INT-NN: 纯 C 实现的整数神经网络库

21307110014-黄悦源　　23302010016-汤腾博　　23300240008-徐兆恺

摘要

本项目提出 **INT-NN——一个用纯 C 实现的轻量级、自包含的整数神经网络训练与推理框架**。与现有方法不同，INT-NN 不依赖显式量化算法或定点数格式，而是直接以整数形式运行。这得益于我们设计的一类适用于整数网络的激活函数（**量化激活**），以及一种新兴的训练方法——**直接反馈对齐（Direct Feedback Alignment, DFA）**。DFA 避免了标准反向传播中常见的整数溢出问题，使各层可独立训练。我们在 MNIST 数据集上对 INT-NN 进行了实证评估，模型准确率达到 97.02%，性能与同样结构的浮点数模型相比仅有 0.68% 的精度损失。INT-NN 无外部依赖，高度可移植，可以很好地兼容低端设备。相关代码已上传至 GitHub[1]。

第 1 章　引言

近年来，深度学习发展迅速，广泛应用于学术与工业界。然而，现代深度神经网络（DNN）模型规模庞大、计算密集，与资源受限的低端设备（如物联网终端、嵌入式系统）需求形成强烈反差。这些设备通常缺乏高性能硬件（如 GPU、大内存），甚至可能不具备浮点运算单元（FPU），难以高效执行模型训练或推理。

TinyML 是专注于低端设备机器学习的研究方向。受限于资源，这类设备通常只能运行简单模型，复杂推理需依赖云端，带来高功耗、通信延迟和额外成本。训练亦面临类似问题：数据需上传至服务器，不仅增加开销，还有暴露隐私的风险。因此，若能在低端设备上直接运行并训练复杂模型，将显著降低成本与能耗，并提升隐私保障，具有重要意义[1]。

在低端设备上运行和训练 DNN 模型面临两大主要挑战：速度与模型大小。由于这类设备计算能力有限、存储紧张，更小、更快的模型尤为关键。量化（Quantization）是解决这一问题的主流方法，其核心是将模型参数由浮点数压缩为更短的整数。尽管精度略有损失，实践表明这对模型性能影响有限[2]。量化不仅显著减小模型体积，还能加快推理速度，因为在多数低端设备上，整数运算远快于浮点运算。许多微控制器（如 Arduino Uno、Zero、Due）不具备浮点单元（FPU），浮点运算需靠软件模拟，性能损耗严重。Arduino 官方网站明确指出，“浮点运算远比整数运算慢，应尽量避免”[3]。

为应对低端设备对本地训练与推理的迫切需求，以及现有工具栈的局限，我们提出了 INT-NN——一种纯整数神经网络算法与纯 C 语言实现的概念验证框架。INT-NN 极为轻量，完全采用 C 编写，无任何外部依赖，具备良好的兼容性与可移植性。作为算法，INT-NN 全部运算基于标准整数类型，无需浮点或定制定点格式，量化以自然方式融入训练流程。其核心在于使用直接反馈对齐（Direct Feedback Alignment, DFA）替代传统反向传播（BP），并结

[1] https://github.com/CanderFlower/int-nn

合我们设计的整数激活函数“量化激活”。DFA 以随机反馈权重训练各层，使每层更新可独立进行，从而规避纯整数算术下 BP 的溢出问题，并带来更高的并行性与简化的实现路径。

简而言之，我们的工作如下：

- 提出 INT-NN，一种在训练和推理中仅使用标准整数类型的神经网络算法；
- 将直接反馈对齐（DFA）融入纯整数计算，并通过定制的量化激活，解决传统反向传播在整数环境下的溢出问题，显著提升训练稳定性；
- 实现一个用纯 C 语言编写、无外部依赖的轻量级整数神经网络框架，适配低端设备。

第 2 章 相关工作

模型量化是 TinyML 领域的主流实践[1]。现有关于模型量化的研究大致可分为两类：仅推理量化和全流程量化。

2.1 推理阶段的量化

第一类研究主要集中于在推理阶段对模型进行量化。这类方法在训练和权重更新时仍使用高精度浮点数，待训练完成后将浮点权重进行量化用于推理。BinaryConnect[4]、Binaryized-NN[5]和 XNOR-Net[6]将权重极致地二值化为 1 比特，而 TTQ[7]则将权重量化为 2 比特。由于 1 或 2 比特对于模型训练而言精度过低，这些方法在训练和权重更新过程中仍使用高精度浮点实数，待训练完成后再将浮点权重进行量化以用于推理。

