图像分类任务中的tricks总结

Jack Stark 机器学习小王子 2019-03-03

计算机视觉主要问题有图像分类、目标检测和图像分割等。针对图像分类任务，提升准确率的方法路线有两条，一个是模型的修改，另一个是各种数据处理和训练的技巧(tricks)。图像分类中的各种技巧对于目标检测、图像分割等任务也有很好的作用，因此值得好好总结。本文在精读论文的基础上，总结了图像分类任务的各种tricks如下：

- Warmup
- Linear scaling learning rate
- Label-smoothing
- Random image cropping and patching
- Knowledge Distillation
- Cutout
- Random erasing
- Cosine learning rate decay
- Mixup training
- AdaBoud
- AutoAugment
- 其他经典的tricks

Warmup

学习率是神经网络训练中最重要的超参数之一，针对学习率的技巧有很多。Warm up是在ResNet论文[1]中提到的一种学习率预热的方法。由于刚开始训练时模型的权重(weights)是随机初始化的(全部置为0是一个坑，原因见[2])，此时选择一个较大的学习率，可能会带来模型的不稳定。学习率预热就是在刚开始训练的时候先使用一个较小的学习率，训练一些epoches或iterations，等模型稳定时再修改为预先设置的学习率进行训练。论文[1]中使用一个110层的ResNet在cifar10上训练时，先用0.01的学习率训练直到训练误差低于80%(大概训练了400个iterations)，然后使用0.1的学习率进行训练。

上述的方法是constant warmup，18年Facebook又针对上面的warmup进行了改进[3]，因为从一个很小的学习率一下变为比较大的学习率可能会导致训练误差突然增大。论文[3]提出了gradual warmup来解决这个问题，即从最开始的小学习率开始，每个iteration增大一点，直到最初设置的比较大的学习率。

Gradual warmup代码如下：

```
from torch.optim.lr_scheduler import _LRScheduler

class GradualWarmupScheduler(_LRScheduler):
    """
    Args:

        optimizer (Optimizer): Wrapped optimizer.
        multiplier: target learning rate = base lr * multiplier
        total_epoch: target learning rate is reached at total_epoch, gradually
        after_scheduler: after target_epoch, use this scheduler(eg. ReduceLROnPlateau)
    """

    def __init__(self, optimizer, multiplier, total_epoch, after_scheduler=None):
        self.multiplier = multiplier
        if self.multiplier <= 1.:
            raise ValueError('multiplier should be greater than 1.')
        self.total_epoch = total_epoch
        self.after_scheduler = after_scheduler
        self.finished = False
        super().__init__(optimizer)

    def get_lr(self):
        if self.last_epoch > self.total_epoch:
            if self.after_scheduler:
                if not self.finished:
                    self.after_scheduler.base_lrs = [base_lr * self.multiplier for base_
                    self.finished = True
                return self.after_scheduler.get_lr()
            return [base_lr * self.multiplier for base_lr in self.base_lrs]

        return [base_lr * ((self.multiplier - 1.) * self.last_epoch / self.total_epoch +

    def step(self, epoch=None):
        if self.finished and self.after_scheduler:
            return self.after_scheduler.step(epoch)
        else:
            return super(GradualWarmupScheduler, self).step(epoch)
```

Linear scaling learning rate

Linear scaling learning rate是在论文[3]中针对比较大的batch size而提出的一种方法。

在凸优化问题中，随着批量的增加，收敛速度会降低，神经网络也有类似的实证结果。随着batch size的增大，处理相同数据量的速度会越来越快，但是达到相同精度所需要的epoch数量越来越多。也就是说，使用相同的epoch时，大batch size训练的模型与小batch size训练的模型相比，验证准确率会减小。

上面提到的gradual warmup是解决此问题的方法之一。另外，linear scaling learning rate也是一种有效的方法。在mini-batch SGD训练时，梯度下降的值是随机的，因为每一个batch的数据是随机选择的。增大batch size不会改变梯度的期望，但是会降低它的方差。也就是说，大batch size会降低梯度中的噪声，所以我们可以增大学习率来加快收敛。

具体做法很简单，比如ResNet原论文[1]中，batch size为256时选择的学习率是0.1，当我们把batch size变为一个较大的数b时，学习率应该变为 0.1 × b/256。

Label-smoothing

在分类问题中，我们的最后一层一般是全连接层，然后对应标签的one-hot编码，即把对应类别的值编码为1，其他为0。这种编码方式和通过降低交叉熵损失来调整参数的方式结合起来，会有一些问题。这种方式会鼓励模型对不同类别的输出分数差异非常大，或者说，模型过分相信它的判断。但是，对于一个由多人标注的数据集，不同人标注的准则可能不同，每个人的标注也可能会有一些错误。模型对标签的过分相信会导致过拟合。

