出神入化：特斯拉AI主管、李飞飞高徒Karpathy的33个神经网络「炼丹」技巧

选自Andrej Karpathy blog

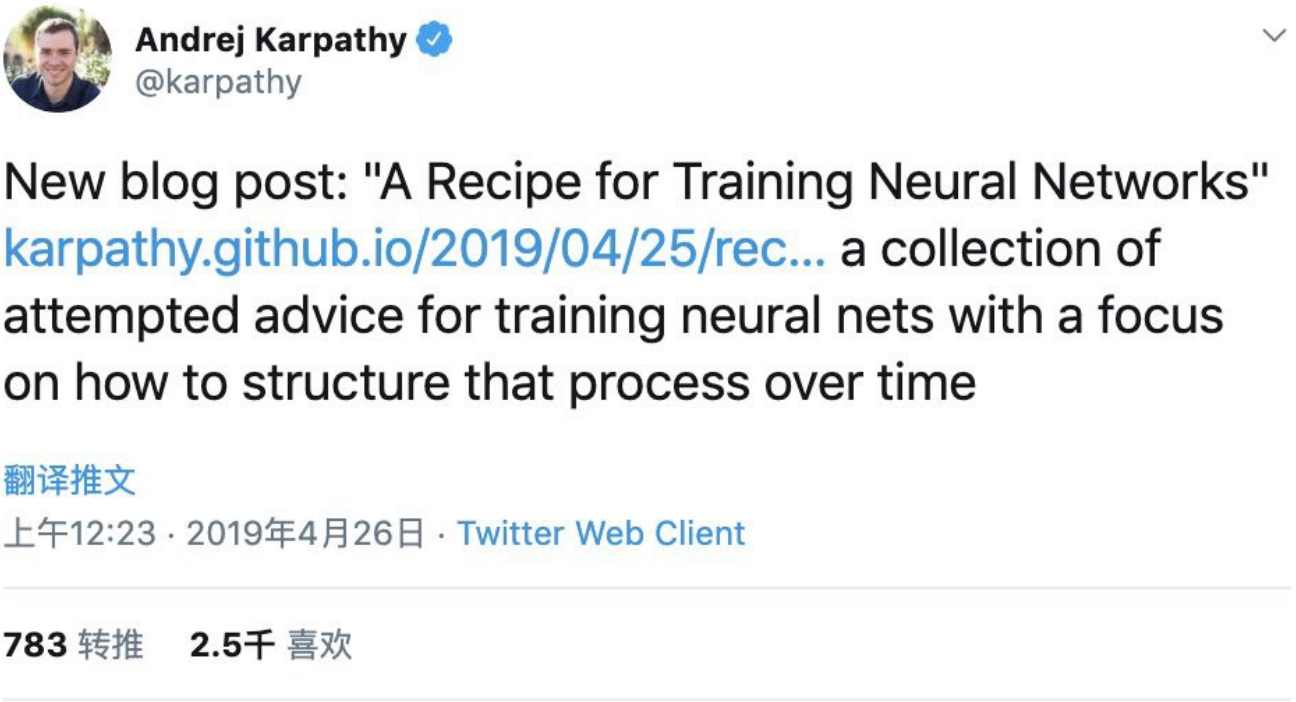
作者：Andrej Karpathy

机器之心编译

特斯拉人工智能部门主管 Andrej Karpathy 发布新博客，介绍神经网络训练的技巧。

Andrej Karpathy 是深度学习计算机视觉领域、生成式模型与强化学习领域的研究员。博士期间师从李飞飞。在读博期间，两次在谷歌实习，研究在 Youtube 视频上的大规模特征学习，2015 年在 DeepMind 实习，研究深度强化学习。毕业后，Karpathy 成为 OpenAI 的研究科学家，后于 2017 年 6 月加入特斯拉担任人工智能与自动驾驶视觉总监。

