神经网络实践的工作流各阶段总结与实现及加速工作流的思路

20 级软件工程辅修班 F201930075021 刘华林

包括神经网络的机器学习方法，在现代实践中已经形成了一种，包括数据预处理与特征工程，过拟合并制定基线，正则化（缩小规模，权重惩罚，dropout）在内的工程方法论。本文欲介绍现代神经网络实践的工作流，并基于此工作流做一个简单的示例。以下就从介绍现代神经网络实践的工作流开始：

（设计）第一步：定义整个问题，包括问题结构和数据结构。

首先是问题结构：已经有成熟方法的有二分类问题（二选一）、多分类问题（打标签）、数值回归问题（预测与插值）、向量回归问题（多物体标注）、翻译问题（语音翻译语种翻译），可以很容易参考已有的方法，更复杂的情形是这些问题在不同情境下的组合，以及开放环境下没有规划好的问题。

数据结构方面，需要面向业务做“特征工程”，根据项目业务所面向的领域，寻找整个问题的关键因素，不找无干紧要的因素，这是最发挥工程人员才智的地方之一，也可以选择用一些机器学习方法（无监督学习的）自动在大量数据中学习特征，但这不意味着工程人员无事可做，无监督学习所用的数据也是经过挑选的。

实际上，2016 年以来许多重要的业界成绩：包括像 AlphaGo(围棋决策)、AlphaFold(蛋白质结构预测)这样的成功模型，都依靠了程序员对特定领域的深度理解，找到了能发挥神经网络高复杂度优势的着力点，超越了传统算法。

这一步的设计要遵循三个假设：

1. **梯度性**，数据中存在的模式/概率分布能使随机出发寻找的输入到输出的映射收敛；
2. **充分性**，数据的量级足够模型收敛(每一轮优化后参数变化都大概率会更小)；
3. **一致性**，模型的测试数据/未来数据中存在的模式，与模型的训练数据/历史数据的模式基本一致；

一个良好的问题定义才能开始用深度学习解决问题，以上三个假设能保证神经网络至少是“**无咎**”的，也即当模型所表现的与预期表现存在差异时，问题至少不会出现在该环节。

反之，如果问题并不满足这些假设，就需要重新思考问题结构是否定义不良好、是否需要重构问题结构（任务类型）以及重新组织数据结构。验证问题是否满足这些假设时，若是低复杂度问题，可以采用其他的运行较快、方便验证的机器学习方法，若是高复杂度问题，可以用 MiniBatch 的思路先使用一部分数据，观察模型运行效果。

（设计）第二步：定义评价方法和评价指标。

先考虑最简单的二分类情形：每一次分类有四种结果：

预测\实际	实际为 1	实际为 0
预测为 1	真阳性	假阳性
预测为 0	假阴性	真阴性

商业损失一般源于预测错误的决策，若预测错误对不同的类别（0 或 1）所造成的损失是一样的，那么准确率的设置就较为简单，但考虑到复杂的商业逻辑，为了回避潜在的信用风险等，分类系统可能被要求有较低的假阳性率或假阴性率。**实际业务中，模型可能需要多种精准度指标，并分配以不同权重。**

除了出于业务需要的指标考虑，还有软件性能方面的和系统性能方面的指标，这些指标也是模型的基本要求，只是在各种领域问题中不是第一位的。软件性能方面包括泛用性、响应速度，故障率是比较重要的指标，系统性能包括稳健性、可维护性、故障处理等指标也需要顾及。

（实现）第三步：数据收集和预处理、表示

模型根据可变性一般分数据集为三部分：

1. 最可变的：数据集；
2. 经常可变的：参数；
3. 较少变动的：超参数。

从这点出发设计三类数据集：验证集、训练集和测试集。验证数据集对应超参数，用于优化超参数；训练数据集对应参数，用于优化神经网络的参数组；测试集对应数据集，用于验证神经网络的性能，数据集也存在一定的性能指标：数据规模、多样性、噪声大小等。