标签平滑(Label-smoothing regularization,LSR)是应对该问题的有效方法之一，它的具体思想是降低我们对于标签的信任，例如我们可以将损失的目标值从1稍微降到0.9，或者将从0稍微升到0.1。标签平滑最早在inception-v2[4]中被提出，它将真实的概率改造为：

$$q_i = \begin{cases} 1-\varepsilon & \text{if } i = y, \\ \varepsilon/(K-1) & \text{otherwise,} \end{cases}$$

其中，ε是一个小的常数，K是类别的数目，y是图片的真正的标签，i代表第i个类别，q_i是图片为第i类的概率。

总的来说，LSR是一种通过在标签y中加入噪声，实现对模型约束，降低模型过拟合程度的一种正则化方法。

LSR代码如下：

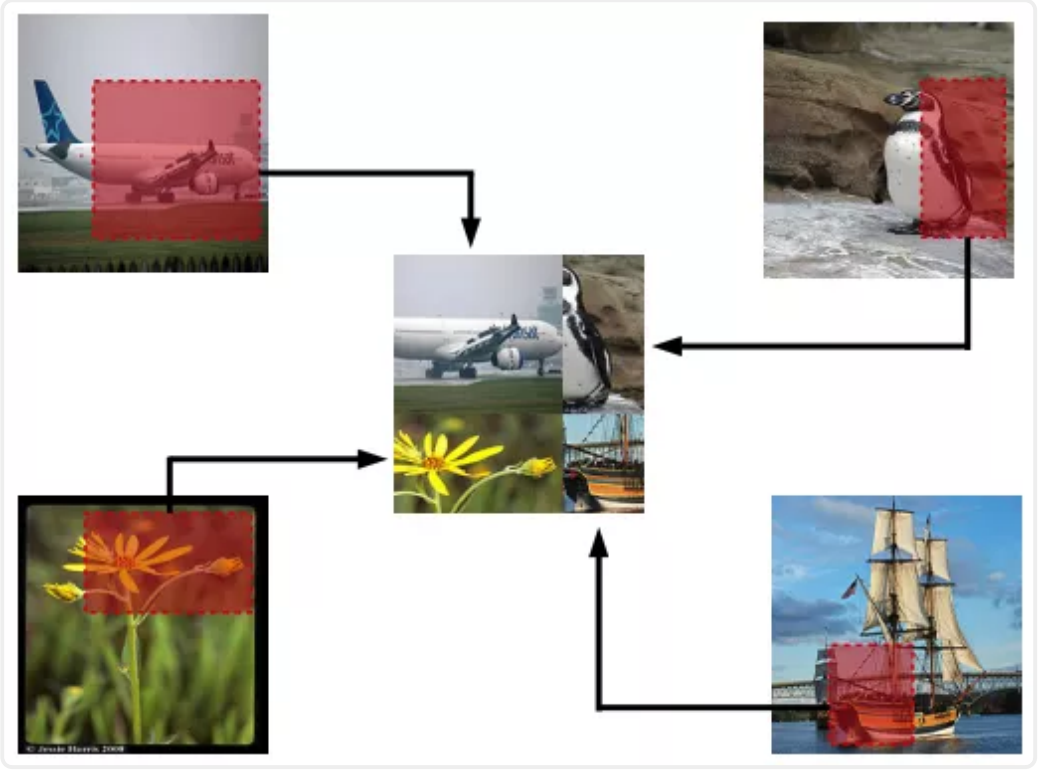
```
import torch
import torch.nn as nn

class LSR(nn.Module):

    def __init__(self, e=0.1, reduction='mean'):
        super().__init__()

        self.log_softmax = nn.LogSoftmax(dim=1)
```

```
        self.e = e
        self.reduction = reduction

    def _one_hot(self, labels, classes, value=1):
        """
            Convert labels to one hot vectors

        Args:
            labels: torch tensor in format [label1, label2, label3, ...]
            classes: int, number of classes
            value: label value in one hot vector, default to 1

        Returns:
            return one hot format labels in shape [batchsize, classes]
        """

        one_hot = torch.zeros(labels.size(0), classes)

        #labels and value_added  size must match
        labels = labels.view(labels.size(0), -1)
        value_added = torch.Tensor(labels.size(0), 1).fill_(value)

        value_added = value_added.to(labels.device)
        one_hot = one_hot.to(labels.device)

        one_hot.scatter_add_(1, labels, value_added)

        return one_hot

    def _smooth_label(self, target, length, smooth_factor):
        """convert targets to one-hot format, and smooth
        them.
        Args:
            target: target in form with [label1, label2, label_batchsize]
            length: length of one-hot format(number of classes)
            smooth_factor: smooth factor for label smooth

        Returns:
            smoothed labels in one hot format
        """
        one_hot = self._one_hot(target, length, value=1 - smooth_factor)
        one_hot += smooth_factor / length

        return one_hot.to(target.device)
```

