Project 1 实验文档

21307140069 田沐钊

一、 反向传播算法

反向传播中的神经元网络抽象，本质上是依托于对应维度的权重矩阵而存在的。故若要实现可伸缩易调整的网络结构，只需根据输入的层数、神经元个数生成对应的权重矩阵、偏置即可。

此外，为了使网络能在不同的结构下均能有效工作，支持不同的参数设计，我们需要推导出权重更新的普遍通解。

（一） 反向传播的参数更新公式推导

以下以损失函数为平方和、激活函数为 sigmoid 函数的情况为例：

我们记 L 为总层数（包括输入层、隐含层与输出层），a_i 为第 i 层的输出向量，b_i 为第 i 层的输入向量。W_i 为第 i 层到第 i+1 层的权重矩阵，β_i 为偏置。

首先求损失函数值对 W_{L-1} 的梯度如下：

$$\frac{\partial Loss}{\partial a_L} = (a_L - y)$$

$$\frac{\partial Loss}{\partial W_{(L-1)j,i}} = \sum_{k=1}^{O_L} \frac{\partial Loss}{\partial a_{Lk}} \cdot \frac{\partial a_{Lk}}{\partial b_{Lk}} \frac{\partial b_{Lk}}{\partial w_{(L-1),j,i}}$$

$$= (a_{Li} - y_i) \times a_{Li}(1 - a_{Li}) \times a_{(L-1),j}$$

$$\text{记} \quad \Delta_i = \begin{bmatrix} a_{i1}(1-a_{i1}) \\ \vdots \\ a_{iO_i}(1-a_{iO_i}) \end{bmatrix} = a_i \odot (1 - a_i)$$

引入新符号，可以表示梯度为

$$\frac{\partial Loss}{\partial W_{L-1}} = \left(\frac{\partial Loss}{\partial a_L} \odot \Delta_L \right) a_{L-1}^T$$

并且易得对偏置的梯度为

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

接着求下一层，有

$$\frac{\partial Loss}{\partial a_{L-1}} = \frac{\partial Loss}{\partial a_L} \cdot \frac{\partial a_L}{\partial a_{L-1}} \cdot \frac{\partial b_L}{\partial a_{L-1}}$$

$$= W_{L-1}^{T}\left((a_L - y) \odot \Delta_L\right)$$

$$\frac{\partial Loss}{\partial W_{(L-2),ij}} = \sum_{k=1}^{n_{L-1}} \frac{\partial Loss}{\partial a_{L-1,K}} \cdot \frac{\partial a_{L-1,K}}{\partial b_{(L-1),K}} \cdot \frac{\partial b_{(L-1),K}}{\partial W_{(L-2),ij}}$$

$$\text{易得} \quad \frac{\partial Loss}{\partial W_{L-2}} = \left(W_{L-1}^{T} \frac{\partial Loss}{\partial a_L} \odot \Delta_L \odot \Delta_{L-1}\right) \cdot a_{L-2}^{T}$$

$$\frac{\partial Loss}{\partial \beta_{L-1}} = W_{L-1}^{T} \frac{\partial Loss}{\partial a_L} \odot \Delta_L \odot \Delta_{L-1}$$

故类推可得通解：

$$\frac{\partial Loss}{\partial \beta_i} = \left(\prod_{k=i+1}^{L-1} W_k^{T}\right) \frac{\partial Loss}{\partial a_L} \odot \prod_{s=i+1}^{L} (\Delta_L)$$

$$\frac{\partial Loss}{\partial W_i} = \frac{\partial Loss}{\partial \beta_i} \cdot a_i^{T}$$

$$\frac{\partial Loss}{\partial \beta_{i-1}} = W_i^{T} \frac{\partial Loss}{\partial \beta_i} \odot \Delta_i$$

其中损失函数对输出层输出的梯度部分对于其他损失函数也一样适用，而$\triangle_L$的第 j 项其实就是 a_{ij} 对 b_{ij} 的导数，根据不同的激活函数进行微调即可。

该形式具有良好且间接的递推性质，很方便在程序中实现。

（二） 代码结构

以下是该实验所创建 BP 网络的类的初始化函数：

```
class back_propagation_:

    def __init__(self, layers, learning_rate=0.05, batch_size=16,
                 epochs=300, dropout=0, classifacation=False,
                 activate='leakyrelu', X_valid=[], y_valid=[]):
        self.layers = layers
        self.num_layers = len(layers)
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.epochs = epochs
        self.dropout = dropout
        self.classifacation = classifacation
        if self.classifacation is True:
            self.loss = 'cross_entropy'
        else:
            self.loss = 'mean_squared_error'
        self.activate = activate
        self.Weights = []
        self.Biases = []
        self.X_valid = X_valid
        self.y_valid = y_valid
        self.initialize_weights_and_biases()
```