该类方法可在高性能服务器上利用海量数据进行训练和量化，但部署至低端设备后无法本地微调或更新。模型改动需在服务器重训并重新下发，训练数据也需上传，带来隐私泄露与通信开销。因此，我们仅考虑在本地完成模型训练与推理。

2.2 训练和推理阶段的量化

第二类研究致力于在训练和推理阶段均进行量化。这种方法在继承第一类优势的基础上，进一步实现了基于整数运算的训练，这对于许多低端设备而言，比浮点运算更快且更兼容。DoReFa-Net[8]在内部维护浮点权重以进行更新，随后将其量化为定点数。FxpNet[9]进一步尝试仅使用定点数，并引入了整数批归一化（integer batch normalization）和定点 ADAM 优化器。然而，它未能完全在其算法中移除浮点实数。WAGE[10]和 NITI[11]也追求类似的目标。

需要指出的是，这类研究项目为了避免浮点实数，通常要么使用复杂的量化算法，要么采用定制的定点表示。例如，WAGE[10]专注于通过映射、位移和随机舍入等方法来量化参数。FxpNet[9]和 NITI[11]则使用两个不同的整数 M（尾数）和 E（指数）来表示一个实数 $M{\cdot}2^E$。这种方法本质上与使用两个整数来模拟 IEEE 754 浮点标准[12]中的一个浮点实数异曲同工。这种以及其他类型的定制定点格式尚未获得普遍认可，因此存在兼容性和可移植性较低的问题。与之相比，INT-NN 直接在整数上进行操作，并且实现了自然的量化过程，无需显式的量化算法。

此外，大多数纯整数训练方法都面临溢出问题。FxpNet[9]和 NITI[11]均提到了这一点。我们分析发现，反向传播（Backpropagation, BP）[13]作为主流训练算法，由于权重更新上界随层数呈指数增长，在纯整数算术下会不可避免地溢出。INT-NN 则借助直接反馈对齐（Direct

Feedback Alignment, DFA)[14]算法和专门设计的量化激活，使得每一层的权重更新互相独立，规避了纯整数下的溢出问题，也使训练过程更简单、更易并行。

从工程实现来看，现有大多数相关工作的开源实现采用 Python 编写，并依赖 TensorFlow[15]、PyTorch[16] 等主流深度学习库。但低端设备通常仅支持 C/C++ 或汇编，直接运行 Python 代码较为困难，而深度学习库的依赖也显著增加了算法移植的复杂度。INT-NN 则完全基于 C 实现，不依赖任何外部库，以最大化平台兼容性。

第 3 章　预备知识

本节将介绍深度神经网络训练中两种权重更新算法：反向传播（BP）和直接反馈对齐（DFA），并比较它们在纯整数下的表现。

3.1　反向传播

反向传播（BP）是训练深度神经网络的事实标准算法。BP 通过一种类似动态规划（Dynamic Programming）的方式，从输出层把误差往前传，迭代地计算每层参数对误差的梯度，进而应用梯度下降优化模型。

接下来我们从数学角度说明这一过程。对于一个输入的行向量 $\mathbf{x} \in \mathbb{R}^{1\times d_0}$，一个 n 层的全连接神经网络对其的处理可以表示为

$$\mathbf{z}^{[1]} = \mathbf{x}W^{[1]} + \mathbf{b}^{[1]} \tag{3.1}$$

$$\mathbf{z}^{[k]} = \mathbf{a}^{[k-1]}W^{[k]} + \mathbf{b}^{[k]} \text{ (for } 2 \le k \le n) \tag{3.2}$$

$$\mathbf{a}^{[k]} = \sigma(\mathbf{z}^{[k]}) \tag{3.3}$$

$$\hat{\mathbf{y}} = \mathbf{a}^{[n]} \tag{3.4}$$

这个过程也被称为前向传播。其中，$W^{[k]} \in \mathbb{R}^{d_{k-1}\times d_k}$ 是第 k 层的权重矩阵，$\mathbf{b}^{[k]}$, $\mathbf{z}^{[k]}$, $\mathbf{a}^{[k]} \in \mathbb{R}^{1\times d_k}$ 分别是第 k 层的偏置，线性输出和激活，$\sigma^{[k]}(\cdot)$ 是第 k 层的非线性激活函数，$\hat{\mathbf{y}} \in \mathbb{R}^{1\times d_n}$ 是网络输出，通常是对一个值 $\mathbf{y}$ 的预测。根据输出我们可以通过损失函数 $L : \mathbb{R}^{1\times d_n} \to \mathbb{R}$ 计算标量损失 $\mathcal{L}$，表示预测与实际的偏离程度。特别地，当 $d_n = 1$ 即 $\hat{\mathbf{y}} \in \mathbb{R}$ 时，可以应用 L2 损失得到 $\mathcal{L} = (\hat{\mathbf{y}} - \mathbf{y})^2/2$。

