模型训练相关参数配置

选项	选择参数
训练轮数	15 或 20 或 30
批大小	64
优化器	$Adam$
学习率	1e-3
数据集划分	训练集，验证集、测试集
模型选择方式	选择 `val` 上准确率最高的

线性层模型结构

MLP 最重要的网络结构为线性层，对于加入 $bias$ 的线性层来说，前向传播的数据变换公式为：

$$y_i = A_i x_i + b_i$$

而在代码中需要考虑 $batch$ 优先（即是第一个维度），因此公式应为：

$$y_i = x_i A_i^T + b_i$$

对于反向梯度传播，我们需要计算出每一个参数的梯度：

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial y_i} * \frac{\partial y_i}{\partial x_i} = \frac{\partial L}{\partial y_i}(A_i^T)^T = \frac{\partial L}{\partial y_i} A_i$$

$$\frac{\partial L}{\partial A_i} = \left(\frac{\partial L}{\partial A_i^T}\right)^T = \left(x_i^T \frac{\partial L}{\partial y_i}\right)^T = (\frac{\partial L}{\partial y_i})^T x_i$$

$$\frac{\partial L}{\partial b_i} = \left(\frac{\partial y_i}{\partial b_i}\right)^T * \frac{\partial L}{\partial y_i} = (1, 1, \cdots, 1) * \left(\frac{\partial L}{\partial y_i}\right) = \left(\frac{\partial L}{\partial y_i}\right).sum(0)$$

因此根据以上正向与反向传播的计算公式，便能够写出 $forward()$ 和 $backward()$ 两个函数，从而构建线性层模型。其中 $\partial L/\partial y_i$ 为已经传到当前层的梯度。

线性层模型结构

对应到代码上为：

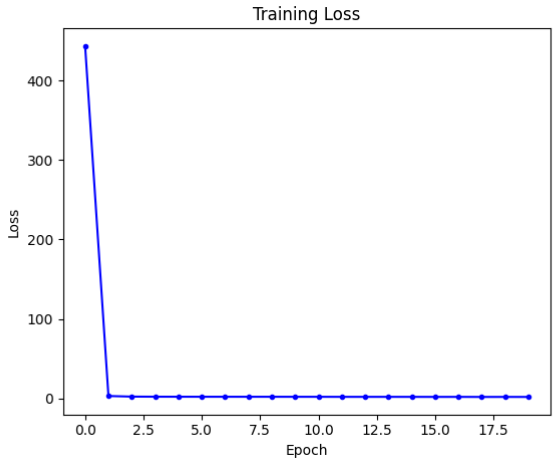
```python
class LinearFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weight, bias=None):
        # The forward pass can use ctx.
        ctx.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    @staticmethod
    def backward(ctx, grad_output):
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None
        if ctx.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0)

        return grad_input, grad_weight, grad_bias
```

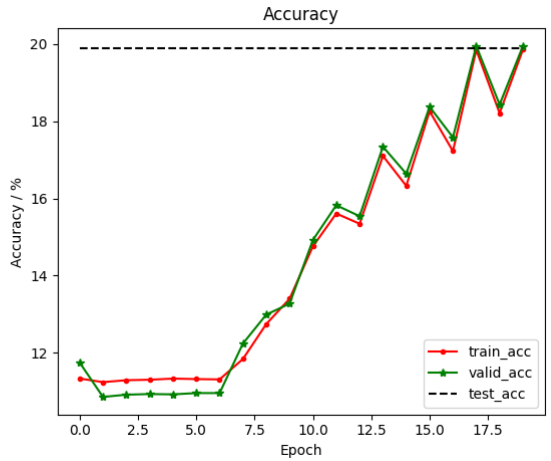
模型结构与问题——难以学习

在基础模块写好后，编写 5 层 MLP 模型，其具体参数结构为：

```
MLP(
  (model): Sequential(
    (0): LinearLayer(in_dim = 28*28, out_dim = 128, bias=True)
    (1): ReLU()
    (2): LinearLayer(in_dim = 128, out_dim = 64, bias=True)
    (3): ReLU()
    (4): LinearLayer(in_dim = 64, out_dim = 32, bias=True)
    (5): ReLU()
    (6): LinearLayer(in_dim = 32, out_dim = 16, bias=True)
    (7): ReLU()
    (8): LinearLayer(in_dim = 16, out_dim = 10, bias=True)
  )
)
```

其中模型参数的初始化是随机的：$torch.randn()$。直接进行训练观察结果：





模型结构与问题——难以学习

通过观察结果发现模型在刚开始并未有很好的效果，直到第六个 $epoch$ 后才开始学到一些内容，但学习效率很低。显然这并不是想得到的结果，尤其是对于 $MNIST$ 这个简单的数据集来说。

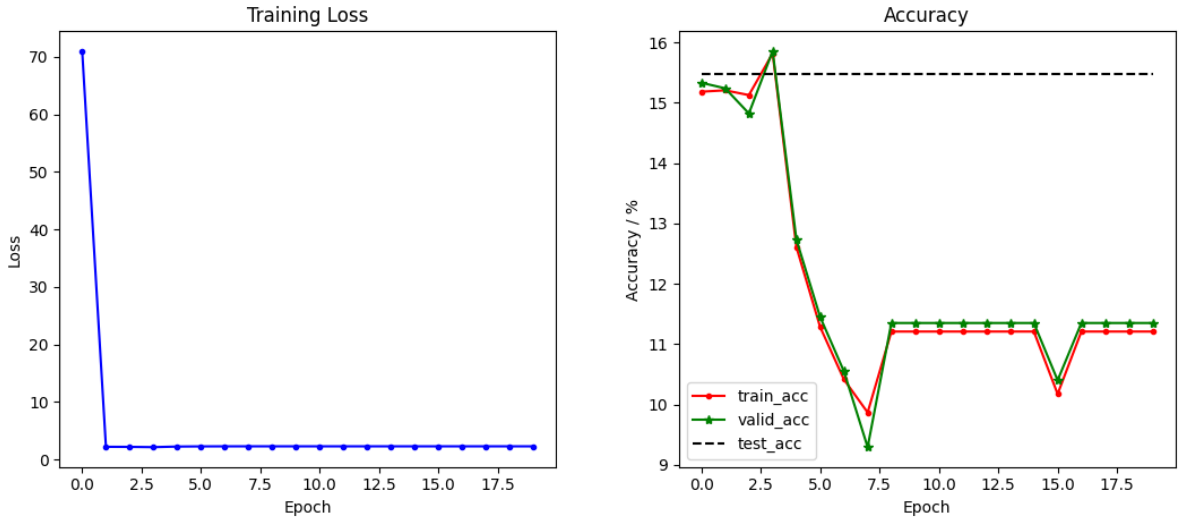
使用 $hook$ 工具观察反向传播最远处的梯度值，即离输出最近的网络模型梯度信息，发现模型在起初介与小梯度和无梯度之间徘徊，直到后期效果变好才慢慢有稳定的梯度。

```
Gradients: (tensor([[ 0.0000,  0.0000, -0.0031,  ...,  0.0000, -0.0008,  0.0023],
        [ 0.0000,  0.0000, -0.0129,  ..., -0.0006,  0.0037,  0.0013],
        [-0.0048,  0.0000,  0.0000,  ..., -0.0038,  0.0010,  0.0000],
        ...,
        [ 0.0000,  0.0000,  0.0000,  ..., -0.0013,  0.0006, -0.0024],
        [ 0.0000,  0.0000, -0.0011,  ...,  0.0034, -0.0026,  0.0000],
        [-0.0018,  0.0000,  0.0021,  ..., -0.0007,  0.0005,  0.0000]],
       device='cuda:0'),)
```

```
Gradients: (tensor([[ 4.4293e-03,  0.0000e+00,  0.0000e+00,  ...,  5.3830e-04,
         -1.3395e-03,  0.0000e+00],
        [ 0.0000e+00, -2.4994e-03,  0.0000e+00,  ...,  2.3082e-03,
         -8.8744e-04, -7.1606e-03],
        [ 1.9665e-03,  0.0000e+00,  0.0000e+00,  ...,  0.0000e+00,
          8.0711e-04,  7.1104e-04],
        ...,
        [-2.4315e-03,  0.0000e+00, -1.3187e-03,  ...,  0.0000e+00,
          5.3508e-04,  8.1932e-04],
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  ...,  9.5049e-04,
          2.1525e-04, -1.2636e-03],
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  ...,  6.2950e-04,
          7.2763e-05,  0.0000e+00]], device='cuda:0'),)
```

