paddle笔记之数据处理

数据处理程序，一般涉及如下五个环节：

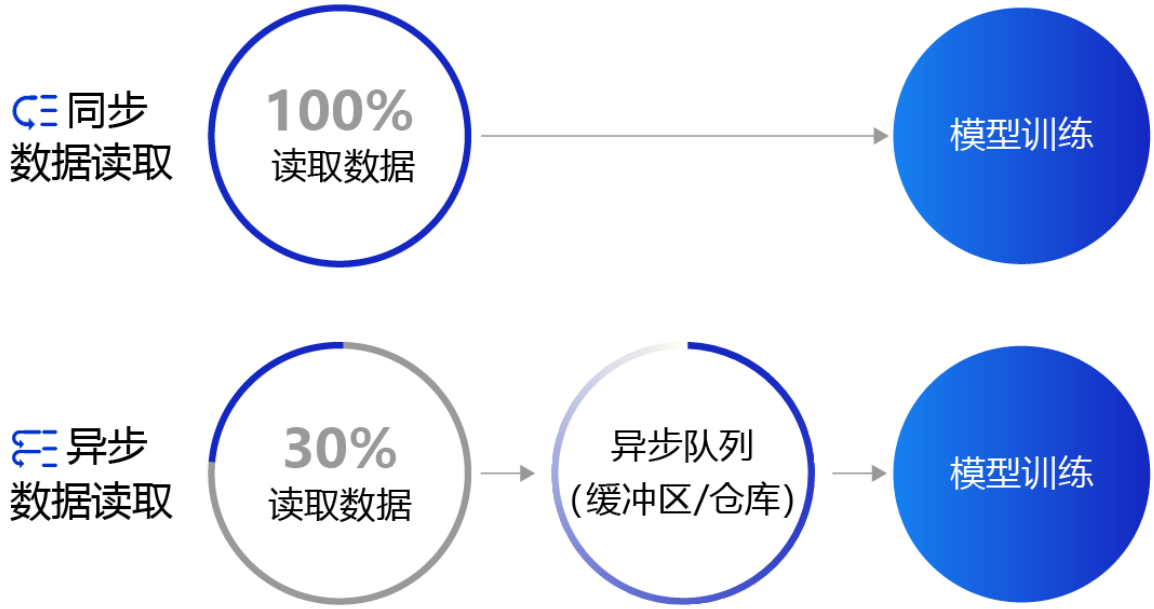
- 读入数据
- 划分数据集
- 生成批次数据
- 训练样本集乱序
- 校验数据有效性

也可以分为

定义数据集

迭代读取数据集

自动将数据集的样本进行分批、乱序等操作，方便训练时迭代读取，同时还支持多进程异步读取功能可加快数据读取速度。在飞桨框架中可使用 paddle.io.DataLoader 迭代读取数据集。对于样本量较大、数据读取较慢的场景，建议采用异步数据读取方式。异步读取数据时，数据读取和模型训练并行执行，从而加快了数据读取速度，牺牲一小部分内存换取数据读取效率的提升



读入数据并且划分数据集

每个数据集的用处

- **训练集**：用于确定模型参数。
- **验证集**：用于调节模型超参数（如多个网络结构、正则化权重的最优选择）。
- **测试集**：用于估计应用效果（没有在模型中应用过的数据，更贴近模型在真实场景应用的效果）。

```
# 声明数据集文件位置
datafile = './work/mnist.json.gz'
```

```
print('loading mnist dataset from {} ......'.format(datafile))
# 加载json数据文件
data = json.load(gzip.open(datafile))
print('mnist dataset load done')
# 读取到的数据区分训练集，验证集，测试集
train_set, val_set, eval_set = data

# 观察训练集数据
imgs, labels = train_set[0], train_set[1]
print("训练数据集数量: ", len(imgs))

# 观察验证集数量
imgs, labels = val_set[0], val_set[1]
print("验证数据集数量: ", len(imgs))

# 观察测试集数量
imgs, labels = val= eval_set[0], eval_set[1]
print("测试数据集数量: ", len(imgs))
print(len(imgs[0]))
```

导入json文件的例子

当我们需要将学术界研发的模型复用于工业项目时，应该如何选择呢？给读者一个小建议：当几个模型的准确率在测试集上差距不大时，尽量选择网络结构相对简单的模型。往往越精巧设计的模型和方法，越不容易在不同的数据集之间迁移。

训练样本乱序、生成批次数据

（1）训练样本乱序： 先将样本按顺序进行编号，建立ID集合index_list。然后将index_list乱序，最后按乱序后的顺序读取数据。

说明：

通过大量实验发现，模型对最后出现的数据印象更加深刻。训练数据导入后，越接近模型训练结束，最后几个批次数据对模型参数的影响越大。为了避免模型记忆影响训练效果，需要进行样本乱序操作。