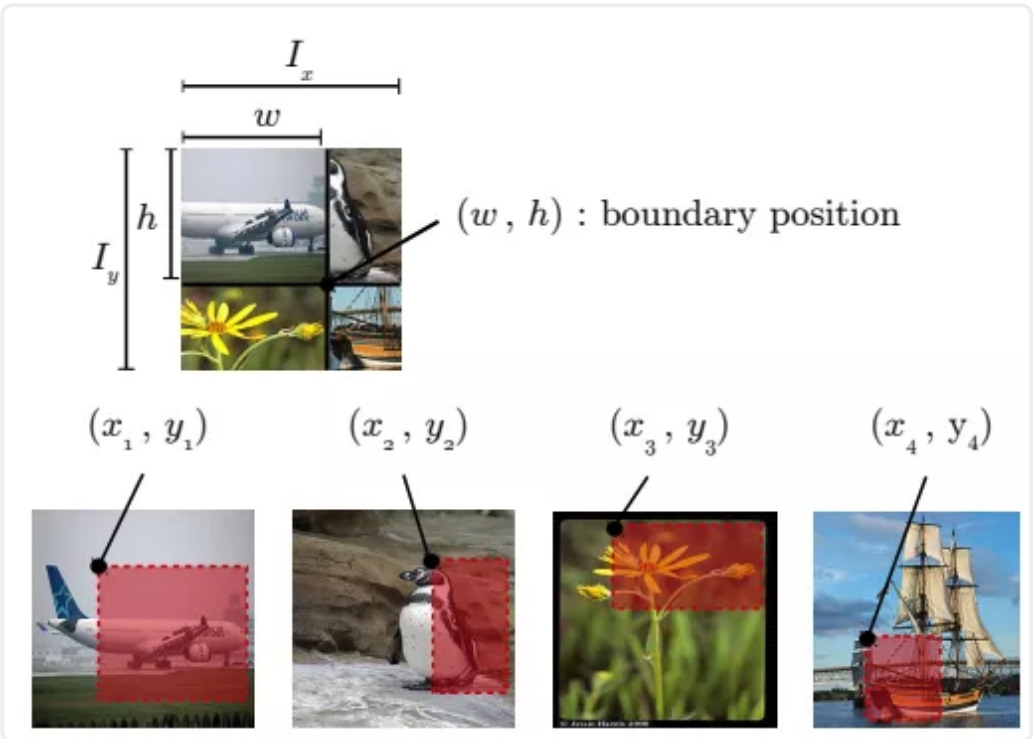
Random image cropping and patching

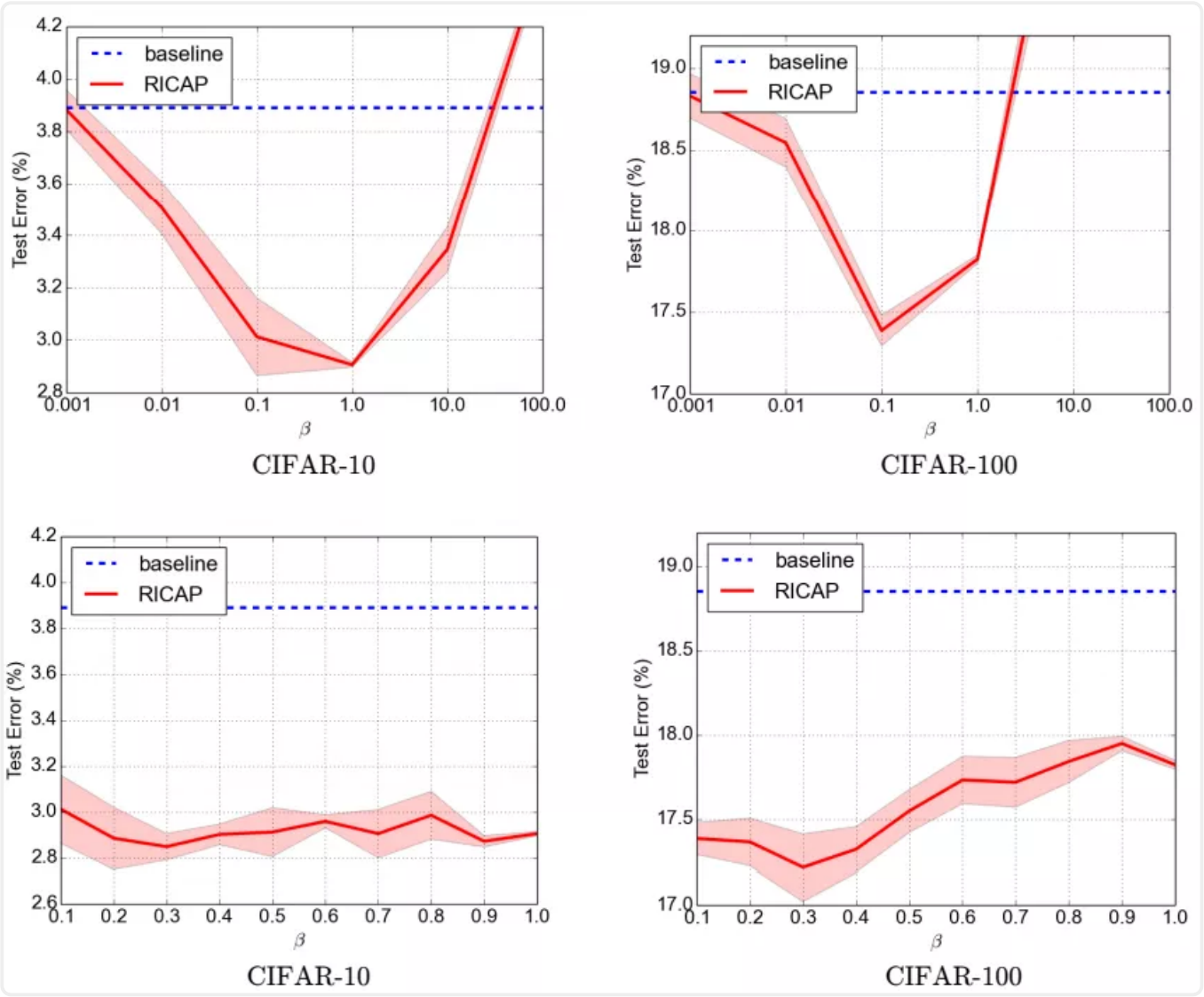
Random image cropping and patching (RICAP)[7]方法随机裁剪四个图片的中部分，然后把它们拼接为一个图片，同时混合这四个图片的标签。

RICAP在caifar10上达到了2.19%的错误率。



如下图所示，Ix, Iy是原始图片的宽和高。w和h称为boundary position，它决定了四个裁剪得到的小图片的尺寸。w和h从beta分布Beta(β, β)中随机生成，β也是RICAP的超参数。最终拼接的图片尺寸和原图片尺寸保持一致。





RICAP的代码如下：

```
beta = 0.3 # hyperparameter
for (images, targets) in train_loader:

    # get the image size
    I_x, I_y = images.size()[2:]

    # draw a boundry position (w, h)
    w = int(np.round(I_x * np.random.beta(beta, beta)))
    h = int(np.round(I_y * np.random.beta(beta, beta)))
    w_ = [w, I_x - w, w, I_x - w]
    h_ = [h, h, I_y - h, I_y - h]

    # select and crop four images
    cropped_images = {}
    c_ = {}
    W_ = {}
    for k in range(4):
        index = torch.randperm(images.size(0))
        x_k = np.random.randint(0, I_x - w_[k] + 1)
        y_k = np.random.randint(0, I_y - h_[k] + 1)