模型结构与问题——难以学习

考虑其形成原因，猜测是参数未初始化可能进入了模型的鞍点，导致一直难以逃离鞍点的束缚。当逐渐远离鞍点后可能会存在模型改善的情况。在这里梯度消失的原因我认为与后面 41 层网络梯度消失的原因是不同的。
当然还有可能是学习率过小导致无法快速学习。因此我将学习率从 0.001 增大到 0.01，再次运行程序：

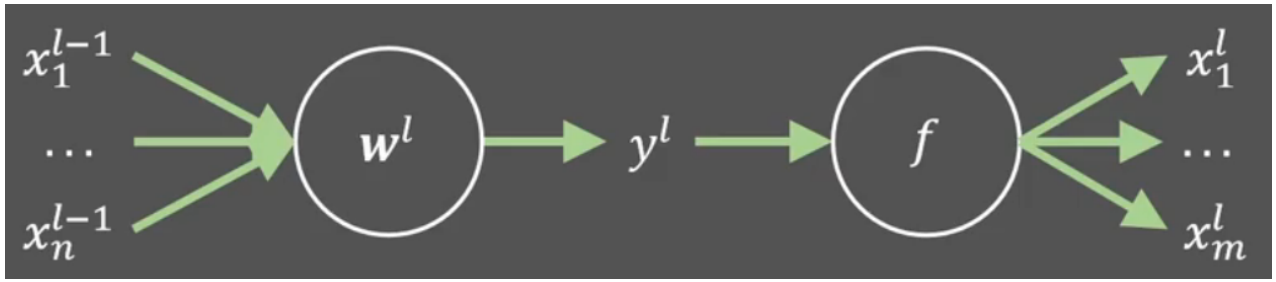


发现效果人没有改观，因此尝试对模型进行参数初始化。特别的，因为在这个模型中激活函数选择的是 $RELU$，因此采用 He 初始化方法。

He 初始化

He 初始化方法是在 2015 年提出的 [1]，特别针对激活函数为 $RELU$ 的网络进行初始化。下面给出推导过程。

考虑一层网络模型：



我们需要始终记住的假设（删除线是从 $Xavier$ 假设不需要的部分）：

- ~~$E(x) = 0$~~，$E(w) = 0$
- ~~f 是对称的且满足 $f'(0) = 1$~~
- x 的方差相等

以及三个重要的数学公式：

- $\mathrm{Var}(X) = E(X^2) - E(X)^2$
- $\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$，当 X 和 Y 互相独立
- $mathrmVar(XY) = \mathrm{Var}(X)\mathrm{Var}(Y) + \mathrm{Var}(X)E(Y)^2 + \mathrm{Var}(Y)E(X)^2$

He 初始化

$$
\begin{aligned}
\mathrm{Var}(y^l) =& n\mathrm{Var}(w_i^l x_i^{l-1}) \\
=& n[\mathrm{Var}(w_i^l)\mathrm{Var}(x_i^{l-1}) + \mathrm{Var}(w_i^l)E(x_i^{l-1})^2 + \mathrm{Var}(x_i^{l-1})E(w_i^l)^2] \\
=& n[\mathrm{Var}(w_i^l)\mathrm{Var}(x_i^{l-1}) + \mathrm{Var}(w_i^l)E(x_i^{l-1})^2] \\
=& n[\mathrm{Var}(w_i^l)E[(x_i^{l-1})^2] - \mathrm{Var}(w_i^l)E(x_i^{l-1})^2 + \mathrm{Var}(w_i^l)E(x_i^{l-1})^2] \\
=& n[\mathrm{Var}(w_i^l)E[(x_i^{l-1})^2]] \\
=& n\mathrm{Var}(w_t^l)E[(f(y^{l-1}))^2]
\end{aligned}
$$

又通过期望的定义可以写成:

$$
\begin{aligned}
E[(f(y^{l-1}))^2] &= \int_{-\infty}^{+\infty} f(y^{l-1})^2 p(y^{l-1}) d(y^{l-1}) \\
&= \int_{0}^{+\infty} (y^{l-1})^2 p(y^{l-1}) d(y^{l-1}) \\
&= E[(y^{l-1})^2] = \mathrm{Var}(y^{l-1}) + E[(y^{l-1})]^2 = \mathrm{Var}(y^{l-1})
\end{aligned}
$$