其中 layers 为一个记录每层神经元个数的数组，程序依据该数组初始化权重矩阵。此外，该网络也支持人为设置学习率、迭代次数、激活函数，并且还具有 dropout 功能。

以下为前向传播函数的实现

```
 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(sel
f.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(s
elf.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])

        return activations
```

以下为后向传播函数的实现

```
def backward_propagation(self, activations, y):
        _, loss_grad = self.loss_compute(y, activations[-1])
        gradients_W = []
        gradients_b = []

        deltas = self.get_deltas(activations)

        # 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] *
activations[-2][:, np.newaxis, :])
        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] *
activations[-i-1][:, np.newaxis, :])

        return gradients_W, gradients_b
```

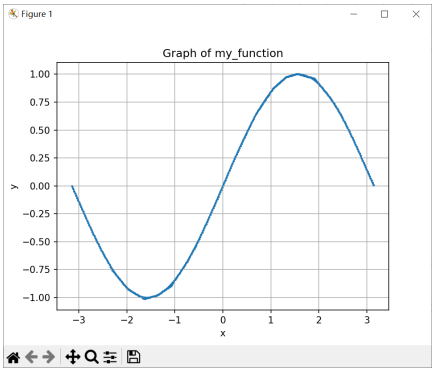
以下为部分训练函数的具体实现：

```
for i in range(self.epochs):
    mapping = np.arange(X.shape[0])
    np.random.shuffle(mapping)

    for j in range(0, X.shape[0], self.batch_size):
        if j + self.batch_size > X.shape[0]:
            X_batch = X[mapping[-self.batch_size:], :]
            y_batch = y[mapping[-self.batch_size:]]
        else:
            X_batch = X[mapping[j:j+self.batch_size], :]
            y_batch = y[mapping[j:j+self.batch_size]]
        activations = self.forward_propogation(X_batch)
        gradients_W, gradients_b = self.backward_propagation(activations, y_batch)
        self.update_weights(gradients_W, gradients_b)
```

（三） 对正弦曲线的拟合：

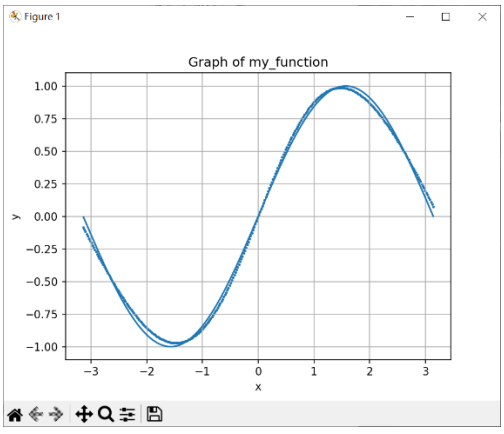
这里通过对正弦函数的拟合来验证网络性能，并尝试比较不同激活函数对拟合效果的影响。

该实验把网络结构设为［ 1，16，32，1 ］。我们首先使用 leakyrelu 函数作为激活函数，将 batch_size 设置为 16，将学习率设置为 0.01，设置 800 个 epoch。得到的拟合效果如下。

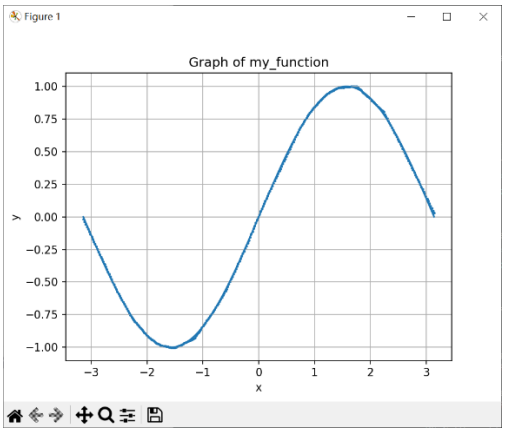


其中散点线为拟合出的图像，可见拟合效果良好。并且对于测试数据求出的平均误差为 0.006447873578030209，说明精度较高。

但当把激活函数换为 simoid 且其它参数不变时，得到的拟合图象如下：



可以看到预测图象与实际图象有较大偏移。将 epoch 增大为 1000，拟合效果得到改善，结果如下：



分析可得一开始 sigmoid 函数拟合效果欠佳的原因是训练次数不够，重复多次后依然是同样的现象。说明 leakyrelu 作为激活函数时，模型的训练速度是快于 sigmoid 函数的。原因可能是因为 Sigmoid 函数在输入远离零时，具有饱和性，即函数的导数接近于零。这会导致在反向传播过程中，当梯度通过网络反向传递时会逐渐变小，导致网络学习缓慢，使梯度信息无法有效地传递到较早的层。

