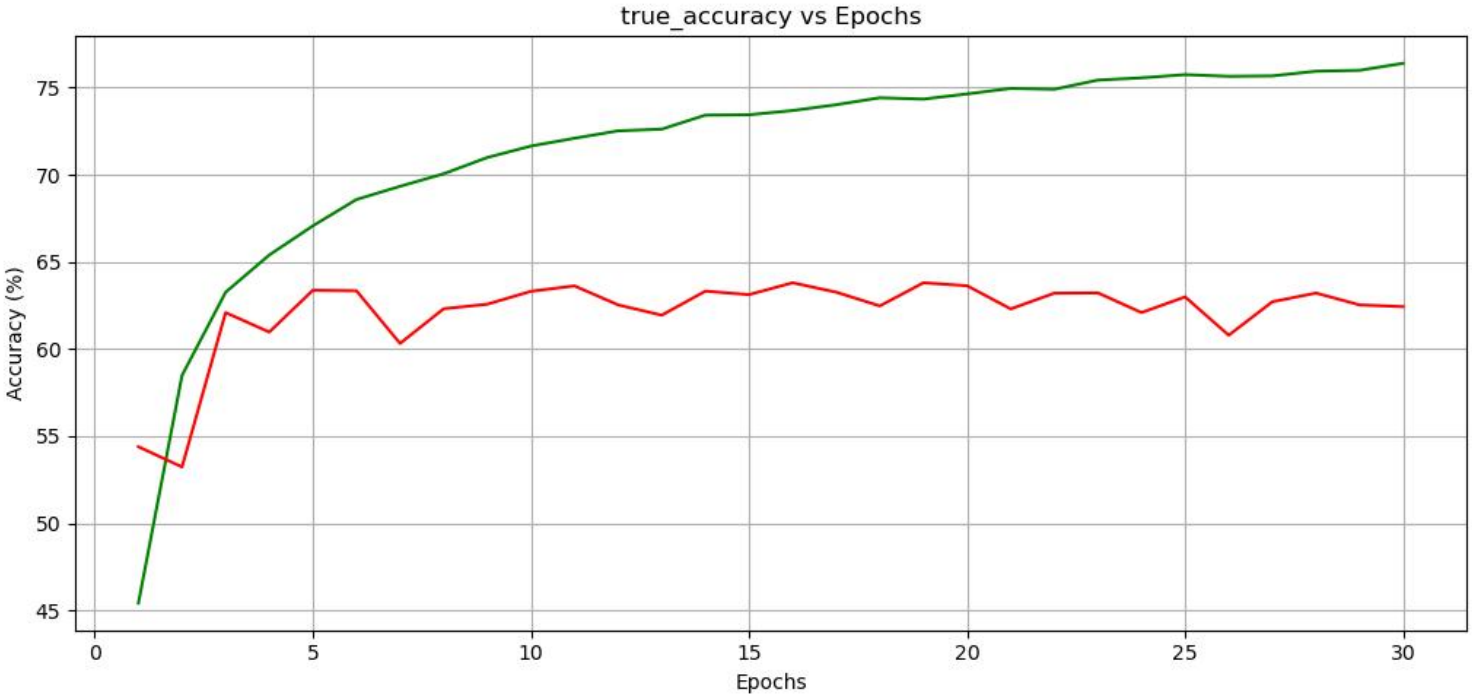
在 epoch 等于 27 的时候在 test 上的表现相当好，就是收敛更慢了。

有 dropout，有 normalization，有正则化，无交叉验证

规范化一个 mini-batch 内每一层的输入，使其均值为 0，方差为 1。
对于每一层的输入，BatchNorm 计算当前 mini-batch 的均值和方差，并根据这些统计量对输入进行标准化。然后，它通过可训练的参数（缩放参数 γ\gammaγ 和偏移参数 θ\betaθ）来恢复或调整标准化后的数据的分布。

- 使用参数以及优化方法
  - 损失函数：交叉熵
  - 优化器：Adam
  - dropout：0.5
  - 学习率：0.001
  - Epoch：30
  - Batch_size:64
  - Steps:50000/64
  - weight_decay=0.001
- 运行结果





- 结果分析

收敛变快了，在 test 上的表现变差了。

4.更深的 Pytorch-CNN 网络＋交叉验证

网络结构（附加题）

学的 AlexNet。输入 32×32×3 卷积 3×3,32 个卷积核，relu 后 2×2 池化，然后卷积 3×3,32 个卷积核，relu 后 2×2 池化，再跟上两层全连接网络，分别是 64 * 8 * 8 到 128，128 到 10。

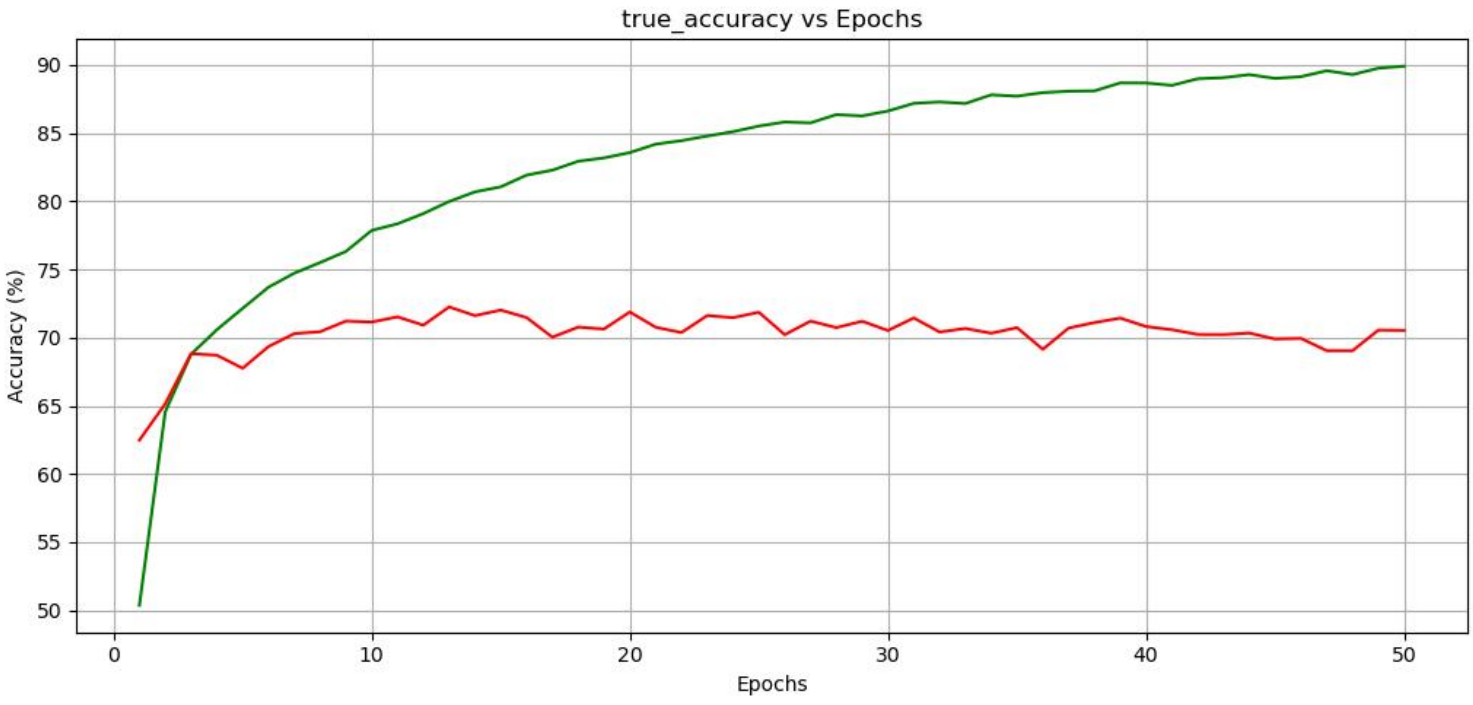
有 dropout，有 batchnorm，有 L2 正则化，不过没有数据增强。

