Project-1 BP网络与卷积层的手动实现

21307140069 田沐钊

出于对代码扩展性和易用性的考虑，本项目选择使用 Python 对模型进行彻底的重写。

下面，本报告将首先对代码的各部分进行介绍说明，并在这个过程中讲解本项目是如何实现要求的各项扩展功能的，接着在最后一部分通过实验验证每项扩展的效果。

Part 1 基础BP网络的实现

以下是BP网络的大致框架。

```
class back_propagation():
    def __init__(self, layers, dropout=0, classifacation=False,
activate='leakyrelu'):
        pass

    def initialize_weights_and_biases(self):
        pass

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def leakyrelu(self, x):
        return np.where(x > 0, x, x * 0.001)

    def softmax(self, X):
        X_exp = np.exp(X - np.max(X, axis=1, keepdims=True))
        row_sums = np.sum(X_exp, axis=1, keepdims=True)
        softmax = X_exp / row_sums
        return softmax

    # 用于将对某一层的输出向量随机置零
    def Dropout(self, X, iftest=0):
        if iftest == 1:
            return X
        else:
            return X * np.random.binomial(1, 1-self.dropout, size=X.shape)

    # 前向传递函数
    def forward_propogation(self, X, iftest=0):
        pass

    # 损失函数值的计算函数
    def loss_compute(self, y_true, y_pred):
        pass

    def square_error(self, y_true, y_pred):
        return np.sum((y_pred - y_true)**2, axis=1)/2
```

```
    def cross_entropy(self, y_true, y_pred):
        return np.sum(np.multiply(-y_true, np.log(y_pred + sys.float_info.epsilon)),
axis=1)

    # 该函数是为了得到计算梯度时所需要的一个部分
    def get_deltas(self):
        pass

    # 反向传播，用于计算参数梯度
    def backward_propagation(self, y):
        pass

    # 这里是使用了 momentum 的版本，用于更新参数
    def update_weights(self, lr, batch_size):
        pass

    # 训练函数
    def train(self, X, y, X_valid, y_valid, batch_size=16, epochs=100, lr=0.02,
l2_regularization=0, momentum_strength=0, valid_cyc=1):
        pass

    # 预测函数
    def predict(self, X):
        return self.forward_propogation(X, iftest=1)

    def accuracy_score(self, y_true, y_pred):
        equal = (y_true == y_pred)
        acc = np.sum(equal) / equal.size
        return acc

    def save(self, path):
        with open(path, 'wb') as f:
            pickle.dump(self, f)

    def load(self, path):
        with open(path, 'rb') as f:
            loaded_obj = pickle.load(f)
        self.Weights = loaded_obj.Weights
        self.Biases = loaded_obj.Biases

    # 以下分别是微调最后一层参数时所要用到的反向传播函数、参数更新函数和微调函数
    def ft_backward(self, y):
        pass

    def ft_update_weights(self, gradient_W, gradient_b, lr, batch_size):
        pass

    def fine_tune(self, X, y, X_valid, y_valid, batch_size=16, epochs=100, lr=0.02,
l2_regularization=0, momentum_strength=0.9):
        pass
```

下面分别对其中的重点部分进行说明。

1.1 模型参数的初始化

模型初始化部分的具体代码实现如下：

```
def __init__(self, layers, dropout=0, classifacation=False,
             activate='leakyrelu'):
    # layers 是一个列表，描述了每一层的神经元个数（包括输入层、隐藏层和输出层），layers的长度即为模型的层数
    self.layers = layers
    self.num_layers = len(layers)
    self.dropout = dropout
    # 表示模型是否用于分类任务，在模型中的作用是决定是否要使用交叉熵作为损失函数
    self.classifacation = classifacation
    if self.classifacation is True:
        self.loss = 'cross_entropy'
    else:
        self.loss = 'mean_squared_error'
    # 指定模型使用什么激活函数
    self.activate = activate
    # 用列表分别记录模型每层的权重矩阵、偏置和对应梯度，以及使用momentum优化策略时的momentum
    self.Weights = []
    self.Biases = []
    self.gradients_W = []
    self.gradients_b = []
    self.momentum_W = []
    self.momentum_b = []
    # 进行权重初始化
    self.initialize_weights_and_biases()

def initialize_weights_and_biases(self):
    for i in range(self.num_layers - 1):
        # 为了对方差进行均衡，使用了 Xaviers 方法进行参数的初始化，控制数据规模，防止出现梯度爆炸等数据不稳定的情况
        Weight = np.random.randn(self.layers[i+1], self.layers[i])*(np.sqrt(2/(self.layers[i+1] + self.layers[i])))
        Bias = np.zeros((self.layers[i+1], 1))
        # 一开始 momentum 均为 0
        weight_momentum = np.zeros((self.layers[i+1], self.layers[i]))
        bias_momentum = np.zeros((self.layers[i+1], 1))
        self.Weights.append(Weight)
        self.Biases.append(Bias)
        self.momentum_W.append(weight_momentum)
        self.momentum_b.append(bias_momentum)
```

在该实现中，用 `layers` 列表来对模型的结构进行描述，可以更加灵活地指定模型层数和神经元个数，从而满足了要求实现的**扩展 1**，支持自行改动模型结构。并且，该实现中对模型的每一层都设置了一个偏置向量，从而等价地实现了要求的 **扩展 6** 。

此外，该实现使用了 Xaviers 初始化方法，其主要思想是根据神经网络的输入和输出的数量来合理初始化参数。它假设前一层的输入和后一层的输出数量应该保持在相对接近的范围内，以避免梯度在网络中过早消失或爆炸。具体而言，对于具有输入数量 n_{in} 和 输出数量 n_{out} 的层，Xavier 初始化从均匀分布中随机采样参数，范围为 $[-limit, limit]$，其中 $limit$ 由以下公式计算得来：

$$limit = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

该公式基于对权重的数学分析，并考虑了激活函数的导数范围。它使得参数的初始化在不同层之间具有相对一致的尺度，有助于更好地控制梯度的传播。

而 `momentum` 的具体实现和使用将在后文说明。

1.2 前向传播与反向传播

我们首先对模型的前向传播及反向传播过程进行一定的数学推导。

记模型第 i 层的神经元个数为 D_i （严谨表述为第 i 层的输出维度）。

记激活函数为 $f(\cdot)$，第 i 层的参数矩阵为 $W_i \in \mathbb{R}^{D_{i+1} \times D_i}$ ，偏置向量为 $\beta_i \in \mathbb{R}^{D_{i+1} \times 1}$ ，则激活前输出为 $b_{i+1} \in \mathbb{R}^{D_{i+1} \times 1}$， 激活后输出为 $a_{i+1} \in \mathbb{R}^{D_{i+1} \times 1}$，则前向传播过程满足：

$$b_{i+1} = W_i a_i + \beta_i$$
$$a_{i+1} = f(b_{i+1})$$

其中 a_0 为模型的最初输入。

记模型的层数为 L ，损失函数为 $loss(\cdot)$，则损失函数的值为 $loss(a_L)$ ，不妨将其简写为 $loss$。

在反向传播过程中，我们计算损失函数值对每个参数的梯度。

首先，我们容易计算得到：

$$\begin{aligned}\frac{\partial\, loss}{\partial\, W_{L-1,i,j}} &= \sum_{k=1}^{D_L} \frac{\partial\, loss}{\partial\, a_{L,k}} \frac{\partial\, a_{L,k}}{\partial\, b_{L,k}} \frac{\partial\, b_{L,k}}{\partial\, W_{L-1,i,j}} \\ &= (\frac{\partial\, loss}{\partial\, a_L})_j \times (\frac{f(b_L)}{a_L})_j \times a_{L-1,i}\end{aligned}$$

其中 i, j 为对矩阵、向量元素的索引标注。

不妨记 $\frac{f(b_L)}{a_L} = \Delta_L$ ，则易得到：

$$\frac{\partial\, loss}{\partial\, W_{L-1}} = (\frac{\partial\, loss}{\partial\, a_L} \odot \Delta_L) a_{L-1}^T$$

其中 $\odot$ 为矩阵的 Hadamard 积。

同理易得：

$$\frac{\partial\, loss}{\partial\, \beta_{L-1}} = \frac{\partial\, loss}{\partial\, a_L} \odot \Delta_L$$

向前类推计算，易得：

$$\frac{\partial\, loss}{\partial\, \beta_i} = (\prod_{k=i+1}^{L-1} W_k^T)\frac{\partial\, loss}{\partial\, a_L} \odot \Delta_L \odot \ \ldots \ \odot \Delta_{i+1}$$

$$\frac{\partial\, loss}{\partial\, W_i} = \frac{\partial\, loss}{\partial\, \beta_i} a_i^T$$

$$\frac{\partial\, loss}{\partial\, \beta_{i-1}} = W_I^T \frac{\partial\, loss}{\partial\, \beta_i} \odot \Delta_i$$

基于这些关系，我们容易写出BP网络的前向传播与反向方法如下：

```
# 用于将对某一层的输出向量随机置零
def Dropout(self, X, iftest=0):
    if iftest == 1:
        return X
    else:
        return X * np.random.binomial(1, 1-self.dropout, size=X.shape)

# 前向传播函数
def forward_propogation(self, X, iftest=0):
    activations = [X]
    if self.activate == 'sigmoid':
        for i in range(self.num_layers - 2):
            activations.append(self.sigmoid(self.Dropout((np.dot(self.Weights[i],
activations[i].T) + self.Biases[i]).T, iftest=iftest)))
    elif self.activate == 'leakyrelu':
        for i in range(self.num_layers - 2):
            activations.append(self.leakyrelu(self.Dropout((np.dot(self.Weights[i],
activations[i].T) + self.Biases[i]).T, iftest=iftest)))

    activations.append((np.dot(self.Weights[-1], activations[-1].T) +
self.Biases[-1]).T)

    if self.classifacation is True:
        activations[-1] = self.softmax(activations[-1])

    self.activations = activations
    return activations[-1]

# 损失函数计算
def loss_compute(self, y_true, y_pred):
    l2_loss = 0
    if self.l2_regularization != 0:
        for i in range(self.num_layers - 1):
            l2_loss += np.sum(np.square(self.Weights[i]))
        l2_loss = self.l2_regularization * l2_loss / 2
    if self.loss == 'mean_squared_error':
        return self.square_error(y_true, y_pred)[0] + l2_loss, y_pred - y_true
    else:
        return self.cross_entropy(y_true, y_pred)[0] + l2_loss, y_pred - y_true
```

```
def square_error(self, y_true, y_pred):
    return np.sum((y_pred - y_true)**2, axis=1)/2

def cross_entropy(self, y_true, y_pred):
    return np.sum(np.multiply(-y_true, np.log(y_pred + sys.float_info.epsilon)),
axis=1)

# 该函数是为了得到计算梯度时所需要的一个部分
def get_deltas(self):
    deltas = []
    if self.activate == 'sigmoid':
        for i in range(self.num_layers - 2):
            deltas.append(np.multiply(self.activations[i + 1], 1 -
self.activations[i + 1]))
    if self.activate == 'leakyrelu':
        for i in range(self.num_layers - 2):
            deltas.append(np.where(self.activations[i+1] >= 0, 1, 0.001))

    return deltas

# 反向传播函数
def backward_propagation(self, y):
    _, loss_grad = self.loss_compute(y, self.activations[-1])
    gradients_W = []
    gradients_b = []

    deltas = self.get_deltas()

    # 第一步的梯度计算（第一步放在循环外是为了防止在循环内部写if语句）
    gradient_b = loss_grad
    gradients_b.insert(0, gradient_b)
    gradients_W.insert(0, gradient_b[:, :, np.newaxis] * self.activations[-2][:,
np.newaxis, :] + self.l2_regularization * self.weights[-1])
    for i in range(2, self.num_layers):
        gradient_b = np.multiply(np.dot(gradient_b, self.weights[-i+1]), deltas[-
i+1])
        gradients_b.insert(0, gradient_b)
        gradients_W.insert(0, gradient_b[:, :, np.newaxis] * self.activations[-i-1]
[:, np.newaxis, :] + self.l2_regularization * self.weights[-i])

    self.gradients_W = gradients_W
    self.gradients_b = gradients_b
```

在前向传播中，我们使用了 Dropout 函数来在训练过程中随机对部分输出进行置零，使得网络的不同部分在每次训练迭代中都能够以一种类似于集成学习的方式进行学习，减少神经元之间的共适应性，强迫网络学习更加鲁棒和泛化性能更好的特征。这有助于防止网络过拟合训练数据，提高其在未见过的测试数据上的性能。在推理阶段，为了获得更稳定的预测结果，通常将所有神经元的输出乘以保留概率，即将训练时关闭的神经元的输出按比例缩放。

为了辨别是否应该使用dropout函数（或者说在训练阶段还是推理阶段），这里我们采用 ifset 来进行标记。这样，我们实现了要求的 **扩展 7**。

此外，在计算损失函数和梯度时，我们将 L2正则项 也纳入了考虑，其为常用的正则化技术，用于在训练神经网络时对权重进行惩罚，以减少过拟合的风险。从而实现了要求的 **扩展 4** 。