今日他发布的这篇博客能为深度学习研究者们提供极为明晰的洞见，在 Twitter 上也引发了极大的关注。



1. 谁说神经网络训练简单了？

很多人认为开始训练神经网络是很容易的，大量库和框架号称可以用 30 行代码段解决你的数据问题，这就给大家留下了（错误的）印象：训练神经网络这件事是非常简单的，不同模块即插即用就能搭个深度模型。

简单的建模过程通常如下所示：

```
>>> your_data = # plug your awesome dataset here>>> model = SuperCrossValidator(SuperDuper.fit, your_data, ResNet50, SGDOptimizer)# conquer world here
```

这些库和示例令我们想起了熟悉标准软件及模块，标准软件中通常可以获取简洁的 API 和抽象。

例如 Request 库的使用展示如下：

```
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))>>> r.status_code200
```

酷！这些库和框架的开发者背负起理解用户 Query 字符串、url、GET/POST 请求、HTTP 连接等的大量需求，将复杂度隐藏在几行代码后面。这就是我们熟悉与期待的。

然而，神经网络不一样，它们并不是现成的技术。我在 2016 年撰写的一篇博客中试图说明这一点，在那篇文章中我认为反向传播是「leaky abstraction」，然而现在的情况似乎更加糟糕了。

Backprop + SGD 不是魔法，无法让你的网络运行；批归一化也无法奇迹般地使网络更快收敛；RNN 也不能神奇地让你直接处理文本。不要因为你可以将自己的问题表示为强化学习，就认为你应该这么做。如果你坚持在不理解技术原理的情况下去使用它，那么你很可能失败。

2. 背着我不 work 的神经网络

当你破坏代码或者错误配置代码时，你通常会得到某种异常。你在原本应该插入字符串的地方插入了整数；导入出错；该关键字不存在……此外，为了方便 debug，你还很可能为某个功能创建单元测试。

这还只是开始。训练神经网络时，有可能所有代码的句法都正确，但整个训练就是不对。可能问题出现在逻辑性（而不是句法），且很难通过单元测试找出来。

例如，你尝试截损失度而不是梯度，这会导致训练期间的异常值被忽视，但语法或维度等检测都不会出现错误。又或者，你弄错了正则化强度、学习率、衰减率、模型大小等的设置，那么幸运的话网络会报错，然而大部分时候它会继续训练，并默默地变糟……

因此，「快速激烈」的神经网络训练方式没有用，只会导致困难。现在，这些经验性困难是使神经网络正常运行的拦路虎，你需要更加周密详尽地调试网络才能减少困难，需要大量可视化来了解每一件事。

在我的经验中，深度学习成功的重要因素是耐心和注重细节。

如何解决

基于以上两点事实，我开发了一套将神经网络应用于新问题的特定流程。该流程严肃地执行了上述两项原则：耐心和注重细节。

具体来说，它按照从简单到复杂的方式来构建，我们在每一步都对即将发生的事作出准确的假设，然后用实验来验证假设或者调查直到发现问题。我们试图尽力阻止大量「未经验证的」复杂性一次来袭，这有可能导致永远也找不到的 bug / 错误配置。如果让你像训练神经网络那样写它的代码，你会想使用非常小的学习率，然后猜测，再在每次迭代后评估整个测试集。

1. 梳理数据

训练神经网络的第一步是不要碰代码，先彻底检查自己的数据。这一步非常关键。我喜欢用大量时间浏览数千个样本，理解它们的分布，寻找其中的模式。幸运的是，人类大脑很擅长做这件事。有一次，我发现数据中包含重复的样本，还有一次我发现了损坏的图像 / 标签。我会查找数据不均衡和偏差。我通常还会注意自己的数据分类过程，它会揭示我们最终探索的架构。比如，只需要局部特征就够了还是需要全局语境？标签噪声多大？

此外，由于神经网络是数据集的压缩 / 编译版本，你能够查看网络（错误）预测，理解预测从哪里来。如果网络预测与你在数据中发现的不一致，那么一定是什么地方出问题了。

在你对数据有了一些感知之后，你可以写一些简单的代码来搜索 / 过滤 / 排序标签类型、标注规模、标注数量等，并沿任意轴可视化其分布和异常值。异常值通常能够揭示数据质量或预处理中的 bug。