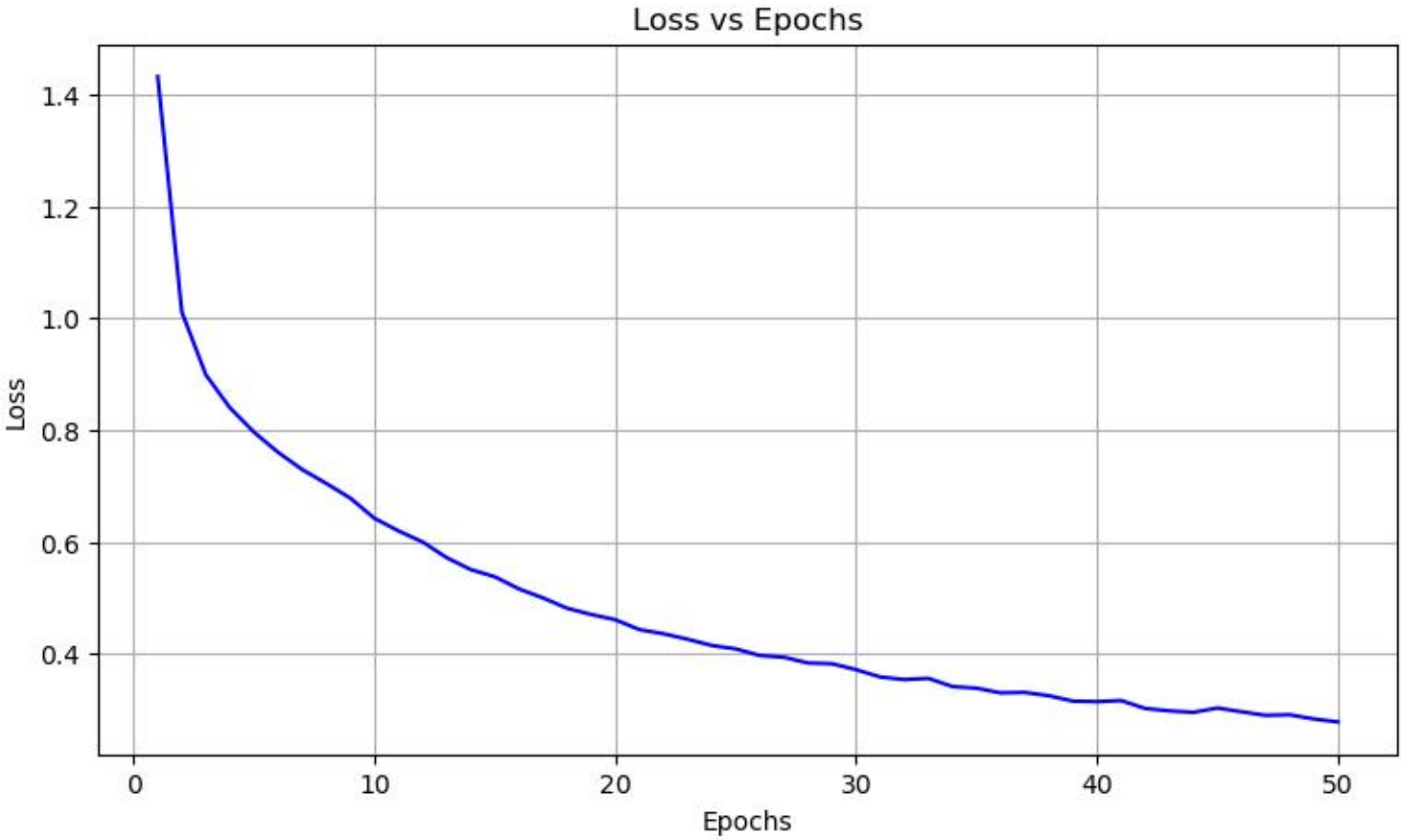
训练，测试一下

- 使用参数以及优化方法

  - 损失函数：交叉熵
  - 优化器：Adam
  - dropout：0.5
  - 学习率：0.001
  - Epoch：50
  - Batch_size:64
  - Steps:50000/64
  - weight_decay=0.001

- 运行结果





- 结果分析

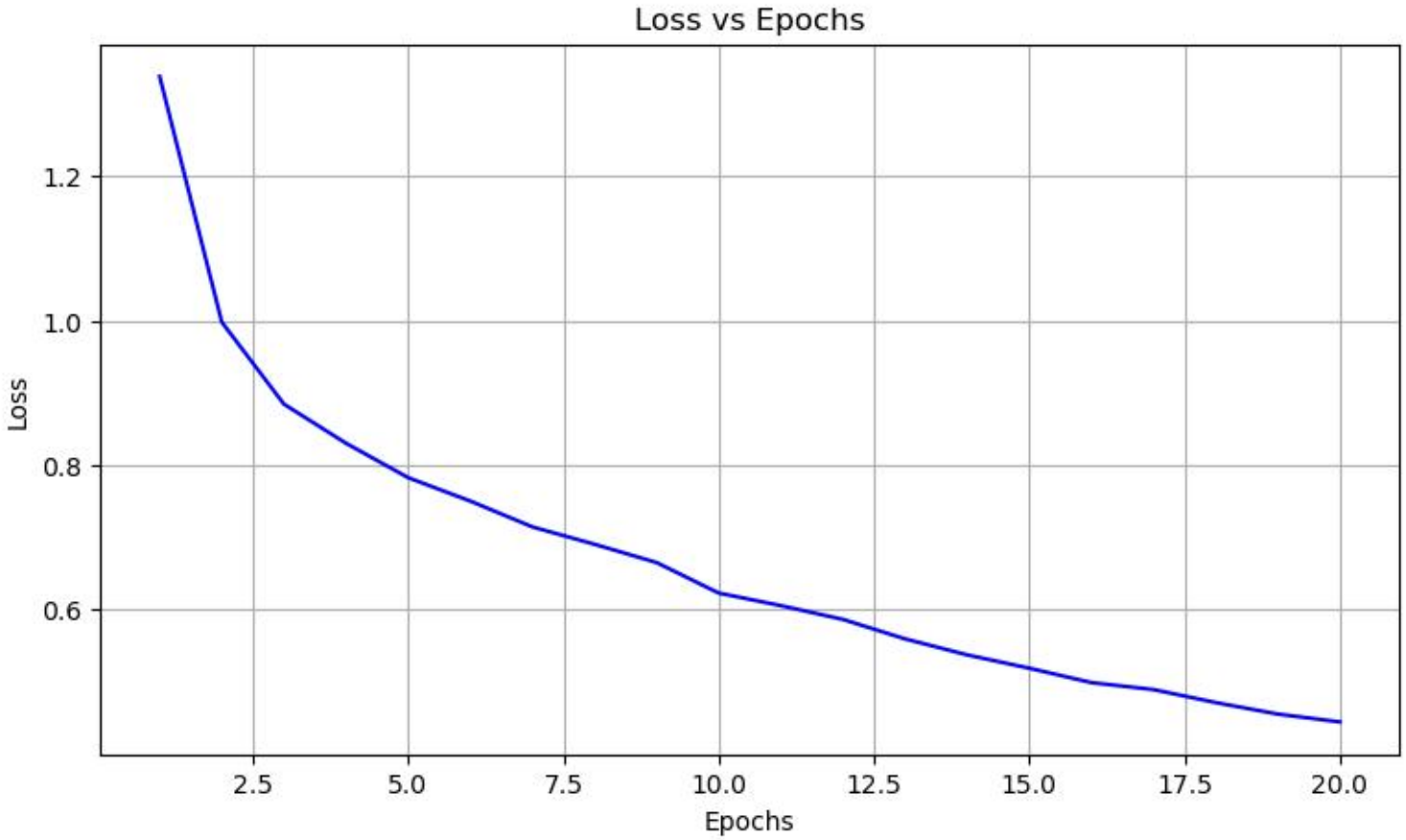
收敛变慢了，但是在 test 上和训练集的表现都变好了，是有史以来的最高值，test 正确率到了 70 上下，训练集 90 上下。算是很满意的结果了。

