第 5 章

CHAPTER 5

算车：自动驾驶模型训练与调优

微课视频 61

微课视频 62

5.0 本章导读

本章主要围绕神经网络模型的训练和优化展开，在人工智能深度学习领域，神经网络模型的设计是基础，而对神经网络模型的训练和优化则是实现模型功能的必经路径。目前有关神经网络模型训练和优化的研究已经非常深入，形成了很多各有特点的“流派”，而本章仅对自动驾驶场景中较为常见的计算机视觉图像分类及智能小车端到端自动驾驶等基础模型进行分析和讲解，对于自动驾驶技术研发中出现的一些新模型及其训练优化方法，本章将不再过多扩展。

前面 4.1 节介绍了自动驾驶模型研发的 4 大环节：数据、模型、损失、优化。第 4 章重点讲述如何完成其中的“模型”部分，即如何构建一个简化的端到端的神经网络[12]模型，把从摄像头采集到的视觉信号转换成智能小车的扭矩、转向角等执行信号。这个端到端神经网络基本结构可以选择 CNN(卷积神经网络)模型，其功能是实现对摄像头输出的图像(路况信息)进行处理，而处理的结果则是对智能小车行驶方向(例如左转、右转等)和速度的“指挥”。当神经网络模型的基本结构建立好之后，神经网络模型的功能并不是随之完成的，初建的模型实际上不具备任何功能，就如同一个初生的婴儿无法说出有实际意义的语言一样。初建的模型要通过数据集的训练才有可能实现特定的功能，而训练的过程就如同儿童认知的过程，需要有良好的学习方法和不断的实践验证，这就是本章重点介绍的内容，对应于模型研发中的“损失”和“优化”的两个环节。

端到端模型训练的基本框架如图 5.1 所示，在这个简化版本的自动驾驶系统中，CNN 模型充当的角色是“大脑”，为了让“大脑”具备功能，则必须经过训练，训练的材料就是数据集。根据第 3 章的介绍，数据集中有来自车辆内部或者外部各类传感器的大量数据，其中最基础的就是摄像头的图片数据、驾驶方向盘的转动和油门的百分比数据，这恰好构成了本章介绍的端到端模型训练和优化的基本数据集。应当指出，真实的自动驾驶模型远比本章介绍的端到端模型复杂，所采用的数据信息也远远超过本章所介绍的范围，而本章介绍的内容

却是更高级别技术研发的基础，学习这些内容有助于读者轻松打开通往自动驾驶技术研发的大门。

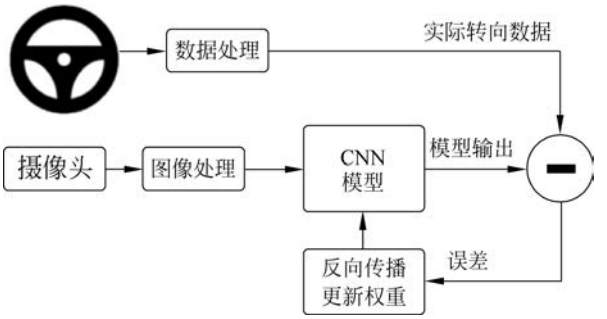

图 5.1 端到端模型训练的基本框架

在图 5.1 中，以汽车的转向控制说明模型训练的基本框架。大量来自车载摄像头的图片数据经过随机位移或者旋转等预处理后输入 CNN 模型中，经过模型运算产生输出结果，开始时模型输出的是像抛硬币一样的随机值，需要与驾驶方向盘实际的操作数据进行比对，利用比对之后的误差信息对 CNN 模型内部的参数进行调整，调整的目的是减少误差，让比对误差趋向于零，即模型的输出动作与采集到的真值数据一致，这个过程就是对模型的训练。当误差越来越趋近于零，小到一个最低阈值时，这个模型就具备了自动驾驶的基本功能，当有新的车载摄像头数据输入时，其输出的结果就可以直接用于控制汽车的方向盘。如何利用误差信息更加有效地调整模型内部参数，这是模型优化需要实现的目标。本章将围绕如何建立训练指标参数，如何完成参数调整以及如何提高运算效率等问题进行相应的介绍。

5.1 模型与训练参数


微课视频 63

5.1.1 模型训练数据

自动驾驶模型训练数据的采集一般是通过人工驾驶装载有采集设备的汽车，在道路行驶过程中一边采集道路环境数据一边记录人工驾驶的操作数据，两方面的数据相结合，最终形成自动驾驶训练所需的数据集。人工驾驶的操作数据将作为自动驾驶生成操作数据的重要参考，而车辆转向和油门的相关数据则是这些操作数据中最基本的构成元素。尽管第 3 章介绍了很多业界的自动驾驶数据集，但对本书使用车道沙盘环境的智能小车而言，这些数据都过于复杂，无论做数据处理还是进行模型训练，所付出的代价都很大，因此本节将对智能小车本身所需的训练数据进行简单的介绍。

车辆行驶中可以利用车辆的转弯半径 r 作为转向角的描述参数，如图 5.2 所示。最小转弯半径 $r_{\min}$ 是指当汽车的方向盘转到某个方向极限的位置时，汽车的行驶路径就会绕着一个圆心画圆，显然 $r_{\min}$ 越小，汽车的转向角就越大。汽车转向角的大小与车辆自身特征

相关，例如对于前后轮距大的汽车，其最小转弯半径相对也大，相同车身尺寸特征的汽车 $r_{\min}$ 越小，其方向盘能够控制的转向轮的转向角也就越大。为了与转向角描述表示保持一致，通常会采用 $1/r_{\min}$ 记录智能小车的最大转向角。同样，在汽车行驶过程中的转向角也可以用 $1/r$ 表示，然而在真实环境中车辆操作的转向角数据是非常复杂的，对于驾驶员在通过不同弯角道路时，转动方向盘的程度是不同的，即使通过相同的转向角道路，车辆的转向操作也会因人而异，与通过时的车速、驾驶员的经验、车辆的载重状态都密切相关。

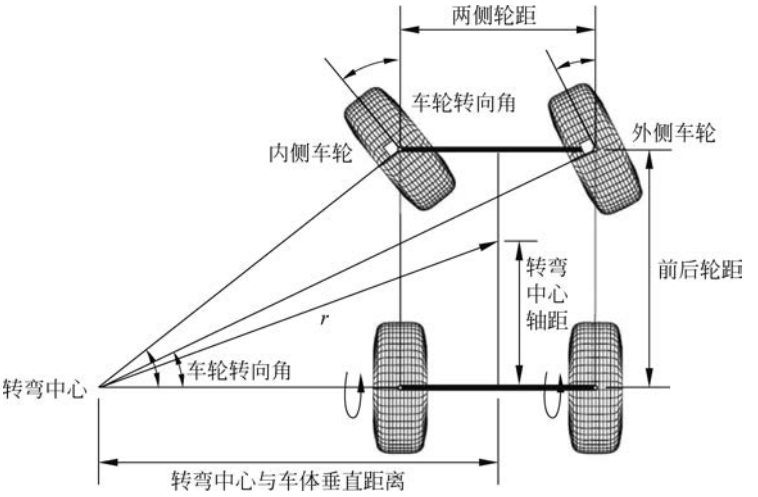

图 5.2　车辆的转弯半径和转向角

因此，类似于转向这种操作数据需要人为降低其复杂度。对复杂操作数据通常可以采用量化编码的方式简化，而最极端的简化就是将转向数据直接判定为“转向”“未转向”两类以区分转向的方向，转向数据最终可以判断为“左转”“直行”“右转”三类。而转向的程度会直接使用 $r_{\min}$ 这个最小转弯半径作为转向角。虽然这种极端的简化会对真实的车辆操作带来严重的影响，但是在本书所介绍的智能小车中，这种极端的简化却是一种可以选择的方案。在智能小车中，自动驾驶模型转向控制参数的输出频率远高于人类驾驶员的操作频率，例如每秒输出控制参数 20 次。使用这种极端的简化方式，智能小车自动驾驶模型每一次控制输出都是极端的，要么左转到底，要么右转到底，但是由于智能小车对控制的响应是有惯性的，这种高频度的操作控制最终展现出的效果并不是智能小车前进时的左右摆动，而是类似于人类驾驶员操控汽车那样令智能小车连贯性转向运动。因此，智能小车所采用的数据集中除了包括车载摄像头记录的第一视角路况图像，同时包括的人工驾驶操作数据可以采用上述极端简化的格式进行记录，例如对应的行进方向用“左转”“直行”“右转”表示，而转向时的角度则用 $1/r_{\min}$ 替代。

在智能小车的自动驾驶实验中，对于给定道路场景 $\boldsymbol{x}$，驾驶员的操作 $\boldsymbol{y}$ 定义了一个样本。通过人工驾驶的方式累计收集了 m 个有标记的样本数据，可以按下面的形式分别定义数据集 D、数据特征集 X 和数据标注集 Y。

$$D=\{(x_i,y_i)\},\quad i=1,2,\cdots,m$$
$$X=\{x_i\},\quad i=1,2,\cdots,m$$
$$Y=\{y_i\},\quad i=1,2,\cdots,m$$

其中，x_i 为特征，在自动驾驶中一般指从道路场景中得到的各种传感器信息，在智能小车中主要是来自摄像头的图像信息；y_i 为在当前道路场景 x_i 下应该采用的驾驶操作，根据候选驾驶操作集的数量定义该向量的维度，例如如果将自动驾驶的智能小车的转向离散化为“左转”“直行”“右转”三个可能值，那么 y_i 可以用独热编码(one-hot encoding)的形式定义为三维向量，并用[1,0,0]、[0,1,0]、[0,0,1]分别表示“左转”“直行”“右转”这三个对应的操作。

由大量道路场景图像和操作标签组成的数据就是智能小车模型训练所需的数据集 D。


微课视频 64

5.1.2 智能小车 CNN 模型

根据第 4 章对卷积神经网络的介绍，下面以自动驾驶智能小车端到端基础模型为例进行讲解。自动驾驶智能小车 CNN 模型(见图 5.3)的输入层将摄像头产生的图像数据转换为适合神经网络计算的张量数据，然后数据连续进入 5 个卷积层(其中激活函数未画出)和后续的处理层。