（四） 对 12 个手写汉字进行分类

这里，为了主动增加训练数据，我们对已有的训练图像采取加噪声、裁剪、平移等操作，作为数据增强。通过这种方式，可以生成更多的训练样本，有助于提高模型的泛化能力和鲁棒性，减少过拟合的风险，帮助模型学习更丰富的特征和模式。

下面是具体的数据增强代码：

```
for j in range(1, 521):
    image_path = os.path.join(subfolder_path, f'{j}.bmp')
    image = Image.open(image_path)
    gray_image = image.convert('L')
    array = np.array(gray_image).flatten()
    image_train.append(array/255)

    noised_image = add_noise(np.array(gray_image)).flatten()
    image_train.append(noised_image/255)

    translated_image = image_translation(gray_image)
    image_train.append(np.array(translated_image).flatten()/255)

    croped_image = random_crop_and_pad(gray_image)
    image_train.append(np.array(croped_image).flatten()/255)
```

在该实验中，我们将从以下几个方面探究网络参数对模型性能的影响：

① 不同激活函数的表现差异。

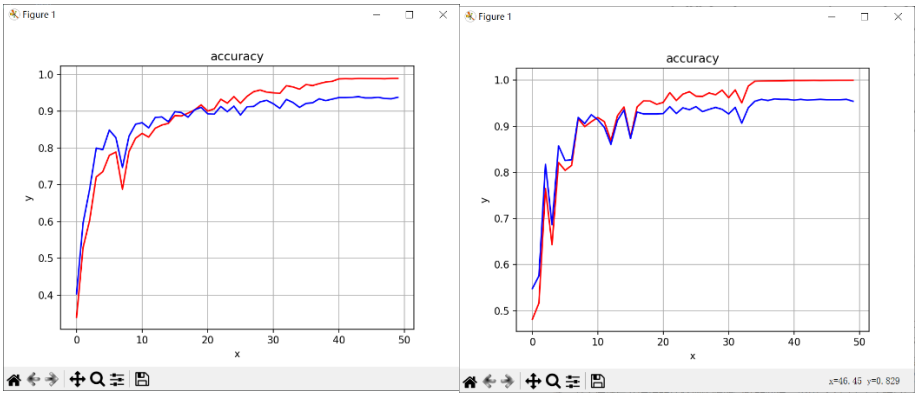
② 网络层数和数据节点数对训练效果的影响。

③ Batch_size 对训练效果的影响。

④ Dropout 对模型训练效果的影响。

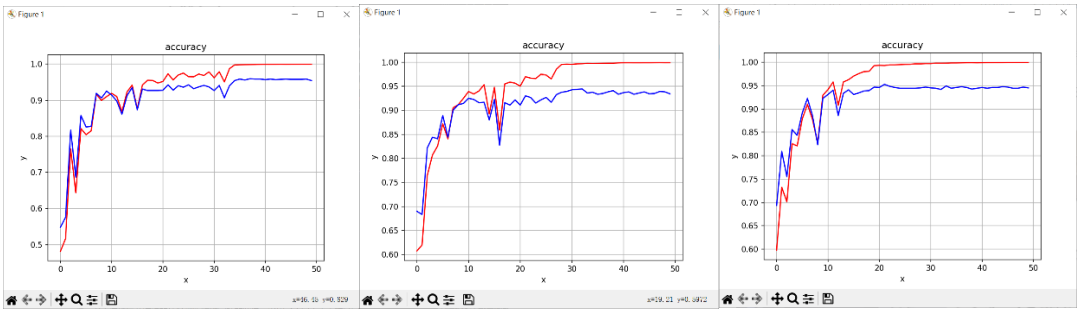
下面我们将逐个展开（以下实验均保证单变量改动）：

（1） 不同激活函数的比较：



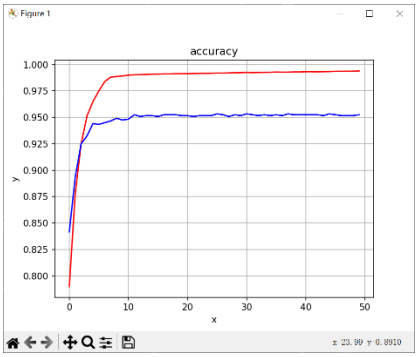
左侧的图像是激活函数是 sigmoid 时的训练结果，右侧则是激活函数为 Leaky ReLU 时的训练结果。其中红色曲线为训练集上的准确率，蓝色曲线为测试集上的准确率。可以看相比于 sigmoid，Leaky ReLU 作为激活函数时的准确率增长更为快速，且收敛得到的准确率结果也较高。分析原因为 Leaky ReLU 函数在正半轴上是线性的，故其它的计算效率更高。此外，由于 sigmoid 函数的导数在接近饱和区域时较小，故可能导致后层网络的信息无法有效回传，导致效率较低，且最后的收敛结果相对较差。