并且，我们提供了softmax和交叉熵的计算函数，从而满足了要求的 **扩展 5** 。

同时，我们在这整个过程中主要进行的均为矩阵乘法运算，大大加快了运算速度，从而满足了要求的 **扩展 3** 。

1.3 训练函数的实现

训练函数的具体实现如下：

```
# 这里是使用了 momentum 的版本
def update_weights(self, lr, batch_size, full_bias):
    for i in range(self.num_layers-1):
        new_momentum_W = self.momentum_strength * self.momentum_W[i] - lr *
(np.sum(self.gradients_W[i], axis=0) / batch_size)
        if full_bias is True or i == 0:
            new_momentum_b = self.momentum_strength * self.momentum_b[i] - lr *
(np.sum(self.gradients_b[i], axis=0)[:, np.newaxis] / batch_size)
        else:
            new_momentum_b = np.zeros_like(self.Biases[i])
        self.Weights[i] += new_momentum_W
        self.Biases[i] += new_momentum_b
        self.momentum_W[i] = new_momentum_W
        self.momentum_b[i] = new_momentum_b

def train(self, X, y, X_valid, y_valid, batch_size=16, epochs=100, lr=0.02,
l2_regularization=0, momentum_strength=0, valid_cyc=1, full_bias=True):
    if self.classifacation is True:
        train_accuracy_scores = []
        valid_accuracy_scores = []
    loss_scores = []
    self.l2_regularization = l2_regularization
    self.momentum_strength = momentum_strength
    i = 0
    for i in range(epochs):
        # 建立随机映射，以实现随机取 batch 的效果
        mapping = np.arange(X.shape[0])
        np.random.shuffle(mapping)

        for j in tqdm(range(0, X.shape[0], batch_size)):
            if j + batch_size > X.shape[0]:
                X_batch = X[mapping[-batch_size:], :]
                y_batch = y[mapping[-batch_size:]]
            else:
                X_batch = X[mapping[j:j+batch_size], :]
                y_batch = y[mapping[j:j+batch_size]]
            # 完成网络的一次前向传播、反向传播和梯度更新
            self.forward_propogation(X_batch)
            self.backward_propagation(y_batch)
            self.update_weights(lr, batch_size, full_bias)
```

```
        # 记录一次损失函数值
        losses, _ = self.loss_compute(y_batch, self.activations[-1])
        loss = np.sum(losses) / batch_size
        loss_scores.append(loss)
        print(f"Epoch {i + 1}: train loss = {loss}, learning_rate = {lr}")

        # 每过 valid_cyc 个 epoch 就使用验证集进行一次验证并打印结果
        if self.classifacation is True and i % valid_cyc == 0:
            train_act = np.array(self.predict(X))
            valid_act = np.array(self.predict(X_valid))
            train_pred = np.argmax(train_act, axis=1)
            train_true = np.argmax(y, axis=1)
            valid_pred = np.argmax(valid_act, axis=1)
            train_accuracy_scores.append(self.accuracy_score(train_true,
train_pred))
            valid_accuracy_scores.append(self.accuracy_score(y_valid, valid_pred))
            print(f"Epoch {i + 1}: train loss = {loss}, train acc =
{train_accuracy_scores[-1]}, valid acc = {valid_accuracy_scores[-1]}, learning_rate
= {lr}")
        elif self.classifacation is False and i % valid_cyc == 0:
            train_act = np.array(self.predict(X))
            valid_act = np.array(self.predict(X_valid))
            valid_loss = self.loss_compute(y_valid, valid_act)[0] / X_valid.shape[0]
            print(f"Epoch {i + 1}: train loss = {loss}, valid loss = {valid_loss},
learning_rate = {lr}")

    if self.classifacation is True:
        return train_accuracy_scores, valid_accuracy_scores, loss_scores
    else:
        return loss_scores
```

在这个参数更新的实现中，我们使用了 momentum 方法，每次更新为：

$$w^{t+1} = w^t - \alpha_t \nabla f(w^t) + \beta_t (w^t - w^{t-1})$$

在不使用的时候，将 momentum_strength 设为0即可。实现了要求的**扩展 2。**

此外这里使用 full_bias 来指定是否要对每层都使用偏置，若full_bias 为 False，则每次将第一层以外的更新梯度设为全零向量，又因为一开始对偏置的初始化都是全零向量，故可以用这种方式保证第一层以外的偏置均为0，即没有偏置。

预测函数的实现较为简单，故本报告不再赘述。

1.4 微调函数的实现

在微调的过程中，我们固定了其余层的参数，仅对最后一层的参数进行优化，其具体实现只需对普通的训练函数进行些微的修改，其所用到的方法如下：

```
def ft_backward(self, y):
    _, loss_grad = self.loss_compute(y, self.activations[-1])
    # 仅计算最后一层的梯度
```

```
    gradient_b = loss_grad
    gradient_W = gradient_b[:, :, np.newaxis] * self.activations[-2][:, np.newaxis,
:] + self.l2_regularization * self.weights[-1]

    return gradient_W, gradient_b

def ft_update_weights(self, gradient_W, gradient_b, lr, batch_size):
    # 仅更新最后一层的参数
    new_momentum_W = self.momentum_strength * self.ft_momentum_W - lr *
(np.sum(gradient_W, axis=0)/batch_size)
    new_momentum_b = self.momentum_strength * self.ft_momentum_b - lr *
(np.sum(gradient_b, axis=0)[:, np.newaxis]/batch_size)
    self.weights[-1] += new_momentum_W
    self.Biases[-1] += new_momentum_b
    self.ft_momentum_W = new_momentum_W
    self.ft_momentum_b = new_momentum_b
```

这样，我们便实现了要求的 **扩展 8**。

Part 2 带卷积层的网络的实现

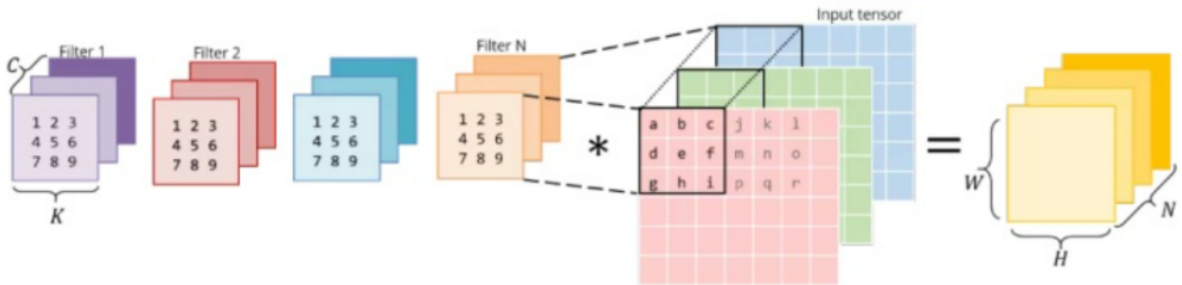
2.1 卷积层的实现

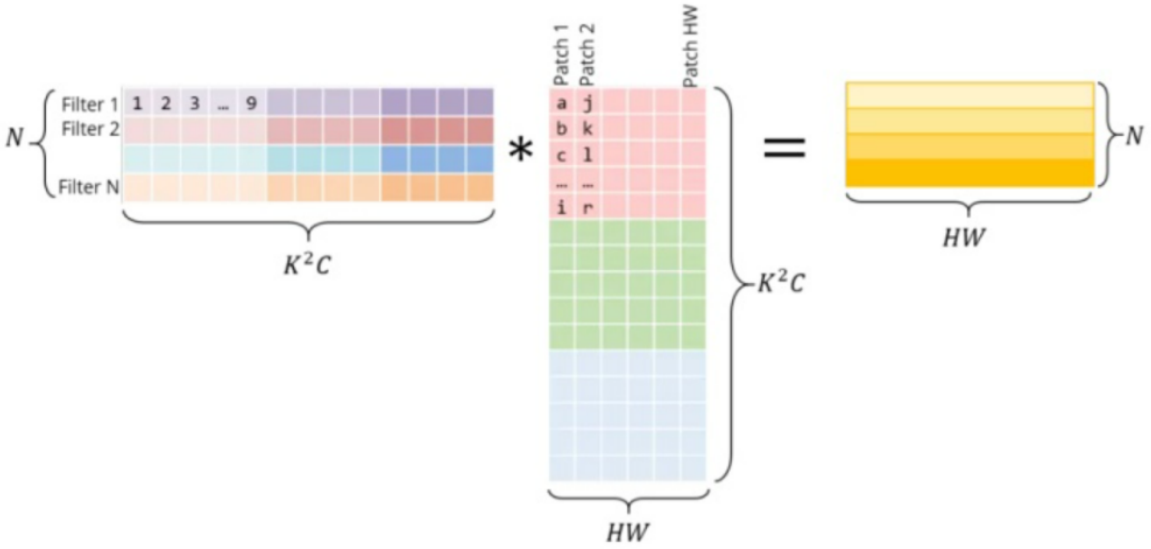
出于对代码扩展性和模块化的考虑，这里我们将卷积层单独作为一个对象进行实现如下：

```
class Conv2d():
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=1,
bias=True):
        # input and output
        self.input = None
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding
        self.X_col = None
        self.w_col = None

        # params
        self.params = {}
        self.params['W'] = None
        self.params['b'] = None
        std = np.sqrt(2 / (self.in_channels + self.out_channels))
        self.params['W'] = np.random.normal(0, std, size=(self.out_channels,
self.in_channels, self.kernel_size[0], self.kernel_size[1]))
        if bias:
            self.params['b'] = np.zeros((self.out_channels, ))
        self.grads = {}

    def forward(self, input):
```

```
        if isinstance(input, np.ndarray) is False:
            input = input.numpy()
        self.input = input
        batchsize, _, in_H, in_W = input.shape

        out_H = (in_H - self.kernel_size[0] + 2 * self.padding) // self.stride + 1
        out_W = (in_W - self.kernel_size[1] + 2 * self.padding) // self.stride + 1

        # 这里使用了快速卷积，利用im2col将卷积运算转换为了矩阵乘法，大大加快了运算速度
        input_col = im2col(input, self.kernel_size[0], self.kernel_size[1],
self.stride, self.padding)
        W_col = self.params['W'].reshape((self.out_channels, -1))
        output = np.dot(W_col, input_col)
        output = np.array(np.hsplit(output, batchsize)).reshape((batchsize,
self.out_channels, out_H, out_W))
        self.W_col = W_col
        self.input_col = input_col

        if self.params['b'] is not None:
            output += self.params['b'][:, np.newaxis, np.newaxis]

        return output

    def backward(self, output_grad, l2_regularization=0):
        batch_size = output_grad.shape[0]
        output_grad_col = output_grad.reshape((output_grad.shape[0] *
output_grad.shape[1], -1))
        output_grad_col = np.array(np.vsplit(output_grad_col, batch_size))
        output_grad_col = np.concatenate(output_grad_col, axis=-1)

        # 同样用矩阵乘法的视角计算梯度，加快计算速度
        self.grads['W'] = np.dot(output_grad_col,
self.input_col.T).reshape(self.params['W'].shape) + l2_regularization *
self.params['W']
        self.grads['b'] = np.mean(output_grad, axis=(0, 2, 3)) + l2_regularization *
self.params['b']
        input_grad = np.dot(self.W_col.T, output_grad_col)
        input_grad = col2im(input_grad, self.input.shape, self.kernel_size[0],
self.kernel_size[1], self.stride, self.padding)
        return input_grad
```

因为传统的卷积操作在实现时需要人为地编写多层循环，运行速度较慢，故这里采用了快速卷积的方法，将卷积运算转化为了高效的矩阵乘法运算。其具体原理如下图所示：





而这个转化过程所需要用到的工具函数如下：

```
# 获取用于im2col的索引
def get_im2col_indices(X_shape, kernel_H, kernel_W, stride, pad):
    # 获取输入和输出的形状
    _, channels, in_H, in_W = X_shape
    out_H = (in_H + 2 * pad - kernel_H) // stride + 1
    out_W = (in_W + 2 * pad - kernel_W) // stride + 1

    level1 = np.tile(np.repeat(np.arange(kernel_H), kernel_W), channels)
    everyLevels = stride * np.repeat(np.arange(out_H), out_W)
    i = level1.reshape(-1, 1) + everyLevels.reshape(1, -1)

    slide1 = np.tile(np.tile(np.arange(kernel_W), kernel_H), channels)
    everySlides = stride * np.tile(np.arange(out_W), out_H)
    j = slide1.reshape(-1, 1) + everySlides.reshape(1, -1)

    k = np.repeat(np.arange(channels), kernel_H * kernel_W).reshape(-1, 1)

    return i, j, k


# 用于将NCHW格式的批次图像转换为列并拼接为二维矩阵，可将卷积运算转换为矩阵乘法
def im2col(X, kernel_H, kernel_W, stride, pad):
    # 将输入图像X用0填充
    X_padded = np.pad(X, ((0, 0), (0, 0), (pad, pad), (pad, pad)), mode='constant')
    # 获取用于im2col的索引
    i, j, k = get_im2col_indices(X.shape, kernel_H, kernel_W, stride, pad)
    # 将填充后的图像X按k,i,j拼接为列
    cols = X_padded[:, k, i, j]
    # 将拼接后的列按最后一维拼接
    cols = np.concatenate(cols, axis=-1)
    return cols


# 用于将列拼接的二维矩阵转换回NCHW格式的原矩阵
```

```
def col2im(input_grad_col, X_shape, kernel_H, kernrl_W, stride, pad):
    # 获取输入和输出的形状
    batch_size, channels, H, W = X_shape
    H_padded, W_padded = H + 2 * pad, W + 2 * pad
    X_padded = np.zeros((batch_size, channels, H_padded, W_padded))

    # 获取用于im2col的索引
    i, j, k = get_im2col_indices(X_shape, kernel_H, kernrl_W, stride, pad)
    # 将输入梯度input_grad_col按batch_size,k,i,j拼接为列
    dX_col_reshaped = np.array(np.hsplit(input_grad_col, batch_size))
    # 将拼接后的列按k,i,j添加到填充后的图像X_padded中
    np.add.at(X_padded, (slice(None), k, i, j), dX_col_reshaped)

    return X_padded[:, :, pad:-pad, pad:-pad]
```