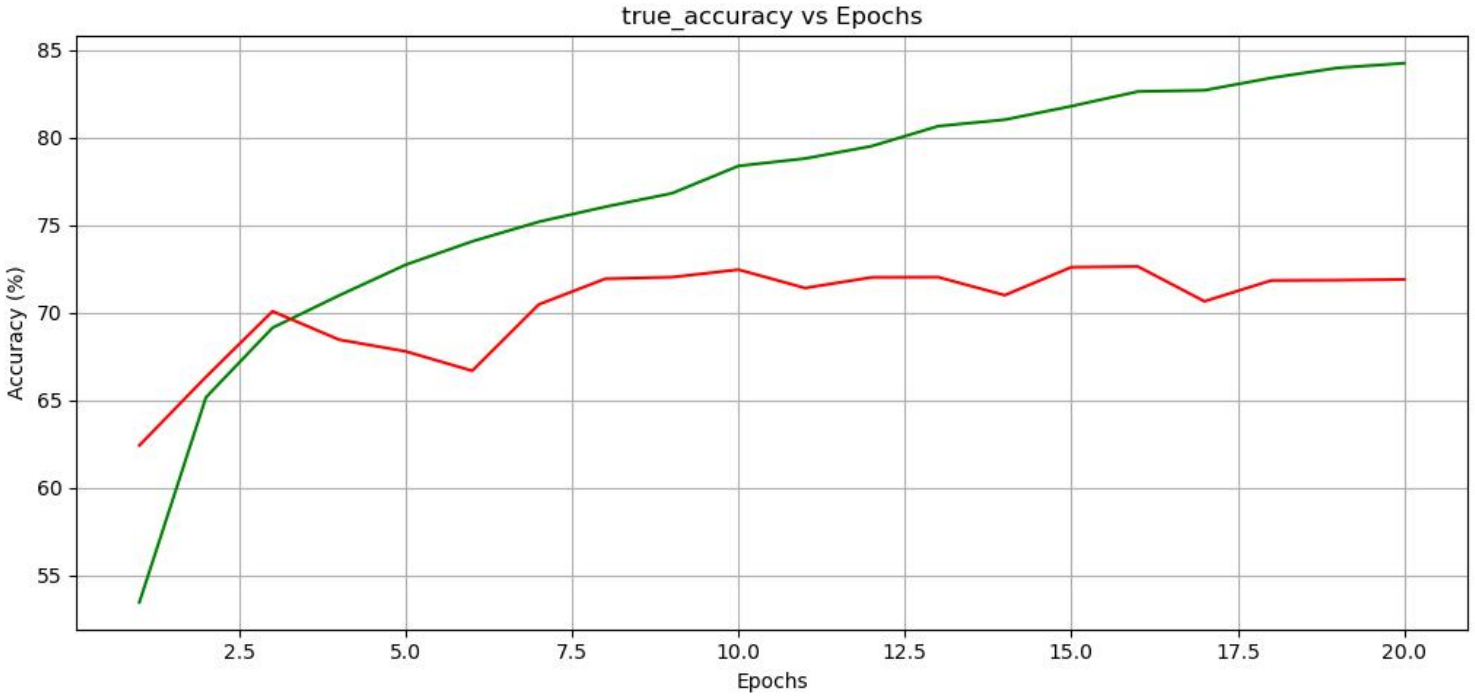
交叉测试找 dropout 最适合的值（附加题）

接下来你会看到Cross-validation Accuracy: x% ± y%的样式，前者是交叉验证过程中得到的平均准确率，后者是准确率的标准差。前者代表了准确率（我好像再说废话），后者代表了稳定性。

- 0.1

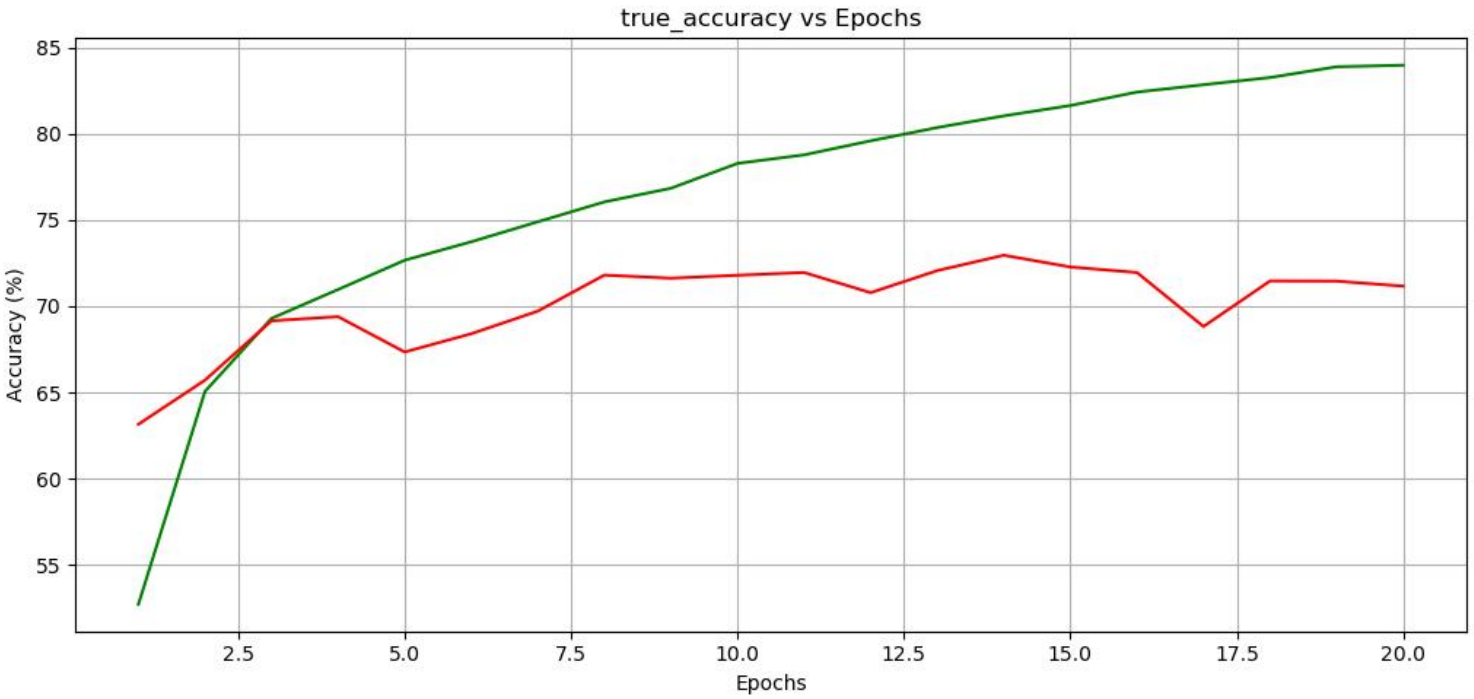
```
Cross-validation Accuracy: 57.16% ± 2.72%
```