所以我们的方差公式变为:

$$
\mathrm{Var}(y^l) = \frac{n}{2}\mathrm{Var}(w_t^l)\mathrm{Var}(y^{l-1})
$$

He 初始化

$$\mathrm{Var}(y^l) = \frac{n}{2}\mathrm{Var}(w_t^l)\mathrm{Var}(y^{l-1})$$

特别的，我们希望保方差，即 $\mathrm{Var}(y^l) = \mathrm{Var}(y^{l-1})$，因此我们能够得到重要结论：

$$\mathrm{Var}(w_t^l) = \frac{2}{n}$$

所以 He 初始化即让参数满足方差为 $\frac{2}{n}$ 的分布，常见的有均值和正态两种选择：

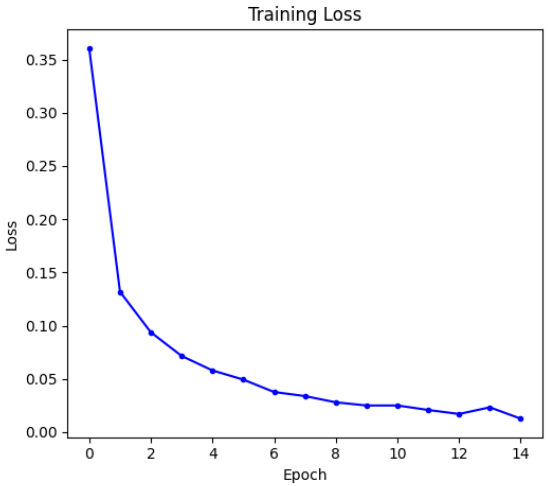
$$\mathcal{N}(0, \frac{2}{n}) \quad \mathcal{U}(-\sqrt{\frac{6}{n}}, \sqrt{\frac{6}{n}})$$

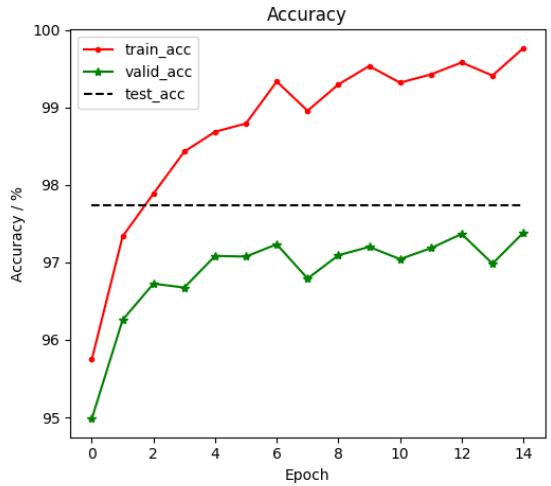
对应到代码中比较简单，因为有函数接口：

```
def reset_parameters(self):
    # He Initialization for weights
    init.kaiming_normal_(self.weight, mode='fan_in', nonlinearity='relu')
    init.zeros_(self.bias)
```

模型效果优化

将加入初始化后的模型运行，观察结果：





能够发现效果有了很明显的改善——**从第一步开始起点就要远高于随机初始化的结果**。

```
Epoch 11/15, Loss:  0.024969, train_acc: 99.32%, val_acc: 97.04%
Epoch 12/15, Loss:  0.020736, train_acc: 99.42%, val_acc: 97.18%
Epoch 13/15, Loss:  0.016992, train_acc: 99.58%, val_acc: 97.37%
Epoch 14/15, Loss:  0.023223, train_acc: 99.41%, val_acc: 96.98%
Epoch 15/15, Loss:  0.012574, train_acc: 99.76%, val_acc: 97.38%
Training complete.
test_acc: 97.74%
```

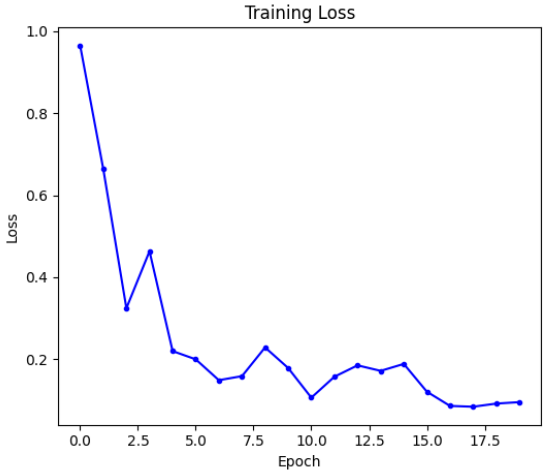
原模型效率低下的情况得到解决，方式即为**初始化模型参数**。

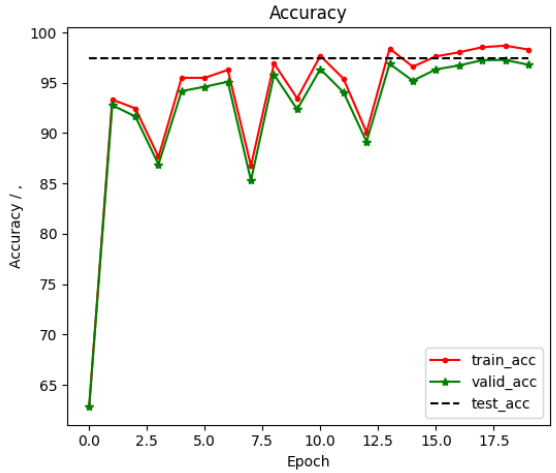
模型结构与问题——震荡收敛

吸取之前训练网络的教训，此处的线性层也使用 He 初始化。
22 层 MLP 的模型结构与 5 层类似，均是一层线性层一层 $RELU$ 激活。只不过增加了层数。特别的其中间每一层的 $hidden\_size$ 为：

$$hidden\_sizes = [28 * 28, 1024, 1024, 2048, 512, 512, 256, 256, 128, 128, 128, 64, 64, 64, 32, 32, 32, 16, 16, 16, 12]$$

整体的网络结构是先变宽后变窄的形状。
直接运行层数变深后的代码，观察运行结果：





模型结构与问题——震荡收敛

```
Epoch 16/20, Loss:  0.121210, train_acc: 97.62%, val_acc: 96.33%
Epoch 17/20, Loss:  0.086539, train_acc: 98.03%, val_acc: 96.72%
Epoch 18/20, Loss:  0.084563, train_acc: 98.53%, val_acc: 97.24%
Epoch 19/20, Loss:  0.092200, train_acc: 98.69%, val_acc: 97.27%
Epoch 20/20, Loss:  0.095773, train_acc: 98.30%, val_acc: 96.79%
Training complete.
test_acc: 97.40%
```

通过结果能够看出模型效果不错，准确率也很高。但观察训练过程能够发现有明显的震荡过程——即存在在某几步之后准确率突然下降的情况，甚至能直接下降 10% ，这说明模型的稳定性是不足的。
因为参数的改变依靠梯度，因此考虑是梯度的不稳定导致训练过程中的震荡效果。其解决方式猜想有两种:

- 减小学习率
- 加入归一化层

因为我希望能够以相同的 epoch 数目得到较好的效果，即不增加训练轮数。而减小学习率代表了收敛会变慢，因此我下面采用加入“batch norm”的方式来尝试解决这个问题。