2.2 网络的实现

这里选择直接将已有的 back_propagation 类和 Conv2d 类进行整合，其具体的代码如下：

```
class Conv_BP():
    def __init__(self, in_channels, out_channels, kernel_size, layers, stride=1,
padding=1, bias=True, dropout=0, classifacation=False,
                 activate='leakyrelu'):
        # 卷积层和线性层
        self.conv = Conv2d(in_channels, out_channels, kernel_size, stride, padding,
bias)
        self.bp_net = back_propagation(layers, dropout, classifacation, activate)
        # 卷积层使用 momentum 方法进行梯度更新时所需要用到的 momentum
        self.momentum_W = np.zeros_like(self.conv.params['W'])
        self.momentum_b = np.zeros_like(self.conv.params['b'])

    def forward(self, X):
        X = self.conv.forward(X)
        self.conv_out_shape = X.shape
        X = self.bp_net.forward_propogation(X.reshape((X.shape[0], -1)))
        return X

    def backward(self, y):
        self.bp_net.backward_propagation(y)
        grad_conv_out = np.dot(self.bp_net.gradients_b[0], self.bp_net.weights[0])
        grad_conv_out = grad_conv_out.reshape(self.conv_out_shape)
        self.conv.backward(grad_conv_out, self.l2_regularization)

    def update_weights(self, lr, batch_size):
        self.bp_net.update_weights(lr, batch_size)
        new_momentum_W = self.bp_net.momentum_strength * self.momentum_W - lr *
(np.sum(self.conv.grads['W'], axis=0) / batch_size)
        new_momentum_b = self.bp_net.momentum_strength * self.momentum_b - lr *
(np.sum(self.conv.grads['b'], axis=0) / batch_size)
        self.conv.params['W'] += new_momentum_W
        self.conv.params['b'] += new_momentum_b
```

```
        self.momentum_W = new_momentum_W
        self.momentum_b = new_momentum_b

    def predict(self, X):
        return self.bp_net.predict(self.conv.forward(X).reshape((X.shape[0], -1)))

    def train(self, X, y, X_valid, y_valid, batch_size=16, epochs=100, lr=0.02,
l2_regularization=0.01, momentum_strength=0.1, valid_cyc=1):
        # 与普通的BP网络的train方法的框架类似，此处不再赘述
        pass

    def save(self, path):
        with open(path, 'wb') as f:
            pickle.dump(self, f)

    def load(self, path):
        with open(path, 'rb') as f:
            loaded_obj = pickle.load(f)
        self.conv = loaded_obj.conv
        self.bp_net = loaded_obj.bp_net
```

这样，我们就实现了要求的 **扩展 10。**

Part 3 模型的实际训练与各项扩展的性能测试

出于对训练效率的考虑，下面所有例子默认情况下均在经过softmax层后使用交叉熵作为损失函数，且默认每层都有一个偏置，并且默认使用向量化的运算。

此外，因为本实验中我们是用 Python 重新实现的网络，故并不方便使用项目文件中给出的 .mat 文件作为数据集。故这里直接使用了 MNIST手写字体数据集，其中每个样本的大小为 28*28。以下是数据的加载和预处理代码：

```
# 读取数据集
train_imgs = read_image("data/train-images.idx3-ubyte")
train_labels = read_label("data/train-labels.idx1-ubyte")
test_imgs = read_image("data/t10k-images.idx3-ubyte")
test_labels = read_label("data/t10k-labels.idx1-ubyte")

# 划分训练集和验证集
valid_imgs = train_imgs[:1000]
valid_labels = train_labels[:1000]
train_imgs = train_imgs[1000:11000]
train_labels = train_labels[1000:11000]
test_imgs = test_imgs[:1000]
test_labels = test_labels[:1000]

train_imgs = train_imgs / 255
train_imgs = np.where(train_imgs > 0.5, 1, 0)
valid_imgs = valid_imgs / 255
valid_imgs = np.where(valid_imgs > 0.5, 1, 0)
```

```
test_imgs = test_imgs / 255
test_imgs = np.where(test_imgs > 0.5, 1, 0)
```

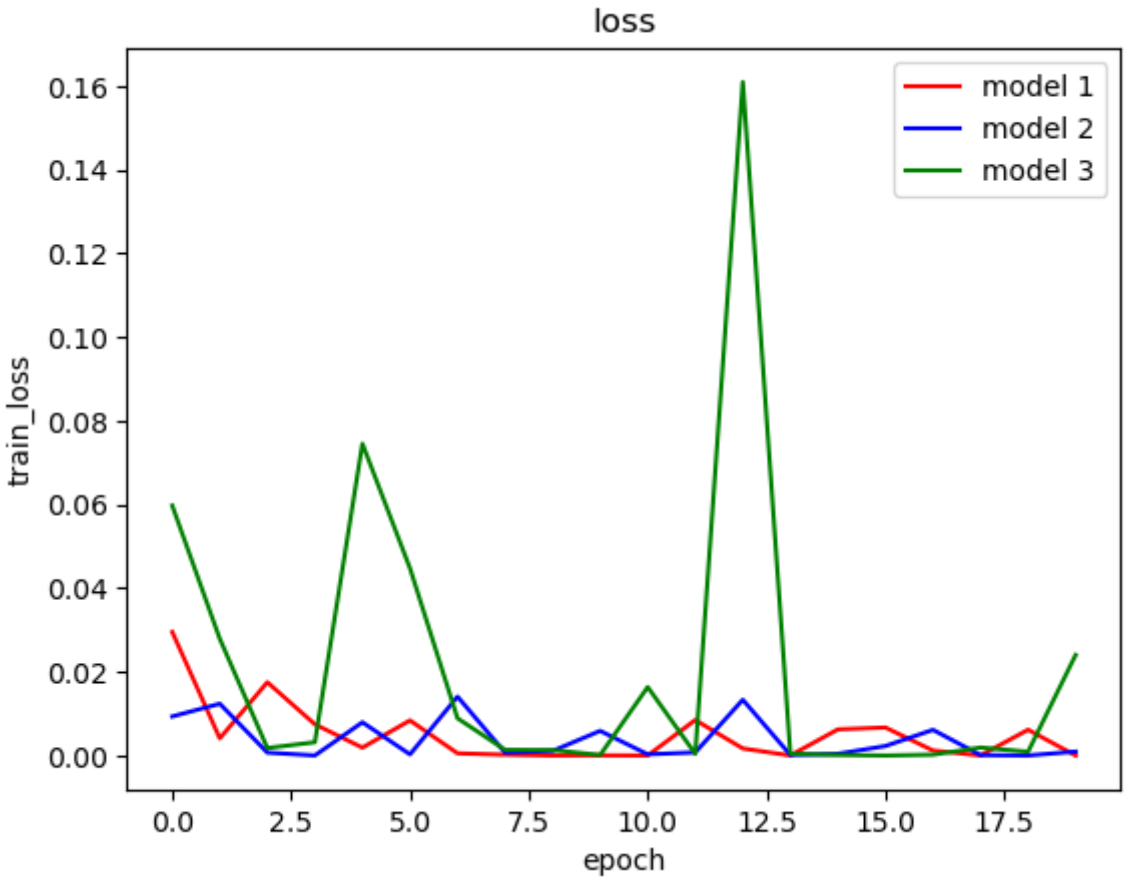
这里对所有数据均进行了二值化处理，从而大大减小了数据规模和复杂度，使数据特征更加易于处理。

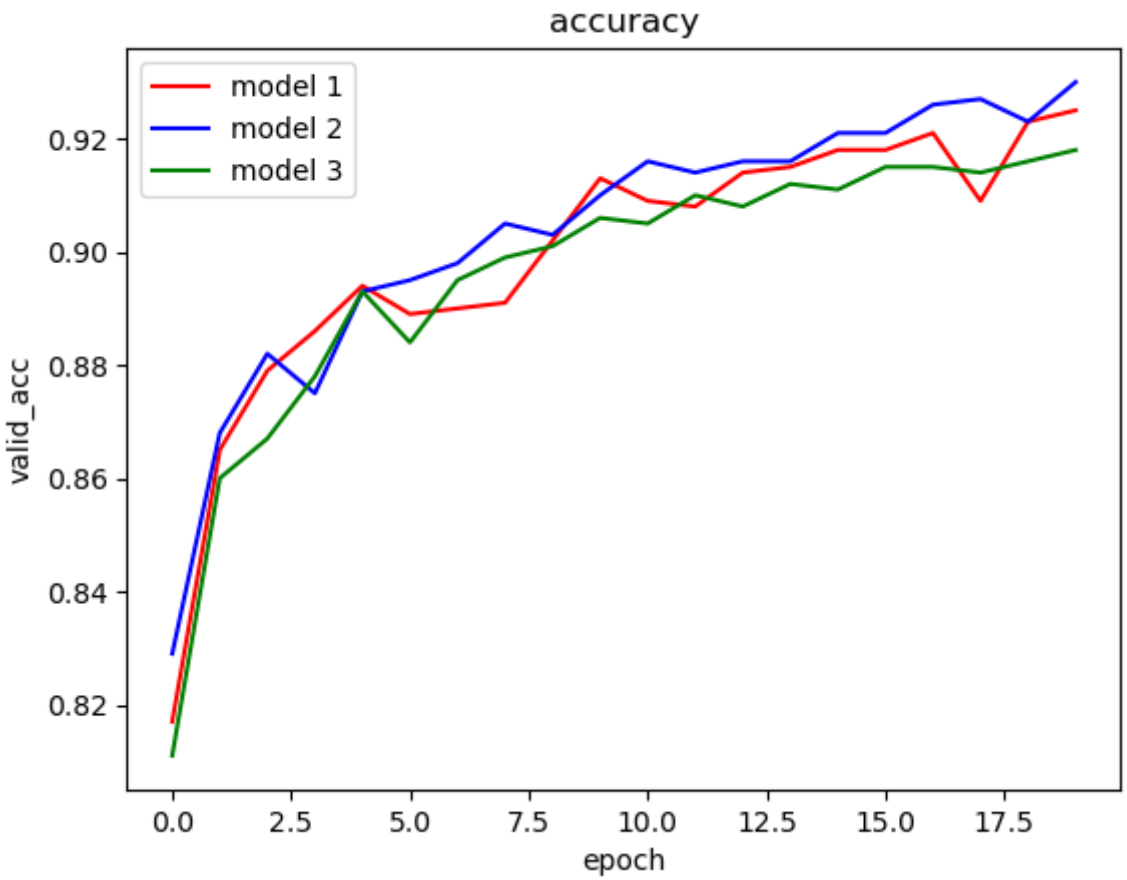
3.1 网络结构的影响

下面分别对三种网络结构的模型进行训练并进行性能比较。其训练代码如下：

```
# 比较不同的模型结构
num_labels = 10
train_label_vectors = np.eye(num_labels)[train_labels]

layers_1 = [28 * 28, 256, 128, num_labels]
layers_2 = [28 * 28, 512, 128, num_labels]
layers_3 = [28 * 28, 256, num_labels]

# 模型中使用了dropout
dropout = 0
bp_net_1 = back_propagation(layers=layers_1, dropout=dropout, classifacation=True)
bp_net_2 = back_propagation(layers=layers_2, dropout=dropout, classifacation=True)
bp_net_3 = back_propagation(layers=layers_3, dropout=dropout, classifacation=True)

# 训练模型
batch_size = 32
epochs = 20
learning_rate = 0.01
l2_reg = 0
momentum = 0
_, valid_accs_1, train_losses_1 = bp_net_1.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum)
_, valid_accs_2, train_losses_2 = bp_net_2.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum)
_, valid_accs_3, train_losses_3 = bp_net_3.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum)
```

其loss曲线、valid accuracy曲线的对比图如下：





其最终在测试集上的预测准确率的对比如下：

```
Accuracy_1: 0.918
Accuracy_2: 0.922
Accuracy_3: 0.904
```

可见 model 1的loss曲线下降得比model 3更为稳定、迅速，最终准确率上model 1也高于 model 3。而model 1和model 2虽然loss曲线无显著差别，但在准确率上依然是model 2略高于model 1。