接下来考虑反向传播。定义第 k 层的误差信号 $\delta^{[k]} \in \mathbb{R}^{1\times d_k}$：

$$\delta^{[k]} := \frac{\partial J}{\partial \mathbf{z}^{[k]}} \tag{3.5}$$

反向过程从后往前逐层计算误差信号。假设我们使用 L2 损失，有

$$\delta^{[n]} = (\hat{\mathbf{y}} - \mathbf{y}) \odot \sigma'^{[n]}(\mathbf{z}^{[n]}) \tag{3.6}$$

$$\delta^{[k]} = \delta^{[k+1]}W^{[k+1]^T} \odot \sigma'^{[k]}(\mathbf{z}^{[k]}) \text{ (for } 1 \le k \le n-1) \tag{3.7}$$

其中 $\odot$ 表示逐元素乘（Hadamard 积），$\sigma'(\cdot)$ 是激活函数的导数。有了这些误差信号，我们可以快速求出各层参数的梯度

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[k]}} = \mathbf{a}^{[k-1]^T} \delta^{[k]} \tag{3.8}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[k]}} = \delta^{[k]} \tag{3.9}$$

从而应用梯度下降，对参数进行更新优化。

当把 BP 应用在整数网络上时，上溢（Overflow）将极易发生。由公式 (3.7)，delta 的计算过程涉及一个递归的关系。假设我们需要用 e 比特表示 $\delta^{[n]}$，用 w 比特表示一个权重，也就是说它们值的上界分别为 $O(2^e)$ 和 $O(2^w)$。那么对于 $\delta^{[n-1]}$，$\delta^{[n-2]}$，一直到 $\delta^{[n-k]}$，它们值的上界会成为 $O(2^{e+w})$, $O(2^{e+2w})$, ..., $O(2^{e+kw})$，呈指数增长，最终上溢。比如当 $w = 8$（即 `int8`）时，隐藏层数一旦高于 4 层，误差信号的上界就会超过 32，对于 `int32` 而言即发生了上溢。实践中上溢的确经常发生在整数 BP 中，这是传统 BP 在纯整数环境下的固有缺陷。

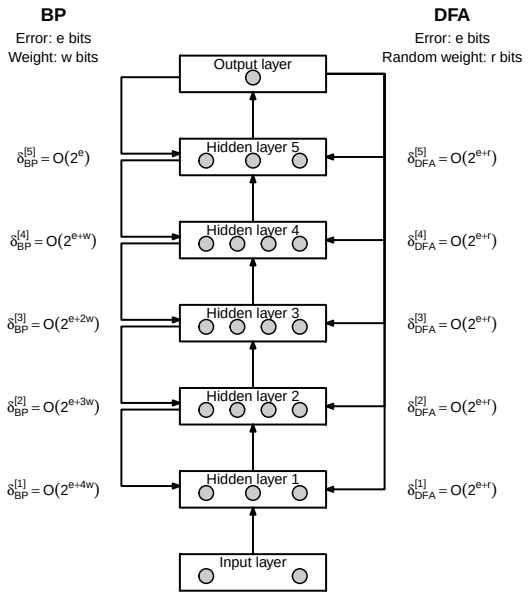


图 3-1 BP 和 DFA 的误差信号值域对比。

3.2 直接反馈对齐

直接反馈对齐（DFA）[14]是一种近年来提出的，旨在解决反向传播算法中误差传播复杂性的新型训练范式，其动机是生物可行性。与 BP 不同，DFA 用随机固定矩阵将输出误差直接反馈到各个隐藏层，替代误差梯度的逐层反向传播。