（2） 不同神经节点数和不同网络层数的比较：



首先是对网络神经节点数的比较，以上三张图对应的网络均有两层隐藏层，而神经节点数分别为（256，128），（128，128），（512，128），可以看到在本实验中，神经元节点较多的网络具有较快的收敛速度。分析其原因，可能是因为较大的网络具有更多的参数和自由度，可以学习更复杂的函数映射，从而更好地适应复杂的数据分布和任务要求。相比之下，较小的网络可能无法充分表达复杂的模式和特征，需要更多的训练迭代来达到较好的性能。此外，神经元节点较少的网络在收敛后期存在一定波动，可能是因为其参数较少，使得优化过程更容易陷入局部最优解。在训练过程中，网络可能会在不同的局部最优解之间跳动，导致准确率波动。但就结果来看，三者均能较好地收敛，得到良好的结果。

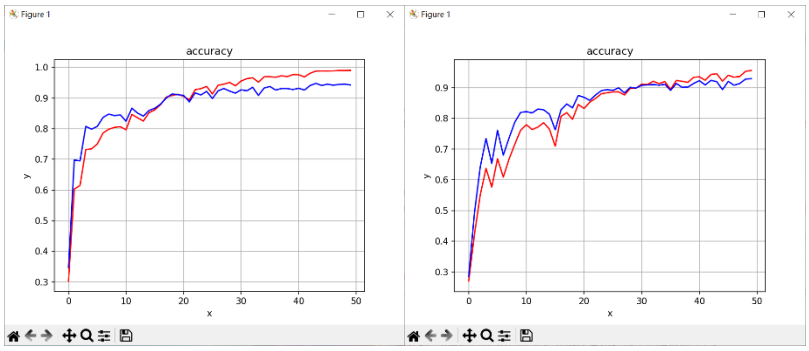
下面将神经网络的层数降低，只使用一层隐藏层，神经元个数为 256。得到的训练结果如下：



可以看到其收敛效果也十分平稳、良好，并且收敛效率相对于二层隐藏层的网络更快。可见对于该问题，一层隐藏层的网络已经能够很好地解决。而盲目地增加隐藏层，只会加大梯度消失、梯度爆炸和过拟合的风险，使网络过于专注训练数据中的噪声和细微变化，无法将性能很好地泛化到训练集以外的数据，导致训练结果的波动。

（3） 不同 batch_size 的比较：

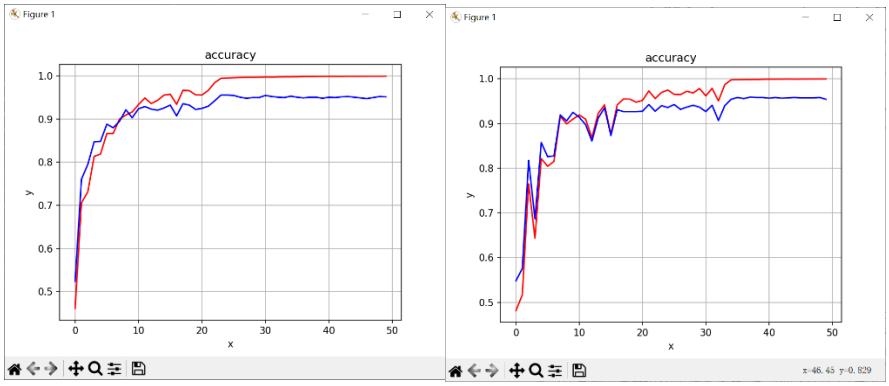
下面对不同的 batch_size 的训练结果进行比较分析。



其中左侧的图像是 batchsize 为 64 时对应的训练结果，而右侧的图像对应的 batchsize 则为 256。可以看到过度提升 batchsize 后，模型的训练效率降低，且结果的收敛稳定性变差，存在明显波动。分析其原因，较大的批量大小可能导致梯度估计的不准确性增加。在每个批次中，通过计算批量内样本的平均梯度来更新模型参数。当批量大小较大时，批内样本的平均梯度可能会更加稳定，但也会因此丧失了一些样本的个体特征。尤其是不同样本计算得到的梯度存在冲突时，求平均可能会使其模型性能变差。这或将导致模型对于个别样本的学习能力下降，降低了模型的泛化能力。