实践上不对这些数据预先做区分，统一获取一个较大的数据集并切分该数据集得到验证集、训练集和测试集。常用的切分方法包括：

1. 简单区分，实践中常用 0.6(测试集)+0.2(验证集)+0.2(测试集)的比例；
2. K 折交叉验证，把总体数据分为 K 个包，逐一选择包作为测试集（同时也是验证集）其余包（K-1）个作为训练集，最后综合 K 个训练结果；
3. 带混洗的 K 折交叉验证，每次 K 折交叉的切分时再对数据做混洗，会带来更大的运算量，但一般可以对性能提升几个百分点。

切分数据集时还应注意切分的子集应可以代表原数据集的数据分布：减少重复复制、保留多样性。

在数据完整性上还应考虑缺失值问题和单位问题，成批的缺失值会使模型预测失误，故可以选择删除样本、历史期平滑、以相似样本的变量补齐等方法解决；单位不统一使模型的预测容易波动，一般对样本集合的各个变量进行归一化，生成均值为 0，标准差为 1 的无量纲数据。

在预处理环节的一个很重要的问题是对象的表示方法，我认为它的重要性不亚于模型的参数训练阶段，甚至在现代的各种复杂问题开放问题涌现的情境，其重要性远超参数训练。

古希腊某位哲学家曾有言：“问问题的人自己已经有了答案”，实际上研究者们常常是带着答案找问题的思路：先确定一些符合现实条件的假设，再根据假设设置一些变量和方程式，这些变量和方程式在假设所描绘的世界中能对应所要研究的目标现象，随后操作这些变量和方程式就足以研究现实世界的目标现象。

在这个过程中，错误分为两类：第一类是假设下完成的推导逻辑上不自洽，存在推导错误，使模型表现与预期现象不一致，对应到软件工程的实践中，就是软件自身的故障和逻辑链条错误，这是逻辑性错误；第二类是假设本身与现实的偏差，假设所依赖的环境条件并不一定为现实所满足，假设所依赖的环境可能只在一部分时期或一部分人群中成立，因此带来的意外现象和异常，是开放性错误。

对于神经网络模型实践而言，训练模型和参数调优，乃至一部分的问题定义，它们的错误属于这个环节：逻辑性错误；而**表示方面的异常则属于开放性错误**。

逻辑性错误基本借助逻辑体系内部的重新调整就可以解决，逐步检查每一个设计就可以理清问题所在，例如检查代码和反复训练模型；但开放性错误一般要求对逻辑体系的推倒重建，无法用简单的步骤就完成对问题的修正而需要重新理解问题。

因此，如何确定对对象的表示一直是神经网络研究的大课题，表示学习（Representation Learning）是一个热门领域，很多成功的神经网络架构都把重点放在表示学习上，经典的 CNN（卷积神经网络）架构分为卷积层部分（表示）和全连接层（分类），卷积层采用卷积核（识

别图片的边缘特征）+池化（粗粒化得到整体特征）的方式，对采用像素级表示(224 像素 x 224 像素)的图片进行再表示，逐层提取重要特征，以方便分类器处理。在 NLP（自然语言处理）领域，从最早的统计语言模型（统计词语组合的出现频率）、词袋模型（对 N 元连续的词语组合的出现频率进行最大似然估计）、到后来的独热表示和词向量(word2vec)表示，在字典中计算词的相似性，不断地对词语这一对象赋予更抽象的定义和更通用的表示。

表示一定程度上与具体的任务相独立，这就带来了迁移学习和预训练的可能性。迁移学习认为不同任务的问题背景是相通的：可以共用或者稍加改动的借用其他任务中已经训练好的表示层，例如识别大量动物的神经网络的表示层可以借用给识别猫和狗的神经网络的表示层。预训练同样基于该种思路，区别在于，大型语言模型使得预训练方法流行于 NLP 领域，而 NLP 领域的流行的大模型预训练实际上其对预训练的理解与 CV（计算机视觉）领域存在一定区别。

（实现）第四步：模型基线制定与大到过拟合

这一步不需要长时间的训练和繁琐地调优，仅是为了确定一些基本的超参数。

基线从评价指标中选择，出于简单性，一般仅选择精准度指标，且设置的基线值并不高，例如对于二分类问题，精准度基线为 0.5 较合适，如果模型连 0.5 的精准度都无法做到，尤其需要考虑数据的质量和规模问题：数据是不是太少或者太烂了？