DFA 的前向过程与 BP 相同，这里不再赘述。DFA 反向过程的核心在于引入一个随机生成的反馈矩阵 $\mathbf{B}^{[k]}$，将输出层的误差信号（这里假设仍用 L2 误差，所以仍然是 $\hat{\mathbf{y}} - \mathbf{y}$）直接映射回每个隐藏层。第 k 层的近似误差信号 $\hat{\delta}^{[k]}$ 可以表示为

$$\hat{\delta}^{[k]} = ((\hat{\mathbf{y}} - \mathbf{y})\mathbf{B}^{[k]}) \odot \sigma'(\mathbf{z}^{[k]}) \text{ (for } 1 \leq k \leq n) \tag{3.10}$$

有了这些近似误差信号，我们可以近似求出每层参数对误差的梯度（虽然本质上不是梯度，但可以这样理解）

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[k]}} \approx \mathbf{a}^{[k-1]^T} \hat{\delta}^{[k]} \tag{3.11}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{[k]}} \approx \hat{\delta}^{[k]} \tag{3.12}$$

然后用类似梯度下降的方式更新参数。一些工作从理论角度讨论了该方法的收敛性[14] [17]。

注意到整个 DFA 优化过程中各个参数之间都是各自独立的，这就提供了并行化的可能，同时也保证了值域的稳定性。将公式 (3.10) 与公式 (3.7) 相比较，可以发现 DFA 的 delta 计算中不再含有递归关系，其值域不会累积。假设我们需要用 e 比特表示 $\hat{\mathbf{y}} - \mathbf{y}$，用 r 比特表示一个随机权重，那么对于任意的 k，$\delta^{[k]}$ 值的上界都是 $O(2^{e+r})$，保持稳定。图 3-1展示了 BP 与 DFA 两者的对比。

第 4 章　方法

本节我们主要从工程角度介绍我们的方法。

4.1　算法

我们用 DNN 作为模型，均方误差作为损失函数（即 L2 损失），DFA 用于参数优化。其中 DNN 的激活函数使用我们自定义的量化激活。

4.1.1　量化激活

常用的激活函数，如 Logistic 函数（Sigmoid，$\sigma(x) = e^x/(e^x+1)$）、双曲正切（Tanh，$\tanh(x) = (e^{2x}-1)/(e^{2x}+1)$）和修正线性单元（ReLU，$f(x) = \max(0, x)$）在纯整数环境下应用受限。前两者由于非线性和过小的值域，需要被近似并重新缩放以适应纯整数；ReLU 则需要加以修改，以起到限制值域的作用。

我们由此设计了一系列专为纯整数 DNN 设计的激活函数，统称为量化激活（Quantized Activation）。它们包括 Q-Sigmoid、Q-Tanh 和 Q-ReLU。这些函数接收整数输入并生成范围在 `int8` 范围内的输出值，以确保值域的一致性（因为线性层的矩阵乘法涉及累加，也会造成值域的增长），避免了上溢或下溢，也无须引入复杂的浮点运算。

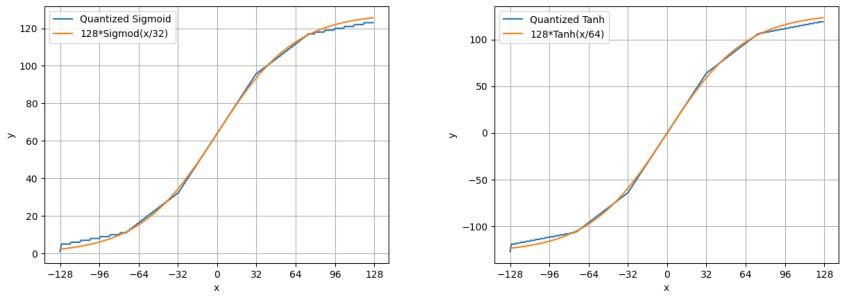
受 Flex-SFU[18]启发，Q-Sigmoid 和 Q-Tanh 被设计为对应主流激活函数（经过缩放）的分段线性近似。Q-Sigmoid 的值域被设计为 $[1, 127]$，以近似 128 Sigmoid$(x/32)$（32 在这里是一个经验性的常数）。其公式为

$$\text{Q-Sigmoid}(x) = \begin{cases} 1 & (x \le -128) \\ x/8 + 20 & (-127 \le x < -74) \\ x/2 + 48 & (-74 \le x < -31) \\ x + 64 & (-31 \le x < 32) \\ x/2 + 80 & (32 \le x < 75) \\ x/8 + 108 & (75 \le x < 128) \\ 127 & (128 \le x) \end{cases} \tag{4.13}$$