（4） 使用 dropout 的效果：

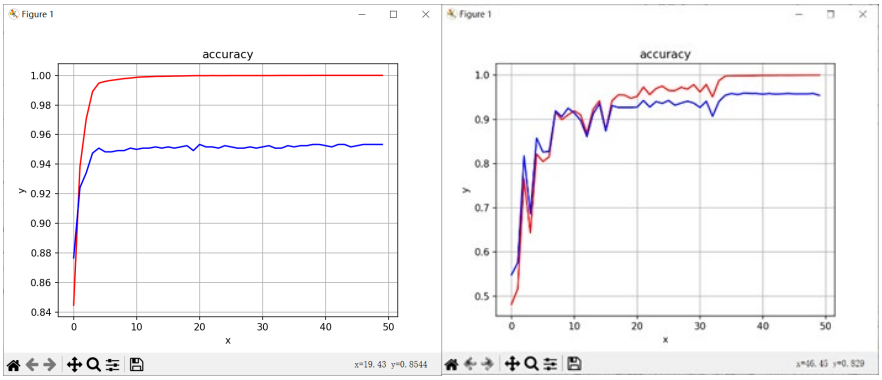
下面对使用 dropout 的模型效果进行分析：



其中左侧图像未使用 dropout，而右侧图像使用了概率为 0.1 的 dropout 层，可以看到使用 dropout 的模型虽然在训练过程中存在波动，但是增长速度相对较快，且收敛结果相对更好。这是因为，通过随机丢弃神经元，dropout 可以减少神经网络中的共适应性，即每个神经元不能过度依赖其他特定的神经元，有助于防止过拟合，使得网络能够更好地泛化到未见过的数据。同时，其可视为一种模型集成方法，通过在每个训练迭代中使用不同的子网络，可以得到多个不同的网络模型。这些子网络的预测结果可以结合起来，从而提高模型的鲁棒性和稳定性，减少对于特定样本的过度依赖，有利于更快地优化和收敛。

除此了网络参数的设置外，还有一个值得发掘的点。在进行数据处理时，由于图像像素的灰度值基本只有 0 和 255，故将其归一化为了 0 和 1 进行训练。在这个过程中，字迹

对应的黑像素点被置为 0，空白背景对应的白像素点被置为 1。但考虑到，对于神经网络来说，输入值为 1 的特征更容易被捕捉到，也更容易对网络产生影响。可以认为在这个过程中网络更关注空白背景的分布。而这明显是不符合直觉上的字体识别常识的，神经网络理应对字迹本身的位置、形状特征更加关心，这些才是更为有效的信息。所以如果将图像中黑像素点置为 1，白像素点置为 0，或许可以得到更好的训练效果。下面我们做实验验证。



其中左侧为对图像进行特殊处理后的训练效果，而右侧未经过特殊处理。可以发现将图像取反后的训练效果极佳，一定程度上说明假设正确。

二、卷积神经网络

（一） 对卷积神经网络的理解

卷积神经网络相对于普通 BP 神经网络，利用了卷积层和池化层的结构，可以对输入数据的局部区域进行感知。卷积层通过共享参数来提取输入数据的局部特征，这样可以显著减少网络中的参数量。并实现参数共享。参数共享的本质是假设输入数据的统计特性在空间上是平稳的，因此可以在不同的位置使用相同的权重来检测相似的特征。这使得 CNN 对于图像、语音等具有局部结构的数据具有更好的建模能力。

此外，由于卷积层中的参数共享，CNN 在平移下具有不变性。这意味着当输入数据在图像中平移时，网络对于相同的特征可以给出相似的响应。对于处理图像中的物体识别、目标检测等任务非常重要。

同时，CNN 通过堆叠多个卷积层和池化层，可以逐渐提取出越来越抽象的特征表示。低层的卷积层可以捕捉到边缘、纹理等底层特征，而高层的卷积层可以捕捉到更高级的语义特征。这种多级表示的特性使得 CNNs 在图像识别、目标检测等任务中能够更好地理解和表达数据。

（二） 基于 pytorch 深度学习框架的 CNN 网络

在该实验中，我使用 torch 提供的 Dataset 类构建了自己的图像加载器，并在加载图像时完成对数据的增强（包括裁剪、平移、添加噪声等操作），类保存在 data_loader 中。并利用其提供的 nn 模型尝试构建了多种 CNN 神经网络。下面是部分核心代码展示：

```python
class Pre_processes():
    def image_normalize(image):
        return np.where(np.array(image) > 0, 0, 1)

    # 随机噪声
    def add_noise(image):
        image = np.array(image)
        rows, cols = image.shape
        noisy_image = np.copy(image)
        indices = np.random.choice(rows * cols, 5, replace=False)
        noisy_image.flat[indices] = 0
        indices = np.random.choice(rows * cols, 5, replace=False)
        noisy_image.flat[indices] = 255
        return np.where(noisy_image > 0, 0, 1)

    # 图像平移
    def image_translation(image):
        x_offset = np.random.randint(-2, 3)
        y_offset = np.random.randint(-2, 3)
        new_image = image.transform(image.size, Image.AFFINE, (1, 0, x_offset, 0, 1, y_offset))
        return np.where(np.array(new_image) > 0, 0, 1)

    # 图像裁剪
    def random_crop_and_pad(image):
        width, height = image.size
        left = random.randint(0, width - 22)
        top = random.randint(0, height - 22)
        right = left + 22
        bottom = top + 22
        cropped_img = image.crop((left, top, right, bottom))

        new_img = Image.new("L", (28, 28), color=255)
        pad_left = (28 - 22) // 2
        pad_top = (28 - 22) // 2
        new_img.paste(cropped_img, (pad_left, pad_top))

        return np.where(np.array(new_img) > 0, 0, 1)
```

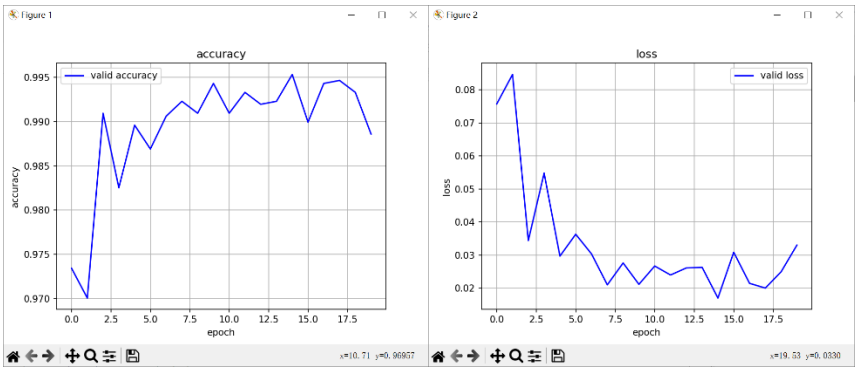
```python
class My_dataset(Dataset):
    # data augmentation
    methods = [Pre_processes.image_normalize, Pre_processes.image_translation, Pre_processes.random_crop_and_pad, Pre_processes.add_noise]

    def __init__(self, root_dir, labels, num_label=12):
        self.root_dir = root_dir
        self.labels = labels
        self.num_label = num_label

    def __len__(self):
        return 620 * len(My_dataset.methods) * self.num_label

    def __getitem__(self, index):
        method_class = index % len(My_dataset.methods)
        method = My_dataset.methods[method_class]

        folder_index = (index // len(My_dataset.methods)) // 620 + 1
        file_index = (index // len(My_dataset.methods)) % 620 + 1

        image_path = os.path.join(self.root_dir, str(folder_index), f"{file_index}.bmp")
        image = Image.open(image_path)
        size = image.size
        image = method(image)
        image = torch.tensor(image).reshape(1, size[0], size[1])

        label = self.labels[index // len(My_dataset.methods)]
        label_vector = torch.zeros(self.num_label)
        label_vector[label] = 1
        return image, label_vector
```

```python
# 定义CNN模型
class CNN(nn.Module):
    def __init__(self, num_label=12):
        super().__init__()
        self.net = nn.Sequential(
            # 两层卷积层，一层隐藏层
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Flatten(),

            nn.Linear(in_features=64 * 7 * 7, out_features=128),
            nn.ReLU(),
            nn.Linear(in_features=128, out_features=num_label)
        )

    def forward(self, x):
        x = self.net(x)
        return x
```

```python
# 定义CNN模型
class CNN(nn.Module): ···

# 使用了残差连接的CNN
class CNN_res(nn.Module): ···

# 使用了批归一化后的CNN
class CNN_normalization(nn.Module): ···

# 使用了dropout的卷积神经网络
class CNN_dropout(nn.Module): ···

# 使用了dropout和批归一化的神经网络
class CNN_dropout_batchnorm(nn.Module): ···
```