        cropped_images[k] = images[index][:, :, x_k:x_k + w_[k], y_k:y_k + h_[k]]
```

```
        c_[k] = target[index].cuda()
        W_[k] = w_[k] * h_[k] / (I_x * I_y)

    # patch cropped images
    patched_images = torch.cat(
                (torch.cat((cropped_images[0], cropped_images[1]), 2),
                torch.cat((cropped_images[2], cropped_images[3]), 2)),
            3)
    #patched_images = patched_images.cuda()

    # get output
    output = model(patched_images)

    # calculate loss and accuracy
    loss = sum([W_[k] * criterion(output, c_[k]) for k in range(4)])
    acc = sum([W_[k] * accuracy(output, c_[k])[0] for k in range(4)])
```

Knowledge Distillation

提高几乎所有机器学习算法性能的一种非常简单的方法是在相同的数据上训练许多不同的模型，然后对它们的预测进行平均。但是使用所有的模型集成进行预测是比较麻烦的，并且可能计算量太大而无法部署到大量用户。Knowledge Distillation(知识蒸馏)[8]方法就是应对这种问题的有效方法之一。

在知识蒸馏方法中，我们使用一个教师模型来帮助当前的模型（学生模型）训练。教师模型是一个较高准确率的预训练模型，因此学生模型可以在保持模型复杂度不变的情况下提升准确率。比如，可以使用ResNet-152作为教师模型来帮助学生模型ResNet-50训练。在训练过程中，我们会加一个蒸馏损失来惩罚学生模型和教师模型的输出之间的差异。