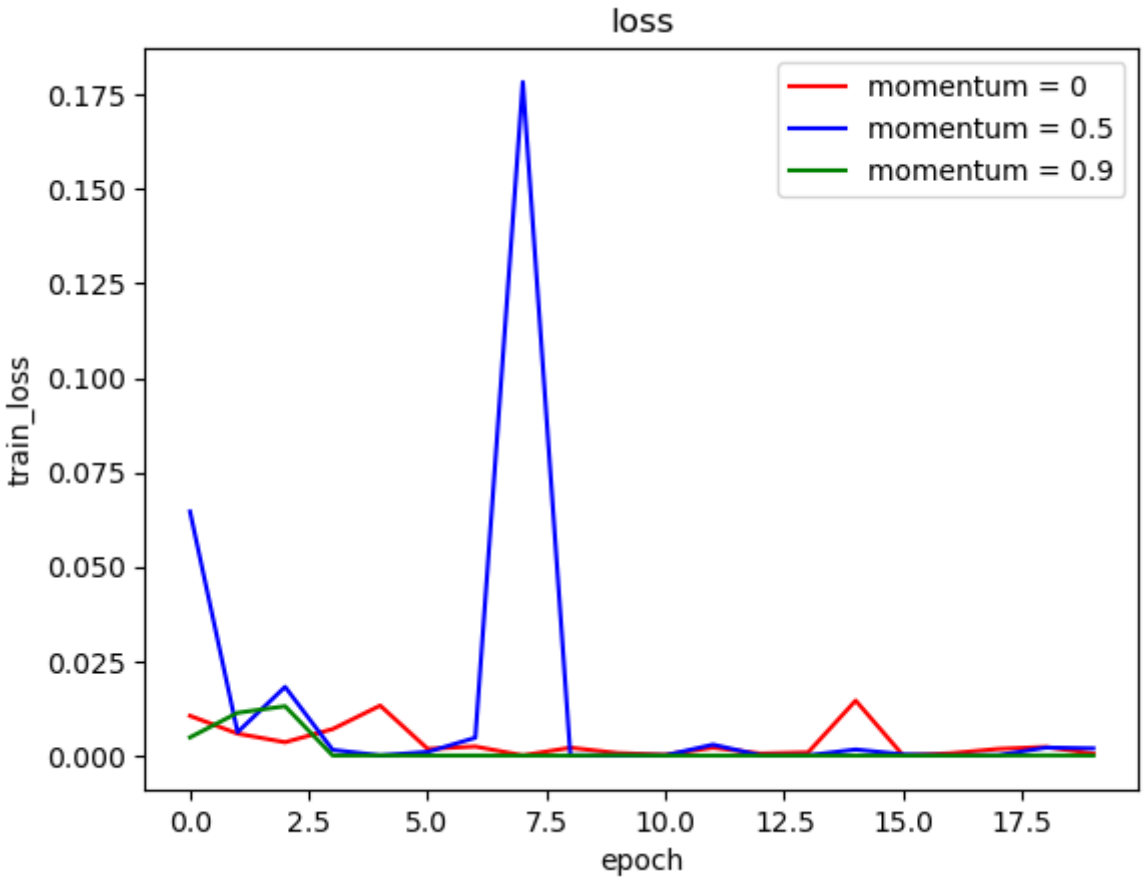
可以认为，增加网络的层数和神经元的个数可以增加网络的表示能力和学习能力。深层网络可以学习到更复杂的特征和抽象表示，从而对复杂的任务具有更强的表达能力，更多的神经元可以提供更高的自由度，使得网络能够学习到更复杂和详细的特征。但增加网络的层数也会增加训练的难度，可能导致梯度消失或梯度爆炸等问题，增加神经元的数量也会增加模型的复杂性和计算成本。过多的神经元可能导致过拟合问题，因此需要根据任务的复杂度和数据集的规模合理选择模型结构。

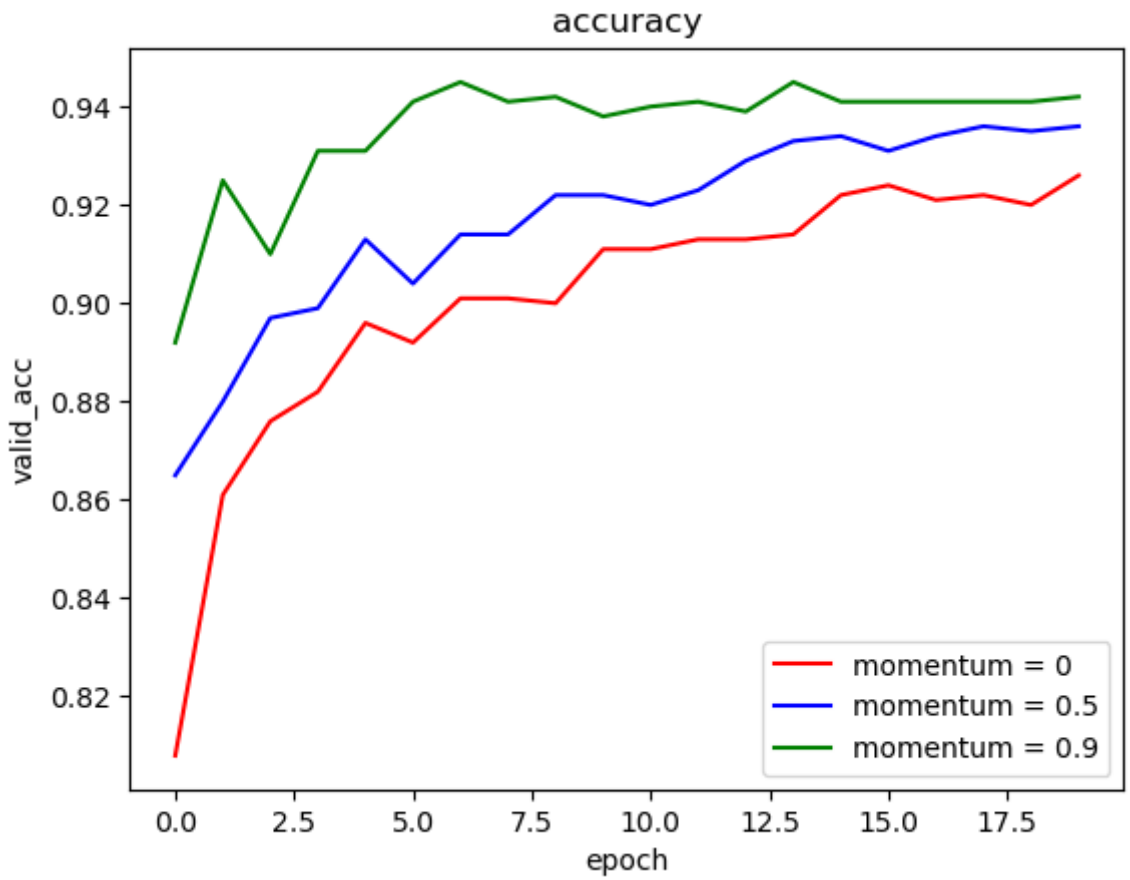
3.2 momentum方法的效果

下面分别对不同momentum strength的模型进行训练并进行性能比较。其训练代码如下：

```
# 比较momentum strength的影响
layers = [28 * 28, 256, 128, num_labels]
bp_net_1 = back_propagation(layers=layers, dropout=dropout, classifacation=True)
bp_net_2 = back_propagation(layers=layers, dropout=dropout, classifacation=True)
bp_net_3 = back_propagation(layers=layers, dropout=dropout, classifacation=True)

momentum_1 = 0
momentum_2 = 0.5
momentum_3 = 0.9
_, valid_accs_1, train_losses_1 = bp_net_1.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum_1)
_, valid_accs_2, train_losses_2 = bp_net_2.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum_2)
_, valid_accs_3, train_losses_3 = bp_net_3.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum_3)
```

其loss曲线、valid accuracy曲线的对比图如下：





其最终在测试集上的预测准确率的对比如下：

```
Accuracy_1: 0.906
Accuracy_2: 0.925
Accuracy_3: 0.946
```

可见随着 momentum strength 的增加，模型在验证集上的预测准确率有显著提升，训练效果也更高效稳定。

可以认为，momentum通过添加一个动量项来考虑历史梯度信息，在更新权重时增加了一种惯性效果，当梯度在连续的迭代中保持一致方向时，动量项的累积效果将增加梯度更新的幅度，从而使权重更快地朝着最优解方向移动。这有助于加快收敛并减少训练时间。此外，动量项可以平滑梯度更新的方向。在梯度变化剧烈的情况下，动量可以减少梯度更新的波动，使权重更新更加平稳。这有助于防止模型在陡峭的梯度表面上震荡，并有助于避免局部最小值。并且，由于动量的积累效果，Momentum方法能够帮助模型跳出局部最小值。当梯度更新受困于局部最小值时，动量项可以提供足够的动力，使得模型能够越过这些局部极小点，并继续朝着全局最小值方向前进。

3.3 对运算进行向量化的说明

在本项目的实现中，我们全部的运算都是高度向量化的，这样做有以下几个好处：

1. 提高计算效率：向量化运算利用现代计算机体系结构中的并行计算能力，可以通过底层优化和并行化指令来加速计算过程。相比于使用循环逐个元素计算，向量化计算可以同时对多个元素进行操作，从而大幅提高计算效率。
2. 简化代码实现：向量化运算可以将复杂的循环结构简化为简洁的矩阵或向量表达式。这样不仅减少了代码的复杂性和冗余性，还提高了代码的可读性和可维护性。向量化计算使得代码更加清晰、简洁，同时减少了出错的可能性。
3. 更好的利用硬件资源：向量化运算可以更好地利用硬件资源，如CPU或GPU的并行计算单元。现代计算机体系结构中的SIMD指令集可以同时对多个数据进行并行计算，从而充分发挥硬件的计算能力。

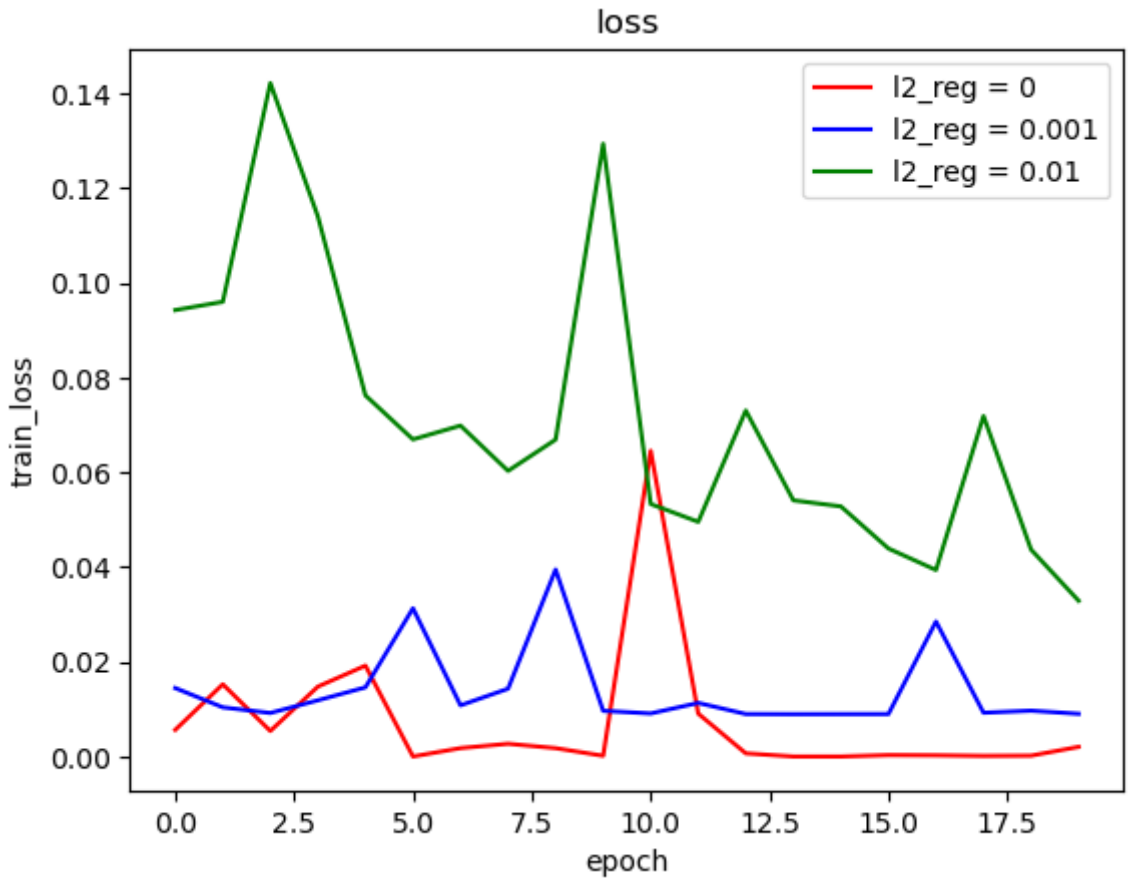
如果不使用矩阵运算，需要人为地手写多重循环来逐个计算神经元的值，显然极其低效，因为本项目是使用 python 对整个模型进行了重写。故在此仅展示并使用向量优化后的方法，而不专门编写手动循环的方法进行对比。