2. 配置端到端训练/评估架构、获取基线结果

现在我们已经理解了数据，那我们就可以开始构建高大上的多尺度 ASPP FPN ResNet 并训练强大的模型了吗？当然还不到时候，这是一个充满荆棘的道路。我们下一步需要构建一个完整的训练、评估架构，并通过一系列实验确定我们对准确率的置信度。

在这个阶段，你们最好选择一些不会出错的简单模型，例如线性分类器或非常精简的 ConvNet 等。我们希望训练这些模型，并可视化训练损失、模型预测和其它度量指标（例如准确率）。当然在这个过程中，我们还需要基于一些明确假设，从而执行一系列对照实验（ablation experiments）。

该阶段的一些技巧与注意事项：

- 固定随机 seed：始终使用固定的随机 seed 能保证很多属性，例如在我们两次运行相同代码时能得到相同的输出。这能消除变化因子，从进行合理的判断。
- 简化：确保禁用不必要的技巧。例如，在这个阶段肯定需要关闭数据增强。数据增强可以在后期引入，并作为一种强大的正则化策略。不过在这个阶段引入的话，它就有机会带来一些愚蠢的 bug。
- 使用多数据、少次数的验证评估：当我们在绘制测试损失时，我们需要在整个比较大的测试集中执行评估。不要过几个批量就绘制一次测试损失，然后再依赖 TensorBoard 的平滑处理。我们虽然追求的是准确率，但也要防止犯这些低级错误。
- 在初始化中验证损失：验证你的损失函数在初始化中有比较合理的损失值。例如，如果你正确地初始化最终层，那么你应该通过-log(1/n_classes) 度量初始化的 Softmax 值。L2 回归和 Huber 损失函数等都有相同的默认值。
- 优秀的初始化：正确地初始化最终层。例如，如果你正在对均值为 50 的一些数据做回归处理，那么初始化的最终偏置项就应该为 50。如果你有一个非平衡数据集（两类样本数 1：10），那么就需要在 logits 上设置偏置项，令模型在初始化时预测概率为 0.1。正确配置这些偏置项将加快收敛速度，因为网络在前面几次迭代中基本上只在学习偏置。
- 人类基线结果：监控损失值等其他度量指标（例如准确度），这些指标应该是人类能解释并检查的。尽可能评估你自己（人类）获得的准确率，并与构建的模型做对比。或者对测试数据进行两次标注，其中一次为预测值，另一次为标注值。
- 独立于输入的基线结果：训练一个独立于输入的基线模型，例如最简单的方法就是将所有输入都设置为 0。这样的模型应该比实际输入数据表现更差，你的模型是否准备好从任何输入中抽取任何信息？
- 在批数据上过拟合：在单个批数据上使得过拟合（两个或多个少样本）。为此，我们需要增加模型拟合能力，并验证我们能达到的最低损失值（即 0）。我还想在同一张图中显示标签和预测值，并确保损失值一旦达到最小，它们就能完美地对齐了。
- 验证训练损失的下降：在这一阶段，你可能希望在数据集上实现欠拟合，该阶段的模型应该是极简的。然后我们尝试增加一点模型的拟合能力，再看看训练损失是否稍微下降了一些。
- 在输入网络前可视化：在运行模型之前，我们需要可视化数据。也就是说，我们需要可视化输入到网络的具体数据，即可视化原始张量的数据和标签。这是唯一的「真实来源」，我有很多次都是因为这个过程而节省了大量时间，并揭示了数据预处理和数据增强过程中的问题。
- 可视化预测过程：我喜欢在训练过程中对一个固定的测试批数据进行模型预测的可视化。这展示了预测值如何变化的过程，能为我们提供关于训练过程的优秀直觉。很多时候，如果网络以某种方式小幅度波动，那么模型最可能在尝试拟合数据，这也展示了一些不稳定性。太低或太高的学习率也很容易注意到，因为抖动量比较大。
- 使用反向传播绘制依赖性：你的深度学习代码通常包括复杂的、矢量化的、Boardcast 操作。一个常见的 bug 是，人们会无意间使用 view 而不是 transpose/permute，从而混合了批量数据中的维度信息。然而，你的网络仍然可以正常训练，只不过它们学会忽略了其它样本中的数据。一种 debug 的方法是将某些样本 i 的损失设置为 1.0，然后运行反向传播一直到输入，并确保第 i 个样本的梯度不为零。更一般的，梯度为我们提供了网络中的依赖性关系，它们在 debug 中非常有用。