（三） 模型的训练效果和优化

下面是基础的卷积神经网络的训练效果，可以看到训练效果相对于 BP 神经网络已经相当优越。仅在 20 个 epoch 内就已经在验证集的预测中达到了极高的准确率。

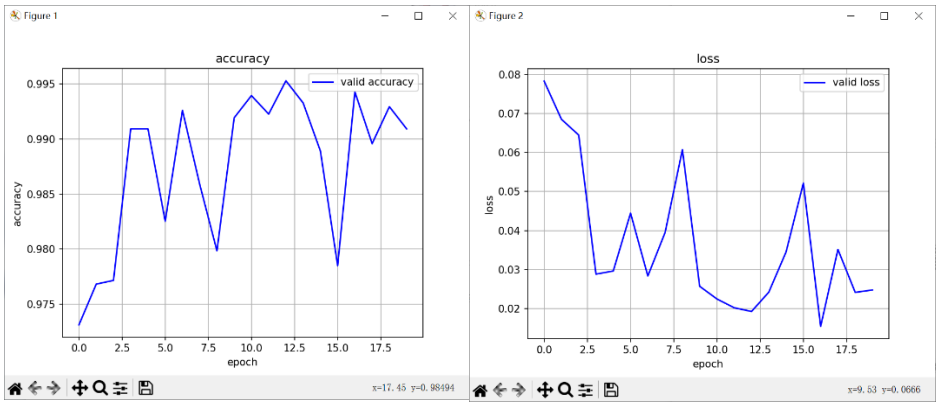


下面我们尝试采用残差连接对其进行优化。其为一种在神经网络中引入跳跃连接的技术，网络中的某一层的输出被直接添加到后续层的输入中，形成了一条跳跃连接路径。其能在一定程度上解决梯度消失和梯度爆炸问题，使得梯度可以更容易地在网络中传播，不会因为网络深度过大而导致梯度在传播过程中被过度放大或衰减，使得网络更容易学习和更新参数，加快优化的速度，并且网络可以更容易地学习到恒等映射，即将输入直接传递到输出，从而减少了特征的丢失和失真。这有助于保留输入的重要特征，降低模型对于噪声和冗余特征的敏感性，提高模型的鲁棒性和泛化性能。

残差连接的实现代码如下：

```
class CNN_res(nn.Module):
    def __init__(self, num_label=12):
        super().__init__()
        self.Conv = nn.Sequential(
            # 两层卷积层，一层隐藏层
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=32),
            nn.ReLU(),

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(num_features=64),
        )

        # 采用平均池化
        self.avgpool = nn.AvgPool2d(kernel_size=4)
        self.flatten = nn.Flatten()

        # 全连接层的序列模块
        self.DNN = nn.Sequential(

            nn.Linear(in_features=64 * 7 * 7, out_features=128),
            nn.ReLU(),
            nn.Linear(in_features=128, out_features=num_label)
        )

        # 残差块
        self.shortcut = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=64, kernel_size=1, stride=1, bias=False),
            nn.BatchNorm2d(num_features=64)
        )
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.Conv(x)
        # 建立残差连接
        out += self.shortcut(x)
        out = self.relu(out)
        out = self.avgpool(out)
        out = self.flatten(out)
        out = self.DNN(out)
        return out
```