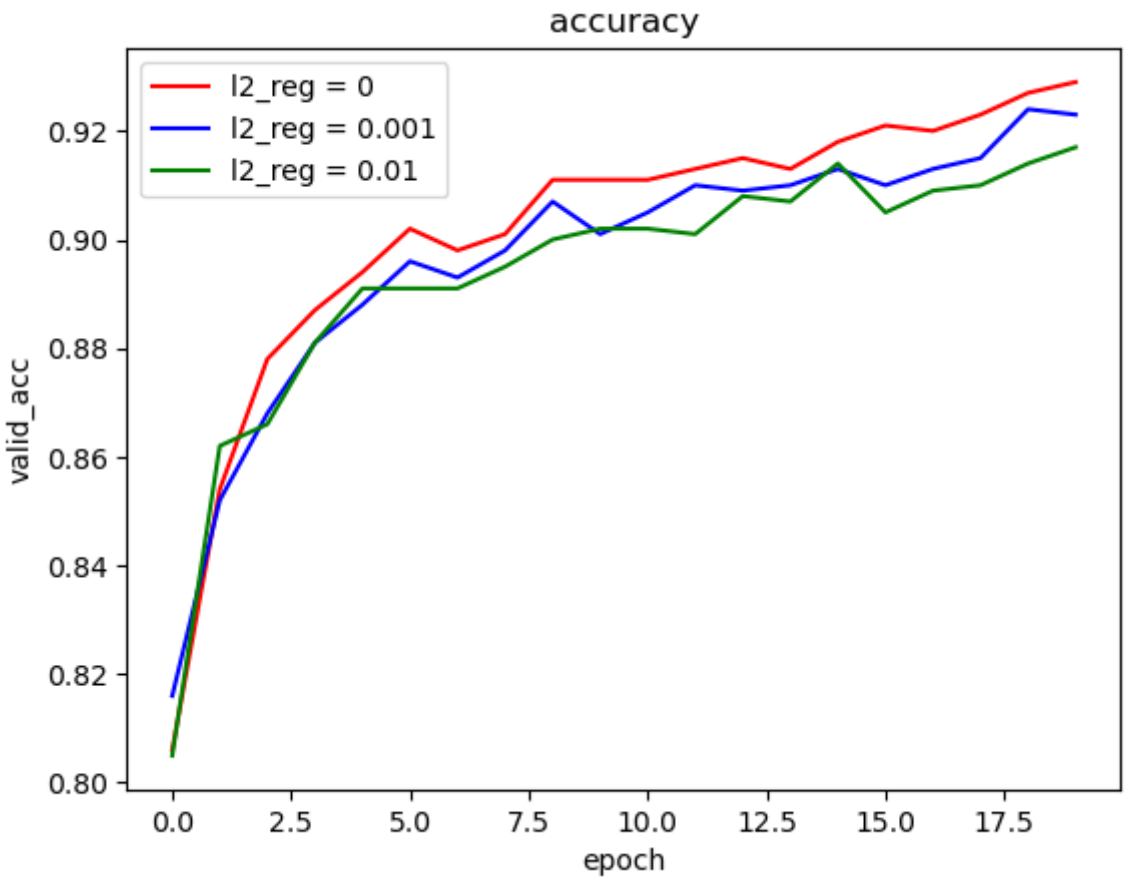
3.4 L2正则项的效果

下面分别对不同L2正则项系数的模型进行训练并进行性能比较。其训练代码如下：

```
# L2正则项的效果比较
bp_net_1 = back_propagation(layers=layers, dropout=dropout, classifacation=True)
bp_net_2 = back_propagation(layers=layers, dropout=dropout, classifacation=True)
bp_net_3 = back_propagation(layers=layers, dropout=dropout, classifacation=True)

l2_reg_1 = 0
l2_reg_2 = 0.001
l2_reg_3 = 0.01
_, valid_accs_1, train_losses_1 = bp_net_1.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg_1, momentum)
_, valid_accs_2, train_losses_2 = bp_net_2.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg_2, momentum)
_, valid_accs_3, train_losses_3 = bp_net_3.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg_3, momentum)
```

其loss曲线、valid accuracy曲线的对比图如下：





其最终在测试集上的预测准确率的对比如下：

```
Accuracy_1: 0.914
Accuracy_2: 0.928
Accuracy_3: 0.899
```

可见加入正则项并提高正则项系数后，模型训练期间的损失函数值有所增大，但正则项系数为0.001的模型的测试准确率较高于不带正则项的模型，而正则项系数为0.01的模型的测试准确率显著低于前两者。

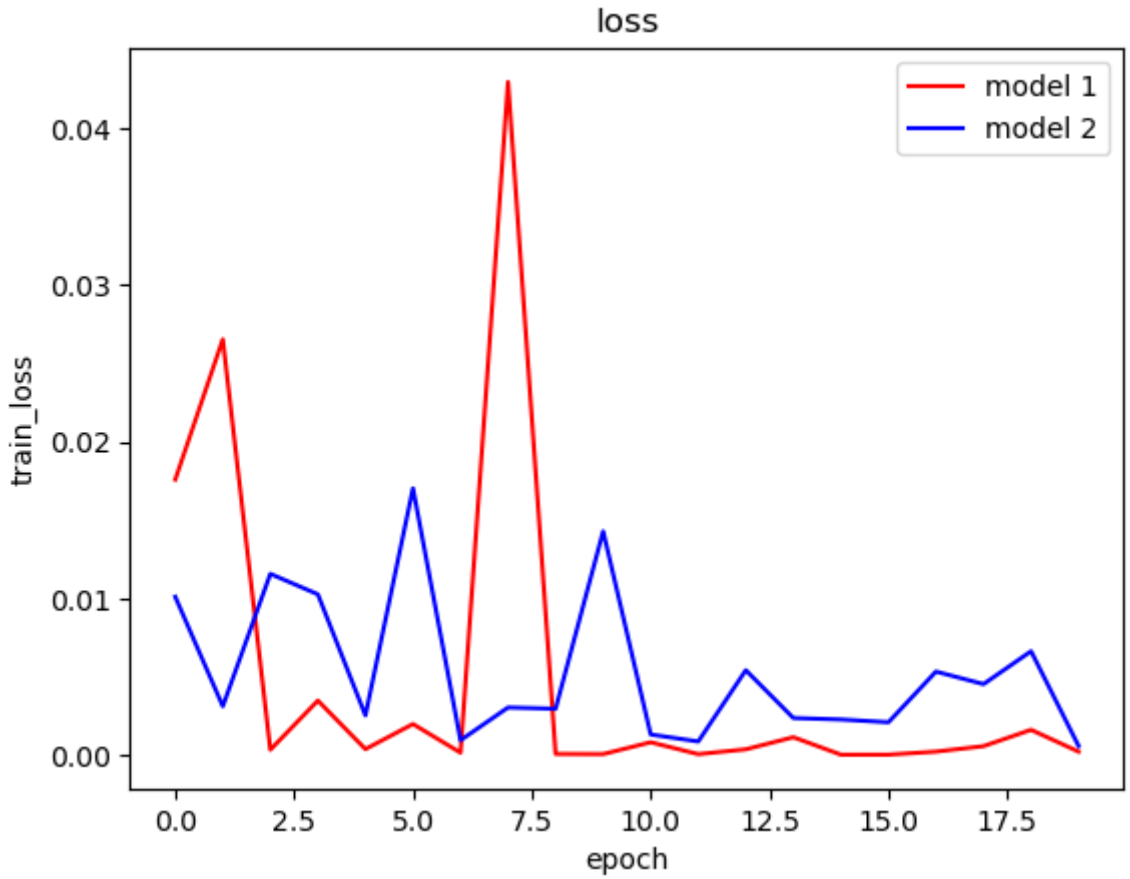
实验结果符合常理。正则化系数可以促使模型的参数向零靠拢。较大的正则化系数会增加参数的惩罚，从而更倾向于选择较小的参数值。这有助于防止模型过度依赖少数特征或参数，降低模型的复杂性，并提高模型的稳定性，但也会一定程度上制约模型向训练集上损失函数更小的方向过快收敛。这种制约可以防止模型对训练数据中的噪声或异常值过度拟合，从而使模型对新数据的预测能力更强。合适的正则化系数可以帮助模型更好地捕捉数据中的一般模式，而不是过分关注训练数据的细节。但过大的正则化系数会鼓励模型选择较简单的参数设置，从而降低模型的复杂性，影响模型的实际性能。故在实际使用中应视具体情况选择适当的系数。

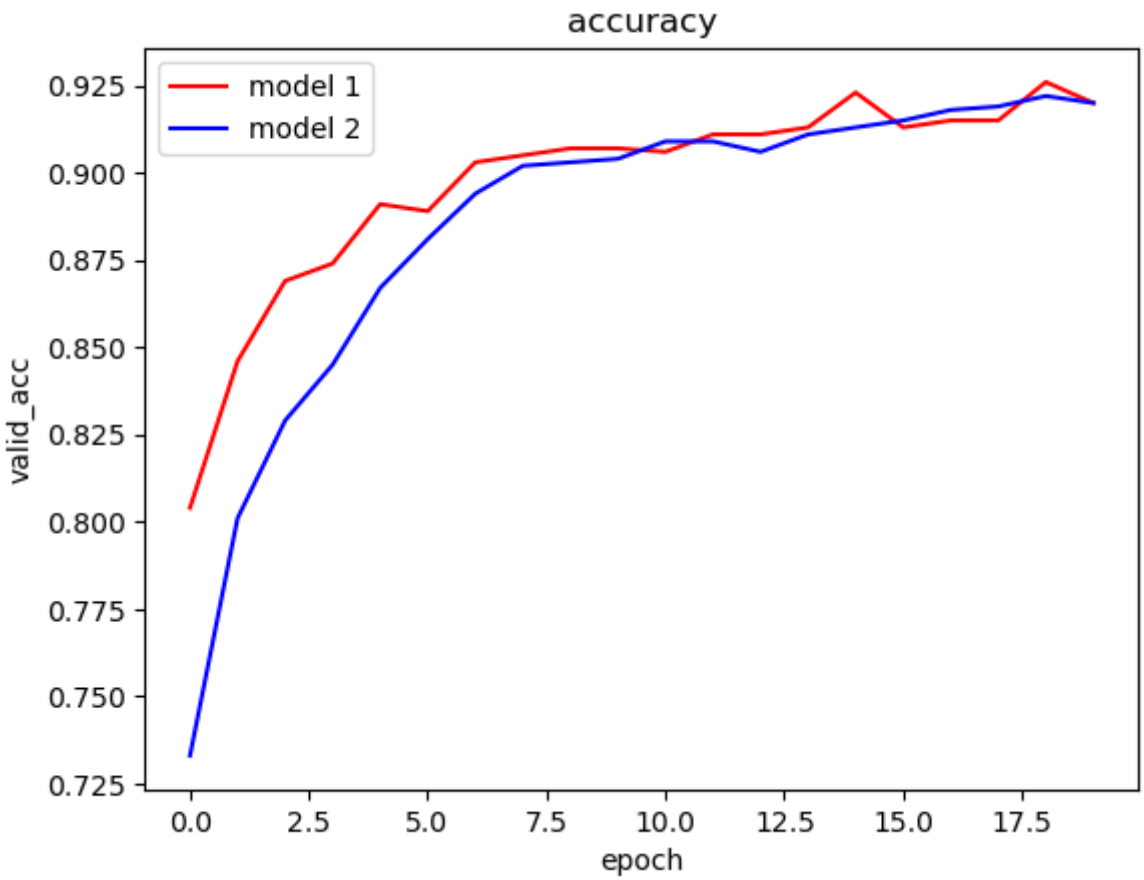
3.5 使用softmax前后的对比

下面分别对以交叉熵作为损失函数的模型和以MSE作为损失函数的模型进行训练并比较两者的性能。其训练代码如下：

```
# 两种损失函数的对比
bp_net_1 = back_propagation(layers=layers, dropout=dropout, classifacation=True)
# 这里只需将classification设为 False 即可选择损失函数为不带softmax的MSE
bp_net_2 = back_propagation(layers=layers, dropout=dropout, classifacation=False)

_, valid_accs_1, train_losses_1 = bp_net_1.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum)
_, valid_accs_2, train_losses_2 = bp_net_2.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum)
```

其loss曲线、valid accuracy曲线的对比图如下：





其最终在测试集上的预测准确率的对比如下：

```
Accuracy_1: 0.914
Accuracy_2: 0.908
```

可见选用softmax和交叉熵函数的网络的loss曲线收敛的更为迅速，且最终对应的loss值也更小，其训练过程中在验证集上的准确率也显著更高，最终在测试集上的预测准确率也高于使用MSE的模型。

这是因为，Softmax交叉熵损失函数适用于多类别分类问题，而均方误差损失函数通常用于回归任务。在多类别分类问题中，Softmax函数可以将模型的输出映射到[0, 1]之间，并且保证所有类别的概率之和为1，使其转化为各个类别的概率分布，更具概率解释性。而MSE损失函数则不适合直接用于多类别分类，因为它对于每个类别只有一个输出值，无法得到更具实际意义的概率分布。

此外，Softmax交叉熵损失函数的梯度特性更适合优化算法。Softmax交叉熵函数对于模型输出的微小变化，会产生较大的梯度信号，这有助于快速收敛和更稳定的训练过程。相比之下，MSE损失函数的梯度在较远离目标值的区域会很小，导致梯度下降过程较慢。

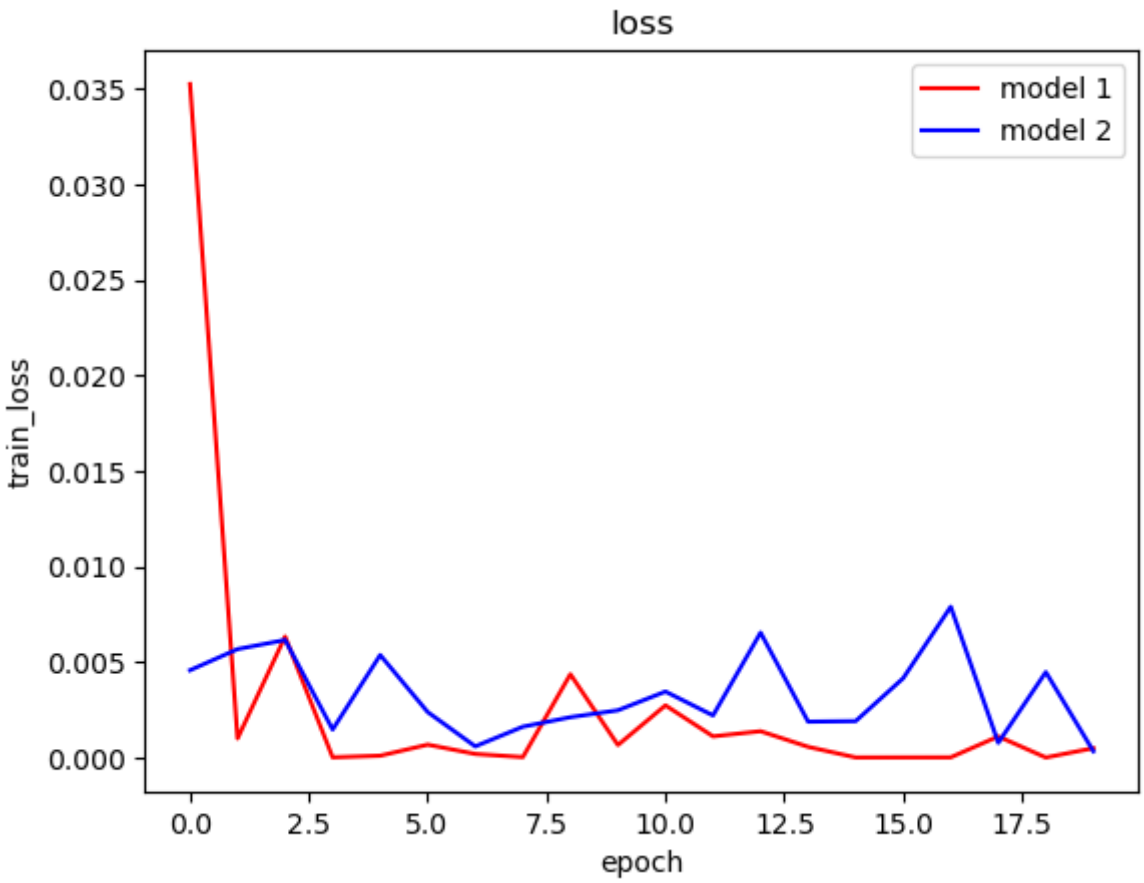
并且，Softmax交叉熵损失函数对于多类别不平衡问题更加鲁棒。在多类别分类中，不同类别的样本数量可能不平衡。Softmax交叉熵损失函数能够充分考虑每个类别的损失贡献，并且对于少数类别的错误分类给予更大的惩罚。而MSE损失函数在类别不平衡问题中可能会导致模型偏向于多数类别。