- 一般化特殊案例：这是一种更为通用的代码技巧，但是我经常看到人们在使用这些技巧时会新产生 Bug，尤其是在从头构建一般函数时。相反，我喜欢直接写非常具体的函数，它只包含我现在需要做的事情。我会先让这个函数能 work，然后再一般化好函数，并确保能取得相同的结果。通常这个过程会体现在向量化代码中，我会先用循环编写某个过程，然后再一次一个循环地将它们转化为向量化化代码。

3. 过拟合

到了这个阶段，我们应该对数据集有所了解了，而且有了完整的训练+评估流程。对于任何给定的模型，我们可以计算出我们信任的度量。而且还为独立于输入的基线准备了性能，一些 dumb 基线的性能（最好超过这些），我们人类的表现有大致的了解（并希望达到这一点）。现在，我们已经为迭代一个好的模型做好了准备。

我准备用来寻找好模型的方法有两个阶段：首先获得足够大的模型，这样它能够过拟合（即关注训练损失），然后对其进行适当的正则化（弃掉一些训练损失以改进验证损失）。我喜欢这两个阶段的原因是，如果我们不能用任何模型实现较低的误差率，则可能再次表明一些问题、bug 和配置错误。

该阶段的一些技巧与注意事项：

- 选择模型：为了达到理想的训练损失，我们可能希望为数据选择一个合适的架构。当我们在挑选模型时，我的第一个建议即别好高骛远。我看到很多人都非常渴望一开始就堆叠一些新的模块，或创造性地用于各种异质架构，从而想一步到位做好。我建议可以找最相关的论文，并直接利用它们的简单架构，从而获得良好性能。后面再基于这个架构做修改和改进，并将我们的想法加进去就行了。
- Adam 是一般选择：在配置基线模型地早期阶段，我喜欢使用 Adam 算法（学习率为 3e-4）。在我的经验中，Adam 对超参数的容忍度更高，不太好的学习率也能获得一般的效果。对于卷积网络来说，一般经过仔细调整的 SGD 几乎总会略优于 Adam，但最佳学习率的可能区域要窄得多。
- 一次复杂化一个：如果你有多个特性插入分类器，我建议你一个个插入，从而确保能获得期待的性能提升。不要在最开始时就一次性全加上，这样你会弄不清楚性能提升到底是哪个特性带来的。还有其它增加复杂性的方法，例如你可以先尝试插入较小的图像，然后再慢慢地加大。
- 别相信默认的学习率衰减：如果你修改来自其它领域的代码，你应该小心使用学习率衰减方法。对于不同问题，你不仅希望使用不同的衰减策略，同时因为 Epoch 的数量不同，衰减过程也会不一样。例如数据集的大小，会影响 Epoch 的数量，而很多学习率衰减策略是直接与 Epoch 相关的。在我自己的工作中，我经常整个地关闭学习率衰减，即使用常数学习率。