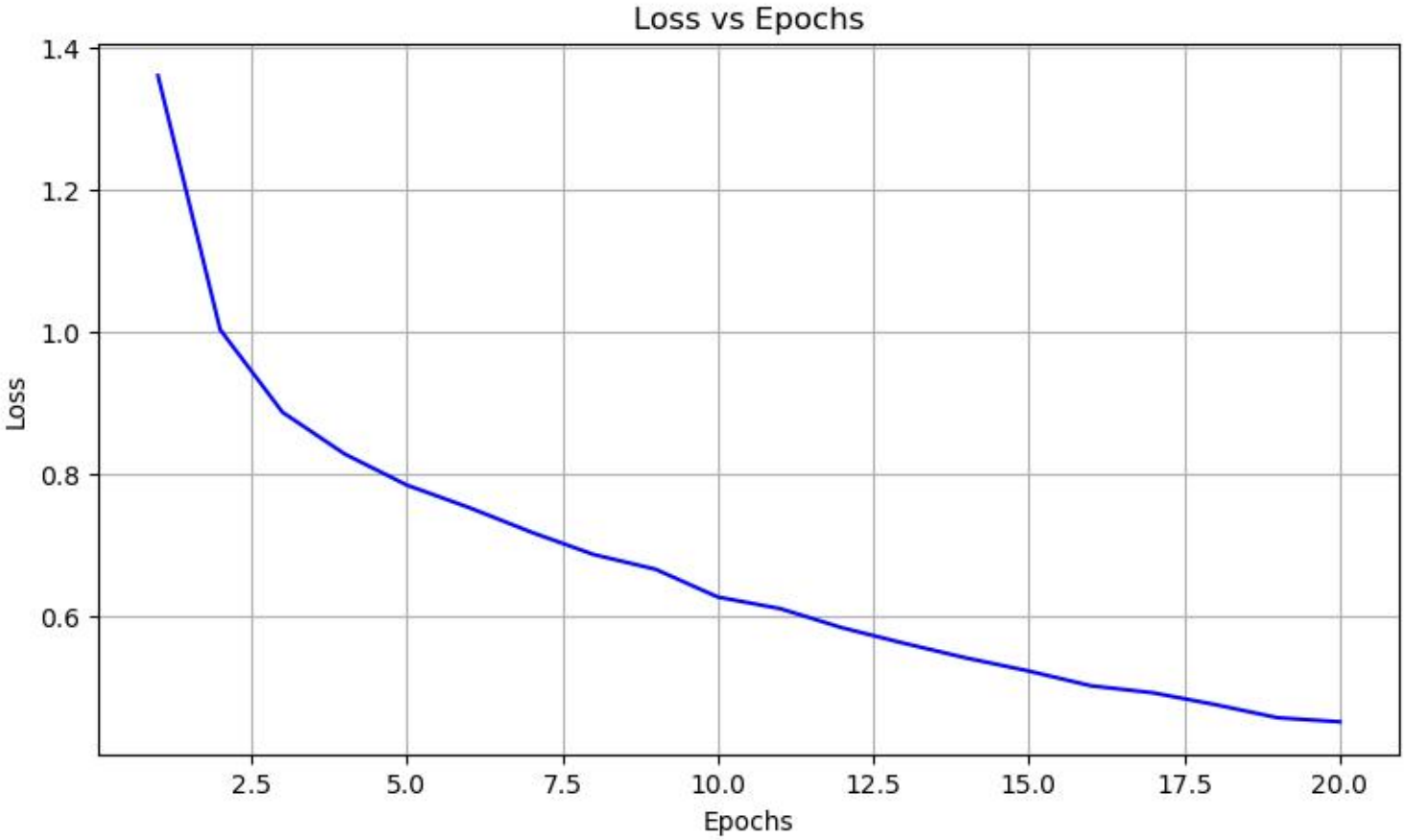




➢ 0.2

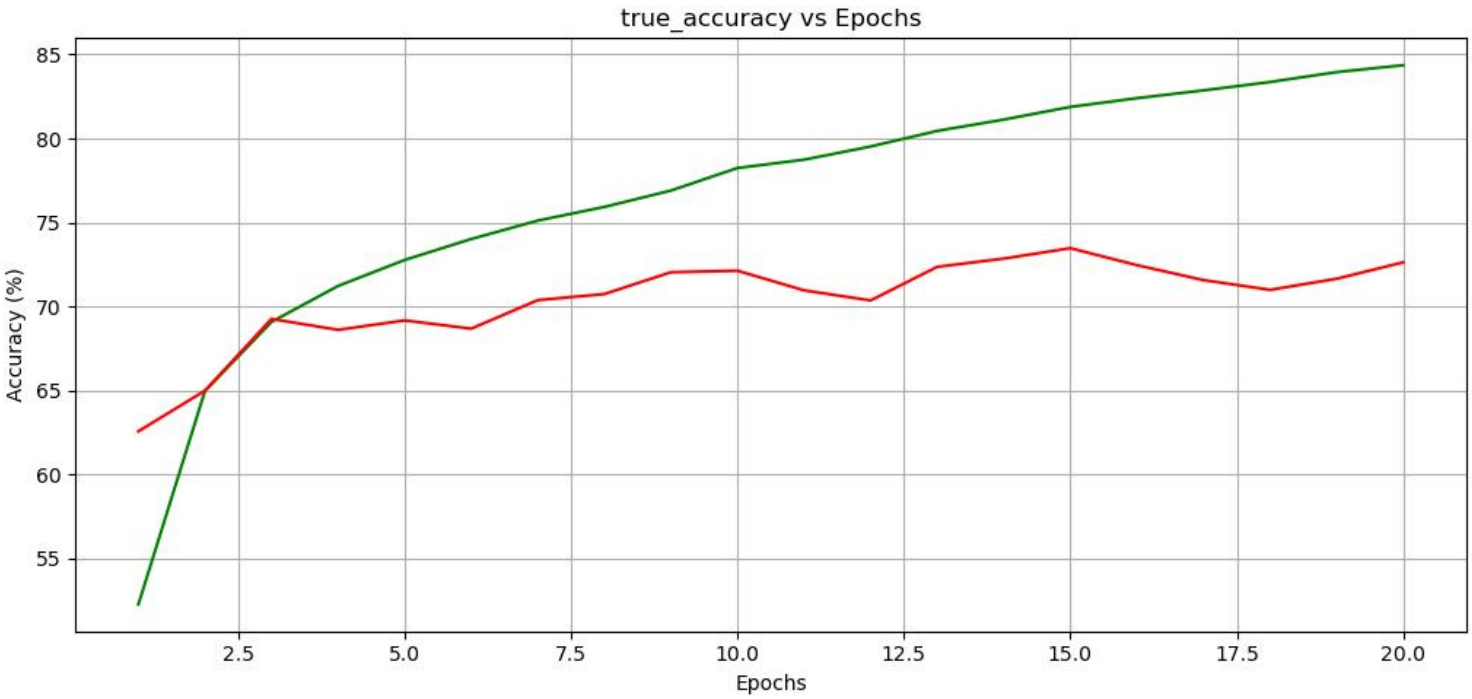
Cross-validation Accuracy: 56.33% ± 4.50%