这里要将前几步的设计转化为一个暂时的架构/超参数设置，有三个关键项：

1. 输出层选取，二分类问题一般用 sigmoid，多类别分类问题一般用 softmax，回归问题采用输出标量的线性层，向量回归问题采用多个标量线性层；
2. 损失函数选择，回归问题一般用 MSE 均方误差，分类问题一般用 CrossEntropy 交叉熵；
3. 优化算法选择，很多任务中 Adam 和 RMSprop 用得比较多，基本都从 SGD（随机梯度下降）思想出发，选择合适的学习率、学习率增长策略。

现在模型的架构基本确定好了，开始做大模型：

1. 加入更多隐层；
2. 已有的层扩宽为更多参数；
3. 训练更多轮。

这么做是**为了保证模型的容量足够大——至少能表示充分多的目标**，这一阶段的目的就是为了把模型做大到过拟合，判断过拟合现象出现的标志是：训练一定的轮数以后，训练集的准确率仍在提高，但验证集的准确率停止并下降。

（实现）第五步：正则化（正则化、参数宽度深度调整、学习率改变、Dropout 等）

保证了模型的容量以后就要考虑模型的泛化性，要做“减法”。这一阶段的正则化可以从三个角度出发：数据调整、参数调整。

数据调整的观点认为过拟合的原因来自于数据的代表性不够、质量不够好、数据不够多，为此可以准备更多数据，或者复制使模型预测错误的样本加大其分量，或者采用**数据增强**的技术，对目标做一些变换得到新的样本：例如对图片做放缩平移旋转变换，对语句做倒序重组，都可以获得更丰富的数据集。

参数调整的观点认为过拟合的原因就是参数的分布或参数的调整策略不当，使神经网络函数的损失（Loss Function）落入了局部极小值（一种几何解释）。解决方法包括调整参数的分布（长度宽度可变性）和参数的调整策略（权重正则化与学习策略）：

1. 参数总体的长宽改变：拉短（减少层数）和拉窄（减少宽度，即减少每层输入的特征数）；
2. 参数的可变性改变：典型的如 Dropout 技术，将某一个隐层的输出随机地屏蔽或归零一个比例（一般是 20%~50%的数据）；

3. 权值正则化：典型的如 L2 正则化，为了防止某些权重过大引起其他权重（特征）失效，在优化过程中加入权重的平方和作为 BP 过程修正项中的惩罚项；
4. 学习策略：学习率每多少轮调整一次，每次改变多少，高度积极的学习策略不易收敛，高度消极的学习策略训练时间过长，合适的策略可以加速并防止过拟合/欠拟合。

以上是一个常见的神经网络模型从启动到投入运营的工作流，接下来展示 Pytorch 下的一个 ResNet18 用于猫狗分类的实践，作为对这个工作流的一个体现，由于这个 DEMO 较小，很多细节并不会体现而只实现一个基本流程。

（1） 问题定义：猫狗二分类

（2） 评价方法：分类精准度

（3） 数据收集和预处理：以下是 Pytorch 代码：

```
import os
import glob
import numpy as np
#数据集整理
path = r"F:\BaiduNetdiskDownload\kaggle\dogs-and-cats"
files = glob.glob(path+"\*.jpg")
print(f'Total number of images {len(files)}')
num_of_images = len(files)
shuffle = np.random.permutation(num_of_images)
#用于保存验证图片集的 validation 目录
os.mkdir(os.path.join(path,'valid/'))
os.mkdir(os.path.join(path,'train/'))
for t in ['train/','valid/']:
    for folder in ['dog/','cat/']:
        os.mkdir(os.path.join(path,t,folder))
#将图片的一小部分子集复制到 validation 文件夹
for i in shuffle[:5000]:
    if "dog" in files[i][45:-6]:
        new_path = path + r"\valid\dog" + files[i][44:]
    else:
        new_path = path + r"\valid\cat" + files[i][44:]
    os.rename(files[i],new_path)
#将图片一小部分复制到 trainng 文件夹
for i in shuffle[5000:]:
    if "dog" in files[i][45:-6]:
        new_path = path + r"\train\dog" + files[i][44:]
    else:
        new_path = path + r"\train\cat" + files[i][44:]
    os.rename(files[i],new_path)
```