4. 正则化

理想情况下，我们现在至少有了一个拟合训练集的大模型。现在是时候对它进行正则化，并通过放弃一些训练准确率来提升验证准确率了。技巧包括：

- 更多数据：首先，在当前任何实际环境中正则化模型的最好方式是增加更多真实的训练数据。在你能收集更多数据时，花费大量工程时间试图从小数据集上取得更好结果是很常见的一个错误。我认为增加更多数据是单调提升一个较好配置神经网络性能的唯一可靠方式。
- 数据增强：比真实数据较次的方法是半假数据，试验下更激进的数据增强。
- 创造性增强：如果半假数据也没有，假数据也还可以。人们在寻求扩展数据集的创造性方法。例如，域随机化、使用模拟数据、把数据插入场景这样机智的混合方法，甚至可以用 GAN。
- 预训练：即使你有足够的数据，你也可以使用预训练网络，基本没什么损失。
- 坚持监督式学习：不要对无监督学习过于激动。据我所知，没有什么无监督学习方法在当前计算机视觉任务上有很强的结果（尽管 NLP 领域现在有了 BERT 和其他类似模型，但这更多归功于文本更成熟的本质以及对噪声比更好的信号）。

- 更小的输入维度：移除可能包含假信号的特征。如果你的数据集很小，任何加入的假输入只会增加过拟合的可能。类似地，如果低级细节作用不大，试试输入更小的图像。
- 更小的模型：在许多情况下，你可以在网络上使用域知识约束来降低模型大小。例如，在 ImageNet 主干网络顶部使用全连接层一度很流行，但它们后来被简单的平均池化取代，消除了这一过程中大量的参数。
- 减小批大小：由于 BN 基于批量大小来做归一化，较小的批量大小具有更强的正则化效果。这主要因为一个批量的统计均值与标准差是实际均值和标准差的近似，所以缩放量和偏移量在小批量内波动地更大。
- drop：增加 dropout。在卷积网络上使用 dropout2d（空间 dropout）。保守谨慎的使用 dropout，因为它对 batch 归一化好像不太友好。
- 权重衰减：增加权重衰减惩罚。
- 早停（early stopping）：基于你得到的验证损失停止训练，从而在即将过拟合之前获取模型。
- 尝试更大的模型：我过去多次发现更大模型最终都会很大程度的过拟合，但它们「早停」后的性能要比小模型好得多。

最后，为了更加确保网络是个合理的分类器，我喜欢可视化网络第一层的权重，确保自己获得了有意义的边缘。如果第一层的滤波器看起来像噪声，那需要去掉些东西。类似地，网络内的激活函数有时候也会揭示出一些问题。

5. 精调

现在你应该位于数据集一环，探索取得较低验证损失的架构模型空间。这一步的一些技巧包括：

- 随机网格搜索：在同时精调多个超参数时，使用网格搜索听起来更诱惑，能够确保覆盖到所有环境。但记住，使用随机搜索反而是最佳方式。直观上，因为神经网络对一些参数更为敏感。在极限情况下，如果参数 a 很重要，改变 b 却没有影响，然后相比于多次在固定点采样，你宁可彻底采样 a。
- 超参数优化：如今社区内有大量好的贝叶斯超参数优化工具箱，我的一些朋友用过后觉得很成功。但我的个人经验是，探索好的、宽的模型空间和超参数的最佳方法是找个实习生。开玩笑而已，哈哈哈。

6. 最后的压榨

一旦你找到最好的架构类型和超参数，依然可以使用更多的技巧让系统变得更好：

- 集成：模型集成是能将准确率稳定提升 2% 的一种好方式。如果你承担不起测试阶段的计算成本，试着使用《Distilling the Knowledge in a Neural Network》中的方法把你的模型蒸馏到一个网络。
- 一直训练：我经常看到一些人在验证损失趋平时会中断模型训练，以我的经验来看，网络会长时间保持非直观的训练。寒假时有一次我忘了关掉模型训练，一月回来后发现它取得了 SOTA 结果。

结论

一旦你做到了这些，你就具备了成功的所有要素：对神经网络、数据集和问题有了足够深的了解，配置好了完整的训练/评估体系，取得高置信度的准确率，逐渐探索更复杂的模型，提升每一步的表现。现在万事俱备，就可以去读大量论文，尝试大量实验并取得 SOTA 结果了。

SYNCED

原文链接：https://karpathy.github.io/2019/04/25/recipe/