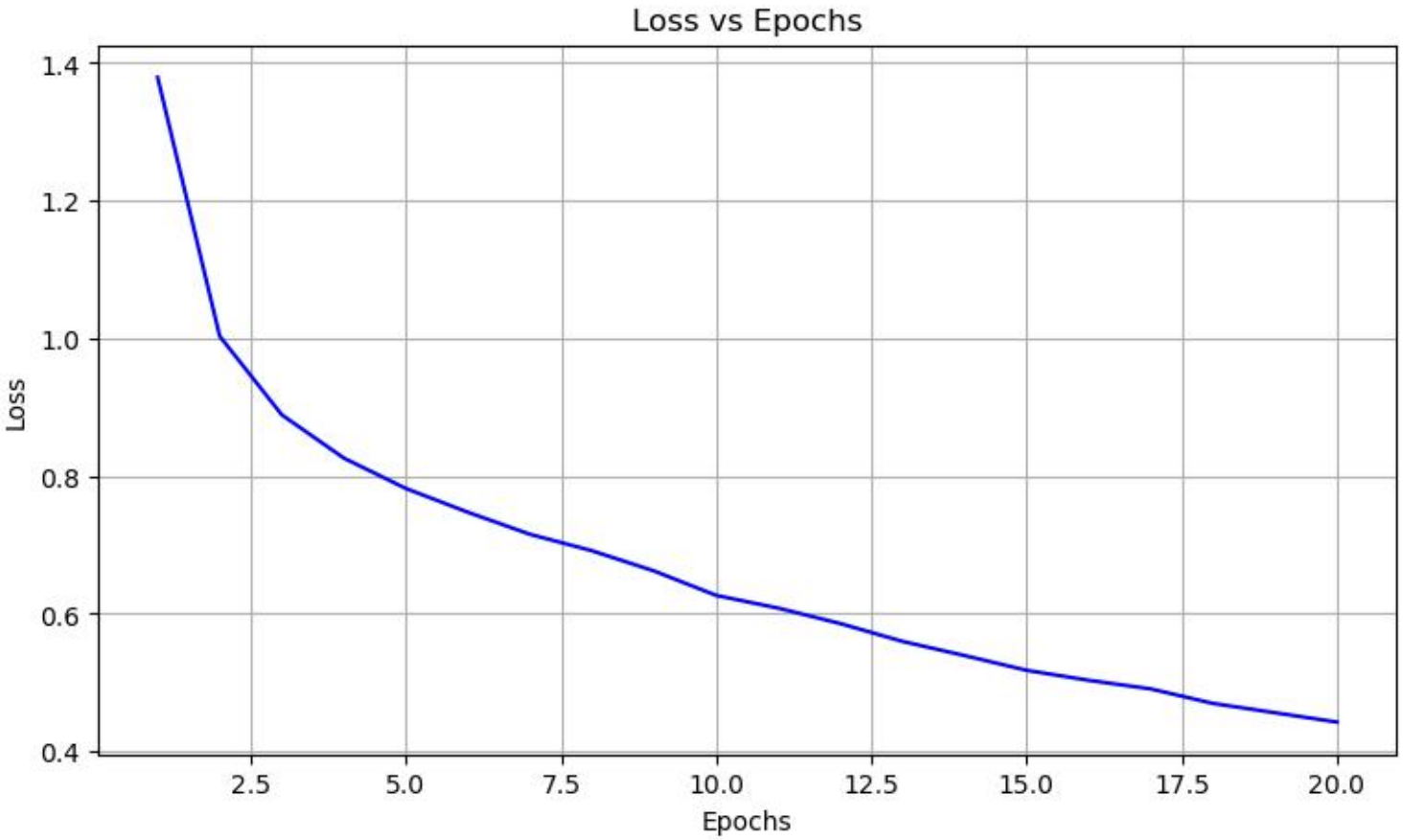




➢ 0.3

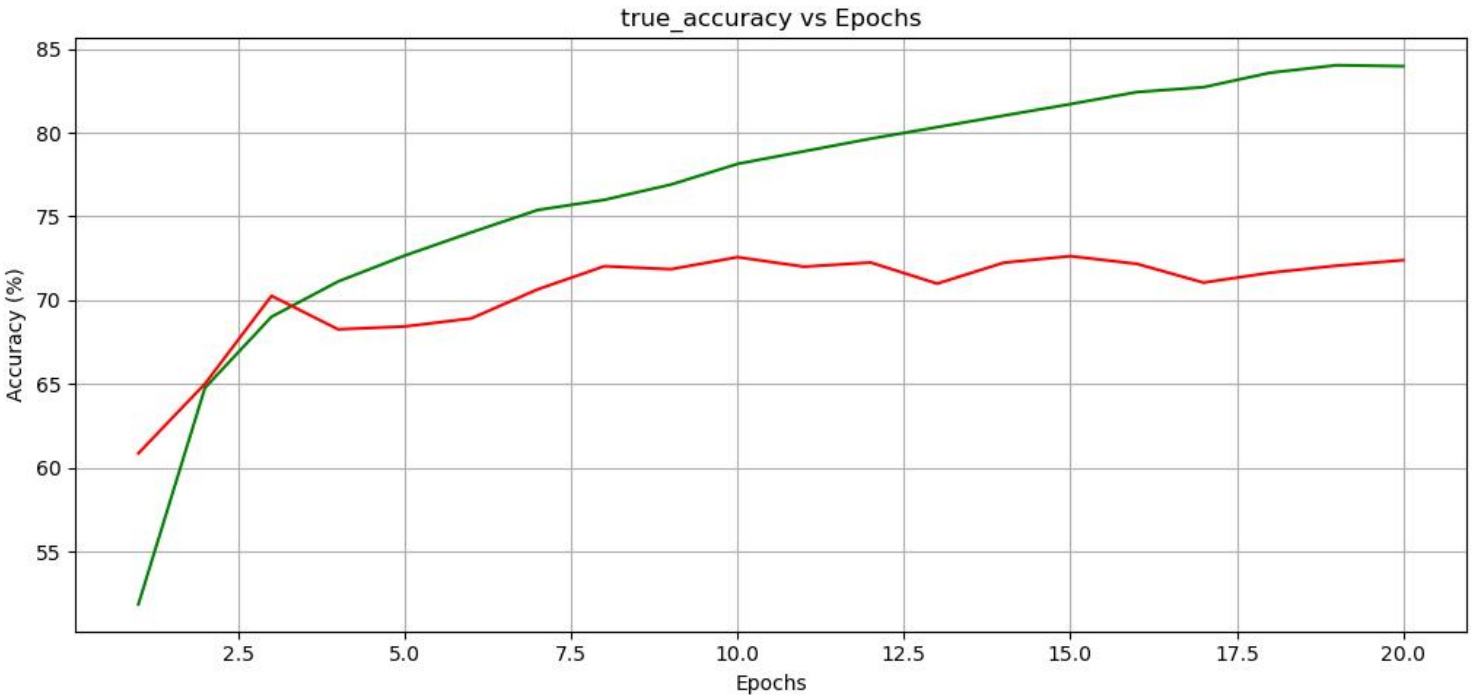
Cross-validation Accuracy: 56.90% ± 4.65%