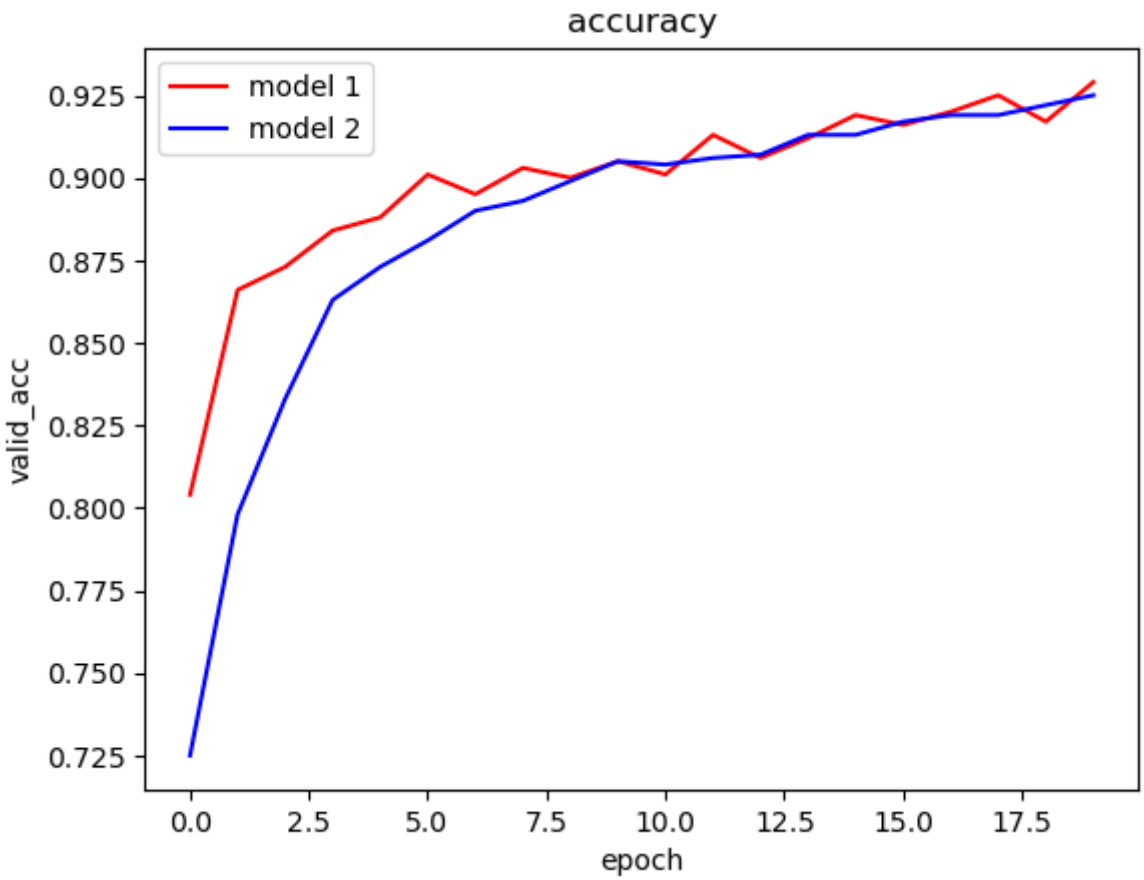
3.6 每层均添加偏置的效果

下面分别对每层均有偏置和仅有第一层有偏置的模型进行训练并比较两者的性能，其训练代码如下：

```
# 对偏置的比较
bp_net_1 = back_propagation(layers=layers, dropout=dropout, classifacation=True)
bp_net_2 = back_propagation(layers=layers, dropout=dropout, classifacation=False)

_, valid_accs_1, train_losses_1 = bp_net_1.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum)
_, valid_accs_2, train_losses_2 = bp_net_2.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum,
full_bias=False)
```

其loss曲线、valid accuracy曲线的对比图如下：





其最终在测试集上的预测准确率的对比如下：

```
Accuracy_1: 0.92
Accuracy_2: 0.9
```

可见每层均带偏置的模型的loss曲线收敛更为迅速稳定，且在验证集上的准确率也更高、提高更快。

这是因为每层都带有偏置可以引入额外的自由参数，从而增加神经网络的表达能力。偏置可以使每个神经元具有不同的偏移量，允许网络在输入空间中学习出更加复杂的决策边界。这有助于提高网络对输入数据的拟合能力和表示能力。

此外，每层都带有偏置可以改善网络的收敛性和学习速度。偏置相当于对输入数据进行了平移，使激活函数在输入空间中更灵活地调整。这有助于减小激活函数的饱和区域，避免梯度消失问题，提高了网络对梯度的敏感性，从而加速了网络的收敛过程。

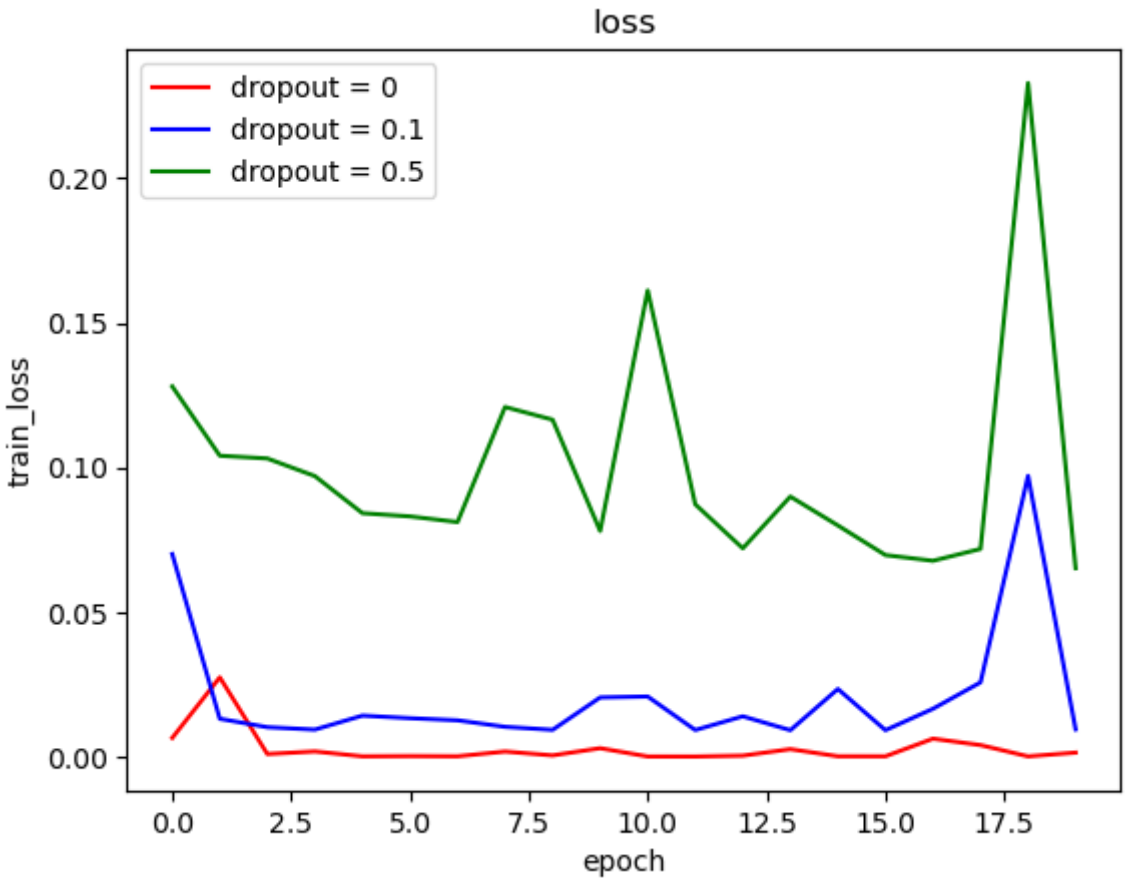
并且，每层都带有偏置可以提高网络的鲁棒性和泛化能力。偏置可以使网络对输入数据的变化更加稳定，降低对输入数据的微小扰动的敏感性。这有助于减少过拟合问题，提高网络在未见过的数据上的泛化能力。

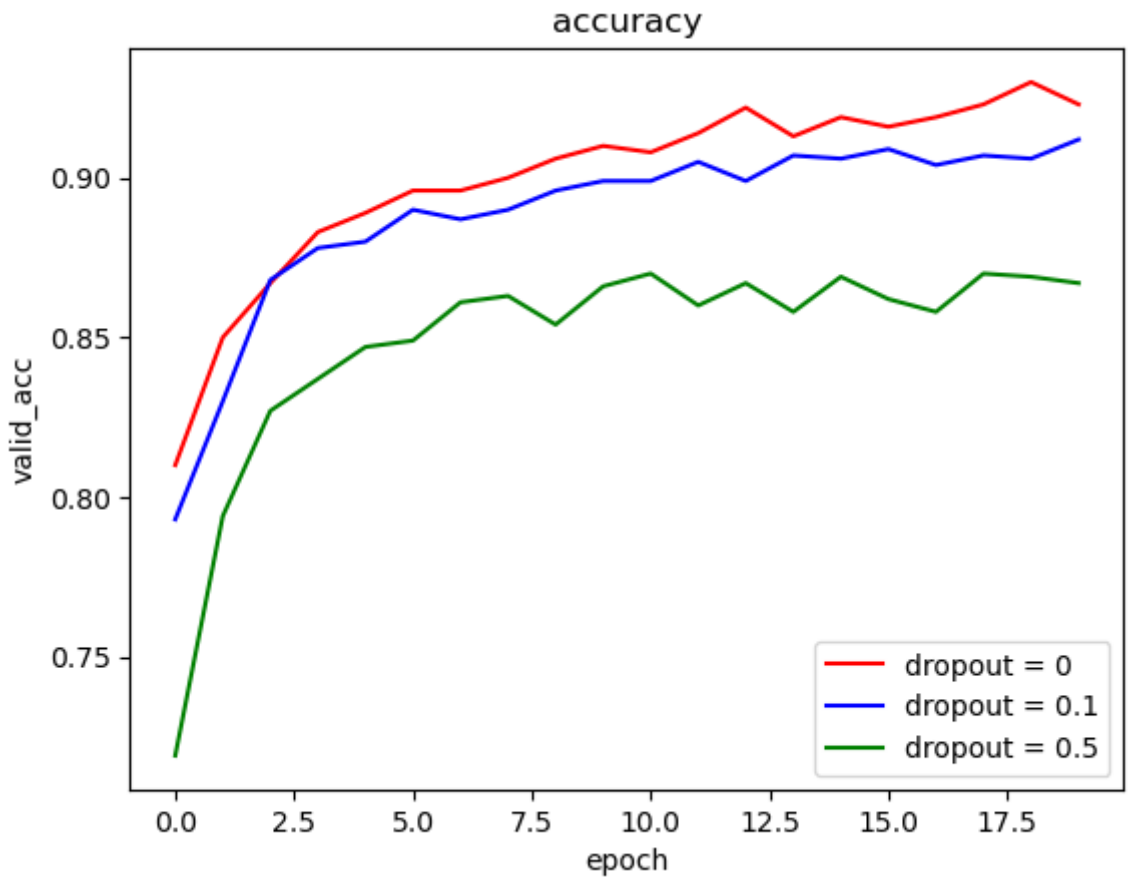
3.7 dropout的效果

下面分别对不同dropout rate的模型进行训练并进行性能比较。其训练代码如下：

```
# 对 dropout 的比较
dropout_1 = 0
dropout_2 = 0.1
dropout_3 = 0.5

bp_net_1 = back_propagation(layers=layers, dropout=dropout_1, classifacation=True)
bp_net_2 = back_propagation(layers=layers, dropout=dropout_2, classifacation=True)
bp_net_3 = back_propagation(layers=layers, dropout=dropout_3, classifacation=True)

_, valid_accs_1, train_losses_1 = bp_net_1.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg_1, momentum)
_, valid_accs_2, train_losses_2 = bp_net_2.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg_2, momentum)
_, valid_accs_3, train_losses_3 = bp_net_3.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg_3, momentum)
```