运行结果：

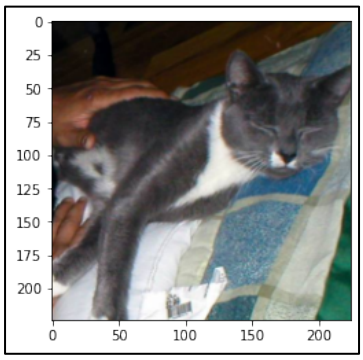
```
Total number of images 25000
```

上述训练集包括训练集和验证集（测试集）两部分，训练集 20000 个样本，验证集 5000 个样本，数据集只有猫狗两类样本，且均匀分布。

（4）模型基线制定与过拟合：迁移 Reset18 的表示层，容量足够大，无需测定基线

（5）正则化：出于简单考虑，仅仅设置了指数学习率策略，剩余部分采用 Resnet18 的正则化方法即可，Pytorch 代码如下：

```
# import numpy
import matplotlib.pyplot as plt
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torchvision.models as models
import time
import gc
path = r"F:\BaiduNetdiskDownload\kaggle\dogs-and-cats"
print(">>>加载训练数据集和验证数据集")
simple_transforms = transforms.Compose([transforms.Resize([224,
224]),transforms.ToTensor(),transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])])
train_path = path + r"\train"
train = datasets.ImageFolder(train_path,simple_transforms)
valid_path = path + r"\valid"
valid = datasets.ImageFolder(valid_path,simple_transforms)
#将张量转换为图片，复原图片
def imageshow(inp):
    inp = inp.numpy().transpose((1,2,0))
    mean = numpy.array([0.485,0.456,0.406])
    std = numpy.array([0.229,0.224,0.225])
    inp = std*inp + mean
    inp = numpy.clip(inp,0,1)
    plt.imshow(inp)
#imageshow(train[50][0])
```




```
#加载数据
train_data_gen = torch.utils.data.DataLoader(train,batch_size=16,num_workers=3)
valid_data_gen = torch.utils.data.DataLoader(valid,batch_size=16,num_workers=3)
```

数据读取采用批处理模式，每批 16 个样本，并采用变换器预处理了图像的张量数据。

```
dataloaders = [train_data_gen,valid_data_gen]
```

```
print("加载完成")
print(">>>加载模型设置")
#加载模型，调整输入特征数：分类数
model_ft = models.resnet18(pretrained=True)
num_features = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_features,2)
```

迁移 Resnet18 架构，只是将分类器层的分类类别数改为 2：仅作猫狗分类。

```
model_ft.half()
model_ft = model_ft.cuda()
```

Half()方法可使模型作半精度计算，加速训练过程，Cuda()使计算在 GPU 执行。

```
#损失函数
learning_rate = 0.001
criterion = nn.CrossEntropyLoss()
#优化器
optimizer_ft = optim.SGD(model_ft.parameters(),lr=learning_rate,momentum=0.9)
exp_lr_scheduler = optim.lr_scheduler.StepLR(optimizer_ft,step_size=7,gamma=0.1)
```