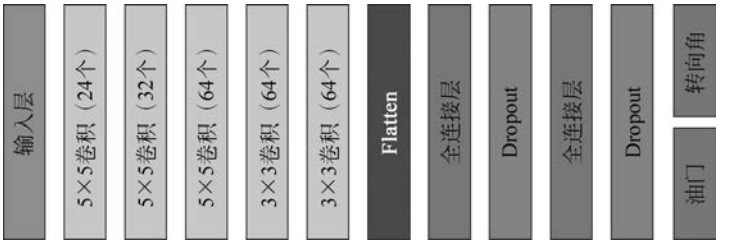

图 5.3 智能小车 CNN 模型

主要说明如下：

(1) 对摄像头获取的图像数据进行重采样，符合网络模型对输入图像分辨率要求后，数据通过输入层进入模型网络。

(2) 数据依次进入 5 个二维卷积层：前 3 层为步幅为 2 的 5×5 卷积，且通道数逐层加深，由输入层的 3 增加到 64。后 2 层为 2 个 3×3、步幅为 1 的卷积层，卷积核数目均为 64。通过这 2 层进一步抽取图像中的特征信息。在 5 层卷积处理之后，沿图像空间尺度方向(宽和高方向)的大小被缩小，信息从空间尺度上浓缩，而在通道维度上用更多的特征图拓展。

(3) 完成卷积的数据依次进入全连接层和处理层：首先是将数据用 Flatten(压平)层将三维张量拉长为一维数组，然后接下来用 Flatten 层将特征图由三维张量拉平为一维数组，并进入 2 层全连接层，用非线性变换将特征的维度降低至 100。为防止出现过拟合，在全连接层后用 Dropout(丢弃)层，将本层的输入特征随机丢弃 10%再向下一层传递。

(4) 网络的最后是转向角和油门两个输出层。在示例网络中，将转向角离散化为 3 个

值(左转、右转、直行)，将油门定义为单一输出值。相当于将前者定义为分类问题，需要用softmax函数产生每种转向动作的概率输出，后者定义为回归问题，直接输出油门的百分比。

5.1.3　参数和超参数


微课视频65

一般而言，神经网络可被视为由若干“层”构成的复合运算网络。在前面定义的端到端CNN模型中，主要包含卷积层、全连接层、Dropout层等，其中卷积层和全连接层都具有可学习的参数，这些参数是神经网络训练过程中要被优化的对象，而Dropout层、Flatten层等是对数据的变换，不具有可学习的参数。假设深度神经网络的模型一共有 L 层，则深度神经网络的参数可以表示为

$$\Theta=\{\theta^{(i)}\},\quad i=1,2,\cdots,L-1$$

其中，$\theta^{(i)}$ 表示连接第 i 层和第 $i+1$ 层的参数矩阵，深度神经网络的全体参数记为 Θ，这里也可以简称为模型参数集或者模型权重集。

训练神经网络的目的就是要找到一套好的模型参数，使之可以将输入数据映射为合适的输出值。现代的神经网络模型，特别是深度学习神经网络模型，通常具有非常大的参数量，典型的深度模型通常都具有百万量级的参数，而当前业界中一些超大模型参数的数量已经开始用“千亿”为单位进行计算了。这样规模的模型参数需要通过计算机用深度学习算法，从数据集中学习到的更新的信息，自动地进行更新。模型训练需要大量数据，用迭代的方式将数据传递给模型并更新其中的参数，这个过程所需的计算量巨大，对计算机的算力也有相应的要求。

如果说深度神经网络中的参数是基本变量，是模型“微观”视角的内部变量，那么超参数(hyperparameter)则是控制深度神经网络模型配置和训练的外部“宏观”变量。例如，神经网络的层数是一个超参数，它影响了模型的容量和学习能力；神经网络训练过程中的学习率影响模型训练的收敛速度和收敛的质量。超参数通常需要预先定义，它们对模型训练的效率和训练的效果都会产生影响。常见的超参数及其对模型训练的敏感性(其数值变化对模型训练的影响能力)如表5.1所示。

表5.1　模型超参数与敏感性

超　参　数	敏　感　性
学习率(learning rate)	高
损失函数(loss function)选择	高
迭代周期数(epoch)	高
数据批量大小(batch-size)	中
隐藏层数(number of hidden layers)	中
隐藏层的单元数/神经元数(number of hidden layer units)	高
优化器(optimizer)选择	低
网络初始化权重(weight initialization)	中

智能小车自动驾驶的控制器选用了 CNN 模型，其模型参数集$\boldsymbol{\Theta}$包括每个卷积核的数值和全连接函数运算时的权重数值(这些数值是神经网络各层连接的参数)。对于一个二维卷积核，通常其参数就是一个矩阵，以尺寸为 3×3 的卷积核为例，其参数矩阵的大小是 3×3，矩阵有 9 个元素变量(模型参数)；对于 3 通道二维卷积核(尺寸 3×3)，其参数矩阵的大小就是 3×3×3，矩阵有 27 个元素变量；对于更多通道(n 通道)二维卷积核，其参数矩阵的大小就是 $m_1 \times m_2 \times n$(m_1，m_2 表示二维卷积核的尺寸)。显然对于图 5.3 中的 CNN 模型，如果输入的图像是 RGB 三通道数据，第一个卷积层中的卷积核尺寸就是 5×5×3，而这一层中这种卷积核的数量是 24 个，总的参数个数为 5×5×3×24。由于这些参数的初始值几乎都不相同，因此同一幅 RGB 图像数据经过第一个卷积层的运算后，24 个卷积核产生的计算结果也完全不同。

那为何需要这么做呢？回顾第 4 章的介绍，可以把每个卷积核都看成一个匹配特定模板的滤镜，透过这些滤镜观察同一幅 RGB 图像，看到的是完全不同的样子。卷积核的参数决定了滤镜的作用，用不同参数的滤镜处理同一幅 RGB 图像，相当于用多个不同的滤镜独立地处理，分别生成对应的模板匹配值。这个输出仍然是多维张量，与输入的 RGB 图像并无二致，因此仍然可以当作图像被下一层卷积层处理，主要的区别在于这种中间特征图不再具有 RGB 颜色通道的含义，且通道数量可以是任意整数。

经过初始化的神经网络中，卷积核参数是随机的，因此经滤镜处理生成的特征图也是没有意义的，那么如何修改滤镜性能使得生成的特征图具有意义呢？这正是模型训练需要解决的问题。显然，整个 CNN 模型中需要调整的参数有很多，每一幅输入图像数据能够提供的调整也十分有限，因此这个调整过程依赖于大量的数据输入和误差校正，需要的计算量也是巨大的，这就是模型训练对算力要求的根本原因。但如果仅仅是保证了算力的提供，也未必就能完成调整。如何调整？调整的规则是什么？这些也是训练中需要解决的关键问题。如表 5.1 所示的超参数，对于模型中参数的调整会产生一定的影响，敏感性高的超参数对参数的调整影响大，反之则小。下面会对所涉及的超参数进行介绍。

5.1.4 损失函数

在自动驾驶模型研发的第 3 个环节中，损失函数用于定量评估当前模型性能的好坏。更具体地说，它表示在当前参数集的取值情况下，给定输入值 x 产生的模型输出响应$\hat{y}$ 与真值 y 的偏离程度。定义损失函数是为了判断每次参数调整后模型输出误差变化的情况。如果模型内部参数的初始设定值是随机的，那么在训练过程中，这些参数的调整必然要遵循一些规则，有些参数调大，有些参数需要调小，经过这些调整后，模型整体的输出误差应该逐步减小。定义损失函数使得计算机有了定量的明确目标，即减小损失函数的数值，从而可以调用优化算法，自动地使用算法中定义的规则完成优化。

以智能小车的 CNN 模型(含模型参数$\boldsymbol{\Theta}$)为例，当输入道路场景图像 $\boldsymbol{x}$ 后，模型推理(前向运算)输出的结果将是对驾驶操作的预测 $\hat{\boldsymbol{y}}$。首先定义在模型推理过程中所产生的一

些中间变量：令 $\boldsymbol{a}^{(i-1)}$ 为神经网络第 $i-1$ 层的输出结果，$\boldsymbol{z}^{(i)}$ 为第 i 层进行线性变换结果，$\boldsymbol{a}^{(i)}$ 是将激活函数 g 作用于 $\boldsymbol{z}^{(i)}$ 的结果。从 $\boldsymbol{a}^{(i-1)}$ 到 $\boldsymbol{z}^{(i)}$ 的变换(线性变换)记为 $\theta^{(i-1)}$；从 $\boldsymbol{a}^{(i-1)}$ 到 $\boldsymbol{a}^{(i)}$ 的变换(先进行线性变换，后通过激活函数进行非线性变换)记为 $h_{\theta^{(i-1)}}$。

$$\boldsymbol{a}^{(i)}=g(\boldsymbol{z}^{(i)})$$
$$\boldsymbol{z}^{(i)}=\theta^{(i-1)}(\boldsymbol{a}^{(i-1)})$$
$$\boldsymbol{a}^{(i)}=h_{\theta^{(i-1)}}(\boldsymbol{a}^{(i-1)})=g(\theta^{(i-1)}(\boldsymbol{a}^{(i-1)}))$$

输入道路场景图像 $\boldsymbol{x}$ 在模型参数$\boldsymbol{\Theta}$ 作用下得到模型推理结果 $\hat{\boldsymbol{y}}$ 的全过程可以记为 $\boldsymbol{H}_{\boldsymbol{\Theta}}$：

$$\hat{\boldsymbol{y}}=\boldsymbol{H}_{\boldsymbol{\Theta}}(x)=h_{\theta^{(L)}}(h_{\theta^{(L-1)}}(h_{\theta^{(L-2)}}(\cdots h_{\theta^{(1)}}(\boldsymbol{x})\cdots)))$$

显然，从道路场景图像 $\boldsymbol{x}$ 到输出模型预测驾驶操作 $\hat{\boldsymbol{y}}$ 的推导的过程路径可简记为

$$\boldsymbol{x}=\boldsymbol{a}^{(1)}\rightarrow\boldsymbol{z}^{(2)}\rightarrow\boldsymbol{a}^{(2)}\rightarrow\cdots\rightarrow\boldsymbol{z}^{(L-1)}\rightarrow\boldsymbol{a}^{(L-1)}\rightarrow\cdots\rightarrow\hat{\boldsymbol{y}}$$

在智能小车 CNN 模型中，输入的道路场景图像 $\boldsymbol{x}$ 先经过卷积和激活函数的计算后，还需要通过全连接层、丢弃层等的计算才能从道路场景图像 $\boldsymbol{x}$ 中构建出关键的特征表示，其中全连接层和丢弃层的计算也可以用类似 $\boldsymbol{h}_{\boldsymbol{\theta}}$ 定义进行表达，并将最后的结果 $\boldsymbol{a}^{(L-1)}$ 输入线性分类器(softmax 函数是一种实现分类常用的算法)以得到道路场景图像 $\boldsymbol{x}$ 对应的驾驶操作预测 $\hat{\boldsymbol{y}}$。考虑到自动驾驶任务的复杂性，模型输出的线性分类器会被定义为多分类问题的类型，智能小车的模型输出结果就有“左转”“直行”“右转”等多个转向分类。对于模型的预测结果 $\hat{\boldsymbol{y}}$，一般取概率值最大的输出节点对应的驾驶操作作为模型的最终输出，例如 5.1.2 节中模型的输出为 $\hat{\boldsymbol{y}}=[0.5,0.3,0.2]$时，具体的结果则应该采用“左转”作为当前道路场景下的驾驶操作选择。