Q-Tanh 的值域被设计为 $[-127, 127]$，近似 $128\tanh(x/64)$（64 同样是一个经验性的常数）。其公式为

$$\text{Q-Tanh}(x) = \begin{cases} -127 & (x \le -128) \\ x/4 - 88 & (-127 \le x < -74) \\ x - 32 & (-74 \le x < -31) \\ 2x & (-31 \le x < 32) \\ x + 32 & (32 \le x < 75) \\ x/4 + 88 & (75 \le x < 128) \\ 127 & (128 \le x) \end{cases} \tag{4.14}$$

如图 4-2a和图 4-2b所示，Q-Sigmoid 和 Q-Tanh 很好地近似了它们经过缩放的对应原函数。



(a) Q-Sigmoid 与经过缩放的 Sigmoid 对比。 (b) Q-Tanh 与经过缩放的 Tanh 对比。

图 4-2 量化激活函数与相应浮点函数的对比。

Q-ReLU 是 ReLU6[19]在 INT-NN 中的纯整数版本，我们将 ReLU 的上限设为 127，即 `int8` 的最大值。其公式为

$$textQ - ReLU(x) = \min(127, \max(0, x)) \tag{4.15}$$

在 DFA 训练中，我们发现 Q-Tanh 通常比 Q-Sigmoid 或 Q-ReLU 表现更好，这与原始 DFA 论文[14]的观察相符。

4.1.2 损失函数

我们使用均方误差的动机仍是纯整数运算：虽然多分类任务的普遍选择是 Softmax 加交叉熵误差，纯整数难以完成 $\exp(x)$ 的精确计算。而实践中我们并没有显式地求平均，而是直接使用了平方的加和，与均方误差有一个常数上的差别。

4.1.3 参数更新

DFA 中也会涉及学习率。为了表示小于 1 的学习率，我们反过来使用整数表示学习率的倒数 lr^{-1}，通过 $\hat{\nabla}W/\text{lr}^{-1}$（/ 是 C 语言默认的整数除法）来代替 $\hat{\nabla}W \times \text{lr}$ 计算参数的更新量。

4.2 框架

我们使用 C 语言实现了矩阵操作，工具，layer 和各种函数类，并且不依赖任何外部库。在 `intnn_actv.c` 中实现了 3 种量化激活；在 `intnn_fc_layer.c` 中实现了神经网络组件的编写，包括正向传播和反向传播，神经网络构建和释放；在 `intnn_loader.c` 中实现了各式文件读入、数据加载等操作，本次主要聚焦于从 mnist 数据集中读取数据；`intnn_loss.c` 中实现了损失函数；`intnn_mat.c` 实现了矩阵的基础运算和操作；`intnn_tools.c` 涵盖了本项目使用到的一些工具函数，包括边界检测，类型转换，整数开方等。

由于 C 是一个相对底层的语言，也没有面向对象支持，我们需要手动进行内存管理，显式地分配和释放资源。每个矩阵由一个 `struct intnn_mat` 表示，其中包括行数、列数以及一个指向数据的二维指针。需要创建矩阵时调用 `intnn_create_mat` 函数，函数中用 `malloc` 为整个矩阵和每个行分配空间，返回一个指向创建好的矩阵结构体的指针。使用完成后需要调用 `intnn_free_mat` 逐行释放空间，最后释放整个结构体。其他数据结构也以相同的方式管理。

我们用 `intnn_fc_layer` 来表示 DNN 中的一层，包括线性层参数、激活函数和用于 DFA 的随机权重矩阵，以及前向和后向指针，分别指向网络中的前一层和后一层——这样整个 DNN 可以视为一个双向链表。构建 DNN 时，我们需要创建需要的各个层，然后设置好它们的前后向指针，以便在前向和反向过程中能正确地遍历网络。

第 5 章 实验

我们用 INT-NN 框架实现了一个 3 层的小型 DNN，每层节点数分别为 784-100-50-10，使用 Q-Tanh 作为激活函数，平方和误差作为损失函数，用 DFA 进行参数优化。我们在 MNIST 上对方法的可行性和有效性进行了验证，并与 Pytorch 实现的基准模型在准确率和性能上进行对比。接下来我们会在 Section 5.1介绍相关设定，Section 5.2呈现实验结果。