BN 模型结构

一般来说，如果模型的输入特征不相关且满足标准正态分布 $N(0,1)$ 时，模型的表现一般较好。我们可以对数据预处理使其尽量满足这个条件，但随着网络的加深，网络的非线性变换使得每一层的结果变得相关了，且不再满足正态分布。甚至隐藏层的特征分布已经发生了偏移。

为了解决这一问题，批归一化通过标准化每一层特征的分布，使得：

- 使得模型训练收敛的速度更快
- 模型隐藏输出特征的分布更稳定，更利于模型的学习

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

BN 模型结构

那么究竟有哪些参数需要反传梯度？

- x_i 当然要反传梯度
- γ, β 是可学习参数，要反传梯度
- 特别的，μ_B, σ_B 也是 x_i 的函数因此要反传梯度

BN 模型结构

通过链式法则求解梯度：

$$\frac{\partial \ell}{\partial \widehat{x_i}} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x_i}} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2} (\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x_i}} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x_i}} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x_i}$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

BN 模型结构

值得注意的一点：在推理过程中传入的是批数据，因此批均值。批方差是能够算出来的。但在 $inference$ 过程中，输入的可能只是一个数据，因此在这种情况下上述两个参数是算不出来的。为了解决这个问题，需要引入两个全局均值与方差：$\hat{\mu}, \hat{\sigma}^2$。
这两个变量在训练每一批数据时候要更新。具体更新过程为：

$$\begin{cases} \hat{\mu} = (1-q) * \hat{\mu} + q * \mu_B \\ \hat{\sigma}^2 = (1-q) * \hat{\sigma}^2 + q * \sigma^2 \end{cases}$$

当在推理过程中便要用到这两个参数进行归一化。
注意：这两个参数不是可学习参数，但在保存模型时仍应该保存，否则在推理过程中若这两个值丢失，会导致训练效果差。

BN 模型结构——向前传播

```python
class BatchNorm1dFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, gamma, beta, moving_mean, moving_var, grad_enable, eps=1e-5, momentum=0.1):
        if not grad_enable:
            # print(len(moving_mean[0]), moving_var[0])
            x_normalized = (x - moving_mean[0]) / torch.sqrt(moving_var[0] + eps)
        else:
            ctx.eps = eps
            ctx.momentum = momentum
            ctx.save_for_backward(x, gamma, beta)

            mean = x.mean(dim=0, keepdim=True)
            var = x.var(dim=0, unbiased=False, keepdim=True)
            x_normalized = (x - mean) / torch.sqrt(var + eps)
            moving_mean[0] = (1 - momentum) * moving_mean[0] + momentum * mean
            moving_var[0] = (1 - momentum) * moving_var[0] + momentum * var
            ctx.save_for_backward(x, gamma, beta, mean, var, x_normalized)

        y = gamma * x_normalized + beta
        return y
```

BN 模型结构——向后传播

```python
@staticmethod
def backward(ctx, grad_output):
    x, gamma, beta, mean, var, x_normalized = ctx.saved_tensors

    N = x.size(0)
    inv_var = 1.0 / torch.sqrt(var + ctx.eps)

    grad_gamma = (grad_output * x_normalized).sum(dim=0, keepdim=True)
    grad_beta = grad_output.sum(dim=0, keepdim=True)
    grad_x_normalized = grad_output * gamma

    grad_var = (grad_x_normalized * (x - mean) * -0.5 * inv_var**3).sum(dim=0, keepdim=True)  # inv_var**3 = (var + ctx.eps)**(-1.5)
    grad_mean = (grad_x_normalized * -inv_var).sum(dim=0, keepdim=True) + grad_var * (x - mean).mean(dim=0, keepdim=True) * -2.0 / N
    grad_x = grad_x_normalized * inv_var + grad_var * 2.0 * (x - mean) / N + grad_mean / N

    return grad_x, grad_gamma, grad_beta, None, None, None
```

模型修正与改善

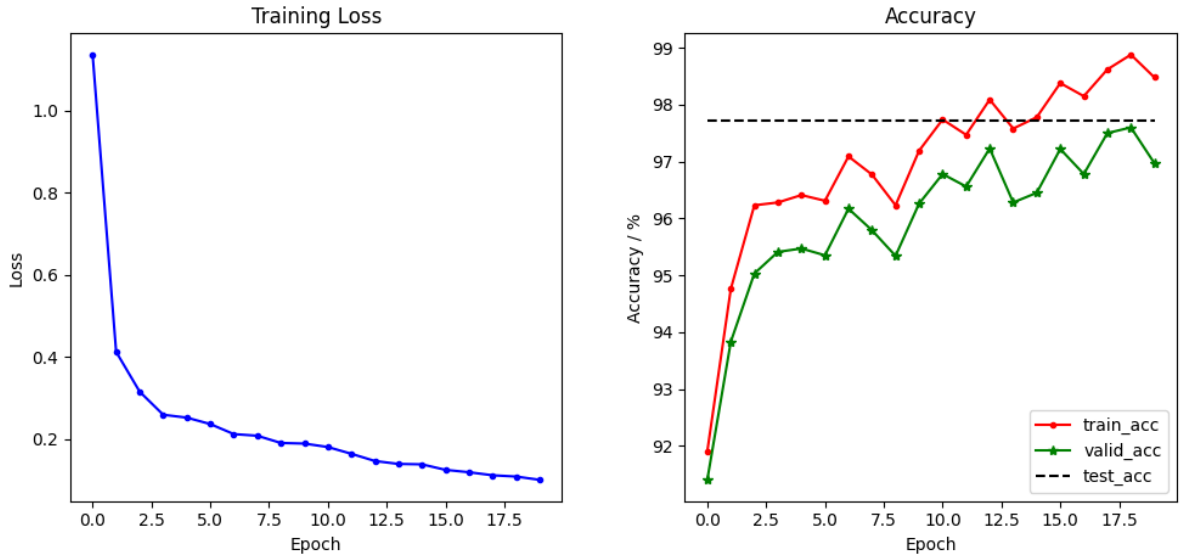
更改网络模型为：

```
MLP(
  (model): Sequential(
    (0): Linear(in_features=28*28, out_features=28*28, bias=True),
    (1): ReLU(),
    (2): BatchNorm1d(28*28),
    (3): Linear(in_features=28*28, out_features=1024, bias=True),
    (4): nn.ReLU(),
    (5): BatchNorm1d(1024),
    # ......
    (60): Linear(in_features=12, out_features=12, bias=True),
    (61): ReLU(),
    (62): BatchNorm1d(12)
    (63): Linear(in_features=12, out_features=10, bias=True)
    )
)
```