（2）生成批次数据： 先设置合理的batch size，再将数据转变成符合模型输入要求的np.array格式返回。同时，在返回数据时将Python生成器设置为`yield`模式，以减少内存占用。

在执行如上两个操作之前，需要先将数据处理代码封装成load_data函数，方便后续调用。load_data有三种模型：`train`、`valid`、`eval`，分为对应返回的数据是训练集、验证集、测试集。

例子

```
imgs, labels = train_set[0], train_set[1]
print("训练数据集数量: ", len(imgs))
# 获得数据集长度
imgs_length = len(imgs)
# 定义数据集每个数据的序号，根据序号读取数据
index_list = list(range(imgs_length))
# 读入数据时用到的批次大小
BATCHSIZE = 100
```

```
# 随机打乱训练数据的索引序号
random.shuffle(index_list)


# 定义数据生成器，返回批次数据
def data_generator():
    imgs_list = []
    labels_list = []
    for i in index_list:
        # 将数据处理成希望的类型
        img = np.array(imgs[i]).astype('float32')
        label = np.array(labels[i]).astype('float32')
        imgs_list.append(img)
        labels_list.append(label)
        if len(imgs_list) == BATCHSIZE:
            # 获得一个batchsize的数据，并返回
            yield np.array(imgs_list), np.array(labels_list)
            # 清空数据读取列表
            imgs_list = []
            labels_list = []


    # 如果剩余数据的数目小于BATCHSIZE，
    # 则剩余数据一起构成一个大小为len(imgs_list)的mini-batch
    if len(imgs_list) > 0:
        yield np.array(imgs_list), np.array(labels_list)
    return data_generator
```

首先打乱，再按照batchsize合并

```
# 声明数据读取函数，从训练集中读取数据
train_loader = data_generator
# 以迭代的形式读取数据
for batch_id, data in enumerate(train_loader()):
    image_data, label_data = data
    if batch_id == 0:
        # 打印数据shape和类型
        print("打印第一个batch数据的维度:")
        print("图像维度: {}, 标签维度: {}".format(image_data.shape,
label_data.shape))
    break
```

校验数据有效性

一般有两种方式：

- 机器校验：加入一些校验和清理数据的操作。
- 人工校验：先打印数据输出结果，观察是否是设置的格式。再从训练的结果验证数据处理和读取的有效性。

机器校验

如下代码所示，如果数据集中的图片数量和标签数量不等，说明数据逻辑存在问题，可使用assert语句校验图像数量和标签数据是否一致。

```
imgs_length = len(imgs)

assert len(imgs) == len(labels), \
        "length of train_imgs({}) should be the same as
train_labels({})".format(len(imgs), len(labels))
```

人工校验

人工校验是指打印数据输出结果，观察是否是预期的格式。实现数据处理和加载函数后，我们可以调用它读取一次数据，观察数据的形状和类型是否与函数中设置的一致。

```
# 声明数据读取函数，从训练集中读取数据
train_loader = data_generator
# 以迭代的形式读取数据
for batch_id, data in enumerate(train_loader()):
    image_data, label_data = data
    if batch_id == 0:
        # 打印数据shape和类型
        print("打印第一个batch数据的维度，以及数据的类型:")
        print("图像维度: {}, 标签维度: {}, 图像数据类型: {}, 标签数据类型:
{}".format(image_data.shape, label_data.shape, type(image_data),
type(label_data)))
    break
打印第一个batch数据的维度，以及数据的类型:
图像维度: (100, 784), 标签维度: (100,), 图像数据类型: <class 'numpy.ndarray'>, 标签数据
类型: <class 'numpy.ndarray'>
```