损失函数可定义为样本(数据)标注 $\boldsymbol{y}$ 和模型预测输出 $\hat{\boldsymbol{y}}$ 之间的误差，即

$$L(\boldsymbol{y},\hat{\boldsymbol{y}})$$
$$\hat{\boldsymbol{y}}=\boldsymbol{H}_{\boldsymbol{\Theta}}(\boldsymbol{x})$$

对于给定的第 i 个样本 x_i，比较根据其深度神经网络预测值 $\hat{\boldsymbol{y}}_i$ 和事先已被标记的样本标注 $\boldsymbol{y}_i$。对于转向角的输出，本章的 CNN 模型将其定义为对离散值的预测，因此损失函数选择使用交叉熵(cross entropy，CE)作为单样本损失 L_i 的定义，则

$$L_i=\mathrm{CE}(\boldsymbol{y}_i,\hat{\boldsymbol{y}}_i)=-\sum_{j=1}^{K}y_{i,j}\cdot\ln\hat{y}_{i,j}+(1-y_{i,j})\cdot\ln(1-\hat{y}_{i,j})$$

其中，$y_{i,j}$ 表示第 i 个样本的标记在第 j 维的取值，例如 $\boldsymbol{y}_i=[1,0,0]$，$y_{i,1}=1$，$y_{i,2}=0$，$y_{i,3}=0$；$\hat{y}_{i,j}$ 表示第 i 个样本的模型预测输出在第 j 维的取值，例如$\hat{\boldsymbol{y}_i}=[0.5,0.3,0.2]$，$\hat{y}_{i,1}=0.5$，$\hat{y}_{i,2}=0.3$，$\hat{y}_{i,3}=0.2$。

通过对全体样本的交叉熵损失函数求平均，不难将单样本交叉熵损失函数推广至全体样本，则

$$J(\boldsymbol{\Theta})=L_{\text{angle}}=L_{\boldsymbol{\Theta}}(\boldsymbol{y},\hat{\boldsymbol{y}})=\frac{1}{m}\sum_{i=1}^{m}L_i=\frac{1}{m}\sum_{i}\mathrm{CE}(\boldsymbol{y}_i,\hat{\boldsymbol{y}}_i)$$
$$=-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K}y_{i,j}\cdot\ln(\hat{\boldsymbol{y}}_{i,j})+(1-y_{i,j})\cdot\ln(1-\hat{\boldsymbol{y}}_{i,j})$$

由此定义的损失函数能够成为判定模型训练好坏的指标，模型在利用大量数据进行反复训练的过程中，如损失函数的输出不断降低，就意味着模型输出的误差越来越小，输出值越来越接近训练样本的标注值，模型将逐步具备预设的功能意义。

类似地，也可以定义并计算油门输出的损失函数。油门采用的是连续值的输出，定义为一个回归问题，因此损失函数采用均方差(mean squared error，MSE)，其定义为

$$\mathrm{MSE}=\frac{1}{m}\sum_{i=1}^{m}(y_i-\hat{y}_i)^2$$

最终总的损失函数应当是两部分输出的损失函数的加权和(具体权重的取值需要经过测试选择合适的数值)，并以此为依据对整个神经网络的参数进行调整。

$$J(\boldsymbol{\Theta})=0.9\times L_{\mathrm{angle}}+0.1\times L_{\mathrm{throttle}}$$

如果训练的结果最终是调整模型内部参数使模型输出的损失函数出现最小值，那么训练问题就转换为求解参数优化问题，即寻找

$$\boldsymbol{\Theta}^{*}=\mathrm{argmin}_{\boldsymbol{\Theta}}J(\boldsymbol{\Theta})$$

其中，$\boldsymbol{\Theta}^{*}$ 表示使函数 $J(\boldsymbol{\Theta})$ 达到最小值时 $\boldsymbol{\Theta}$ 的取值。现实中无法在模型训练过程中达到真正的最小值，一般通过优化迭代将损失函数减小到一定范围即可。

通常在设计深度神经网络过程中，对模型进行训练需要预先确定模型的超参数。选择恰当的超参数，能够提高模型参数优化问题的求解效率。这个求解过程通常是利用优化算法通过反向传播(backward propagation，BP)算法计算出关于 $\boldsymbol{\Theta}$ 的梯度，基于 $\boldsymbol{\Theta}$ 的梯度，使用随机梯度下降(stochastic gradient descent，SGD)算法更新 $\boldsymbol{\Theta}$ 值，进一步最小化损失函数，从而完成对深度神经网络内部参数的优化。

5.2 神经网络模型训练

模型训练的过程就是在超参数已确定的前提下，求解 $\min_{\boldsymbol{\Theta}}J(\boldsymbol{\Theta})$，从而确定 $\boldsymbol{\Theta}$ 的过程。而 $J(\boldsymbol{\Theta})$ 又是典型的多极值非凸函数，在实际训练模型过程中，会发现即使超参数一致也会有每次优化的结果不一样的情况。根据实际经验，学习率的选择会显著影响模型优化的最终结果，优化器的求解和数据样本的分批量大小对优化效果也有一定的影响。本节将介绍梯度下降和反向传播算法求解梯度。在深度学习框架中，反向传播算法一般属于完全后端实现的模块，不对用户暴露接口，而梯度下降的过程则需要考虑到上述超参数的选择，具体超参数的优化选择会在后面介绍，本节重点在于理解这些超参数对优化结果的影响。

用 Python 程序表达的智能小车 CNN 模型训练的主要代码如下(可参见每行代码行的注释，后续代码将不再进行逐行注释)。

```
1  # 根据 n_epoch 变量的数值进行多次循环运行，每次循环的序号存储在 epoch 变量中
2  for epoch in range(n_epoch):
3      # 将模型设置为训练模式
4      model.train()
```

```
5
6       #初始化 losses 损失量数组
7       losses = []
8       # 将训练数据集 train_loader 中的训练数据提取到 data 变量,
9       # 并重复循环直至训练数据集中所有数据都被提取完成
10      for data in train_loader:
11          # 将本轮训练数据 data 中的图像数据转移到目标设备(例如 GPU)
12          x_img = data['image'].to(device)
13          y_ang = data['angle'].to(device)
14          y_thr = data['throttle'].type(torch.float32).to(device)
15
16          # 将优化器 optimizer 状态重新归零
17          optimizer.zero_grad()
18          # 将 x_img 输入模型,并得到运算结果 out_ang, out_thr
19          out_ang, out_thr = model(x_img)
20          # 用负对数似然损失函数工具获得转向变量的损失值
21          loss_ang = F.nll_loss(out_ang, y_ang, reduction = 'mean')
22          # 用均方差损失函数工具获得油门变量的损失值
23          loss_thr = F.mse_loss(out_thr.squeeze(), y_thr, reduction = 'mean')
24          # 加权转向和油门变量的损失值生成总损失值
25          loss = 0.9 * loss_ang + 0.1 * loss_thr
26          # 损失函数反向传播
27          loss.backward()
28          # 执行优化步骤,更新模型权重
29          optimizer.step()
```

下面将展开阐述这段代码背后的原理。

5.2.1　梯度下降迭代


微课视频 66

模型训练的代码在整体逻辑上是一个两层的循环：外层循环重复 n_epoch 次，把完整的数据集传给内层的训练逻辑，内层循环把完整的数据集拆分为多个小批量，迭代更新模型内部的参数。在每一次小批量的迭代中，包含以下几个子步骤：

(1) 将数据复制到计算设备(通常是 GPU)上。

(2) 将优化器的状态重新归零。

(3) 进行前馈计算，将输入传给 CNN 模型，经各层计算之后得到两个输出。

(4) 分别计算两个输出的损失函数，加权求和得到总损失函数值。这一步将得到在当前小批量的数据中，模型预测的输出值与真值的差异。

(5) 反向传播，得到损失函数关于各参数的梯度值，即每一个参数应该往哪个方向调整，以及在这个位置损失函数对该参数的敏感程度。

(6) 按照预定义的优化器算法，执行参数更新。

1. 迭代优化

可以看到，上述步骤整体的逻辑都是在为最后两个步骤服务的，训练的核心步骤就是通过反向传播得到损失函数关于每一个参数的偏导数，并以此为依据对参数进行更新，迭代多次直到损失函数减小到可接受的范围。之所以需要把数据拆成小批量并且每次迭代更新一

小步，有几方面的原因：①神经网络的损失函数是一个关于模型参数的非凸函数，且具有数量庞大的优化变量，因此其优化方法有别于传统的凸函数的最优化方法；②将问题转化为对Θ取优，使其损失值稍微减少，那么问题的难度就大大降低了；③通过拆分成小批量，每一步的计算都可以快速地在GPU有限的内存中完成，以便有效地利用硬件加速器的算力，实现更快的收敛。

2. 迭代优化的最优方向——梯度

梯度是一个向量，通常表示某个函数在该点处的方向导数的最大值，也就是沿着该方向（此梯度的方向）函数在该点处变化最快，变化率（梯度的模）最大。用梯度解决优化问题是非常有效率的一种方法。以图5.4为例，损失函数构成的曲面在局部有很多极值点，找到这些极值点并求出最小值就可以完成优化问题的求解。显然如果从曲面上任意一点开始移动，若想用最快速度移动到最近的极值点，从这个点的局部信息来看，沿着梯度是最好的办法。在移动前，先求取当前位置点的梯度，然后沿着下降梯度方向移动到下一个点，再重复求取当前位置的梯度，再继续沿着梯度方向移动，最后如果收敛，就会抵达极值点。但为什么有时所抵达的终点不一定是最小值点呢？这是因为这个曲面上可能存在若干个极值点，当出发起始点的位置选择不恰当时，沿着梯度前进未必就能正好找到最小值点，很可能是进入了最靠近起始点位置的极值点。一旦进入极值点，如果还是单纯使用原先的梯度方法，更大的可能性是陷入这个极值点所在的“坑”中无法走出来，移动迭代过程会被判定为结束，而此时所找到的点不是最小值，也无法得到全局最优解。对于具有百万量级甚至更高数据量级的神经网络来说，损失函数的优化问题会更加复杂，因为高维空间中存在更多鞍点（即偏导数为0，但仍未达到局部最优），这时优化器很难从这种状态中跳出。尽管如此，梯度法目前仍然是最有效的方法，只不过在具体实施过程中需要采用一些策略进行调整，以避免过早地陷入局部极值点上，后续将对此做进一步的分析。