给定输入，假定p是真正的概率分布，z和r分别是学生模型和教师模型最后一个全连接层的输出。之前我们会用交叉熵损失l(p,softmax(z))来度量p和z之间的差异，这里的蒸馏损失同样用交叉熵。所以，使用知识蒸馏方法总的损失函数是

$$\ell(p, \mathrm{softmax}(z)) + T^2 \ell(\mathrm{softmax}(r/T), \mathrm{softmax}(z/T))$$

上式中，第一项还是原来的损失函数，第二项是添加的用来惩罚学生模型和教师模型输出差异的蒸馏损失。其中，T是一个温度超参数，用来使softmax的输出更加平滑的。实验证明，用ResNet-152作为教师模型来训练ResNet-50，可以提高后者的准确率。

Cutout

Cutout[9]是一种新的正则化方法。原理是在训练时随机把图片的一部分减掉，这样能提高模型的鲁棒性。它的来源是计算机视觉任务中经常遇到的物体遮挡问题。通过cutout生成一些类似被遮挡的物体，不仅可以让模型在遇到遮挡问题时表现更好，还能让模型在做决定时更多地考虑环境(context)。

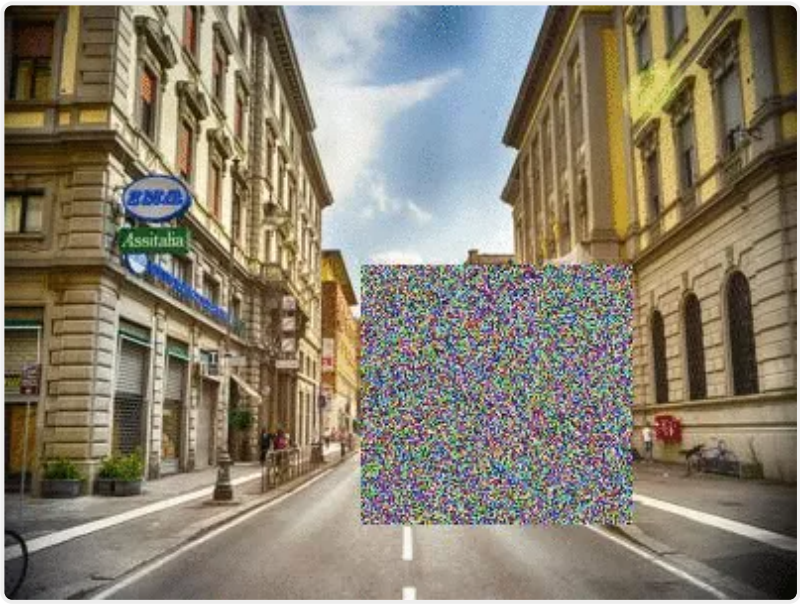
代码如下：

```
import torch
import numpy as np

class Cutout(object):
    """Randomly mask out one or more patches from an image.
    Args:
        n_holes (int): Number of patches to cut out of each image.
        length (int): The length (in pixels) of each square patch.
    """
    def __init__(self, n_holes, length):
        self.n_holes = n_holes
        self.length = length

    def __call__(self, img):
        """
        Args:
            img (Tensor): Tensor image of size (C, H, W).
        Returns:
            Tensor: Image with n_holes of dimension length x length cut out of it.
        """
        h = img.size(1)
        w = img.size(2)

        mask = np.ones((h, w), np.float32)

        for n in range(self.n_holes):
            y = np.random.randint(h)
            x = np.random.randint(w)

            y1 = np.clip(y - self.length // 2, 0, h)
            y2 = np.clip(y + self.length // 2, 0, h)
            x1 = np.clip(x - self.length // 2, 0, w)
            x2 = np.clip(x + self.length // 2, 0, w)

            mask[y1: y2, x1: x2] = 0.

        mask = torch.from_numpy(mask)
        mask = mask.expand_as(img)
        img = img * mask

        return img
```

效果如下图，每个图片的一小部分被cutout了。

Random erasing

Random erasing[6]其实和cutout非常类似，也是一种模拟物体遮挡情况的数据增强方法。区别在于，cutout是把图片中随机抽中的矩形区域的像素值置为0，相当于裁剪掉，random erasing是用随机数或者数据集中像素的平均值替换原来的像素值。而且，cutout每次裁剪掉的区域大小是固定的，Random erasing替换掉的区域大小是随机的。