指数学习率调整策略，每 7 步使学习率乘一个 gamma 因子，即每 7 步使学习率缩小 10 倍，这是一种学习率的正则化策略。

```
print("加载完成")
#总训练器
def train_model(model,criterion,optimizer,scheduler,dataloaders,num_epochs=10):
    since = time.time()
    best_model_weights = model.state_dict()
    best_acc = 0.0
    datasets_sizes = [len(dataloaders[0].dataset.imgs),len(dataloaders[1].dataset.imgs)]
    for epoch in range(num_epochs):
        gc.collect()
        print("Epoch {}/{}".format(epoch+1,num_epochs))
        print('-'*10)
        #每轮都有训练和验证阶段
        for phase in ['train','valid']:
            if phase == 'train':
                model.train(True)
            else:
                model.train(False)
            #开始训练/验证
            running_loss = 0.0
            running_corrects = 0
            #在数据上迭代
            phase_int = 0 if phase == train else 1
            for data in dataloaders[phase_int]:
                torch.cuda.empty_cache()
                #获取输入
                inputs, labels = data
```

```
                #封装为 Torch 变量(有 CUDA 可以放在 GPU 上)
                inputs, labels = Variable(inputs.cuda().half()), Variable(labels.cuda().half())
                #梯度参数清零
                optimizer.zero_grad()
                #前向传播
                print("输入尺寸：",inputs.size())
                outputs = model(inputs)
                print("输出尺寸：",outputs.size())
                _, preds = torch.max(outputs.data,1)
                print("标签尺寸：",labels.size())
                loss = criterion(outputs,labels.long())
                #只在训练阶段反向优化
                if phase == 'train':
                    loss.backward()
                    optimizer.step()
                    scheduler.step()
                #统计损失和正确率
                running_loss += loss.data.item()
                running_corrects += torch.sum(preds == labels.data)
            #每一轮的统计
            epoch_loss = running_loss / datasets_sizes[phase_int]
            epoch_acc = running_corrects / datasets_sizes[phase_int]
            print("{} Loss: {:.4f} Acc: {:.4f}".format(phase,epoch_loss,epoch_acc))
            #深度复制模型
            if phase == 'valid' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_weights = model.state_dict()
        print()
    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed//60,time_elapsed%60))
    print('Best val Acc: {:4f}'.format(best_acc))
    #加载最优权重
    model.load_state_dict(best_model_weights)
    return model
train_model(model_ft,criterion,optimizer_ft,exp_lr_scheduler,dataloaders)
print("Success!")
```

运行结果:

```
>>>加载训练数据集和验证数据集
加载完成
>>>加载模型设置
加载完成
轮数： 1/10
----------
train 损失: 0.1653 Acc: 0.4990
```

valid 损失: 0.1646 Acc: 0.5010

轮数： 2/10
----
train 损失: 0.1649 Acc: 0.5010
valid 损失: 0.1646 Acc: 0.5010

轮数： 3/10
----
train 损失: 0.1649 Acc: 0.5010
valid 损失: 0.1646 Acc: 0.5010

轮数： 4/10
----
train 损失: 0.1649 Acc: 0.5010
valid 损失: 0.1646 Acc: 0.5010

轮数： 5/10
----
train 损失: 0.1649 Acc: 0.5010
valid 损失: 0.1646 Acc: 0.5010

轮数： 6/10
----
train 损失: 0.1649 Acc: 0.5010
valid 损失: 0.1646 Acc: 0.5010

轮数： 7/10
----
train 损失: 0.1649 Acc: 0.5010
valid 损失: 0.1646 Acc: 0.5010

轮数： 8/10
----
train 损失: 0.1649 Acc: 0.5010
valid 损失: 0.1646 Acc: 0.5010

轮数： 9/10
----
train 损失: 0.1649 Acc: 0.5010
valid 损失: 0.1646 Acc: 0.5010

轮数： 10/10

```
train 损失: 0.1649 Acc: 0.5010
valid 损失: 0.1646 Acc: 0.5010

训练完成，用时： 10m 5s
最佳的验证精确度: 0.501000
训练完毕!
```