封装数据读取与处理函数

```
def load_data(mode='train'):
    datafile = './work/mnist.json.gz'
    print('loading mnist dataset from {} ......'.format(datafile))
    # 加载json数据文件
    data = json.load(gzip.open(datafile))
    print('mnist dataset load done')

    # 读取到的数据区分训练集，验证集，测试集
    train_set, val_set, eval_set = data
    if mode=='train':
        # 获得训练数据集
        imgs, labels = train_set[0], train_set[1]
    elif mode=='valid':
        # 获得验证数据集
        imgs, labels = val_set[0], val_set[1]
    elif mode=='eval':
        # 获得测试数据集
```

```
        imgs, labels = eval_set[0], eval_set[1]
    else:
        raise Exception("mode can only be one of ['train', 'valid', 'eval']")
    print("训练数据集数量: ", len(imgs))

    # 校验数据
    imgs_length = len(imgs)

    assert len(imgs) == len(labels), \
          "length of train_imgs({}) should be the same as
train_labels({})".format(len(imgs), len(labels))

    # 获得数据集长度
    imgs_length = len(imgs)

    # 定义数据集每个数据的序号，根据序号读取数据
    index_list = list(range(imgs_length))
    # 读入数据时用到的批次大小
    BATCHSIZE = 100

    # 定义数据生成器
    def data_generator():
        if mode == 'train':
            # 训练模式下打乱数据
            random.shuffle(index_list)
        imgs_list = []
        labels_list = []
        for i in index_list:
            # 将数据处理成希望的类型
            img = np.array(imgs[i]).astype('float32')
            label = np.array(labels[i]).astype('float32')
            imgs_list.append(img)
            labels_list.append(label)
            if len(imgs_list) == BATCHSIZE:
                # 获得一个batchsize的数据，并返回
                yield np.array(imgs_list), np.array(labels_list)
                # 清空数据读取列表
                imgs_list = []
                labels_list = []

        # 如果剩余数据的数目小于BATCHSIZE，
        # 则剩余数据一起构成一个大小为len(imgs_list)的mini-batch
        if len(imgs_list) > 0:
            yield np.array(imgs_list), np.array(labels_list)
    return data_generator
```

下面定义一层神经网络，利用定义好的数据处理函数，完成神经网络的训练。

```
# 数据处理部分之后的代码,数据读取的部分调用Load_data函数
# 定义网络结构,同上一节所使用的网络结构
class MNIST(paddle.nn.Layer):
    def __init__(self):
        super(MNIST, self).__init__()
        # 定义一层全连接层，输出维度是1
        self.fc = paddle.nn.Linear(in_features=784, out_features=1)
```

```
    def forward(self, inputs):
        outputs = self.fc(inputs)
        return outputs
# 训练配置，并启动训练过程
def train(model):
    model = MNIST()
    model.train()
    #调用加载数据的函数
    train_loader = load_data('train')
    opt = paddle.optimizer.SGD(learning_rate=0.001,
parameters=model.parameters())
    EPOCH_NUM = 10
    for epoch_id in range(EPOCH_NUM):
        for batch_id, data in enumerate(train_loader()):
            #准备数据，变得更加简洁
            images, labels = data
            images = paddle.to_tensor(images)
            labels = paddle.to_tensor(labels)

            #前向计算的过程
            predits = model(images)

            #计算损失，取一个批次样本损失的平均值
            loss = F.square_error_cost(predits, labels)
            avg_loss = paddle.mean(loss)

            #每训练了200批次的数据，打印下当前Loss的情况
            if batch_id % 200 == 0:
                print("epoch: {}, batch: {}, loss is: {}".format(epoch_id,
batch_id, avg_loss.numpy()))

            #后向传播，更新参数的过程
            avg_loss.backward()
            opt.step()
            opt.clear_grad()

    # 保存模型
    paddle.save(model.state_dict(), './mnist.pdparams')
# 创建模型
model = MNIST()
# 启动训练过程
train(model)
```

迭代读取数据集

（1）同步数据读取：数据读取与模型训练串行。当模型需要数据时，才运行数据读取函数获得当前批次的数据。在读取数据期间，模型一直等待数据读取结束才进行训练，数据读取速度相对较慢。

（2）异步数据读取：数据读取和模型训练并行。读取到的数据不断的放入缓存区，无需等待模型训练就可以启动下一轮数据读取。当模型训练完一个批次后，不用等待数据读取过程，直接从缓存区获得下一批次数据进行训练，从而加快了数据读取速度。

（3）异步队列：数据读取和模型训练交互的仓库，二者均可以从仓库中读取数据，它的存在使得两者的工作节奏可以解耦。

通过飞桨`paddle.io.Dataset`和`paddle.io.DataLoader`两个API可以轻松创建异步数据读取的迭代器。

- batch_size：每批次读取样本数，示例中 batch_size=64 表示每批次读取 64 个样本。
- shuffle：样本乱序，示例中 shuffle=True 表示在取数据时打乱样本顺序，以减少过拟合发生的可能。
- drop_last：丢弃不完整的批次样本，示例中 drop_last=True 表示丢弃因数据集样本数不能被 batch_size 整除而产生的最后一个不完整的 batch 样本。
- num_workers：同步/异步读取数据，通过 num_workers 来设置加载数据的子进程个数，num_workers的值设为大于0时，即开启多进程方式异步加载数据，可提升数据读取速度。