即每次激活后加入 BN 层对数据进行调整。

模型修正与改善

观察模型结果：



效果依旧不错，震荡情况大幅改善了。尽管也有训练震荡，但已经在可接受范围内——从原本的 10% 下降到近 1% ，大大改善了训练效果。

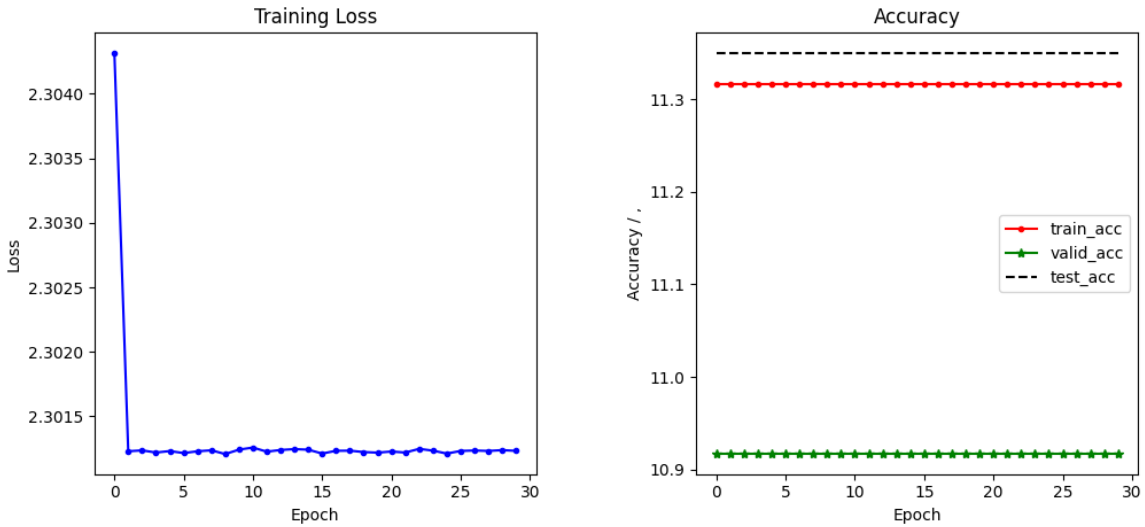
尽管是否加入 BN 结果很相近，但因为数据集过于简单，导致模型在 2 3 $epoch$ 就已经达到了较好的情况。而对于更复杂的数据集是不会这样的，而震荡的训练不利于模型的收敛，因此加入批归一化后的效果事实上是优于不加归一化的。

模型结构与问题——梯度消失

在 22 层中，BN 使得模型更加稳健，但事实上不加其结果也不错，只不过过程比较崎岖。因此对于更深层次的网络，首先尝试不加 BN 层，直接在原基础上加深网络。特别的，隐藏层大小参数为：

$$\begin{aligned} hidden\_sizes =& [28*28, 1024, 1024, 512, 512] + [256, 512, 1024, 512, 256] * 5 \\ &+ [64, 64, 64, 32, 32, 32, 16, 16, 12, 12] \end{aligned}$$

共 41 层 MLP。
观察训练结果：



模型结构与问题——梯度消失

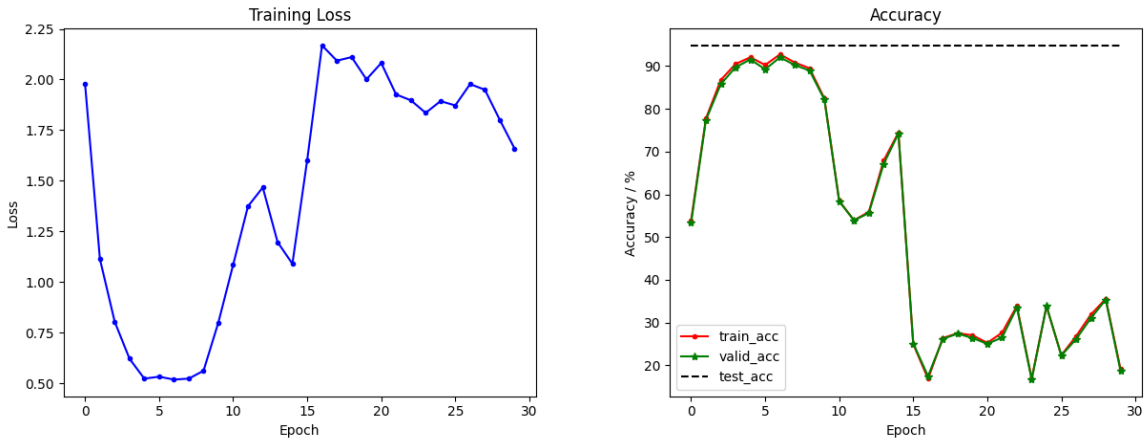
能够观察到模型根本没有学习，与之前 5 层 MLP 不同，其正确率始终几乎不波动。因为网络过深，因此考虑是存在梯度消失的情况。使用 $hook$ 进行查看：

```
-5.7788e-05, -1.5397e-05]], device='cuda:0'),)
Gradients: (tensor([[-2.1924e-05, -1.2902e-05,  0.0000e+00,  ...,  2.6019e-06,
         -2.5385e-05,  0.0000e+00],
        [ 2.1589e-06,  1.0188e-05,  0.0000e+00,  ...,  0.0000e+00,
          0.0000e+00,  3.3610e-06],
        [-5.5153e-06,  0.0000e+00,  0.0000e+00,  ..., -1.4065e-05,
         -3.5968e-05,  0.0000e+00],
        ...,
        [-2.8958e-05,  0.0000e+00,  0.0000e+00,  ...,  0.0000e+00,
          0.0000e+00,  8.3317e-07],
        [ 9.9271e-06,  0.0000e+00,  0.0000e+00,  ..., -3.7589e-06,
          1.3643e-05,  0.0000e+00],
        [-1.5427e-06,  0.0000e+00,  3.2996e-08,  ...,  0.0000e+00,
          0.0000e+00, -2.9689e-05]], device='cuda:0'),)
Gradients: (tensor([[ 1.5033e-05, -3.4242e-05,  0.0000e+00,  ...,  1.4805e-05,
          0.0000e+00,  0.0000e+00],
        [ 8.3613e-05,  0.0000e+00,  0.0000e+00,  ...,  0.0000e+00,
          7.2148e-05,  0.0000e+00],
        [ 0.0000e+00, -9.2777e-06,  0.0000e+00,  ..., -2.5303e-05,
          0.0000e+00,  0.0000e+00],
        ...,
        [-1.6583e-05, -1.7396e-05,  0.0000e+00,  ...,  7.7239e-05,
          0.0000e+00,  4.8927e-05],
        [ 9.5275e-05,  0.0000e+00,  0.0000e+00,  ..., -8.5893e-05,
         -1.4127e-05,  0.0000e+00],
        [-5.7867e-05,  0.0000e+00,  0.0000e+00,  ...,  3.4328e-05,
          0.0000e+00,  0.0000e+00]], device='cuda:0'),)
```