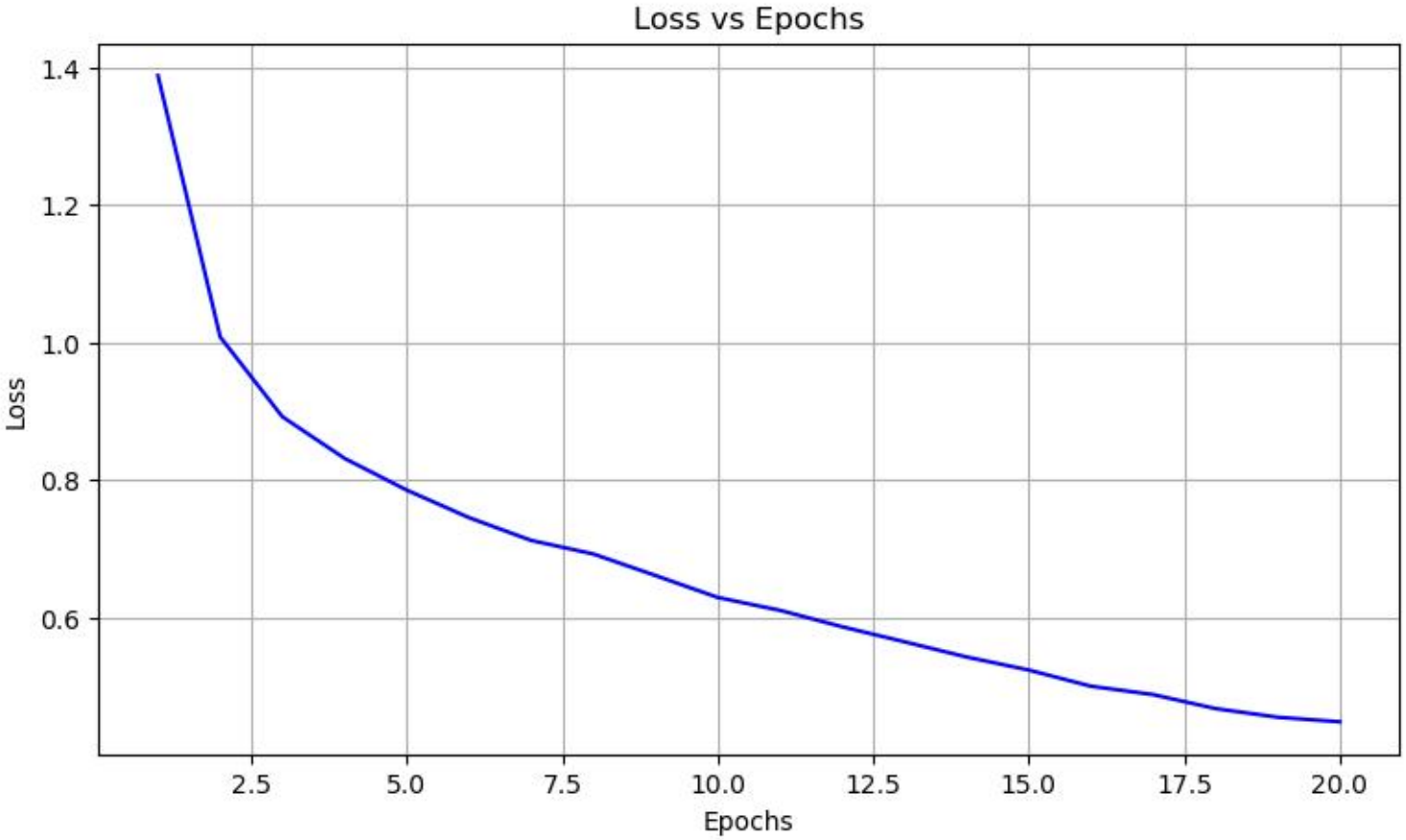




➢ 0.4

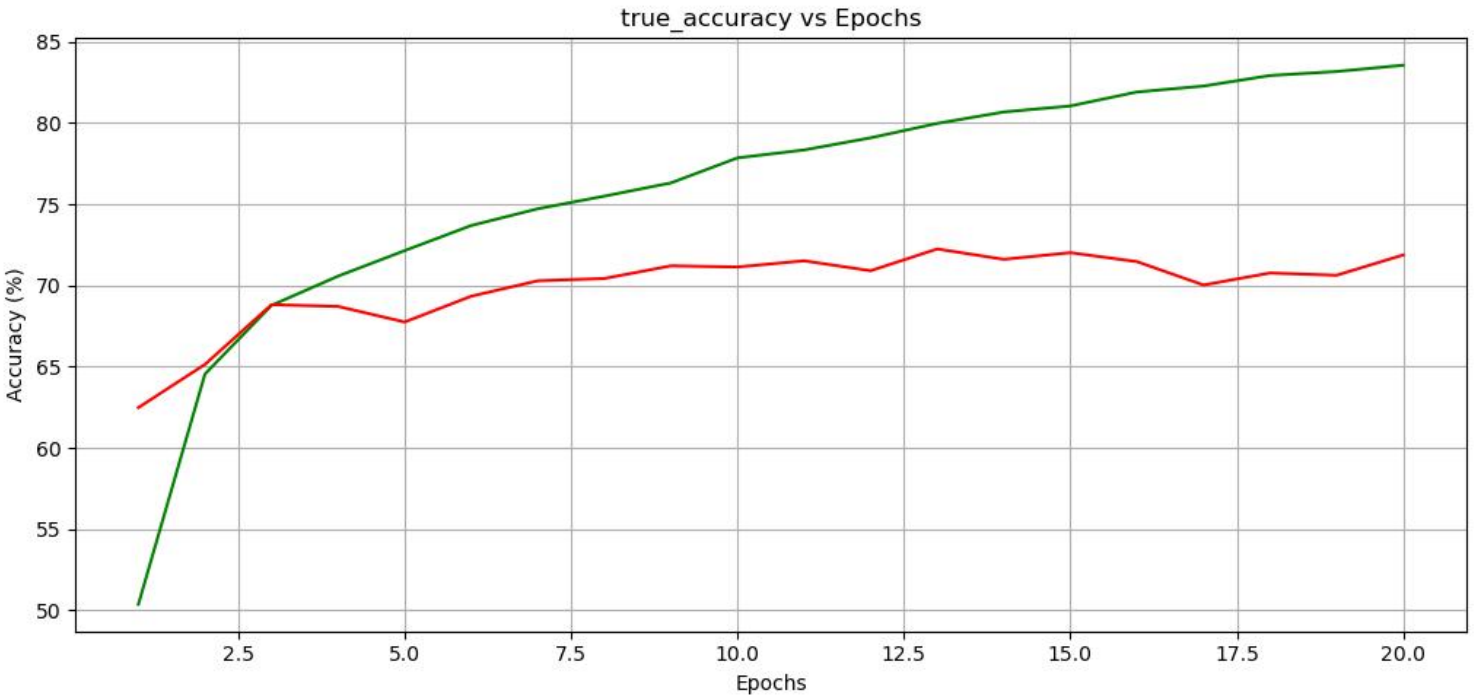
Cross-validation Accuracy: 56.74% ± 3.70%