5.1 实验设置

运行环境：我们在 Windows 和 MacOS 上对框架进行验证，这里主要介绍后者，也就是用 CPU 运行我们的框架，用 MPS（Metal Performance Shaders）运行 pytorch 实现的基准模

型。使用 Cmake 编译项目，开启-O3 编译优化选项。我们发现虽然指定了随机数种子，不同平台的运行结果仍然会有细微区别。

超参数：使用 `epochs = 20, batch_size = 20, lr = 1/1000`。

数据集：我们使用 MNIST 作为数据集。MNIST 由手写数字的灰度图像组成，包含 60000 张图像的训练集和 10000 张图像的测试集，在 TinyML 领域被广泛用于验证新方法。

基准模型：我们使用 Pytorch 定义了一个具有相同结构的 DNN，分别用 BP 和 DFA 作为优化方法，并相应调整了学习率。值得一提的是，使用均方误差在 BP 下难以收敛到较好效果，因此对于 BP 的基准模型我们改用了更主流的交叉熵误差，并把最后一层的激活适配为 Softmax。

5.2 实验结果

实验证明我们的方法不仅在准确率上与浮点的基准模型几乎没有区别（表 5-1），还在性能上表现出了显著的优势（表 5-2）。实验结果还表明 INT-NN 与同样结构的浮点模型相比收敛速度更快，在第 9 个 epoch 就达到了最佳的测试准确率，显著优于 BP 基准（第 31 个 epoch）和 DFA 基准（第 47 个 epoch）。与每个 epoch 的运行时间优势结合，INT-NN 可以在训练阶段实现大约 10 倍的提速。

表 5-1 MNIST 上各模型的准确率对比。

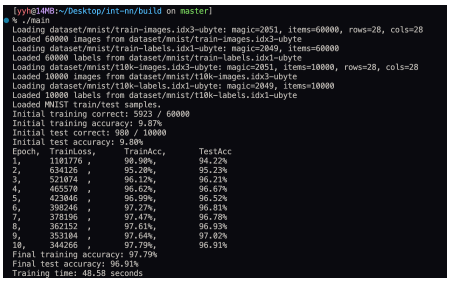
Methods	Train Accuracy	Test Accuracy	Reached Epoch
Baseline (BP)	99.17%	97.70%	31th
Baseline (DFA)	97.50%	96.30%	47th
INT-NN	97.64%	97.02%	9th

表 5-2 MNIST 上各模型的性能对比。

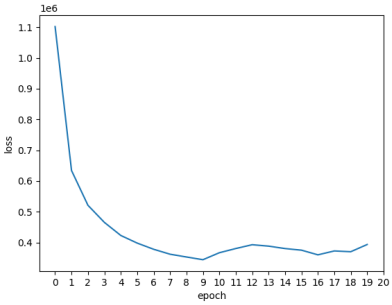
Methods	#Epoch	Training Time (Seconds)	Time Per Epoch
Baseline (BP)	40	367.99	9.20
Baseline (DFA)	50	496.67	9.93
INT-NN	10	48.58	4.86

有趣的是，DFA 的基准模型与 INT-NN 除数据类型外几乎没有区别，却在收敛速度上显著落后于 INT-NN（我们已尝试多种学习率）。这或许是因为，在分类任务中使用均方误差时，整数网络中经过放大后的误差信号仍能保持一定的明确性，而在浮点网络中则难以清晰指引优化方向。通过这一点，我们可以管中窥豹地看出整数网络的独特性。

在 MacOS 上编译并运行`./main`的结果如图 5-3a所示。训练过程中误差变化趋势如图 5-3b所示。



(a) INT-NN 训练运行结果截图。



(b) INT-NN 训练过程中误差变化曲线。

第 6 章 局限

我们目前只实现了较为简单的网络结构，应用场景比较有限。未来可以扩展更多复杂的网络结构，如卷积神经网络（CNN），循环神经网络（RNN）等。此外我们没有探索近似其他损失函数（如交叉熵）的方法，如果能在整数运算下实现类似损失，或许可以在分类任务上取得更好的表现。

第 7 章 结论

本项目开发了 INT-NN，一个纯 C 语言的整数神经网络训练与推理框架，为低端设备上的深度学习应用提供了高效可靠的解决方案。我们创新性地引入了直接反馈对齐（DFA）算法并配合量化激活，成功解决了传统 BP 算法在纯整数环境下固有的梯度溢出难题，显著提高了训练的稳定性和可行性。作为一个轻量级、无需外部库的纯 C 框架，INT-NN 确保了在微控制器等低端设备上的极致兼容性与可移植性。通过赋予设备端本地训练和推理能力，INT-NN 有效克服了数据隐私、网络依赖和额外成本等关键障碍，为资源受限的边缘设备提供了自主学习和适应环境的强大潜能。

参考文献

[1] LIN J, ZHU L, CHEN W M, et al. Tiny machine learning: Progress and futures [feature] [J/OL]. IEEE Circuits and Systems Magazine, 2023, 23(3): 8–34. http://dx.doi.org/10.1109/MCAS.2023.3302182. DOI: 10.1109/mcas.2023.3302182.

[2] LANG J, GUO Z, HUANG S. A comprehensive study on quantization techniques for large language models[A/OL]. 2024. arXiv: 2411.02530. https://arxiv.org/abs/2411.02530.

[3] Arduino Documentation Team. float - arduino documentation[EB/OL]. 2025. https://docs.arduino.cc/language-reference/en/variables/data-types/float/.

[4] COURBARIAUX M, BENGIO Y, DAVID J P. Binaryconnect: Training deep neural networks with binary weights during propagations[A/OL]. 2016. arXiv: 1511.00363. https://arxiv.org/abs/1511.00363.

[5] COURBARIAUX M, HUBARA I, SOUDRY D, et al. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1[A/OL]. 2016. arXiv: 1602.02830. https://arxiv.org/abs/1602.02830.

[6] RASTEGARI M, ORDONEZ V, REDMON J, et al. Xnor-net: Imagenet classification using binary convolutional neural networks[A/OL]. 2016. arXiv: 1603.05279. https://arxiv.org/abs/1603.05279.

[7] ZHU C, HAN S, MAO H, et al. Trained ternary quantization[A/OL]. 2017. arXiv: 1612.01064. https://arxiv.org/abs/1612.01064.

[8] ZHOU S, WU Y, NI Z, et al. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients[A/OL]. 2018. arXiv: 1606.06160. https://arxiv.org/abs/1606.06160.

[9] CHEN X, HU X, ZHOU H, et al. Fxpnet: Training a deep convolutional neural network in fixed-point representation[EB/OL]. 2017: 2494-2501. DOI: 10.1109/IJCNN.2017.7966159.

[10] WU S, LI G, CHEN F, et al. Training and inference with integers in deep neural networks [A/OL]. 2018. arXiv: 1802.04680. https://arxiv.org/abs/1802.04680.

[11] WANG M, RASOULINEZHAD S, LEONG P H W, et al. Niti: Training integer neural networks using integer-only arithmetic[A/OL]. 2022. arXiv: 2009.13108. https://arxiv.org/abs/2009.13108.

[12] Ieee standard for binary floating-point arithmetic[J/OL]. ANSI/IEEE Std 754-1985, 1985: 1-20. DOI: 10.1109/IEEESTD.1985.82928.

[13] RUMELHART D E, HINTON G E, WILLIAMS R J. Learning representations by back-propagating errors[J/OL]. Nature, 1986, 323(6088): 533-536. https://doi.org/10.1038/323533a0.

[14] NøKLAND A. Direct feedback alignment provides learning in deep neural networks[A/OL]. 2016. arXiv: 1609.01596. https://arxiv.org/abs/1609.01596.

[15] ABADI M, AGARWAL A, BARHAM P, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems[A/OL]. 2016. arXiv: 1603.04467. https://arxiv.org/abs/1603.04467.

[16] PASZKE A, GROSS S, MASSA F, et al. Pytorch: An imperative style, high-performance deep learning library[A/OL]. 2019. arXiv: 1912.01703. https://arxiv.org/abs/1912.01703.

[17] REFINETTI M, D'ASCOLI S, OHANA R, et al. Align, then memorise: the dynamics of learning with feedback alignment[A/OL]. 2021. arXiv: 2011.12428. https://arxiv.org/abs/2011.12428.

[18] REGGIANI E, ANDRI R, CAVIGELLI L. Flex-sfu: Accelerating dnn activation functions by non-uniform piecewise approximation[A/OL]. 2023. arXiv: 2305.04546. https://arxiv.org/abs/2305.04546.

[19] KRIZHEVSKY A. Convolutional deep belief networks on cifar-10[Z]. 2012.