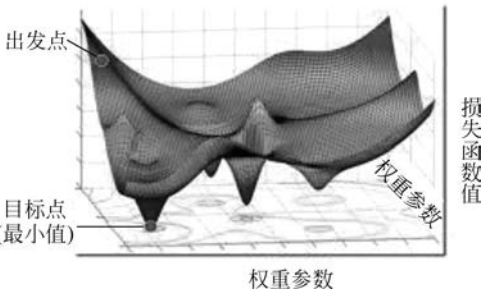

图5.4 损失函数参数优化问题示意图

而迭代优化的方向不需要随机寻找，因为可以直接计算出最好的方向，这就是从数学上计算出当前局部最陡峭的方向，这个方向就是损失函数的梯度。假设在反向传播中已经得到参数$\theta^{(i)}$的梯度的$\nabla_{\theta^{(i)}}$（计算方法会在后续详细介绍），可用如下迭代公式更新参数的值

$$\theta^{(i)} = \theta^{(i)} - \eta \times \nabla_{\theta^{(i)}}$$

其中，η为学习率或步长。

3. 学习率

梯度指明了函数在哪个方向是变化率最大的，但是没有指明在这个方向上应该走多远。

选择步长(也叫作学习率)将会是神经网络训练中最重要(也是最困难)的超参数设定之一。这就好比，人们可以感觉到脚朝向的不同方向，地形的倾斜程度不同，但是该跨出多长的步长呢？不确定。如图 5.5 所示，如果谨慎地小步走，可以比较稳定地收敛到某个局部最优解，但是进展较慢，需要较多的迭代次数和时间。相反，如果想尽快下山，那就大步走吧，但结果有时也会适得其反。因为梯度只指示了在当前局部位置的最陡峭方向，并不是指向真正的最优值，在某些点，如果步长过大，反而可能越过最低点导致更高的损失值。

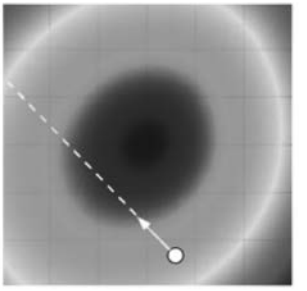

图 5.5　学习率与梯度下降

学习率是模型训练优化器的重要参数，在智能小车模型训练中，具体代码如下(其中，变量 lr 就是学习率)：

```
1  model = Net()
2  optimizer = torch.optim.Adam(model.parameters(), lr = 0.0005)
```

4. 随机梯度下降

梯度下降算法的基本思想是跟随负梯度方向，重复地计算梯度然后对参数进行更新。梯度下降算法是对神经网络的损失函数最优化处理中最常用的方法。模型参数Θ一直跟着梯度走，直到结果不再变化。正如 5.2 节代码所示，这个简单的循环已经被现在的深度学习框架例如 PyTorch 作了高度的封装，使用者并不需要手动实现其中的细节。

在自动驾驶深度神经网络的训练过程中，训练数据可以达到百万量级。根据损失函数计算的公式 $L=\frac{1}{m}\sum_i L_i$，每一次计算损失函数都需要对完整的数据集进行遍历，计算每一个数据点上的损失函数，再计算算术平均。如果计算整个的训练集才获得仅仅的一次参数更新，从算力成本的角度来看，这是十分浪费的，因此一个常用的方法是选取训练集中的小批量(mini batch)数据进行单独的计算和参数更新，即小批量数据梯度下降(mini batch gradient descent)算法。这个算法的本质是用小批量数据中的损失函数作为全量数据损失函数的近似，通过在训练集中多次选取不同的小批量数据进行训练来大幅提高参数更新的效率。例如在目前常用的卷积神经网络中，整个训练集包含几十万甚至上百万个样本，而对于小批量数据中的样本数，其典型值仅为 256 个，每次计算一个小批量数据就可以实现一次参数的更新，而每次迭代计算所需的时间远远少于整个训练集训练所需的时间，参数更新的效率大大提高。这个算法之所以行之有效，是因为训练集中的数据都是相关的，小批量数据的梯度就是对整个训练集梯度的一个近似。因此，在实践中通过计算小批量数据的梯度可以实现更快速的收敛，并以此来进行更频繁的参数更新。小批量数据策略中有个极端情况，就是将小批量中样本数量设置为 1 个，即每次迭代都只随机选取一个样本，并根据单个样本的梯度信息来更新模型参数，这种策略被称为随机梯度下降(stochastic gradient descent，SGD)算法，有时候也被称为在线梯度下降算法。这种策略在实际情况中极少采用，因为当前的计算训练通常都是在 GPU 等加速器中进行向量化的操作，一次计算 100 个数据的向

量比 100 次计算 1 个数据的标量要高效得多。因此，通常所说的 SGD 算法指的是小批量数据梯度下降算法。

小批量数据中包含的样本数的大小（即批量大小，batch size）是一个超参数，在实际应用中由于其影响相对较小，有时候并不通过交叉验证调用参数，而是在 GPU 内存允许的情况下选择尽可能大的批量大小，使得 GPU 可以尽量在满负荷的状态下运行，或者有时干脆设置为固定大小，例如 32、64、128 等。

此外，超参数 epochs 表示遍历全体样本集的次数（迭代周期数）。为了避免在出现无法收敛的情况时，数据的训练过程无法终止，通常会人为设置一个重复遍历全体样本集的次数作为训练运算的上限，这个上限就是 epochs。如果 epochs 过小，很有可能训练并未达到优化迭代的最佳次数就终止，此时得到的结果并不是最优解。如果训练超过一定次数时，训练也很有可能出现过拟合的现象，过拟合会导致模型最终的适应性降低。如果在训练中发现产生了过拟合现象时，训练通常需要提前终止，而不一定要达到 epochs 所设定的上限值。

在神经网络基于梯度下降算法的优化中，还有一个重要概念是“动量”（momentum）。因为每一个小批量的迭代计算中，小批量数据提供的梯度信息仅仅是真实梯度信息的近似，还存在一定的随机噪声，如果单纯地用梯度下降算法优化，优化的路径将会受噪声影响变得十分曲折，收敛很慢。一个有效的解决办法是对移动路径进行移动平均，这样可以使得优化路径变得更加光滑。动量方法正是基于这个思想，对每一步的梯度信号进行移动指数平均，减小局部噪声的影响。这种情形就像在从山上下坡时，每一个局部点的梯度都会受到各种小石头的影响，使得局部提供的梯度信息与全局梯度不一致。而动量方法就像是一个有质量的小球滚下山，它本身的动量可以帮助小球冲过局部的坑洼，更快地滚下山。因为动量的有效性，它也是深度学习框架中 SGD 优化器的自带的重要参数。

除了原始的 SGD 算法以外，实际中也有一些其他基于 SGD 算法的改进优化算法，其中的代表是 Adam 算法。Adam 算法的主要特点是可以根据优化情况自适应地独立调整每一个参数的学习步长。因为这个特性，它相比 SGD 算法可以更不依赖人工调整参数，很多时候默认设置就能得到较好的效果，因此 Adam 算法也是目前应用最广泛的优化算法之一。


微课视频 67

5.2.2 反向传播梯度计算

反向传播（back propagation，BP）算法是深度学习神经网络中的常用学习算法，它是梯度下降算法的基础。反向传播算法是用自动微分（automatic differentiation）实现精确梯度计算的方法之一，适用于神经网络中的梯度计算。它是各种深度学习框架最基础的功能，虽然各框架在底层反向传播算法的设计上都有一些不同的取舍，但原理上采用的都是反向传播算法。

如图 5.6 所示，反向传播算法由正向传播（前馈计算）和反向传播（梯度计算）这两个环节反复进行循环迭代而构成，这个循环迭代会一直持续到网络对输入的响应输出达到预定的目标范围为止[13]。在正向传播过程中，输入数据沿着正向路径，经过输入层、隐藏层传向输出层，得到输出结果，然后将输出结果与真值比较，计算损失函数值。然后从损失函数开

始，进入反向传播环节。通过逐层求出目标函数对网络模型中各神经元权重值的偏导数，构成目标函数对权重值向量的梯度，作为修改权重值的依据。

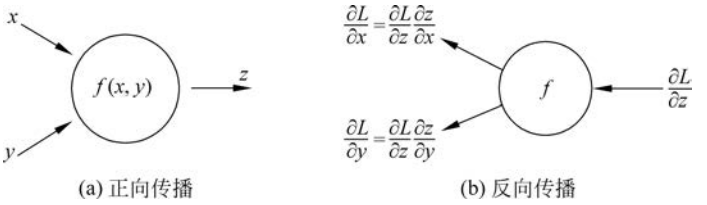

图 5.6　反向传播算法

1. 求导的链式法则

反向传播可以看成微积分中复合函数求导的链式法则在神经网络中的应用，而神经网络正好可以看作由一层层函数嵌套而成的复合函数。

在神经网络的梯度计算中，最终的目标是求出损失函数 L 相对于每一个参数的偏导数。图 5.6 给出了复合函数视角下神经网络中正向传播和反向传播在一个神经元局部的计算方法。图 5.6(a)的正向传播的输入变量为 x 和 y，输出 z 为函数 $f(x,y)$的计算结果。图 5.6(b)是反向传播，对于输出求偏导数$\partial L/\partial z$，即相对于 z 的损失函数的梯度，损失函数中对 x 和 y 的梯度可以通过链式法则计算。

2. 反向传播算法——前馈计算

以如图 5.7 所示的神经网络模型为例介绍神经网络的前馈计算和反向传播过程。不失一般性地，这里假定神经网络为一个 4 层的全连接网络，激活函数为 sigmoid 函数，最后一层输出层的激活函数为 softmax 函数。

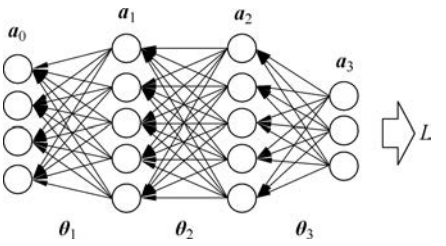

图 5.7　4 层神经网络模型

神经网络的前馈计算指的是由输入 $\boldsymbol{a}_0$ 经各层变换计算得到输出 $\boldsymbol{a}_3$ 的过程。这里以 $\boldsymbol{a}_1$ 到 $\boldsymbol{a}_2$ 为例，展开公式如下：

$$\boldsymbol{a}_2=\sigma(\text{Linear}(\boldsymbol{a}_1;\boldsymbol{\theta}_1))=\sigma(\boldsymbol{W}_1\boldsymbol{a}_1+\boldsymbol{b}_1)$$

其中，$\sigma(x)=1/(1+\mathrm{e}^{-x})$为 sigmoid 函数；Linear 为全连接层变换，是对输入的仿射变换，其参数$\boldsymbol{\theta}_1$包含 $\boldsymbol{W}_1$ 和 $\boldsymbol{b}_1$ 两部分。

其余各层之间也是用相同的变换完成计算，唯一的区别是输出层，其激活函数不是 sigmoid 而是 softmax 函数(用于产生概率预测值)。

3. 反向传播算法——反向传播

反向传播是在完成前馈计算之后计算损失函数相对所有模型参数的偏导数($\partial L/\partial\theta$，其中 θ 为模型的任一参数)的过程。这里同样只对 $\boldsymbol{a}_2$ 到 $\boldsymbol{a}_1$ 的反向传播过程展开讨论。记

$$\boldsymbol{o}_2=\boldsymbol{W}_1\boldsymbol{a}_1+\boldsymbol{b}_1$$

则前馈计算可以表达为 $\boldsymbol{a}_2$ 和 $\boldsymbol{o}_2$ 两部分

$$\boldsymbol{a}_2=\sigma(\boldsymbol{o}_2);\quad \boldsymbol{o}_2=\boldsymbol{W}_1\boldsymbol{a}_1+\boldsymbol{b}_1$$

假定$\partial L\partial \boldsymbol{a}_2$ 已知，则可以推导出当前层参数的偏导数为

$$\frac{\partial L}{\partial W_1}=\frac{\partial L}{\partial \boldsymbol{a}_2}\times\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{o}_1}\times\frac{\partial \boldsymbol{o}_1}{\partial W_1};\frac{\partial L}{\partial \boldsymbol{b}_1}=\frac{\partial L}{\partial \boldsymbol{a}_2}\times\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{o}_1}\times\frac{\partial \boldsymbol{o}_1}{\partial \boldsymbol{b}_1}$$

其中$\partial \boldsymbol{a}_2/\partial \boldsymbol{o}_1$ 就是 sigmoid 函数的偏导数，其解析公式为$\partial \boldsymbol{a}_2/\partial \boldsymbol{o}_1=\boldsymbol{a}_2(1-\boldsymbol{a}_2)$；而$\partial \boldsymbol{o}_1/\partial \boldsymbol{W}_1=\boldsymbol{a}_1$，$\partial \boldsymbol{o}_1/\partial \boldsymbol{b}_1=1$。因此在假定后一层导数$\partial L/\partial \boldsymbol{a}_2$ 已知的情况下，很容易代入上式计算出损失函数相对于当前层参数的导数。

同样地，为了计算前一层参数的导数，需要用到$\partial L/\partial \boldsymbol{a}_1$，也可以很容易地用下式推算得到

$$\frac{\partial L}{\partial \boldsymbol{a}_1}=\frac{\partial L}{\partial \boldsymbol{a}_2}\times\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{o}_1}\times\frac{\partial \boldsymbol{o}_1}{\partial \boldsymbol{a}_1}$$

回顾这里的推导过程，对本层梯度（也就是对本层所有参数的偏导数）的计算是一个递归过程，它假定损失函数对本层输出量 $\boldsymbol{a}_2$ 的偏导数已知，就可以计算出损失函数对本层所有参数的偏导数，并且给出损失函数对本层输入量 $\boldsymbol{a}_1$ 的偏导数，用于前一层的梯度计算。

按照完全相同的原理，反向传播就是从最右端的损失函数开始，逐层地往前计算，递归地计算出损失函数相对每个参数的偏导数。对于其他结构的神经网络，例如卷积神经网络，采用的是完全相同的原理进行反向传播计算，差别只在于局部梯度的计算方法有所不同。从使用者的角度来说，绝大多数算子（如卷积、ReLU、softmax 算子）都可以被认为已经内置在软件框架中，其中反向传播的具体细节都被封装在 nn. Module 中，实际使用时只需要通过 loss. backward()函数调用就可以完成。只有在需要用到自定义算子（例如某种新的激活函数）时，才需要手动实现其局部导数。

注意在上述反向传播的计算中会用到前馈计算的中间值，例如计算$\partial \boldsymbol{a}_2/\partial \boldsymbol{o}_1$ 需要用到 $\boldsymbol{a}_2$ 的值，这不仅是为了简便，也是多数情况下代码中实际的实现方式，可以避免在反向传播中的重复计算，代价是需要用更多的内存保存这些中间值。


微课视频 68

5.2.3 训练参数调整实例分析

模型训练参数批量大小（batch size）和学习率（learning rate）的选择对结果会产生直接的影响。

关于学习率的选择：学习率设置得过高会导致学习不充分，学习率设置得过低会降低学习的效率。一般而言，可以结合不同的批量大小分别从小到大地设置学习率，并记录不同设置的损失值，当发现损失值曲线有明显的持续降低时，该设置就是一个比较合适的学习率及相应的批量大小组合，如图 5.8 所示。

由于每一个小批量数据能提供的梯度信息都是全局梯度的近似，相当于在“真实梯度”的基础上包含一定的噪声。一方面这种噪声会使得优化方向经常偏离全局梯度方向，使得收敛更慢，但另一方面，适量的噪声也有助于帮助优化器跳出局部的鞍点。需要先通过分析

或实验预估每个批量的数据所能产生的噪声大小，并以此决定小批量的大小。如果批量大小设置得过小，噪声相对前进方向较大，容易产生损失值抖动的问题，没有办法收敛到最优值，甚至不会收敛；如果批量大小设置得较大（例如 batch size＞1000，具体数值与问题的性质相关，没有固定的范围），会导致训练次数的不足，使模型收敛的时间成本增加；如果批量大小设置过大，模型的收敛也会受到影响。

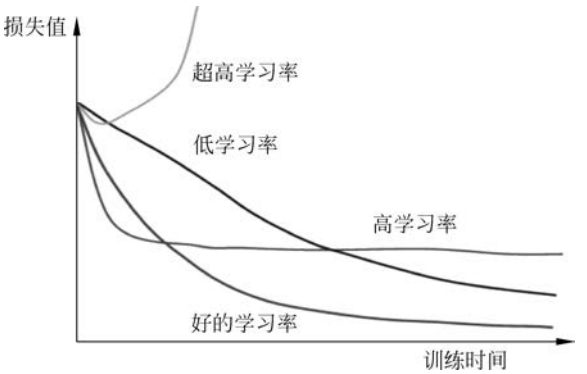

图 5.8　不同学习率下的训练结果

下面给出了 MNIST 手写数字分类代码：

```
import torch
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import torch.nn.init

device = 'cuda' if torch.cuda.is_available() else 'cpu'
# 设置固定 seed(种子)以便于重复实验
torch.manual_seed(777)
if device == 'cuda':
    torch.cuda.manual_seed_all(777)
# 设置主要的超参数
learning_rate = 0.001
training_epochs = 15
batch_size = 100

# 加载 MNIST 数据集
mnist_train = dsets.MNIST(root = 'MNIST_data/',
                          train = True,
                          transform = transforms.ToTensor(),
                          download = True)

mnist_test = dsets.MNIST(root = 'MNIST_data/',
                         train = False,
                         transform = transforms.ToTensor(),
                         download = True)
data_loader = torch.utils.data.DataLoader(dataset = mnist_train,
                                          batch_size = batch_size,
                                          shuffle = True,
```

```
                                          drop_last = True)

# 建立包含 2 层卷积层的 CNN 模型
class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer1 = torch.nn.Sequential(
            torch.nn.Conv2d(1, 32, kernel_size = 3, stride = 1, padding = 1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer2 = torch.nn.Sequential(
            torch.nn.Conv2d(32, 64, kernel_size = 3, stride = 1, padding = 1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size = 2, stride = 2))
        # 用全连接层将 7×7×64 个输入映射到 10 个输出
        self.fc = torch.nn.Linear(7 * 7 * 64, 10, bias = True)
        torch.nn.init.xavier_uniform_(self.fc.weight)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)  # 进入全连接层前将数据转为一维
        out = self.fc(out)
        return out

# 实例化 CNN 模型,并转移到计算设备(GPU)中
model = CNN().to(device)
# 定义损失函数和优化器
# softmax 已经包含在 CE 函数中
criterion = torch.nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)

# 训练模型
total_batch = len(data_loader)
for epoch in range(training_epochs):
    avg_cost = 0

    for X, Y in data_loader:
        X = X.to(device)
        Y = Y.to(device)
        optimizer.zero_grad()
        hypothesis = model(X)
        cost = criterion(hypothesis, Y)
        cost.backward()
        optimizer.step()
        avg_cost += cost / total_batch

    print('[Epoch: {:>4}] cost = {:>.9}'.format(epoch + 1, avg_cost))

# 检查模型在验证集上的精度
with torch.no_grad():
    X_test = mnist_test.test_data.view(len(mnist_test), 1, 28, 28).float().to(device)
```

```
Y_test = mnist_test.test_labels.to(device)
prediction = model(X_test)
correct_prediction = torch.argmax(prediction, 1) == Y_test
accuracy = correct_prediction.float().mean()
print('Accuracy:', accuracy.item())
```

通过调整变量 batch_size(批量大小)和软件包自带的优化工具 Adam 函数的输入参数,可以设置不同的 batch_size 和 lr(学习率)的值,进而可以观察和比较损失值在训练集、验证集和测试集上的不同收敛过程。

MNIST 不同超参数设置的训练结果如表 5.2 所示。

表 5.2　MNIST 不同超参数设置的训练结果

批量大小	5000	2000	1000	500	256	100	50	20	10	5	2 1
总迭代周期数	200	200	200	200	200	200	200	200	200	200	
总迭代次数	1999	4999	9999	19 999	38 999	99 999	199 999	499 999	999 999	1 999 999	
200 周期用时	1	1.068	116	138	1.75	3.016	5.027	8.513	13.773	24.055	
达到 0.99 精度的周期数	—	—	135	78	41	45	24	9	9	—	
达到 0.99 精度的用时	—	—	2.12	1.48	1	1.874	1.7	1.082	1.729	—	不能收敛
最佳训练分数	0.015	0.011	0.01	0.01	0.01	0.009	0.0098	0.0084	0.01	0.032	
最佳训练的周期数	182	170	198	100	93	111	38	49	51	17	
最佳测试成绩	0.014	0.01	0.01	0.01	0.01	0.008	0.0083	0.0088	0.008	0.0262	
最终测试误差(200 周期)	0.0134	0.01	0.01	0.01	0.01	0.009	0.0082	0.0088	0.008	0.0662	

通过对比表 5.2 中数据可知:

(1) 批量大小为 1 和 2 时,模型无法收敛。

(2) 从精度(accuracy)达到 0.99 的周期数来看,批量大小设置得较大时,需要更多的周期才能达到一个比较好的效果。

(3) 从最佳测试成绩中可以看出,批量大小设置得比较小(例如 batch_size<5)时,模型并不能收敛到最优值;实际上,当批量大小设置过大(例如 batch_size>5000)时,也有可能出现模型不能收敛的情况。

5.3　模型超参数优化

从之前的介绍中不难发现超参数的设定会直接影响训练的结果。通常所指的超参数主要包括深度学习模型本身的结构参数(不同于模型内部的参数,而是激活函数的类型、网络层数、每层的神经元数等)和模型训练过程的参数(优化方法、学习率、批量大小、正则化参数)等。在自动驾驶场景中,由于训练样本的获取和模型训练的成本都较高,所以超参数的设定十分重要,合适的超参数能够进一步优化整体的训练,将更多的算力集中在当前有限的

数据集上。本节将重点介绍如何优化超参数以最终获得一个具有良好泛化能力的自动驾驶决策模型。

超参数优化(hyperparameter optimization)是找到一组较优的超参数,使得在该超参数组合下求解 $\min_{\Theta} J(\Theta)$ 得到 Θ 最优的过程。这是一个两层的优化问题:内层是神经网络的训练过程,优化变量是神经网络中的参数(典型数量为百万量级);外层是超参数优化过程,优化变量是各种超参数(典型数量为几个到十几个)。由于评估每一组超参数的组合,都需要完成内层的优化,即进行模型训练求得 Θ,因此超参数优化主要存在下面几方面的困难:

(1) 搜索空间很大:因为很多超参数是离散的变量,例如神经网络的层数、优化算法的选择等,所以超参数优化是一个组合优化问题,搜索空间是指数级的。

(2) 单次评估比较耗时:评估一组超参数配置需要完成一次训练,其时间成本很高,特别是对于大模型来说效率很低。

(3) 可用的优化算法受限:梯度下降算法无法用于带有离散变量的组合优化问题中,其他一些依赖多样本数据点评估的常见算法,例如遗传算法也不适用。


微课视频 69

5.3.1 常见超参数优化方法

1. 网格搜索与随机搜索

网格搜索(grid search)是超参数优化中的常见策略。它通过对每个超参数选择若干数值,然后遍历所有备选超参数组合,从而寻找一组最优的超参数配置。超参数中往往同时有离散变量和连续变量,例如网络层数是离散变量,而学习率是连续变量。如果所有的变量都是离散变量,那么超参数优化最简单的方法是遍历所有的参数组合。当某些超参数可能的取值过多,或者是连续变量时,可以对这些超参数进行采样,例如在可行范围内均匀地取值,或者根据经验人为选择一些典型值。通过降采样,把搜索空间的大小控制在合理的范围内。然后通过网格搜索用遍历的方式对这些离散的组合分别进行模型训练,并评估每个超参数组合所训练出模型在验证集上的性能,最终选取最优的配置。

网格搜索实质上是把超参数空间进行离散化,然后用遍历的方法寻找最优值。它的搜索策略是公平地探索在超参数空间中人为选择的离散点。这种做法虽然在实现上非常简单,但需要遍历的组合数为每个超参数可能取值数的乘积,很容易就超出了允许的范围。例如有 8 个超参数,为简化讨论,这里假设每个超参数都可能取 5 个值,则需要遍历的参数组合数为 $5^8=390\,625$,即使是比较保守的估计也很快超出了可行范围。不同超参数对模型性能的影响有很大差异,例如学习率对模型训练效果的影响通常大于正则化系数,这时候公平搜索的效率就比较低,可能会在低敏感度的区域中耗费过多的时间。

网格搜索的主要优点在于实现非常简单,但由于上述缺点,在实际中的使用率小于另一种策略——随机搜索(random search)。如果说网格搜索是在指定空间内用均匀网格采样,随机搜索则是把均匀网格采样替换为随机采样。具体来说,随机搜索在每个超参数组合中都对每个超参数进行独立的随机取值,通过模型训练选择出一个性能最好的配置。相对网

格搜索，随机搜索的主要优势包括：

（1）随机搜索的搜索次数是直接人为给出的，而网格搜索的搜索次数随超参数的数量呈指数增长。

（2）每一次超参数组合的计算中，对连续变量来说都搜索了不同的数值，而网格搜索则只限定在几个固定的数值中重复计算，因此随机搜索有更大概率以更少的搜索次数找到更优的解。

网格搜索和随机搜索都是比较简单的策略，搜索过程中并没有根据已经搜索到的信息进行反馈，再继续决定如何搜索，而只是简单地根据预设的规则进行搜索。这样的好处是每次搜索是独立的，可以很简单地并行搜索，适合大规模集群的并行化搜索，但不足之处也很明显，搜索效率相对较低。

2. 贝叶斯优化

提高搜索效率的关键在于充分利用已搜索组合给出的信息，以此为指导决定后续如何搜索。在超参数优化问题上，贝叶斯优化（bayesian optimization）是一种常见的自适应优化算法。与神经网络训练所采用的梯度下降算法不同，后者需要依赖梯度决定优化的方向，而贝叶斯优化属于黑盒优化算法，它把被优化函数当成一个黑盒函数，只需要函数给出指定输入的输出即可——对应超参数优化场景，这个黑盒函数的输入是一组超参数组合，输出就是该组合对应的所训练出模型的性能。贝叶斯优化算法的大体流程是：

（1）选择一些初始点$\{x\}$（超参数组合），评估这些点上对应的函数值$\{y\}$（模型性能），构成初始数值集$\mathcal{D}$；

（2）用已有数据集拟合黑盒函数的代理模型 $p(y|\boldsymbol{x},\mathcal{D})$；

（3）将代理模型代入一个预定义的采集函数$\mathcal{S}(x,p(y|\boldsymbol{x},\mathcal{D}))$，并令下一个搜索的点 $x_{i+1}=\mathrm{argmax}_x(\mathcal{S}(x,p(y|\boldsymbol{x},D)))$；

（4）计算下一个搜索点对应的函数值 y_{i+1}，这一步需要训练神经网络，是最耗时的步骤；

（5）将(x_{i+1},y_{i+1})合并到数据集中，返回第（2）步，直至达到预设的迭代次数或性能目标。

贝叶斯优化算法整体的思路是利用已搜索的点中所包含的信息，选择下一个能达到最大收益的点，从而减少优化迭代的次数[第（4）步的次数]。完成这一目标有两个关键：一是黑盒函数的代理模型（surrogate model）；二是采集函数（acquisition function）。

贝叶斯优化中采用的代理模型用于根据已有的数据近似拟合黑盒函数，常用的代理模型包括高斯过程（gaussian process，GP）回归、树状结构 Parzen 估计（tree-structured parzen estimator，TPE）方法、随机森林等。其中高斯过程是目前应用较多的一种，但其明显的缺点是当数据量稍大时，计算复杂度就急剧上升，导致其优化效率下降。

在有了代理模型之后，并不是直接用代理模型求最优值，因为代理模型只是基于已有数据对黑盒函数的近似，求出来的最优值并不准确，还需要采集函数帮助决定下一组评估的点。同样地，采集函数有不同的选择，常用的一种称为期望改善（expectation improvement，

EI)函数。设目前已搜索数据集中的最优值为 y^*，则 EI 函数计算公式为

$$\mathrm{EI}(x,\mathcal{D})=\int_{-\infty}^{\infty}\max(y^*-y,0)P(y\mid x,\mathcal{D})\mathrm{d}y$$

期望改善函数是定义一个样本 x 在当前模型 $P(f(x)|x,H)$ 下，$f(x)$ 超过最好结果 y^* 的期望。

贝叶斯优化中有一个很重要的问题是探索与开发的权衡(exploration-exploitation trade-off)。开发(exploitation)指的是根据已知的信息寻找最有可能是最优点的位置，而探索(exploration)指寻找最不确定的位置，因为这些位置最有可能让目前的估计(代理函数)更准确，从而可以帮助定位到更优的解。开发与探索都有助于找到最优解，但两者天然存在矛盾，因此贝叶斯优化的核心问题就是对两者进行权衡。具体的选择需要根据问题的要求来定，当允许的探索次数较少时(训练非常耗时，或资源有限时)，倾向于开发，反之则倾向于探索。类似的思想在人工智能的其他领域(例如强化学习)中也会遇到。

3. 神经网络架构搜索

前面介绍的超参数优化方法是基于已经确定的网络结构，或者有少量网络结构参数(例如网络层数、卷积核数)的情况，而事实上神经网络的结构本身就是一个对性能影响巨大的超参数。例如在第 4 章介绍 CNN 网络可以有效地处理图像识别任务，但在卷积层内叠加不同尺度的卷积核(Inception 模型)和在卷积层之间增加残差连接(ResNet)都可以有效地提升网络性能。神经网络架构搜索(neural architecture search，NAS)是一种自动实现神经网络架构设计调优的方法。与前述超参数优化方法一样，NAS 的自动化网络搜索需要定义一个搜索空间，然后按一定的策略搜索该空间，找到最(较)优的配置组合。对 NAS 而言，它有 3 个核心要素：搜索空间、搜索策略和性能评估方法。

搜索空间定义的是可行的网络结构类型，一般包括网络的拓扑结构(有多少层、每层多大、层与层之间如何连接)以及层的类型(全连接、卷积、激活函数)等。搜索策略指的是如何迭代优化，可以采用前面介绍的贝叶斯优化，而遗传算法、强化学习及基于梯度的方法(需要将离散优化问题转化为连续优化问题)是常见的方法。由于 NAS 中需要的迭代次数较多，如果每次都进行完整的训练会过于耗时，因此通常需要一些方法加快性能评估，例如用前面若干训练步的性能外推，或者共享一部分相似子网络的参数加速模型的收敛等。


微课视频 70

5.3.2 超参数优化工具

对于一般的小规模超参数优化，通常采用的是专家经验结合简单的网格或随机搜索策略，但对于更大规模的超参数优化问题，需要借助自动化库帮助搜索。目前主流的超参数优化库都集成了贝叶斯优化等算法，并提供了不同的选项构建优化策略。较常用的贝叶斯优化库包括 HyperOpt、BayesianOptimization、Spearmint，以及最近由华为诺亚方舟实验室开源的 HEBO 等。

Auto-PyTorch 是一个针对 PyTorch 框架的超参数优化工具，它的主要特色是结合了

一般的超参数搜索和神经网络架构搜索，大幅降低了超参数优化中所需的手动工作。

安装完 Auto-PyTorch 后，即可用于优化模型的性能。以来源于 Auto-PyTorch 官方示例的代码片段为例，演示了在 FashionMNIST 数据集上对模型超参数优化的过程。在这个例子中，优化之后，可以观察到超参数优化过程中模型性能的不断提升，模型在测试集上的验证精度可以从初始模型的 0.93 逐步提升至 0.99 左右。具体代码如下：

```
import numpy as np
import sklearn.model_selection
import torchvision.datasets

from autoPyTorch.pipeline.image_classification import ImageClassificationPipeline

trainset = torchvision.datasets.FashionMNIST(
    root='../datasets/', train=True, download=True)
data = trainset.data.numpy()
data = np.expand_dims(data, axis=3)
dataset_properties = dict()
# 定义用于图像分类的 pipeline 对象
pipeline = ImageClassificationPipeline(dataset_properties=dataset_properties)

# 拆分训练集和验证集
train_indices, val_indices = sklearn.model_selection.train_test_split(
    list(range(data.shape[0])),
    random_state=1,
    test_size=0.25,
)

pipeline_cs = pipeline.get_hyperparameter_search_space()
print("Pipeline CS:\n", '_' * 40, f"\n{pipeline_cs}")
config = pipeline_cs.sample_configuration()
print("Pipeline Random Config:\n", '_' * 40, f"\n{config}")
pipeline.set_hyperparameters(config)

print("Fitting the pipeline...")
# 调用 pipeline 对象的 fit 方法，完成真正的优化过程
pipeline.fit(X=dict(X_train=data,
             is_small_preprocess=True,
             dataset_properties=dict(mean=np.array([np.mean(data[:, :, :, i]) for i in range(1)]),
                 std=np.array([np.std(data[:, :, :, i]) for i in range(1)]),
                 num_classes=10,
                 num_features=data.shape[1] * data.shape[2],
                 image_height=data.shape[1],
                 image_width=data.shape[2],
                 is_small_preprocess=True),
             train_indices=train_indices,
             val_indices=val_indices,
             )
)

print(pipeline)
```

5.4 训练效率与推理效果

模型的训练需要耗费大量的数据和算力，此时系统更加注重训练的效率问题，超参数优化是提升效率的途径之一，而最大限度有效利用此前训练的积累（模型迁移）也是一条提升效率的途径。

训练结果的优劣最终还是体现在实际应用中模型推理的准确性上，即推理效果是否理想。推理过程是将场景的新数据（非训练数据集或测试数据集数据）输入模型，并将得到的结果用于解决场景的应用问题，如果问题得以顺利解决则说明推理的效果达到预期。

与模型训练可以通过后台服务器进行离线计算不同，模型推理过程通常需要在应用现场进行在线计算，对模型计算的实时性也提出了要求。


微课视频 71

5.4.1 离线计算与在线计算

模型计算可以分为离线计算和在线计算两个模块，表 5.3 从效果和效率指标对模块进行了对比分析。其中效果指标方面：主要关注深度学习模型分类的准确性；效率指标方面：主要关注深度学习模型离线训练的经济性和模型在线推理的实时性和功耗。

表 5.3 模块指标对比

离线/在线计算模块	效果指标	效率指标	算力来源
离线计算模块	深度学习模型分类准确率	使用云服务资源的经济成本（使用的是通用型加速器）	云服务资源
在线计算模块	验证 AI 芯片逻辑运算的正确性	模型在线推理实时性和功耗（使用的是专用型加速器）	自动驾驶车、边缘服务器

离线计算一般指模型的离线训练和超参数调优，一般用于解决模型功能实现的“效果”问题。自动驾驶领域离线计算模块一般用模型分类准确率评估“效果”，即

$$\text{模型分类准确率}=\frac{\text{累计正确自动驾驶决策数量}}{\text{累计自动驾驶决策数量}}$$

离线计算通常部署在云端，基于大规模标注的训练数据，通过 GPU、CPU 等通用型加速器和分布式训练的算法求得“效果”较优的深度神经网络模型。

在实践中，离线计算的“效率”通常要考虑经济成本问题，更关注对云服务计算资源的利用效率。采用迁移学习的方法则可以重复使用在大规模通用数据上训练而成的预训练模型，并将预训练模型在特定实例数据上进行微调以降低离线计算的经济成本。

在线计算一般指模型的在线推理和计算，通常部署在自动驾驶车辆上，或者以车路协同的方式在边缘服务器上进行模型推理，此时更注重计算的“效率”。在这里，效率主要指低延

迟、低功耗、低成本三方面。低延迟可以通过优化模型在线推理的计算时间改进。例如在红绿灯识别场景中(如图 5.9 所示)，红绿灯的状态是实时发生变化的，必须用低延迟保证自动驾驶车对当前红绿灯转换的快速响应。

图 5.9　红绿灯识别场景

在线计算的功耗可用单位功耗下的运算能力进行描述。当前处于研发阶段的自动驾驶原型车都搭载了大量的传感器和高性能的处理器，并使用高精度的神经网络模型，这些都需要消耗大量电能，自动驾驶系统的耗电量高达 4kW，对纯用电池供电的自动驾驶汽车的续航能力形成了巨大挑战。在成本方面，由于自动驾驶技术还未最终成熟，原型车还在使用通用型加速器，也导致了自动驾驶系统硬件成本居高不下。这表明从低延迟、低功耗和低成本三方面来看，在线计算的效率都不尽理想。

5.4.2　模型迁移

微课视频 72

对于深度学习神经网络的训练，“迁移学习”的概念近年来得到了广泛的应用，迁移学习直观上可理解为是老手与新手之间的“知识转移”过程。在 5.1 节的相关介绍中提到神经网络的模型参数需要经过数据训练才能使其有意义，在训练过程中通过大量数据计算和反向传播不断更新模型参数，这些模型参数构成了神经网络中各个连接的权重，记录了对数据学习的结果。如果能将这些权重提取出来，就可以迁移到其他的神经网络模型中，“迁移”了学习的结果，如此一来就不需要从零开始重新训练一个神经网络模型了。本节主要介绍“预训练模型(pretrained model)＋微调(fine tune)”这一迁移学习的范式。

所谓预训练模型是前人为了解决类似问题所创造出来的模型。在解决问题时，不用从零开始训练一个新模型，可以通过选择类似问题训练出的模型作为基础进行更新。例如研发一款图像识别的应用产品，可以花数年时间从零开始构建一个性能优良的图像识别算法，也可以从 Google 在 ImageNet 数据集上训练得到的 Inception 模型(一个预训练模型)起步完成图像识别功能。一个预训练模型可能对应用并不是 100％的准确匹配，但是它可以节省大量设计和训练成本。

在神经网络模型的训练过程中，如果希望无须通过多次正向传播和反向传播的反复迭

代就能找到合适的模型参数(权重),则可以通过使用之前在大数据集上经过训练的预训练模型,这些模型可以直接提供相应的网络结构和权重,为解决目前正在面对的问题提供帮助。这个过程则是“迁移学习”,将预训练的模型“迁移”到目前正在应对的特定问题中。在选择预训练模型的时候需要非常仔细,如果问题与预训练模型训练情景有很大出入,那么模型所得到的预测结果将会不准确。

ImageNet 数据集目前已经被广泛用作训练集,因为它规模足够大(包括 120 万张图片),有助于训练普适模型。ImageNet 的训练目标是将所有的图片正确地划分到 1000 个分类条目下。这 1000 个分类条目基本上都来源于日常生活,例如猫或狗的种类、各种家庭日用品、日常通勤工具等。通过迁移学习,这些预训练的模型对于 ImageNet 数据集以外的图片也表现出了良好的泛化性能。既然预训练模型已经能够达到很好的效果,那么就无须在短时间内去修改过多的模型参数,在迁移学习中使用这些预训练模型时,往往只需进行微调处理就足够了。模型的修改通常会采用比一般训练模型更低的学习率。

当有了可用于“迁移学习”的预训练模型时,可以采用如下方法微调模型:

(1) 特征提取。可以将预训练模型作为特征提取装置来使用,具体的做法是:将输出层去掉,然后将剩下的整个网络作为一个固定的特征提取机应用到新的数据集中。

(2) 采用预训练模型的结构。此时微调模型的具体做法是:保留预训练模型的结构,将原有的权重随机化,然后依据自己的数据集进行训练。

(3) 训练特定层并冻结其他层。这是一种对预训练模型进行部分训练的微调,具体的做法是:将模型起始的一些层的权重保持不变,重新训练后面的层得到新的权重。在微调过程中可以进行多次尝试,以便依据结果找到冻结层和特定层之间的最佳搭配。

如何使用预训练模型,这是由数据集大小以及新旧数据集(要解决的数据集与预训练的数据集)之间数据的相似度所决定的。灵活地使用预训练模型能够最大限度地增强解决问题的能力,可以作为开放性思考的重要参考。


微课视频 73

5.4.3 硬件加速器

为了使人工智能真正应用在实际生活中,就需要解决深度神经网络在训练和推理过程中的效率问题。效率主要由运算速度、功耗以及计算成本等因素决定,这些因素与其计算平台的硬件息息相关。而训练与推理对加速器的需求也稍有不同,前者更注重在强大算力支撑下的大吞吐量(throughput),可以在一定时间内完成尽可能多数据的训练,后者更注重单批数据处理的端到端时延(latency)。这两个指标都直接跟加速器的计算能力相关,但又存在本质区别,由此分别发展出了专门针对训练和推理的加速芯片。

在人工智能领域,计算平台中负责处理大量运算的硬件加速器[14]通常分为通用型加速器和专用型加速器两种。

通用型加速器是面向多种应用场景所设计的通用硬件设备,以 CPU 和 GPU 为代表,通常能够处理和应对多种不同的计算任务。自动驾驶任务是人工智能应用中的一个场景任务,类似的应用场景还有很多,例如刷脸认证支付、智能音箱语音识别等。虽然这些计算任务中的

神经网络模型的结构各不相同，但通用型加速器都能较好地完成对这些模型的训练和推理。

专用型加速器是面向单一应用场景所设计的专用硬件设备，以各种 ASIC 芯片（如 Google 的 TPU）为代表。专用型加速器虽然功能单一，但其在自身支持的计算任务中的效率却远高于通用型加速器。

1. 通用型加速器

通用型加速器的核心部件一般是通用处理器，例如传统的中央处理器（central processing unit，CPU）、图形处理器（graphics processing unit，GPU）等，被广泛用于深度神经网络的离线训练。而在很多深度神经网络的推理场景应用中，通用型加速器却存在实时性差、功耗高、成本高等问题，特别在自动驾驶任务的商用落地场景中这些问题更为突出。例如自动驾驶需要识别道路、行人、红绿灯等状况，如果使用 CPU 作为加速器，实时性差的问题将严重影响车辆自动驾驶的安全性，如果换成 GPU，虽然能显著加速推理过程，但其功耗大的问题对于存储能源有限的车辆而言却是沉重的负担。

GPU 最早出现在 1999 年，由英伟达公司首先提出。作为图形处理器，GPU 能处理绝大部分图形数据运算，其内部有远超 CPU 比例的逻辑运算处理单元，虽然这些处理单元的功能相对单一却数量庞大。这个特点让 GPU 更擅长处理大批量且高度统一的数据，能够实现连续的大规模计算任务，深度神经网络的训练和推理恰好属于此类计算任务。GPU 仍然属于通用型处理器，其任务的实现依赖于装载的程序代码，不同的任务可以通过不同的程序代码实现。为保障大型计算任务的实施，GPU 一般会保留大规模的逻辑处理单元，这是其功耗大的主要原因。

现场可编程门阵列（field programmable gate array，FPGA）[15] 也可以作为一种通用型的加速器，在自动驾驶等场景中多有应用。由于自动驾驶的人工智能模型虽然计算密集，但其运算精度的需求却不高，相对于 GPU 或 CPU 中使用固定的高精度运算结构，FPGA 可以更加灵活地配置运算结构，达到了节省计算成本的目的。而且在固定的计算任务中，FPGA 比 CPU 更快，比 GPU 功耗更低，这些特点也构成了它本身的优势。相对于开发专用 AI 芯片，FPGA 开发周期更短，对于小批量应用而言，其成本也相对更便宜，因此，自动驾驶开发厂商通过 FPGA 尝试不同架构、不同策略的方案，在研发阶段取得了较好的实验结果。然而，一旦自动驾驶技术成熟落地，需要在大批量的车辆上安装应用时，再使用 FPGA 作为加速器就没有优势了。

2. 专用型加速器

当自动驾驶技术成熟落地时，开发专用集成电路（application specific integrated circuit，ASIC）芯片[16] 实现车辆的自动驾驶功能是最终的方案选择。实现自动驾驶功能的深度神经网络模型在训练过程中需要使用大量的算力处理海量的数据，而训练完成的模型在车辆中实现自动驾驶功能时就转入了模型推理过程，模型推理的计算量远小于模型训练所需的计算量。模型推理应用对计算的实时性要求非常高，对能耗也十分敏感，因此当自动驾驶模型需要大规模部署在车辆中时就不能再使用通用型加速器作为载体了，而采用专用型加速器就成

为了最佳的选择。类似情况在目前成熟的图像识别、语音识别等应用场景中已经发生了。

对于投入市场的电子产品，提高其性价比常用的方法就是将产品的核心功能用 ASIC 芯片实现。如果产品的市场投放量足够多，达到数万、数十万量级，开发 ASIC 芯片作为产品的电子核心是最佳方案。为加速深度学习算法的推理运算，目前已经出现了很多物美价廉的专用 AI 芯片（一般是为实现 AI 算法而专门设计出的 ASIC 芯片），这类 AI 芯片体积小，功耗低，成本也不高，尤其适用于图像识别、语音识别等应用场景。

车端的自动驾驶芯片由于巨大的市场潜力和不断成熟的技术，目前处于快速发展的阶段，芯片的发展日新月异。以当前英伟达的旗舰产品 Orin 为例，它可以提供 254 TOPS 的运算能力，已经可以支撑复杂自动驾驶模型的实时推理。自动驾驶模型仍在快速发展，可以预见未来将会出现更大、更复杂，需要更强大算力支撑的模型。


微课视频 74

5.5 开放性思考

1. 智能小车模型的迁移与微调

1）智能小车模型迁移

结合使用智能小车采集的数据，请读者分析下列情况下自动驾驶任务可以采用的模型迁移方法分别是什么？

（1）数据集小，数据相似度高（与预训练模型的训练数据相比而言）。

提示：在这种情况下，因为数据与预训练模型的训练数据相似度很高，因此不需要重新训练模型。只需要将输出层改制成符合问题情境下的结构就可以，使用预处理模型作为特征模式提取器。例如使用在 ImageNet 上训练的模型辨认一组新照片中的猫或狗。在这里，需要被辨认的图片与 ImageNet 库中的图片类似，但是所输出的结果只需包括两项——猫或狗。此时需要做的就是把全连接层和输出层的输出从 1000 个类别改为 2 个类别。

（2）数据集小，数据相似度不高。

提示：在这种情况下，可以冻结预训练模型中的前 k 层中的参数，然后重新训练后面的 $n-k$ 层，当然最后一层也需要根据相应的输出格式进行修改。因为数据的相似度不高，重新训练的过程就变得非常关键。而由于新数据集小，所以要通过冻结预训练模型的前 k 层进行弥补。

（3）数据集大，数据相似度不高。

提示：在这种情况下，如果有一个很大的数据集，神经网络模型的训练会比较高效。然而，因为实际数据与预训练模型的训练数据之间存在很大差异，采用预训练模型的效率不高，此时最好还是将预处理模型中的参数全初始化后在新数据集的基础上从头开始训练。

（4）数据集大，数据相似度高。

提示：这种情况最为理想，采用预训练模型会非常高效。最好的方式是保持预训练模型原有的结构和参数不变，随后在新数据集的基础上进行训练。

2）智能小车模型微调

请读者尝试在 PyTorch 中选择一个在 ImageNet 预训练的深度学习模型，并通过智能小车采集得到的数据在预训练模型上进行微调，根据模型的准确率、离线训练时间等两个指标评估和比较迁移学习与从零开始训练两种不同方法所得到模型的结果。

2. GPU 模型训练加速实验

首先请读者对 GPU 的模型训练环境进行如下配置：

（1）如图 5.10 所示，使用 nvidia-smi 命令查看 GPU 是否可用（或在模型训练时，用此命令观察 GPU 的资源利用率）。

```
xiongyu@ubuntu:~$ watch nvidia-smi
|    0     47741      C   ./mtcnn_c                                   461MiB |
Every 2.0s: nvidia-smi

Fri Jun 29 11:25:43 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 390.48                 Driver Version: 390.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla M40           On   | 00000000:03:00.0 Off |                    0 |
|  0%   57C    P0   217W / 250W |  10806MiB / 11448MiB |     99%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla M40           On   | 00000000:04:00.0 Off |                    0 |
|  0%   58C    P0   200W / 250W |   9992MiB / 11448MiB |     99%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla M40           On   | 00000000:84:00.0 Off |                    0 |
|  0%   56C    P0   222W / 250W |   9955MiB / 11448MiB |     99%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla M40           On   | 00000000:85:00.0 Off |                    0 |
|  0%   57C    P0   203W / 250W |   9960MiB / 11448MiB |     98%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     35346      C   python2                                    110MiB |
|    0     55323      C   python                                   10673MiB |
|    1     55323      C   python                                    9970MiB |
```

图 5.10　在 Linux 系统中查看英伟达 GPU 信息

（2）根据 GPU 版本下载兼容 CUDA 并安装，根据 CUDA 版本下载兼容的 cudnn 版本并安装。

（3）根据 CUDA 版本安装兼容的 PyTorch。

（4）查看 GPU 是否对 PyTorch 可用。

```
import torch
torch.cuda.is_available()
torch.cuda.current_device()
torch.cuda.device(0)

torch.cuda.device_count()
torch.cuda.get_device_name(0)

# 检查 PyTorch 使用的 CUDA 版本
```

```
torch.version.cuda

# 查看 GPU 数量
torch.cuda.device_count()

# 尝试使用第 0 个 GPU,如果获取失败则退回使用 CPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# 或 device = torch.device("cuda:0")
for batch_idx, (img, label) in enumerate(train_loader):
    img = img.to(device)
    label = label.to(device)
```

配置好 GPU 环境后,对于 5.2.3 节中的实例,在代码头部加上如下代码,即可以使用 GPU 进行模型训练,其中“0,1,2,3”分别代表 GPU 对应的 ID。

```
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "0,1,2,3"
```

请读者比较 GPU 加速后的模型训练和使用 CPU 的模型训练的情况,分析训练速度和效率的变化。


微课视频 75

5.6 本章小结

如图 5.11 虚线框所示,本章重点介绍了对自动驾驶算法模型的训练和优化。利用大量数据完成对算法模型的训练是机器学习的关键环节,对于模型的功能实现有重要的影响。随着人工智能技术的发展,所用算法模型的结构正变得越来越复杂,内含的参数越来越多,上亿参数的模型也在不断出现,因此每次训练所需的算力也越来越大。模型训练所消耗的巨大算力和海量数据都将计入研发成本,但并非成本投入越大所得到的结果就越好,这还取决于模型训练时的优化策略以及对优化工具的使用等。

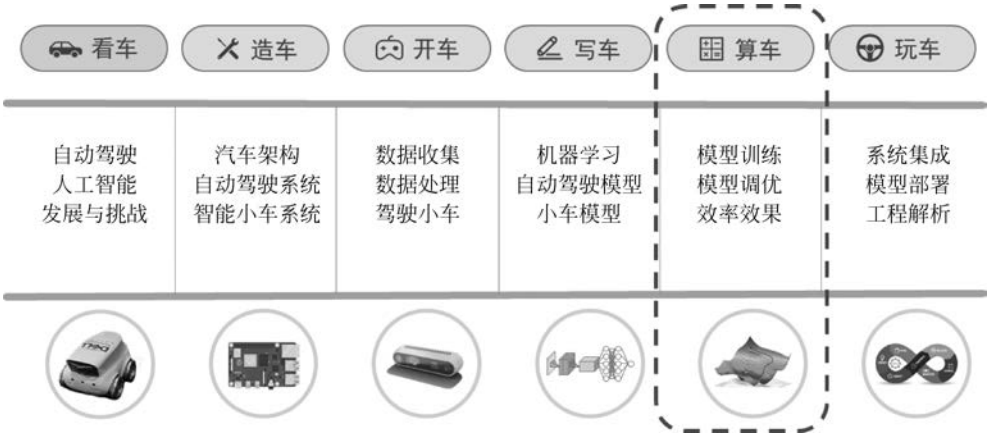

图 5.11 章节编排

本章对自动驾驶算法模型和训练优化进行了相关讨论，其目的是提升自动驾驶系统的“效果”和“效率”。实际上，“效果”和“效率”亦是全书所关注的重点问题之一。为了提升自动驾驶系统的“效果”，在硬件方面可通过使用高清摄像头获取更清晰的道路场景数据（提高采集图像的分辨率），用多目广角摄像头获取更多的道路场景角度，从原始数据上提升对实际自动驾驶场景的表征能力；在软件算法方面，可以通过使用高质量的标记数据、更复杂的深度神经网络结构等方法提高算法的准确性和适用性。

显然，为了提升自动驾驶系统的“效果”所引入的硬件更多、更精密，深度学习自动驾驶模型参数相对规模也会更大，计算成本快速上升，这些对自动驾驶系统的“效率”提出了更高的要求。对模型进行优化是有效降低研发成本的路径之一。有效使用优化工具可以大幅减少训练时间，提高训练成效，这些优化工具不仅仅限于本章所介绍的超参数优化工具或模型迁移等，还包括模型推理时对模型的裁剪工具和加速工具等，这里不再展开介绍，有兴趣的读者可自行查阅相关资料。