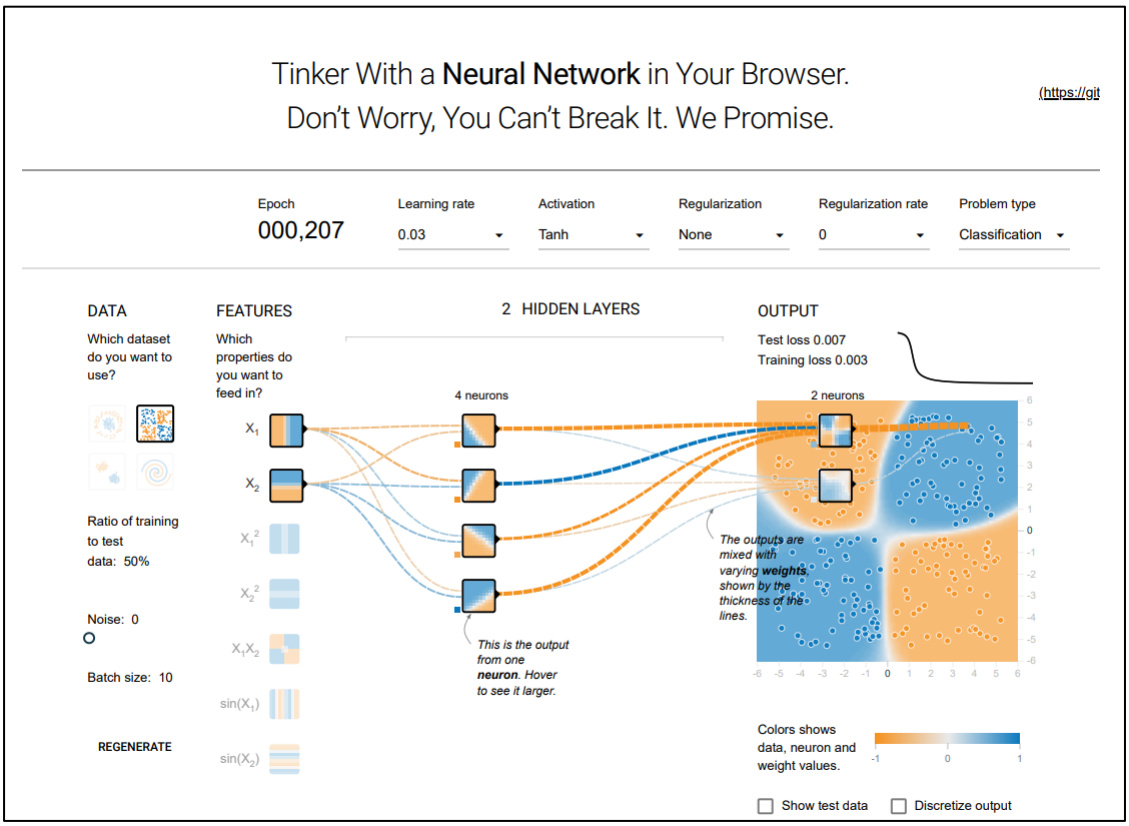
以上是我实现的一个 ResNet18 退化版（仅用于猫狗二分类），可以观察到，训练与验证损失在第二轮后就未曾改变，精准度一直维持在基准水平，显然这个神经网络模型在学习率设置方面存在一定问题，但限于调试时间，不再对这个 Resnet 迁移的改进作进一步讨论。

神经网络存在着一定的工作流模式，那么能否加速神经网络的工作流呢？前文已陈述我的观点：耗时最长的是训练过程，但实际上最困难的是数据预处理过程（表示学习）；虽然神经网络训练的水平决定了其性能上限，但快速进行的表示学习可以加速开发周期：对于重复的问题可以进行迁移学习和预训练，对于非重复的开放问题，快速的表示学习可以帮助开发者迅速试错确定有效方案，而不需要辛苦调试等待结果。

因此，加速神经网络工作流的关键在于加速对表示学习部分的验证，能可视化地、可解释地、实时地察看不同的表示学习方案带来的神经网络学习能力的变化，为此，展示一个基于 TensorFlow 和 JavaScript 的简单神经网络训练网页版。

网址：http://playground.tensorflow.org/

网站外观：



该网址可以自由选择任务类型（分类/回归），数据集（多种数据分布），输入特征，输出特征，隐层数量和宽度，训练过程的学习率，激活函数，正则化项和正则化率。在确定参数后会以极快的速度完成运算并反馈以用户：测试损失，训练损失，权重变化的动画效果。

先使用极小但极多样的数据集，为欲搭建的神经网络架构的各个模块赋予一定自由度，并设计一个实时可视化的人机交互界面，使设计人员可以快速验证其想法。

面向验证(表示)而非面向性能(训练)，这是加速神经网络模型工作流的一条有效路径。