数据增强/增广

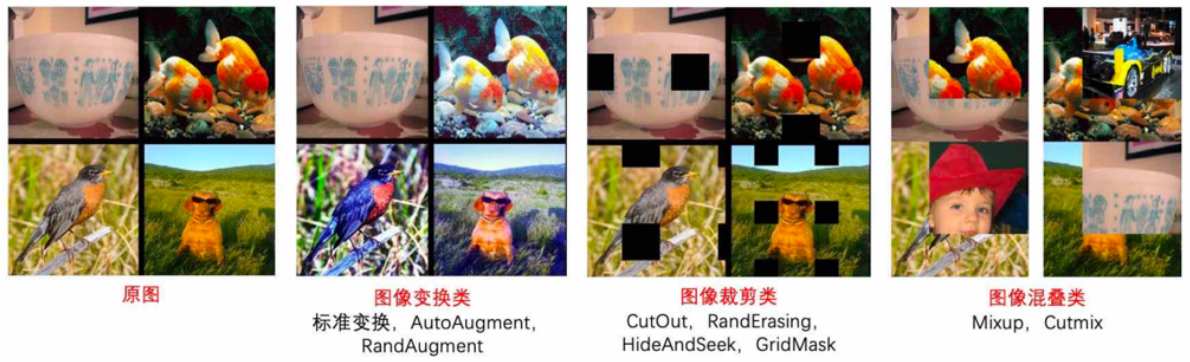
> 任何数学技巧都不能弥补信息的缺失
> -Cornelius Lanczos（匈牙利数学家、物理学家）

基础的数据增强/增广方法和实践

图5所示为一些基础的图像增强方法，如果我们发现数据集中的猫均是标准姿势，而真实场景中的猫时常有倾斜身姿的情况，那么在原始图片数据的基础上采用旋转的方法造一批数据加入到数据集会有助于提升模型效果。类似的，如果数据集中均是高清图片，而真实场景中经常有拍照模糊或曝光异常的情况，则采用降采样和调整饱和度的方式造一批数据，回有助于提升模型的效果。

高阶的数据增强/增广方法和实践

图6展示了一些高阶的图像增强方法，裁剪和拼接分别适合于“数据集中物体完整，但实际场景中物体存在遮挡”，以及“数据集中物体背景单一，而实际场景中物体的背景多变”的两种情况。具体数据增广实现，参见PaddleClas。



（1）文本识别的数据增强方法

图7展示了专门针对文本识别的数据增强方法TIA（Text Image augmentation），对应到"数据集中字体多是平面，而真实场景中的字体往往会在曲面上扭曲的情况，比如拿着相机对一张凸凹不平摆放的纸面拍摄的文字就会存在该效果"。

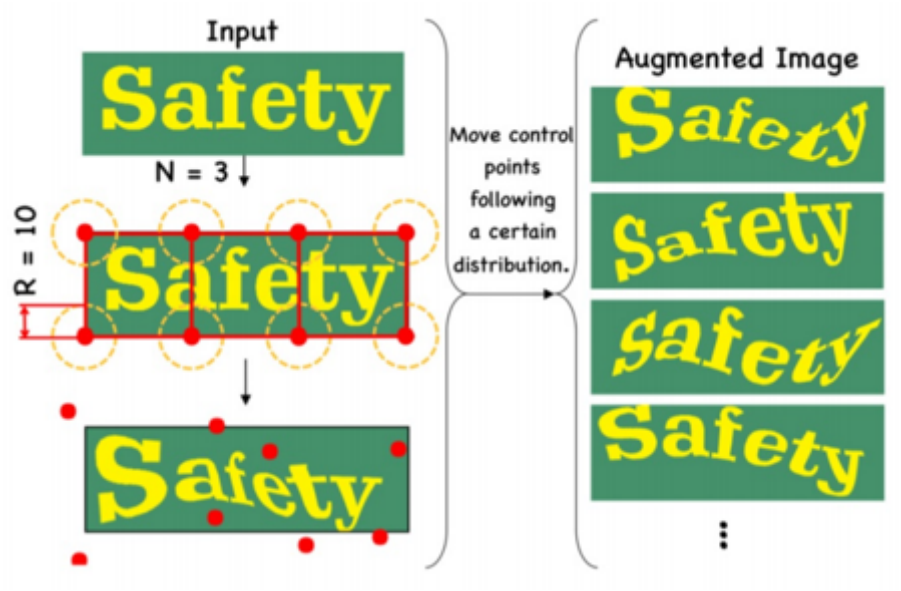


图7：文本识别数据增强方法

（2）一种新颖的数据增强技巧CopyPaste

现实中的文字检测要面临复杂多样的背景，比如店铺牌匾上的文字周围的背景可能是非常多样的。我们将部分文本区域剪辑出来，随机摆放到图片的各种位置来生成数据用于训练，会大大提高模型在复杂背景中，检测到文字内容的能力。

图8：一种新颖的数据增强技巧CopyPaste

根据实际应用场景来实现数据增强