下面是使用残差连接的训练效果，可以看到其性能相较于基础卷积神经网络更加突出。

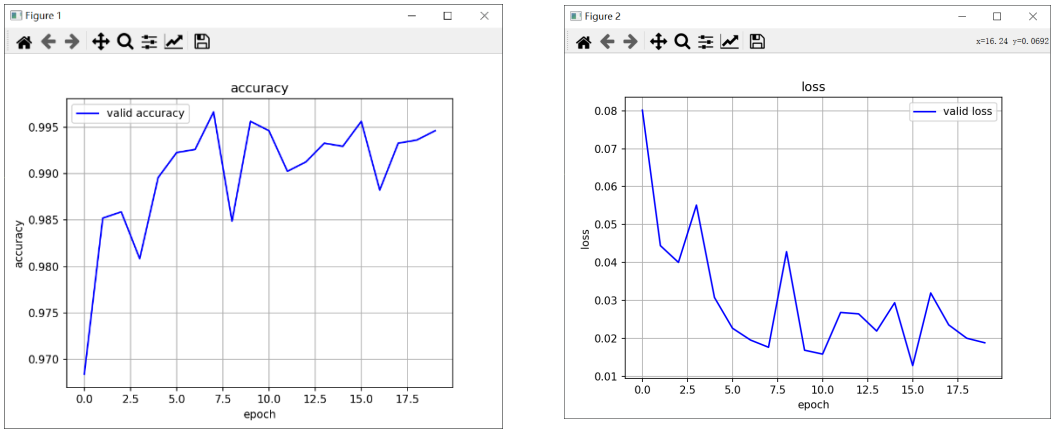


（四） 避免过拟合

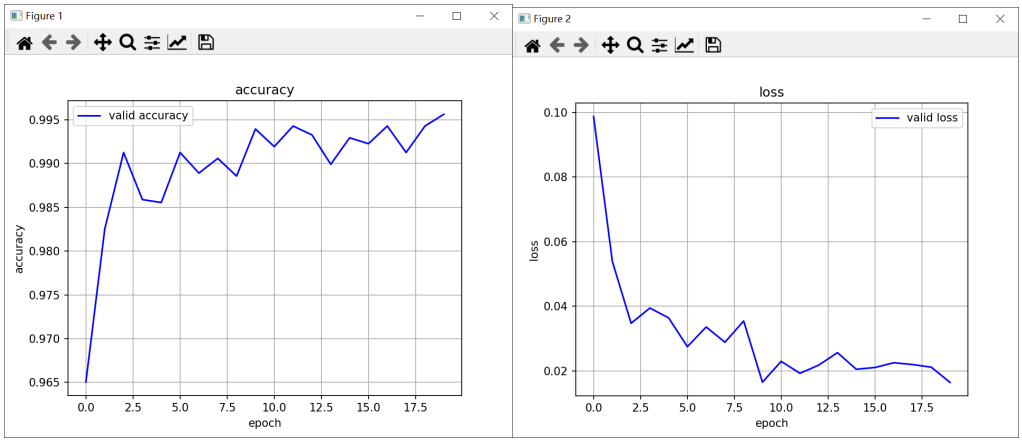
为了避免过拟合，下面我们分别采用批归一化和 dropout 对卷积神经网络进行优化。

批归一化在训练过程中对网络的中间层进行归一化处理。其通过将每个小批量输入数据进行归一化，使得网络中间层的输入数据的分布更加稳定，避免了不同层之间分布差异过大的问题，减少训练过程中的梯度爆炸和梯度消失问题，使得网络更容易优化和收敛。减少过拟合的风险。通过标准化每个中间层的输出，批归一化可以使得网络对于输入数据的小变化更加鲁棒。这有助于网络学习到更通用的特征表示，减少对于特定样本的过度拟合，提高模型的泛化性能。

使用了批归一化的卷积网络的训练效果如下：



使用了 dropout 的卷积网络的训练效果如下：

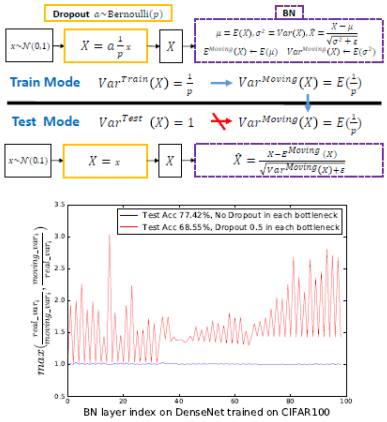


此外，在尝试同时在一个网络中使用批归一化和 dropout 时，我们发现，模型的训练效果欠佳，经过排查后发现代码本身并没有出现错误。而之后在查找资料中，我们在

Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift 这篇论文中发现了问题的答案。

简而言之，在网络中增加 dropout 层，会使该层输出的方差产生一定的变化。而批归一化本身会在训练的过程中计算并保存每个批次数据的均值和方差，并根据记录进行指数加权平均得到 moving variance ，在测试过程中对测试数据进行归一化（因为测试时往往不具有成批次的数据以计算有效方差）。但是在测试过程中并没有 dropout 操作，所以模型在训练过程中得到的 moving variance 并不适用。且训练过程越充分，这种基于方差的偏差就越大。因此会导致模型在测试集上的表现较差。

下面是论文中的一个图解说明：



（五） 手写 CNN 的尝试：

在手写卷积神经网络的过程中，最具挑战性的步骤实际上是反向传播通用理论的推演和通道数的处理。下面是我对卷积神经网络过程中的反向传播的规律总结：

记 F 为卷积核，由于对 X 做带步长的卷积操作有时需要考虑舍去部分元素，故在此记舍去后的 X 为 X_。

由

$$O = Conv(X, F)$$

$$d = \frac{\partial Loss}{\partial O}$$

$$X\_ = X[n{:}n, n{:}n]$$

基于 d 构建新矩阵 D，以 d 的大小为 2*2、步长为 2 的情况为例，构建过程如下

$$d=\begin{bmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix} \Rightarrow D=\begin{bmatrix} d_{11} & 0 & d_{12} \\ 0 & 0 & 0 \\ d_{21} & 0 & d_{22} \end{bmatrix}$$

通过在 d 的元素之间填充 0，使 D 的每行和每列的有效元素之间的间隔为（步长-1）。

通过简单归纳，可以得到梯度求解公式如下：

$$\frac{\partial L}{\partial F} = Conv(X\_, D)$$

$$\frac{\partial L}{\partial X} = full\_conv(F_r, D)$$

其中 Fr 是经过 180° 旋转后的卷积核。

而对于经过池化层的求导，只需要在池化时保存一个与池化前矩阵尺寸相同的记录矩阵 records，对于被保留的元素位置赋值为 1，其余位置赋值为 0。，这样，在求导时只需进行 hadamard 乘积即可。

$$\frac{\partial L}{\partial O'} = \frac{\partial L}{\partial O} \odot records$$

但出于时间问题，本实验自己手写的 CNN 网络未能调整至有效运行，或将在之后进行补充。