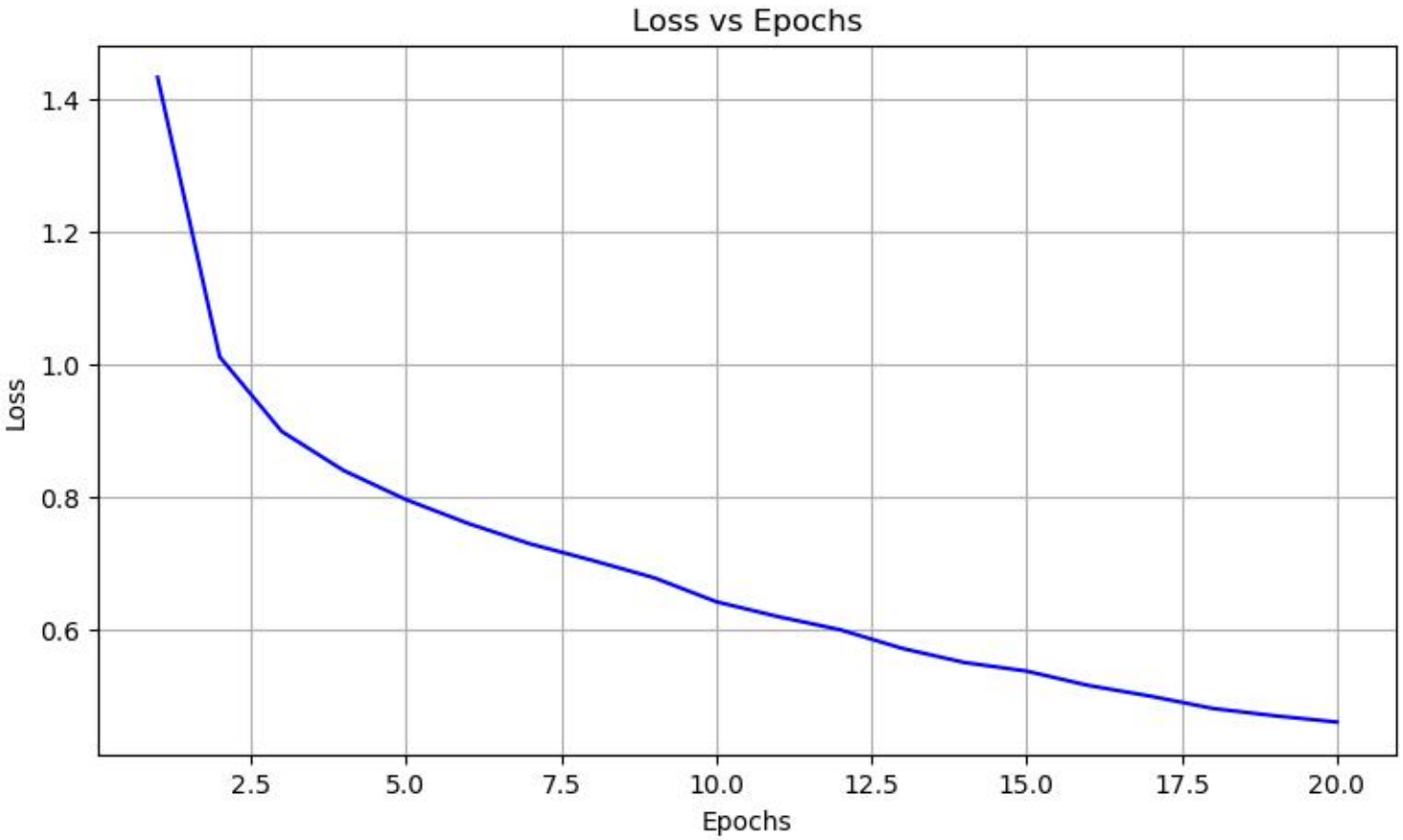




➢ 0.5

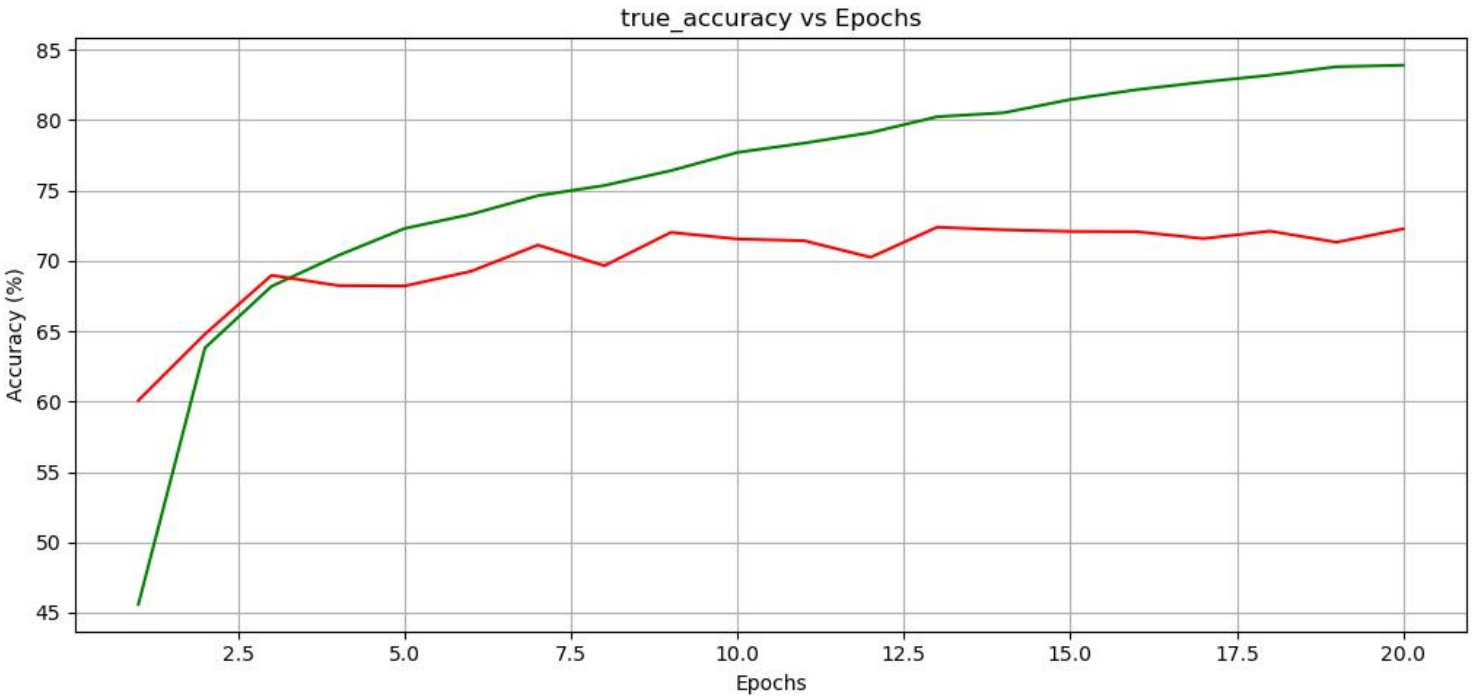
Cross-validation Accuracy: 55.13% ± 3.09%