发现确实梯度几乎都为 0，这也解释了模型为什么始终不进步。

加入 BN 层——模型退化

此网络的特点就是深度很深，是 20 层的两倍，是 5 层的八倍！因此面对如此深得网络，若干小于 1 的梯度相乘导致梯度消失。为了解决这个问题，将模型内加入 BN 层，从而缓解梯度消失问题。再次运行查看结果：

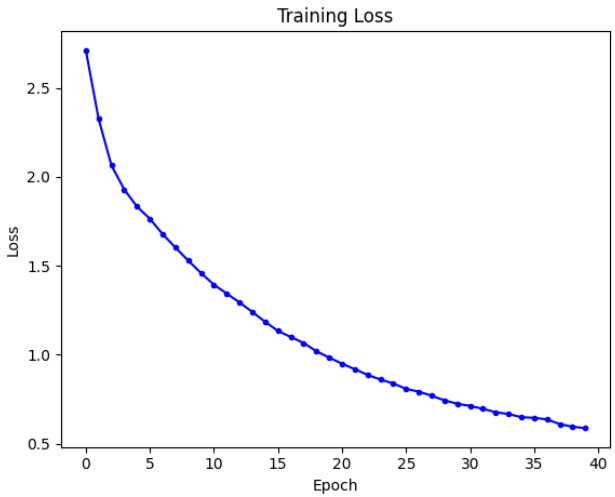


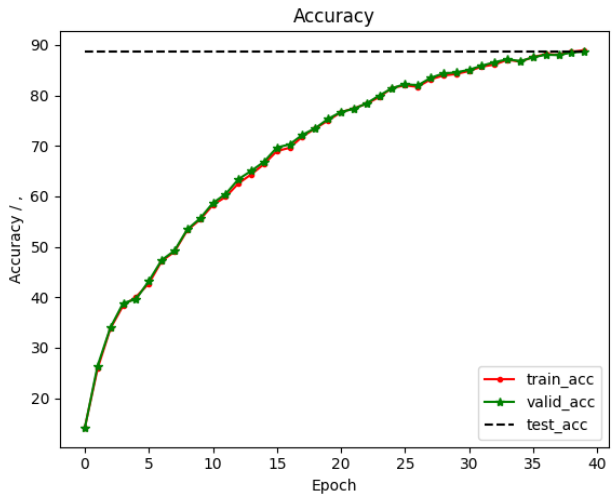
尽管 test 的结果依旧很高，但这是因为我选取的是 val 正确率高的模型。但从图中看，第 7 个 $epoch$ 后模型便开始退化了。正确率越来越不准，甚至比不训练的第一步还要效果差。

起初怀疑是模型代码写错了，全部替换成官方的代码后，依旧出现退化情况，因此问题在其他地方。

退化问题分析

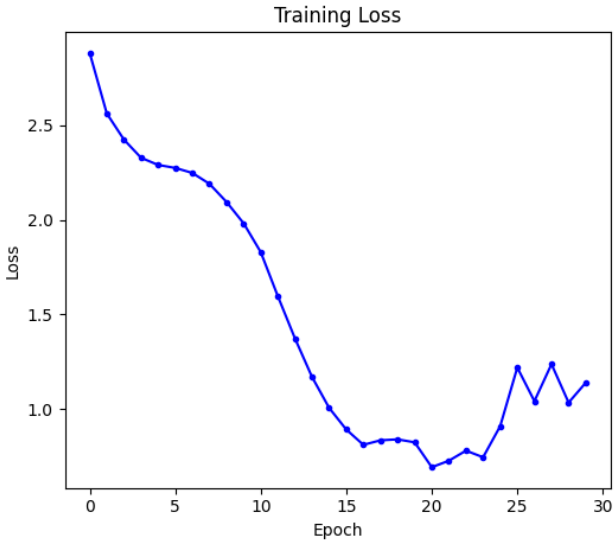
因为我观察到模型是先收敛再发散，因此我想探究这份收敛是不是偶然。多次运行后都是先收敛后发散，因此我怀疑收敛不是偶然！我选用收敛速度相较于 $Adam$ 慢的 SGD 优化器进行训练，结果表明，模型的收敛确实是必然，在模型未达到最优情况时是能够稳步收敛的。

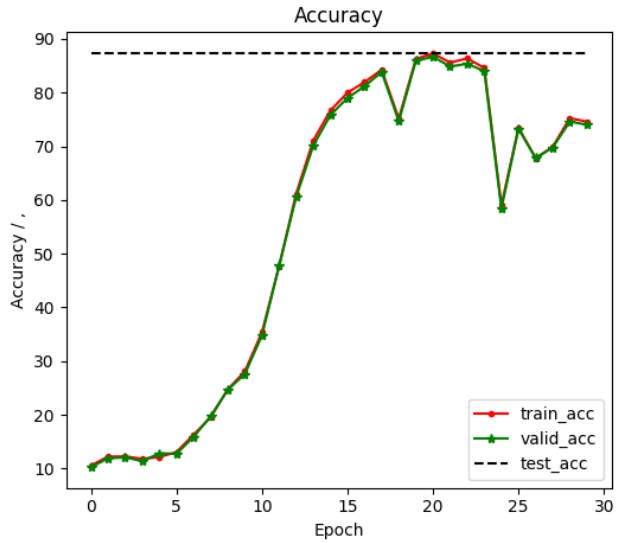




退化问题分析

那么既然其会收敛，我推测是因为如此深的网络拟合后应该是一个极值点复杂的情况，又因为模型在努力收敛到极值点，因此如果收敛后极值对函数值很敏感——即微微的参数变化就能游离出极值点，便可能会导致这种越来越不准的情况。
所以初步解决措施也就呼之欲出了——减小学习率。