其loss曲线、valid accuracy曲线的对比图如下：





其最终在测试集上的预测准确率的对比如下：

```
Accuracy_1: 0.919
Accuracy_2: 0.912
Accuracy_3: 0.866
```

可见随着dropout rate的提高，模型的loss曲线反而更加不稳定，在验证集和测试集上的效果也反而更差。

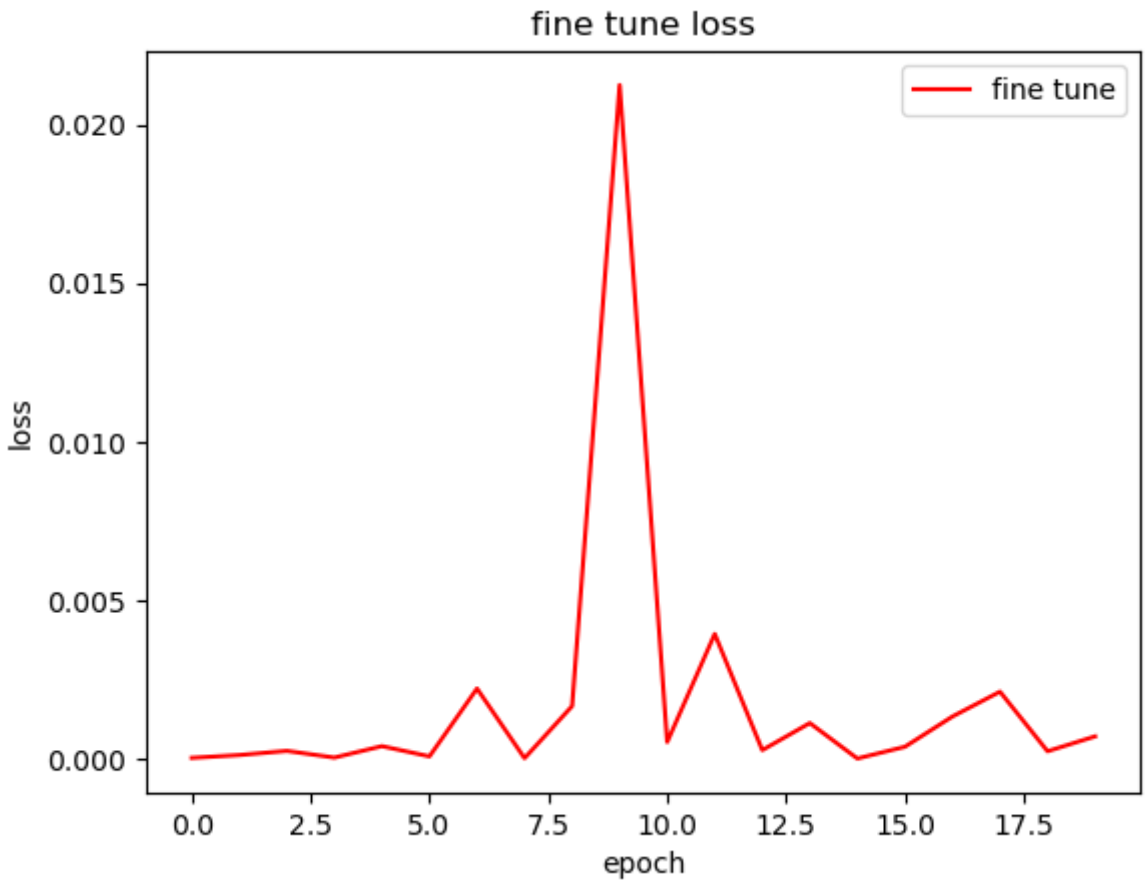
我们猜测，是因为Dropout会随机地将一部分神经元的输出置为零，这意味着在训练过程中一些重要的特征信息可能会被丢弃。特别是在网络较浅或数据集较小的情况下，高Dropout率可能会导致模型难以学习到足够的有效特征表示，从而影响性能，并且较高的Dropout率会增加模型的随机性，使得模型更加不稳定。故dropout在较小的模型上并不一定拥有更好的效果。

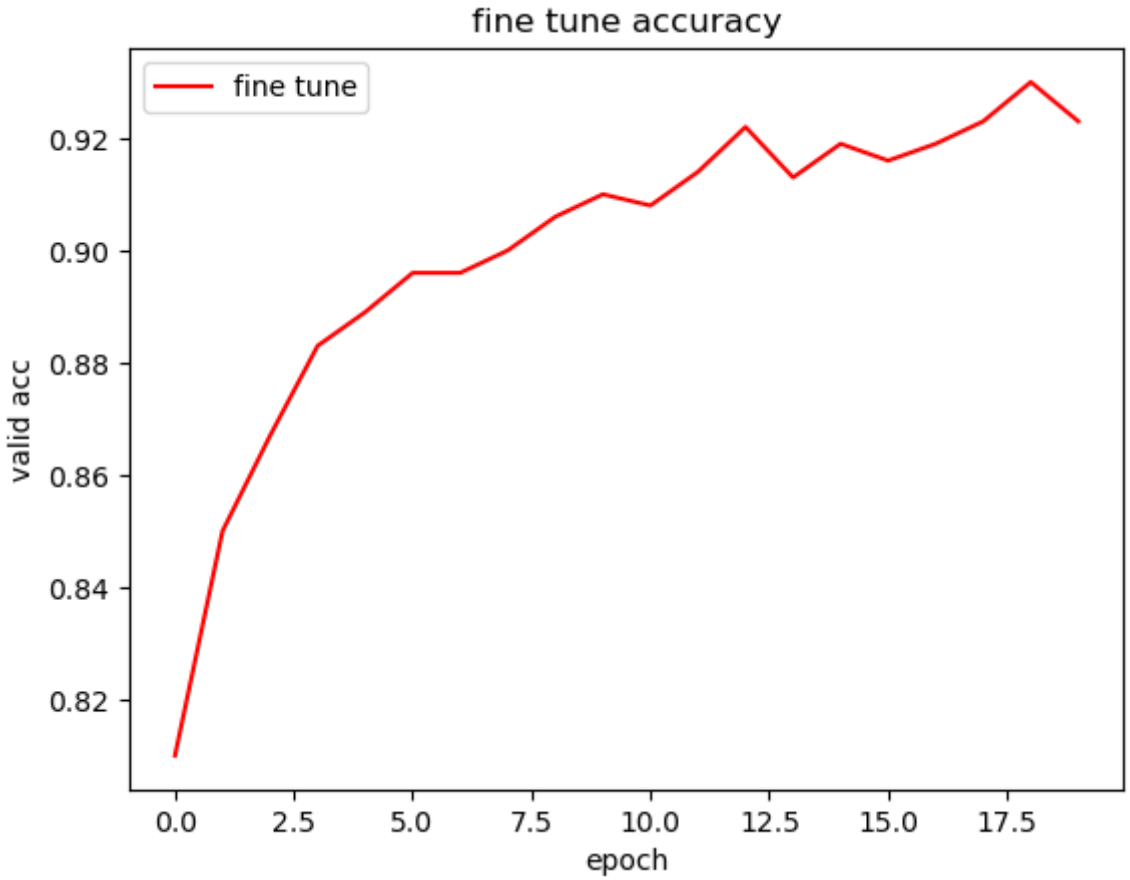
3.8 BP网络的微调效果

模型的微调代码如下：

```
# 模型训练
bp_net = back_propagation(layers=layers, dropout=dropout, classifacation=True)
_, valid_accs, train_losses = bp_net.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum)

# 对 bp_net 最后一层进行微调
_, valid_accs, ft_losses = bp_net.fine_tune(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size, epochs, learning_rate, l2_reg, momentum)
```

其loss曲线、valid accuracy曲线如下：





其与微调前的模型在测试集上的预测准确率的对比如下：

```
Accuracy_0: 0.902
Accuracy_1: 0.916
```

可见虽然模型在微调期间的损失函数值没有显著下降，但其在验证集上的准确率得到了显著提升，在测试集上的准确率也高于微调前的模型。

这是因为通过微调最后一层，可以使网络更好地适应具体的任务，进一步优化模型的输出，提高模型在该任务上的性能，并能加速网络的收敛速度。底层的隐藏层通常具有较大的参数量和较强的特征提取能力，而微调最后一层可以使网络更快地适应任务，减少训练时间和计算成本。