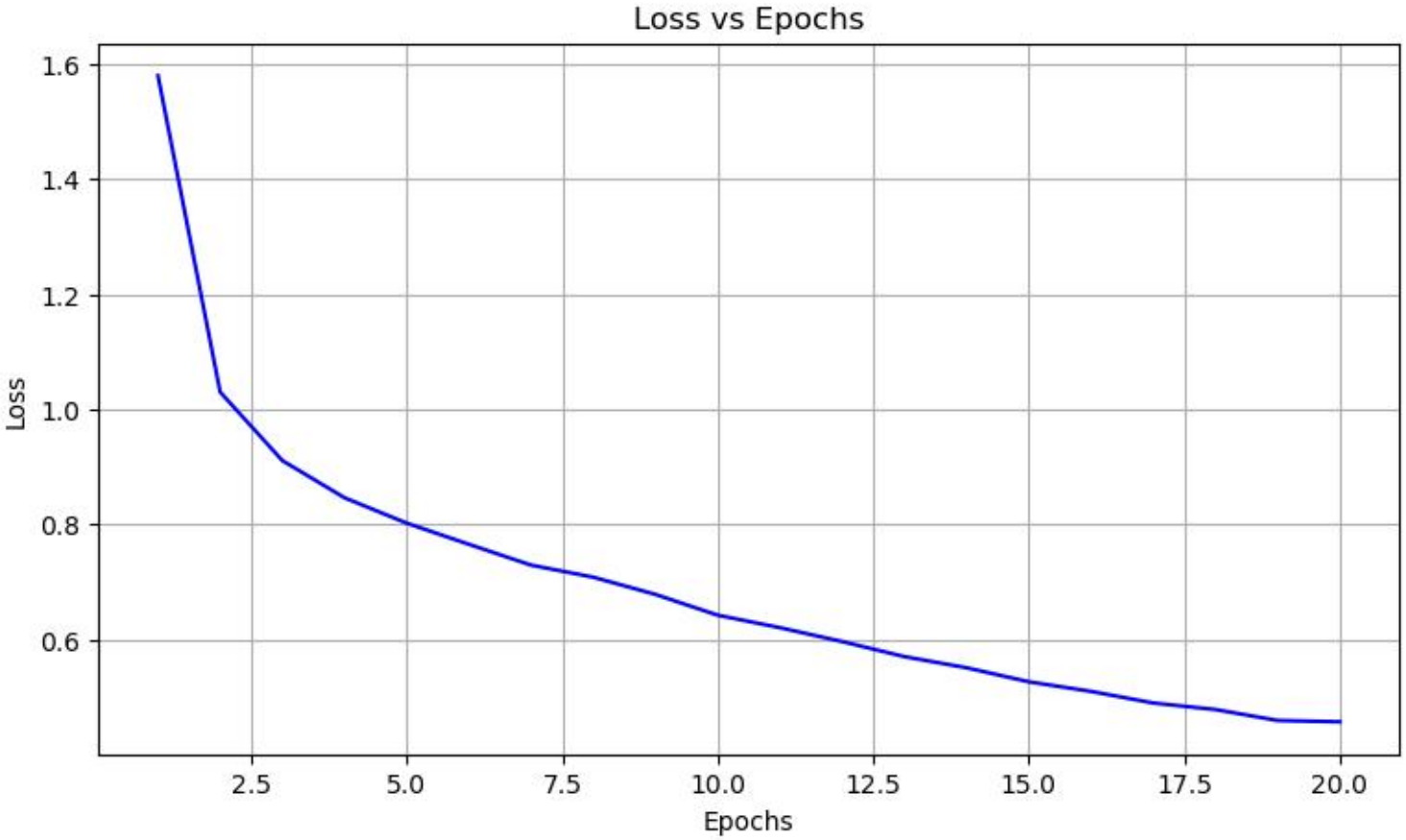




➢ 0.6

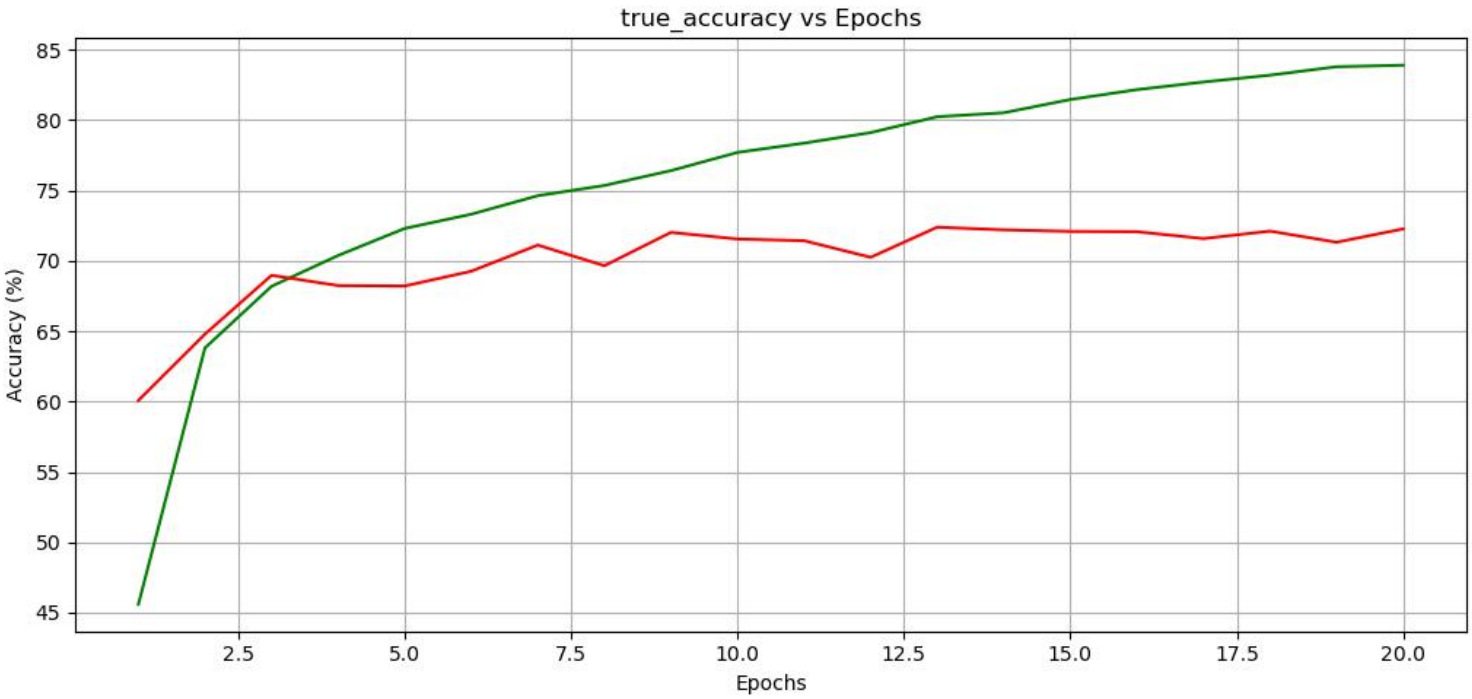
Cross-validation Accuracy: 54.31% ± 4.28%

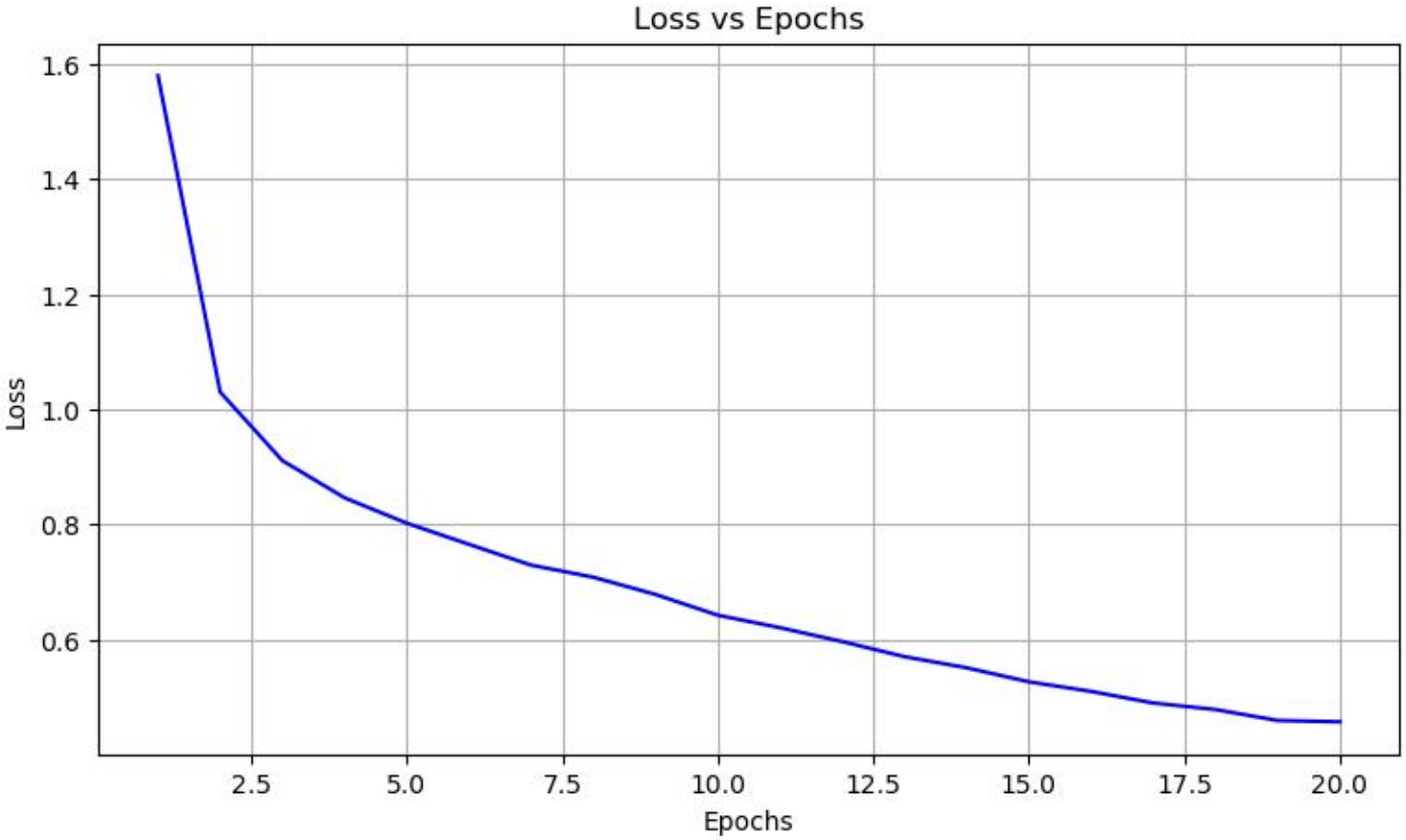




➢ 0.7

Cross-validation Accuracy: 53.87% ± 0.84%





Drop=0.4 和 0.5 都还行，0.4 准确率最高，0.5 最稳定。