能够发现模型起初收敛，收敛后又开始进行退化，只不过退化的时间步滞后了。一味减小学习率是不现实的，需要自动调整学习率。

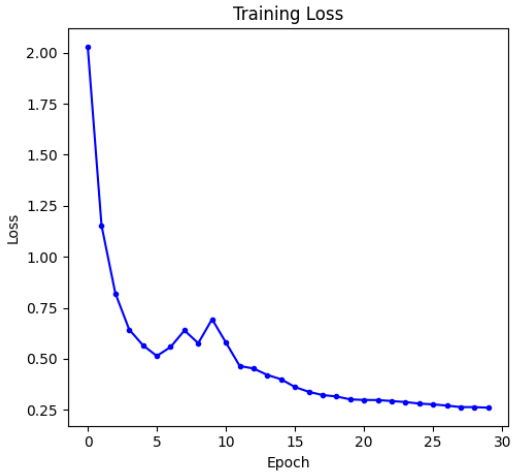
自适应调整学习率法

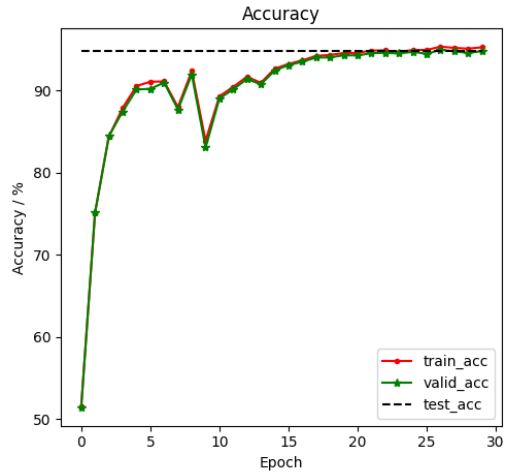
选用学习率调整函数；

```
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min',
patience=3, factor=0.1, verbose=True)
```

其含义为：$scheduler$ 检测一个值，这个值越小越好。当这个值" 未变小三次时"，便调整学习率。这里我设置其检测的变量是 val 集合的 $loss$ 值。
观察模型效果。

自适应调整学习率法





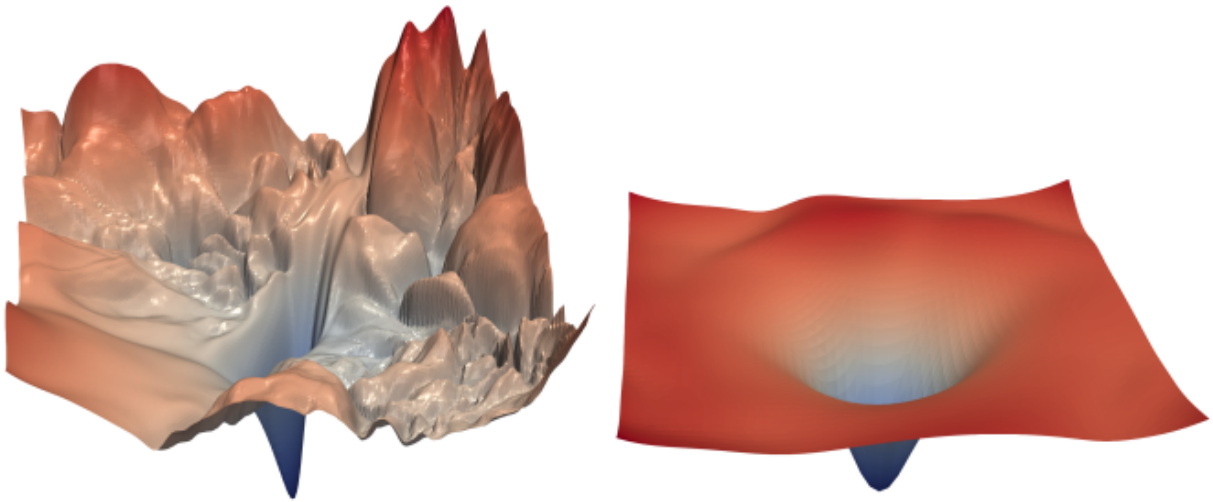

```
Epoch 9/30, Loss:  0.576400, train_acc: 92.39%, val_acc: 91.97%
Epoch 10/30, Loss:  0.693940, train_acc: 83.91%, val_acc: 83.07%
Epoch 00010: reducing learning rate of group 0 to 1.0000e-04.
Epoch 11/30, Loss:  0.579648, train_acc: 89.31%, val_acc: 89.03%
Epoch 12/30, Loss:  0.464598, train_acc: 90.46%, val_acc: 90.17%
Epoch 13/30, Loss:  0.452258, train_acc: 91.69%, val_acc: 91.41%
Epoch 14/30, Loss:  0.420641, train_acc: 90.95%, val_acc: 90.77%
Epoch 00014: reducing learning rate of group 0 to 1.0000e-05.
Epoch 15/30, Loss:  0.399345, train_acc: 92.67%, val_acc: 92.45%
```

注意到模型调整了两次的学习率，分别在 10 和 14 轮。观察这两步的上下情况，发现在到达这个步数之前模型效果均有变差迹象，而在调整学习率后模型有很快的改善了效果，使得效果比调整前的最优值还要好。仿佛就是当模型变差之时，学习率调整成功将模型从变差边缘拉了回来，进一步收敛。

这也进一步印证了之前的猜测——模型的变差与学习率有着密切的关系。

残差连接法

受到残差网络的启发 [2]，残差结构能够很好地缓解因为网络过深而导致的梯度消失问题。用论文 [3] 中的截图能够直观的观察到有着残差结构的网络其可行域表面变得更加光滑，即利于网络优化。



(a) without skip connections (b) with skip connections

进一步考虑反向传播情况，在反向传梯度时，第 l 层的梯度不止传给上一层，还可以通过残差连接传给较远的一层网络——这样尽管那层网络离输出较远，但因为残差连接的缘故，梯度能够很好地通过这个桥梁传到前面。

残差连接法

首先声明一个 $block$ 块的结构

$$block = linear + relu + BN$$

接下来便可以以其为最小单元构建残差模块

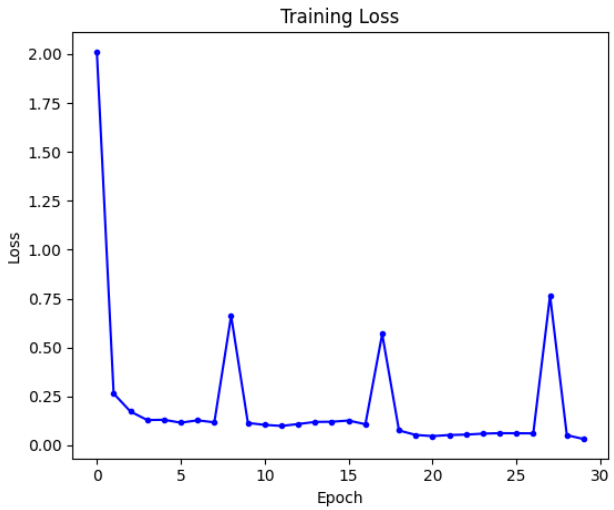
```
    ____________________________ Linear ____________________________
    |                                                               |
x->--------block------block------block------block------block-------- + ----> y
```