Random erasing代码如下：

```
from __future__ import absolute_import
from torchvision.transforms import *
from PIL import Image
import random
import math
import numpy as np
import torch

class RandomErasing(object):
    '''
    probability: The probability that the operation will be performed.
    sl: min erasing area
    sh: max erasing area
    r1: min aspect ratio
    mean: erasing value
    '''
    def __init__(self, probability = 0.5, sl = 0.02, sh = 0.4, r1 = 0.3, mean=[0.4914, 0
        self.probability = probability
        self.mean = mean
        self.sl = sl
        self.sh = sh
        self.r1 = r1

    def __call__(self, img):

        if random.uniform(0, 1) > self.probability:
            return img

        for attempt in range(100):
            area = img.size()[1] * img.size()[2]

            target_area = random.uniform(self.sl, self.sh) * area
            aspect_ratio = random.uniform(self.r1, 1/self.r1)

            h = int(round(math.sqrt(target_area * aspect_ratio)))
            w = int(round(math.sqrt(target_area / aspect_ratio)))
```

```
            if w < img.size()[2] and h < img.size()[1]:
                x1 = random.randint(0, img.size()[1] - h)
                y1 = random.randint(0, img.size()[2] - w)
                if img.size()[0] == 3:
                    img[0, x1:x1+h, y1:y1+w] = self.mean[0]
                    img[1, x1:x1+h, y1:y1+w] = self.mean[1]
                    img[2, x1:x1+h, y1:y1+w] = self.mean[2]
                else:
                    img[0, x1:x1+h, y1:y1+w] = self.mean[0]
                return img

        return img
```

Cosine learning rate decay

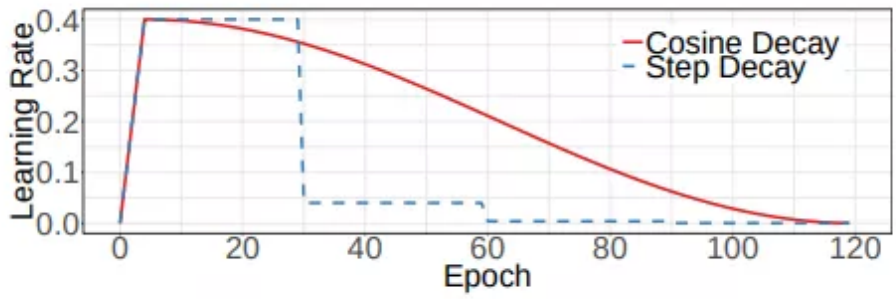
在warmup之后的训练过程中，学习率不断衰减是一个提高精度的好方法。其中有step decay和cosine decay等，前者是随着epoch增大学习率不断减去一个小的数，后者是让学习率随着训练过程曲线下降。

对于cosine decay，假设总共有T个batch（不考虑warmup阶段），在第t个batch时，学习率η_t为：

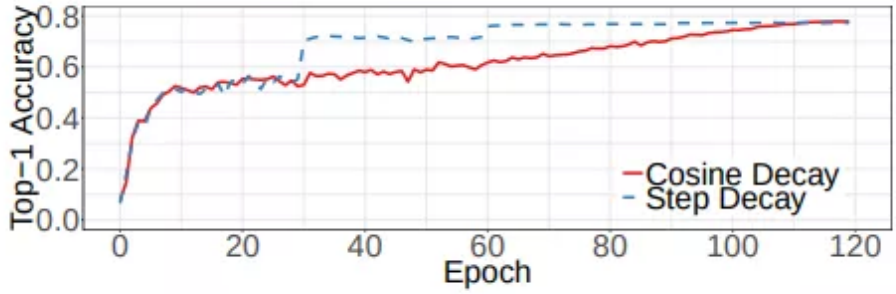
$$\eta_t = \frac{1}{2}\left(1+\cos\left(\frac{t\pi}{T}\right)\right)\eta$$

这里，η代表初始设置的学习率。这种学习率递减的方式称之为cosine decay。

下面是带有warmup的学习率衰减的可视化图[4]。其中，图(a)是学习率随epoch增大而下降的图，可以看出cosine decay比step decay更加平滑一点。图(b)是准确率随epoch的变化图，两者最终的准确率没有太大差别，不过cosine decay的学习过程更加平滑。



(a) Learning Rate Schedule



(b) Validation Accuracy

在pytorch的torch.optim.lr_scheduler中有更多的学习率衰减的方法，至于哪个效果好，可能对于不同问题答案是不一样的。对于step decay，使用方法如下：

```
# Assuming optimizer uses lr = 0.05 for all groups
# lr = 0.05     if epoch < 30
# lr = 0.005    if 30 <= epoch < 60
# lr = 0.0005   if 60 <= epoch < 90

from torch.optim.lr_scheduler import StepLR
scheduler = StepLR(optimizer, step_size=30, gamma=0.1)
for epoch in range(100):
    scheduler.step()
    train(...)
    validate(...)
```