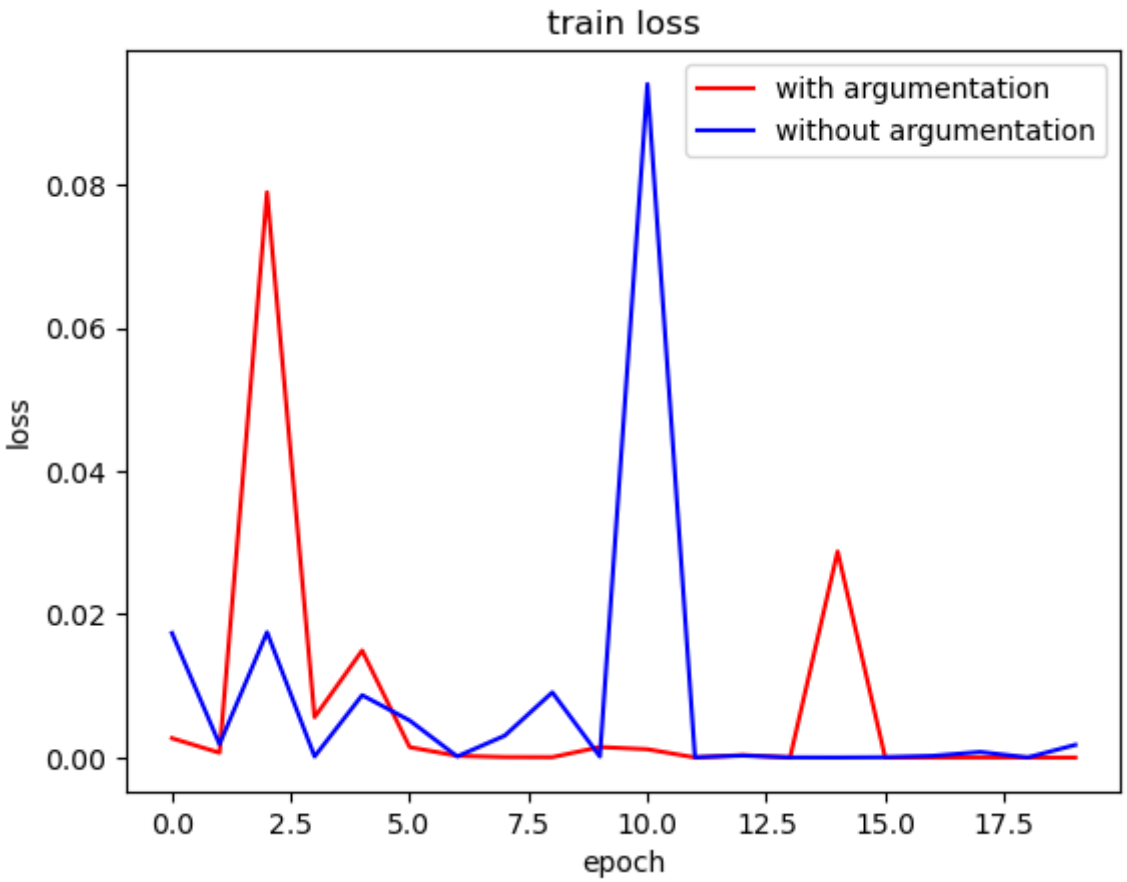
3.9 数据增强的效果

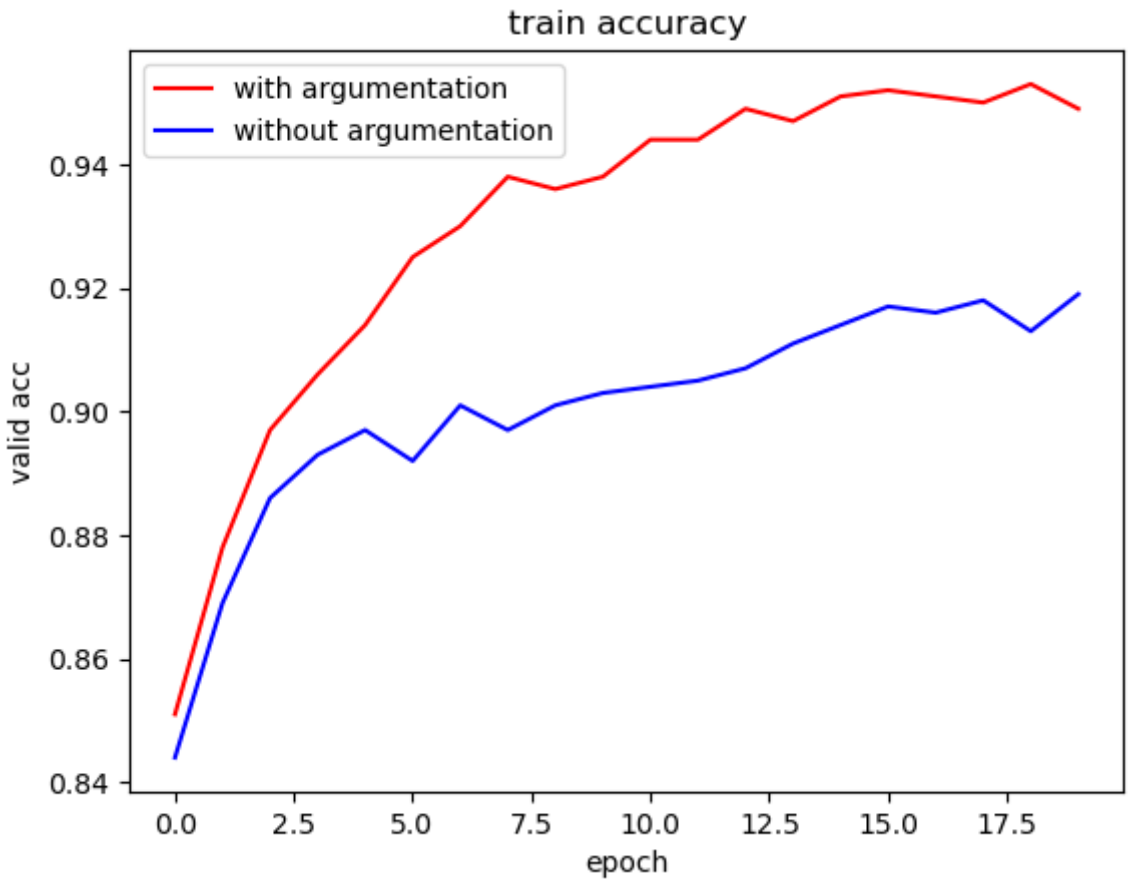
下面是数据增强部分的代码：

```
# 对训练集做数据增强（加噪，平移和裁剪）
shape = (28, 28)
noised_imgs = add_noise(train_imgs, noise_rate=0.2)
trans_imgs = image_translation(train_imgs, shape)
cropped_imgs = random_crop_and_pad(train_imgs, shape)
# 合并训练集
train_imgs = np.concatenate((train_imgs, noised_imgs, trans_imgs, cropped_imgs),
axis=0)
train_labels = np.concatenate((train_labels, train_labels, train_labels,
train_labels), axis=0)
train_label_vectors = np.eye(num_labels)[train_labels]

# 以下方法均在utils.py文件中
# 添加噪声
def add_noise(matrix, noise_rate):
    rows, cols = matrix.shape
    noisy_matrix = np.copy(matrix)
    indices = np.random.choice(rows * cols, int(rows * cols * noise_rate),
replace=False)
    noisy_matrix.flat[indices] = 0
    indices = np.random.choice(rows * cols, int(rows * cols * noise_rate),
replace=False)
    noisy_matrix.flat[indices] = 255
    return noisy_matrix

# 图像平移
def image_translation(images, shape):
    trans_images = np.copy(images).reshape((images.shape[0], ) + shape)
    for i in range(trans_images.shape[0]):
        x_offset = np.random.randint(-2, 3)
        y_offset = np.random.randint(-2, 3)
        new_image = shift(trans_images[i], [y_offset, x_offset], cval=255)
        trans_images[i] = new_image
    return trans_images.reshape(images.shape)

# 图像的裁剪与填充
def random_crop_and_pad(images, shape):
```

```
    width, height = shape
    left = random.randint(0, width - 22)
    top = random.randint(0, height - 22)
    right = left + 22
    bottom = top + 22

    cropped_imgs = np.copy(images).reshape((images.shape[0],) + shape)
    for i in range(images.shape[0]):
        cropped_img = cropped_imgs[i, top:bottom, left:right]

        new_img = np.ones((28, 28), dtype=np.uint8) * 255
        pad_left = (28 - 22) // 2
        pad_top = (28 - 22) // 2
        new_img[pad_top:pad_top + cropped_img.shape[0], pad_left:pad_left +
cropped_img.shape[1]] = cropped_img
        cropped_imgs[i] = new_img

    return cropped_imgs.reshape(images.shape)
```

下面是使用原始数据集和数据增强后的数据集进行训练的模型的loss曲线、valid accuracy曲线的对比图：





其最终在测试集上的预测准确率的对比如下：

```
Accuracy: 0.907
Accuracy: 0.948
```

可见经过数据增强后，我们的训练集的数量整整提升到了原来的四倍，模型的loss曲线收敛得更加稳定迅速，并且在验证集和测试集上的准确率也显著高于原始模型。

这是因为，通过数据增强，可以生成更多的训练样本，从而扩充训练集的规模。更多的样本可以提供更多的信息，使模型更好地学习数据的分布和特征，减少过拟合的风险。此外还能增加数据多样性，模型能够更全面地学习数据的不同变化模式和特征，提高模型对多样数据的泛化能力。

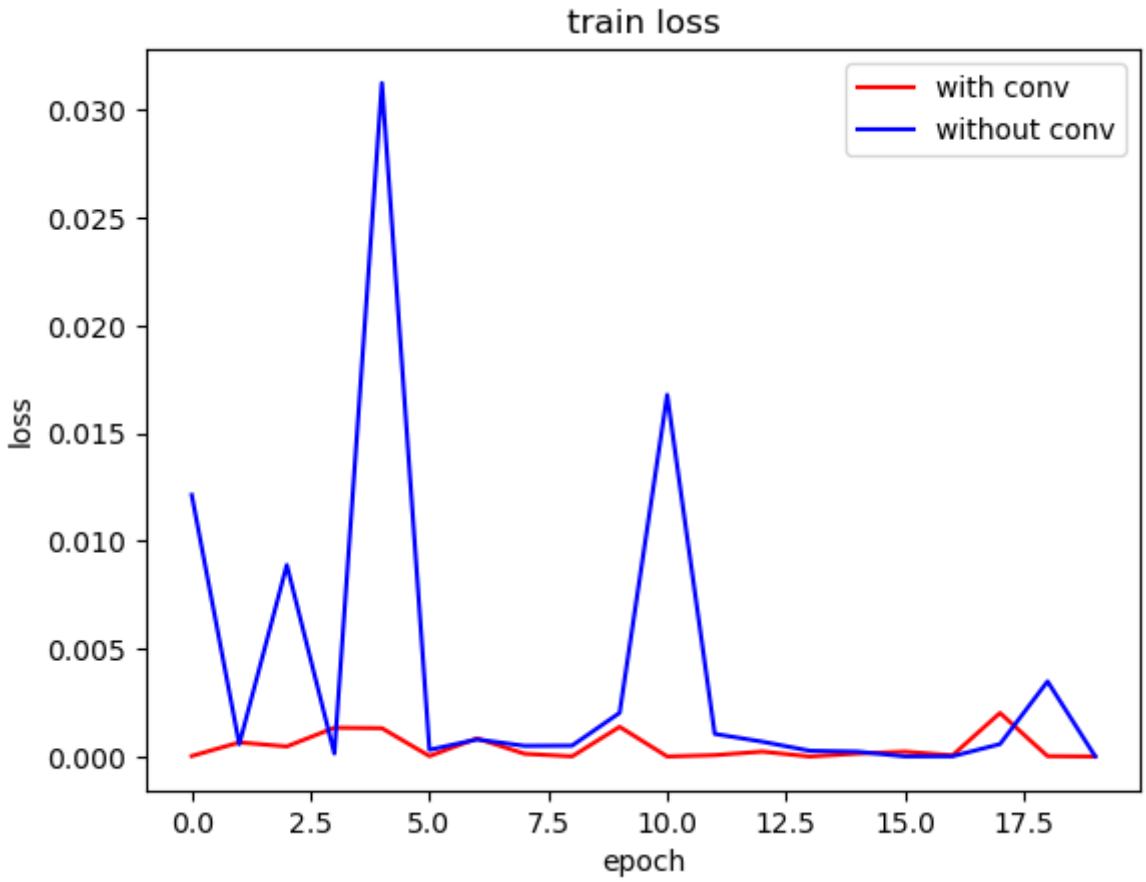
并且，数据增强能够引入一定程度的随机性，模型在训练过程中能够更好地适应不同的输入变化。这提高了模型对噪声、图像变形、光照变化等干扰的鲁棒性，使得模型在实际应用中更具可靠性。数据增强还可以有效缓解过拟合问题，尤其在数据量较少的情况下。通过增加样本的多样性和数量，数据增强使得模型难以过多地记住训练样本的细节，而更关注于学习数据的一般特征和模式，从而提高了模型的泛化能力。

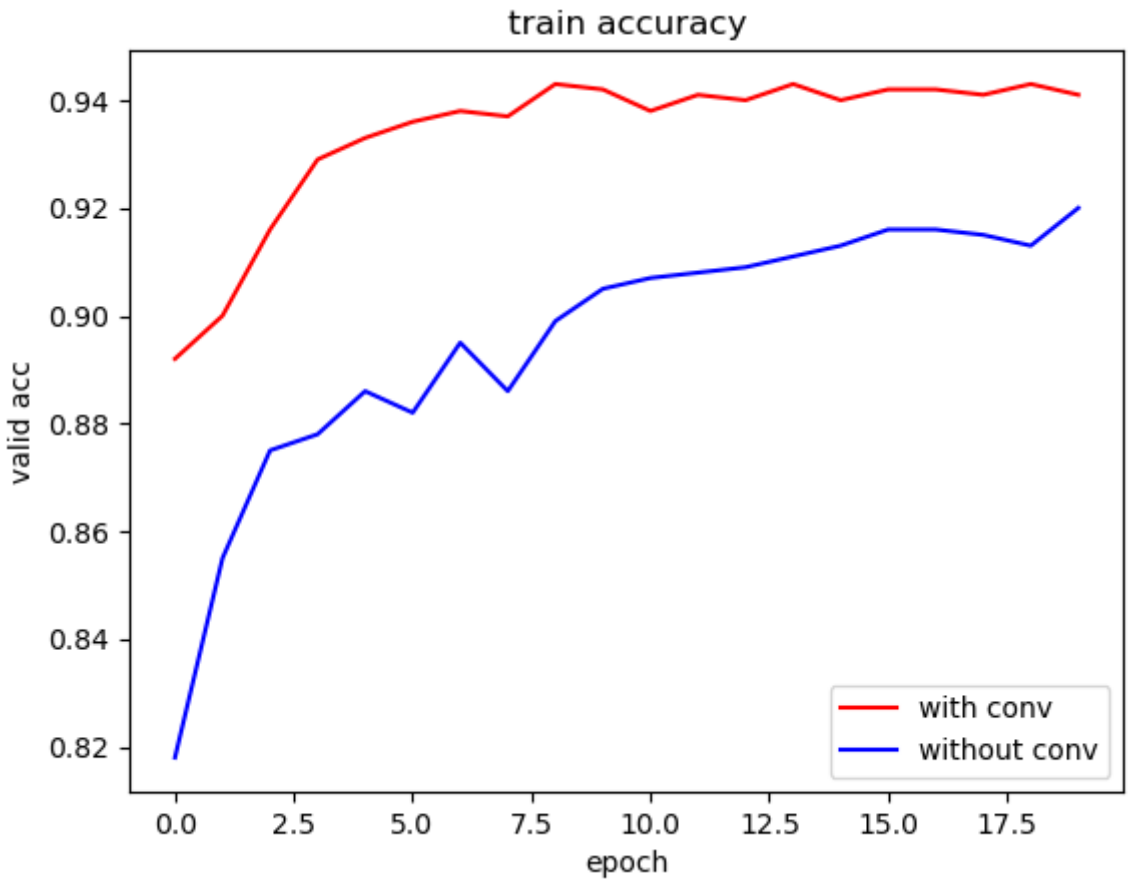
3.10 添加卷积层的效果

以下是带卷积层网络的训练代码：

```
# 添加卷积层后的效果
in_channels = 1
out_channels = 3
kernel_size = (5, 5)
layers = [out_channels * 28 * 28, 256, 128, num_labels]
conv_bp = Conv_BP(in_channels=in_channels, out_channels=out_channels,
kernel_size=kernel_size, layers=layers, padding=2, dropout=dropout,
classifacation=True)

# 训练模型
train_imgs = np.reshape(train_imgs, (train_imgs.shape[0], 1, 28, 28))
valid_imgs = np.reshape(valid_imgs, (valid_imgs.shape[0], 1, 28, 28))
test_imgs = np.reshape(test_imgs, (test_imgs.shape[0], 1, 28, 28))
_, valid_accs, train_losses = conv_bp.train(train_imgs, train_label_vectors,
valid_imgs, valid_labels, batch_size=batch_size, epochs=epochs, lr=learning_rate,
l2_regularization=l2_reg, momentum_strength=momentum)
```

以下是两种网络的loss曲线、valid accuracy曲线的对比图：





以下是两者在最终测试集上的准确度的对比：

```
Accuracy: 0.907
Accuracy: 0.934
```

可见使用卷积层后，模型的loss曲线收敛得更加稳定迅速，并且在验证集和测试集上的准确率也显著高于原始模型。

这是因为卷积层可以有效地提取输入数据中的局部特征。在手写字体分类任务中，每个字符的局部特征对于正确分类非常重要。通过卷积层，网络可以自动学习到这些局部特征的模式，从而更好地表示和区分不同的手写字符。

此外，卷积层具有参数共享的特性，即卷积核在不同位置上共享权重。这样可以极大地减少网络的参数量，使得网络更加紧凑和高效。对于手写字体分类任务，由于字符的局部模式通常在整个图像中具有相似性，参数共享可以更好地捕捉到这些共享模式，提高特征提取的效率和准确性。

并且，卷积层具有平移不变性的特性，即对输入的平移操作不敏感。在手写字体分类中，字符的位置可能会有所变化，但字符的形状和结构保持不变。卷积层可以通过局部感受野和参数共享，使得网络对字符的平移具有鲁棒性，提高模型的泛化能力。