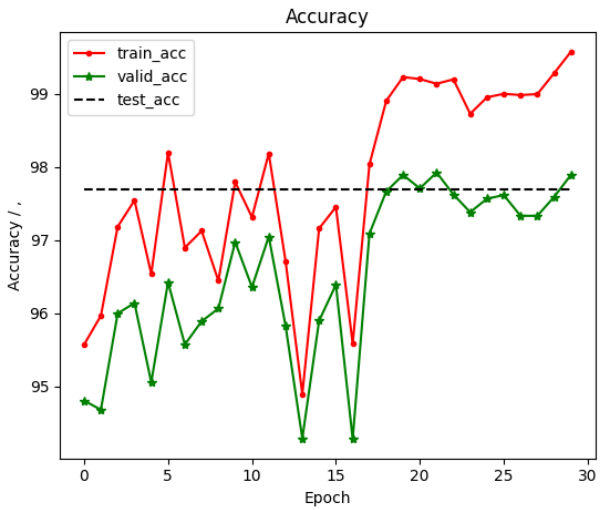
每五个块进行一个残差连接形成一个残差模块。其中在支路上的线性层是为了保持数据对齐，只有相同 $shape$ 的数据才可以相加。
整个模型共由八个残差块组成，再在残差部分整体的前后各添加两个线性层作为 $begin$ 和 end 模型结构就完整了。

```
x ->-- begin -> residual1 -> residual2 -> ... ->--residual7 -> end -> pred
```

残差连接法

运行模型查看结果：





能够发现模型效果也很好，甚至比之前效果更优秀。尽管存在一定的波动，但仍在正常范围内——正确率波动不超过 3%，可以接受。且能够观察到模型收敛速度很快。

残差连接法

综上我用了两种方式最终成功训练出了较深的 MLP 网络。

- 改变学习率：这种方式不需要增加额外的层运用改变学习率的 $trick$ 得到了较好的效果。
- 残差结构：通过引入残差结果，将梯度能够完整的传到较远模型区域，缓解了模型过深导致的梯度消失问题。其实残差连接的网络深度可以理解为变浅了，不再是像第一种方式“纯粹”的深网络。

模型参数数量

模型	参数数量	模型层数
5层MLP，无 BN 层	111514	9
5层MLP，有 BN 层	111994	13
22层MLP，无 BN 层	6165038	43
22层MLP，有 BN 层	6179334	64
41层MLP，无 BN 层	10246202	80
41层MLP，有 BN 层	10280202	120
41层MLP，残差连接	10758910	-----

万能近似定理

Cybenko 等人于 1989 年证明了具有隐含层 (最少一层) 感知机神经网络在激励函数 (也称激活函数) 为 $Sigmoid$ 函数的情况下具有逼近任何函数的作用 [4]。

Hornik 等人在 1991 年进一步推广了其结论，得到著名的万能近似定理（universal approximation theorem）[5]：

定理 (Universal Approximation Theorem)

一个前馈神经网络如果具有线性输出层和至少一层具有任何一种”挤压”性质的激活函数（非常数函数）的隐藏层，只要给予网络足够数量的隐藏单元，它可以以任意的精度来近似任何从一个有限维空间到另一个有限维空间的 $Borel$ 可测函数。

所以这个定理事实上已经告诉我们：无论我们试图学习什么函数，大的 MLP（宽度）一定能够表示这个函数。但事实上学习过程总是失败的：

- 用于训练的优化算法可能找不到用于期望函数的参数值
- 训练算法可能由于过拟合而选择了错误的函数

具有单层的前馈网络足以表示任何函数，但是网络层可能大得不可实现，并且可能无法正确地学习和泛化——因此我们在深度上进一步探索，试图用深的网络得到更好的效果。

深度 $RELU$ 网络的拟合能力

关于深度网络，也有诸多研究。对于深层网络来说：在神经元或参数数量相同的前提下，增加深度相比增加宽度能够获得更多的表达能力。例如同样拟合一个目标函数 f ，使得误差小于 ϵ ，二层 MLP 需要 1000 个神经元，但是深度 MLP 可能只需要 100 个神经元。

在 Yarotsky 论文中 [6]，作者在 $Sobolev$ 空间的逼近设置中建立了新的严格上下界，证明了深度 ReLU 网络比浅层网络更有效地逼近平滑函数。
在 Lu 等人的论文中 [7]，文章证明存在一些宽网络类别，这些网络不能被深度不超过多项式界的窄网络实现。但在另一方面，文章通过大量实验表明，尺寸超过多项式界的窄网络可以高精度地逼近宽和浅的网络。文章最终说明：对于 $RELU$ 网络的表达能力而言，深度比宽度更有效。

个人想法

从万能逼近理论我们可以知道理论上浅层 MLP 加上激活函数就可以拟合任意函数，但是现实中的计算效率、计算工具、时间成本等因素让我们不可能按照理论一般训练任意宽度的神经网络。

为了更好的训练效果，便开始在深度上下功夫，其确实有了更好的表现。但这不代表越深越好。尽管从认知上感觉越深拟合能力越强，但现实的训练难度也确实展现出来。包括这次的大作业，我们已经能够深切的感受到训练一个深度网络不是简单的把浅层网络进行延续，我们仍面临更多问题。而这也就引导了例如残差网络的诞生。

当然越深、越宽的网络不一定是最好的，是要按照现实任务来决定。
观察机器学习的发展过程——理论推动、实践突破、遇到瓶颈、解决问题的过程是始终存在的。例如现在所说的“知识驱动神经网络”、“GPT4”、“大模型的领域精细化”等等。突破问题前没人知道是否会成功，但当确实解决后一定是科技的一大飞跃。

科技不会止步!

References

[1]He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 770-778.
[2] He K, Zhang X, Ren S, et al. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification[C]//Proceedings of the IEEE international conference on computer vision. 2015: 1026-1034.
[3] Li H, Xu Z, Taylor G, et al. Visualizing the loss landscape of neural nets[J]. Advances in neural information processing systems, 2018, 31.
[4] Cybenko G. Approximation by superpositions of a sigmoidal function[J]. Mathematics of control, signals and systems, 1989, 2(4): 303-314.
[5] Hornik K, Stinchcombe M, White H. Multilayer feedforward networks are universal approximators[J]. Neural networks, 1989, 2(5): 359-366.
[6] Yarotsky D. Error bounds for approximations with deep ReLU networks[J]. Neural Networks, 2017, 94: 103-114.
[7] Lu Z, Pu H, Wang F, et al. The expressive power of neural networks: A view from the width[J]. Advances in neural information processing systems, 2017, 30.