Mixup training

Mixup[10]是一种新的数据增强的方法。Mixup training，就是每次取出2张图片，然后将它们线性组合，得到新的图片，以此来作为新的训练样本，进行网络的训练，如下公式，其中x代表图像数据，y代表标签，则得到的新的xhat, yhat。

$$\begin{aligned} \hat{x} &= \lambda x_i + (1-\lambda)x_j, \\ \hat{y} &= \lambda y_i + (1-\lambda)y_j, \end{aligned}$$

其中，λ是从Beta(α, α)随机采样的数，在[0,1]之间。在训练过程中，仅使用(xhat, yhat)。

Mixup方法主要增强了训练样本之间的线性表达，增强网络的泛化能力，不过mixup方法需要较长的时间才能收敛得比较好。

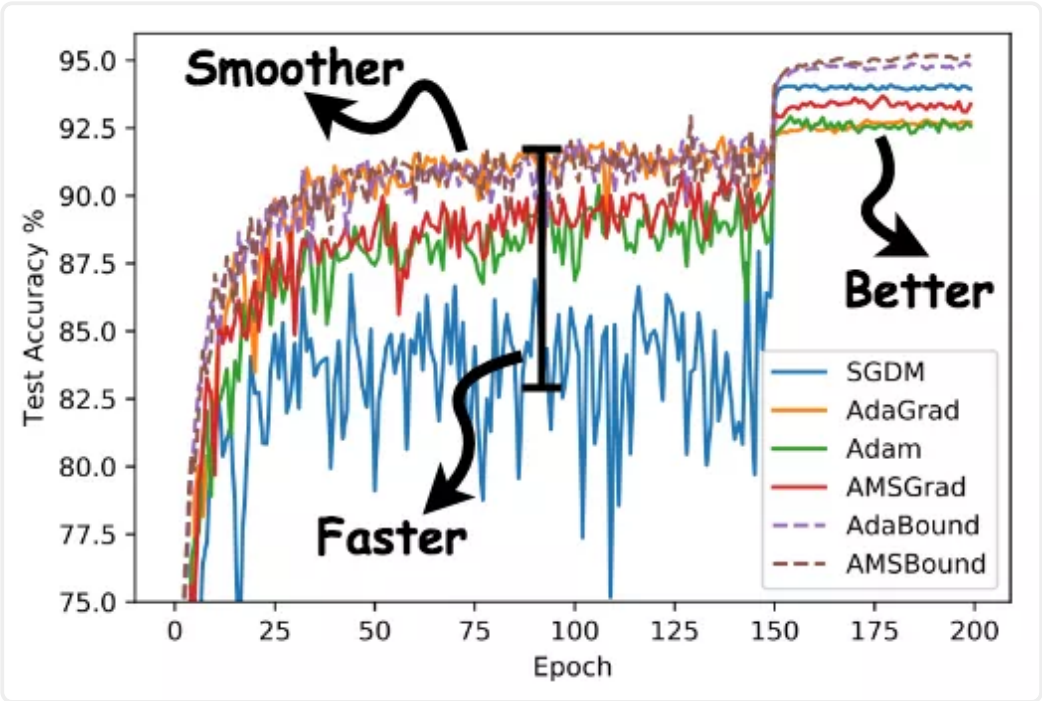
Mixup代码如下：

```
for (images, labels) in train_loader:

    l = np.random.beta(mixup_alpha, mixup_alpha)

    index = torch.randperm(images.size(0))
    images_a, images_b = images, images[index]
    labels_a, labels_b = labels, labels[index]

    mixed_images = l * images_a + (1 - l) * images_b

    outputs = model(mixed_images)
    loss = l * criterion(outputs, labels_a) + (1 - l) * criterion(outputs, labels_b)
    acc = l * accuracy(outputs, labels_a)[0] + (1 - l) * accuracy(outputs, labels_b)[0]
```

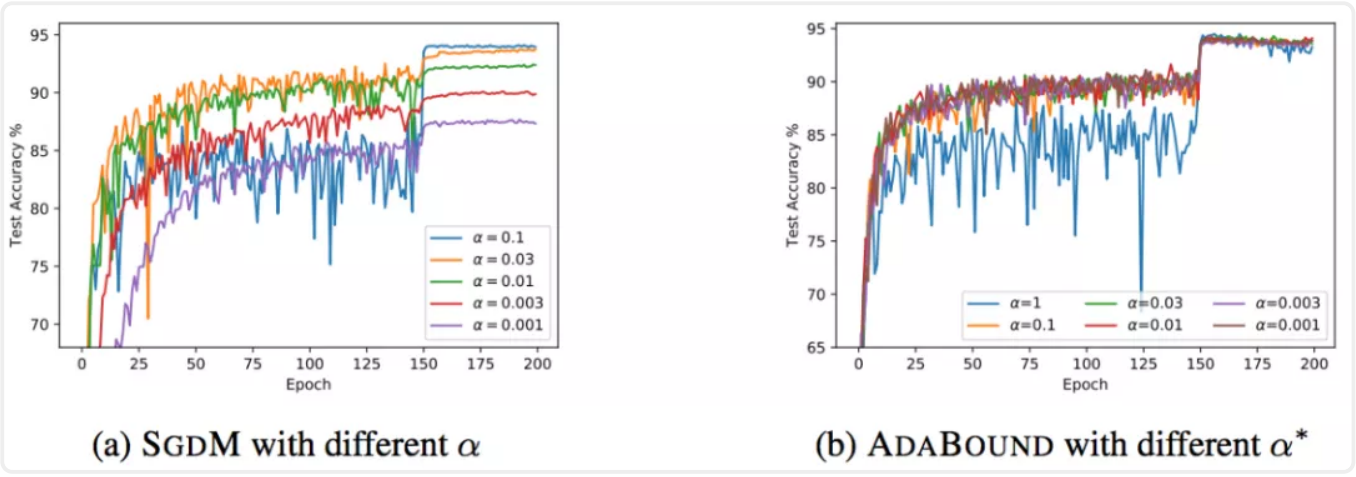
AdaBound

AdaBound是最近一篇论文[5]中提到的，按照作者的说法，AdaBound会让你的训练过程像adam一样快，并且像SGD一样好。

如下图所示，使用AdaBound会收敛速度更快，过程更平滑，结果更好。



另外，这种方法相对于SGD对超参数的变化不是那么敏感，也就是说鲁棒性更好。但是，针对不同的问题还是需要调节超参数的，只是所用的时间可能变少了。



(a) SGDM with different α (b) ADABOUND with different α^*

当然，AdaBound还没有经过普遍的检验，也有可能只是对于某些问题效果好。

使用方法如下：

安装AdaBound

```
pip install adabound
```

使用AdaBound(和其他PyTorch optimizers用法一致)

```
optimizer = adabound.AdaBound(model.parameters(), lr=1e-3, final_lr=0.1)
```

AutoAugment

数据增强在图像分类问题上有很重要的作用，但是增强的方法有很多，并非一股脑地用上所有的方法就是最好的。那么，如何选择最佳的数据增强方法呢？ AutoAugment[11]就是一种搜索适合当前问题的数据增强方法的方法。该方法创建一个数据增强策略的搜索空间，利用搜索算法选取适合特定数据集的数据增强策略。此外，从一个数据集中学到的策略能够很好地迁移到其它相似的数据集上。

AutoAugment在cifar10上的表现如下表，达到了98.52%的准确率。

Model	Baseline	Cutout [25]	AutoAugment
Wide-ResNet-28-10 [57]	3.87	3.08	2.68
Shake-Shake (26 2x32d) [59]	3.55	3.02	2.47
Shake-Shake (26 2x96d) [59]	2.86	2.56	1.99
Shake-Shake (26 2x112d) [59]	2.82	2.57	1.89
AmoebaNet-B (6,128) [21]	2.98	2.13	1.75
PyramidNet+ShakeDrop [60]	2.67	2.31	**1.48**

其他经典的tricks

常用的正则化方法为

- Dropout
- L1/L2正则
- Batch Normalization
- Early stopping
- Random cropping
- Mirroring
- Rotation
- Color shifting
- PCA color augmentation
- ...

其他

- Xavier init[12]
- ...

参考

[1] Deep Residual Learning for Image Recognition(https://arxiv.org/pdf/1512.03385.pdf)

[2] http://cs231n.github.io/neural-networks-2/

[3] Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour(https://arxiv.org/pdf/1706.02677v2.pdf)

[4] Rethinking the Inception Architecture for Computer Vision(https://arxiv.org/pdf/1512.00567v3.pdf)

[4]Bag of Tricks for Image Classification with Convolutional Neural Networks(https://arxiv.org/pdf/1812.01187.pdf)

[5] Adaptive Gradient Methods with Dynamic Bound of Learning Rate(https://www.luolc.com/publications/adabound/)

[6] Random erasing(https://arxiv.org/pdf/1708.04896v2.pdf)

[7] RICAP(https://arxiv.org/pdf/1811.09030.pdf)

[8] Distilling the Knowledge in a Neural Network(https://arxiv.org/pdf/1503.02531.pdf)

[9] Improved Regularization of Convolutional Neural Networks with Cutout(https://arxiv.org/pdf/1708.04552.pdf)

[10] Mixup: BEYOND EMPIRICAL RISK MINIMIZATION(https://arxiv.org/pdf/1710.09412.pdf)

[11] AutoAugment: Learning Augmentation Policies from Data(https://arxiv.org/pdf/1805.09501.pdf)

[12] Understanding the difficulty of training deep feedforward neural networks(http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf)