Application Note

神经网络处理单元 (NPU) 指南

Pranav Siddappa, Nima Eskandari, Nikhil Dasan, Tushar Sharma, Adithya Thonse

摘要

本神经网络处理单元指南提供了在 F28P55x NPU 上部署机器学习解决方案的综合框架，专为汽车和工业应用而设计。借助该片上硬件加速器，C2000™Ware 客户可实现预测性维护、异常检测、传感器融合和先进控制系统的实时推理，同时保持在这些领域中至关重要的确定性性能。本指南以正弦函数逼近为实用案例，引导工程师完成从架构设计到硬件验证的全流程，重点介绍 NPU 在内存和计算受限的情况下的功能。本文档阐述了有效使用 NPU 所必需的量化技术、基于 TI 工具链的编译流程，以及在 CCS 项目中的集成策略。汽车和工业领域客户将获得开发高效嵌入式 ML 应用所需的实践知识，从而满足特殊环境下对实时性、功耗及可靠性的严苛要求。

内容

商标

1 简介

1.1 NPU 定义和用途

F28P55x 神经处理单元 (NPU) 是一款集成在 TI C2000 微控制器架构中的专用硬件加速器，专为高效执行神经网络计算而设计。这款专用定制芯片支持在嵌入式器件上直接完成机器学习推理，无需依赖外部处理器或云端连接。作为 C2000 生态系统的核心组成部分，NPU 与主 CPU、模拟前端及控制外设协同工作，为实时控制系统提供高级智能能力。这款 NPU 体现了 TI 的承诺，即在确定性性能至关重要的资源受限环境中，提供先进的机器学习能力。

1.2 关键功能

F28P55x NPU 具备多项关键能力，可显著增强 C2000 应用的性能：

- *加速神经网络推理：*通过硬件优化执行神经网络运算，其性能显著优于主 CPU 上的软件实现，根据模型架构不同，典型加速比最高可达 70 倍。
- *基于整数的计算：*专为高效定点算术运算设计，可在嵌入式资源受限的环境下，实现功耗优化的推理处理。
- *实时处理：*具备确定性执行能力，可满足汽车与工业领域控制系统必需的可预测时序要求。
- *外设集成：*可与 ADC、DAC 及其他 C2000 外设无缝协同工作，实现完整的信号处理与控制流程。
- *并行运行：*能够在主 CPU 处理其他任务的同时，独立执行神经网络计算，从而最大限度地提高系统吞吐量。
- *汽车/工业场景聚焦：*专为满足这些高要求领域所需的严苛可靠性、工作温度范围和长期供货能力而设计。

1.3 技术限制

F28P55x NPU 性能优异，但存在以下影响应用设计的约束：

- *架构限制：*与 LSTM 或 Transformer 等复杂架构相比，具有 ReLu 激活的 CNN 和 MLP 等神经网络拓扑受到的支持度更高。
- *精度权衡：*与浮点实现相比，NPU 执行所需的量化会引入精度损耗，需通过精细训练策略维持精度。
- *开发工作流程复杂性：*与标准微控制器编程相比，模型编译和部署的专用工具链要求会增加额外的开发步骤。

这些功能和限制共同界定了汽车和工业嵌入式系统中 F28P55x NPU 的实际应用空间，其中平衡计算能力和资源限制对于成功实施至关重要。

2 开发流程概述

F28P55x NPU 应用的开发工作流程遵循一个结构化流程，该流程将机器学习模型开发与嵌入式系统部署连接起来。本节简要概述了完整的开发周期，后续章节将对其进行更为详尽的深入探讨。

2.1 模型开发阶段

NPU 开发流程从专为嵌入式部署优化的模型设计与训练开始：

- *模型架构设计：*构建平衡应用需求与 NPU 硬件约束的神经网络架构，通常倾向于采用更小的网络和优化的层类型。
- *数据集准备：*整理覆盖部署预期全工作范围的代表性训练数据，特别注意采用与量化兼容的输入归一化策略。
- *量化感知训练：*实施在训练过程中纳入量化效应的训练流程，使模型能够学习到在 NPU 纯整数运算约束下仍表现良好的参数。
- *模型验证：*采用与目标应用相关的指标验证模型性能，评估量化环境下的精度与计算效率。

2.2 模型编译阶段

训练完毕后，必须将模型转换为与 F28P55x NPU 兼容的格式：

- *ONNX 导出：*将训练完成的模型转换为开放神经网络交换 (ONNX) 格式，该格式是编译工具链的互换标准。
- *TI NPU 编译器配置：*通过配置文件定义编译参数，这些配置文件指定目标器件、优化策略和 I/O 要求。
- *编译执行：*通过 TI 的神经网络编译器 (TI MCU NNC) 处理 ONNX 模型，以生成与 NPU 硬件兼容的 C/C++ 编译产物。
- *编译输出验证：*验证生成的头文件和库以确保正确转换，尤其是回归任务中反量化等关键方面。

2.3 应用程序集成阶段

最终阶段将编译后的模型集成至完整嵌入式应用中：

- *CCS 工程设置：*建立一个 Code Composer Studio 工程，整合生成的模型文件与应用专属代码。
- *硬件外设配置：*配置所需外设（ADC、DAC、通信接口），为神经网络提供输入并处理其输出。
- *应用逻辑实现：*开发用于协调外设和 NPU 之间的数据流的主应用逻辑，包括适当的输入和输出缩放。
- *硬件测试：*在实际工作条件下，验证实际 F28P55x 硬件上的端到端功能。
- *性能优化：*微调应用，实现推理速度、精度、功耗的最优平衡。
- *部署打包：*为汽车或工业环境中的生产部署准备最终固件包。

3 模型创建示例 (Python)

本节通过构建正弦函数逼近器来演示 NPU 开发原则的实际应用。该示例是一个参考实现，展示了从模型设计到在 F28P55x NPU 上部署的完整工作流程。

3.1 模型选型依据

正弦函数特意被选为演示示例，核心原因如下：

- *数学复杂性：*尽管看似简单，但正弦函数为非线性数学关系，需神经网络具备超越简单线性逼近的建模能力。
- *有界输出范围：*由于输出限制为 -1 到 +1，正弦函数提供了一个适合量化策略的明确范围。
- *可视化验证：*虽然本示例采用示波器波形可视化方式（非常适用于正弦函数），但其他应用需采用与其具体任务匹配的验证方法，例如，分类任务可使用混淆矩阵、异常检测可使用热力图，而控制系统、信号处理和运动控制等汽车与工业领域常见的回归任务，则可采用误差分布图。
- *完整的流水线演示：*正弦函数可演示以 NPU 为中央处理元件的完整的模数模信号处理链。

3.2 模型架构设计

正弦函数逼近器采用多层感知器 (MLP)，专为 F28P55x NPU 进行资源约束设计：

```
SineApproximator(
  (regressor): Sequential (
    (0): Linear(in_features=1, out_features=64, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=64, out_features=64, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=64, out_features=1, bias=True)))
```

代码 1：Sine_64_Model 的正弦逼近器核心架构

该架构包含几个专门用于 NPU 部署的关键设计决策：

- *输入层：*单个神经元输入，接受角度值（映射到 0-2π 弧度）。
- *隐藏层：*选用两层各 64 神经元的隐藏层，以兼顾精度与资源效率；而该 NPU 实际可支持每层最高 128 神经元的模型。
- *激活函数：*选择 ReLU 激活函数，以兼顾计算效率与量化友好性。
- *输出层：*单个神经元输出，用于生成预测的正弦值（范围为 -1 至 +1）。

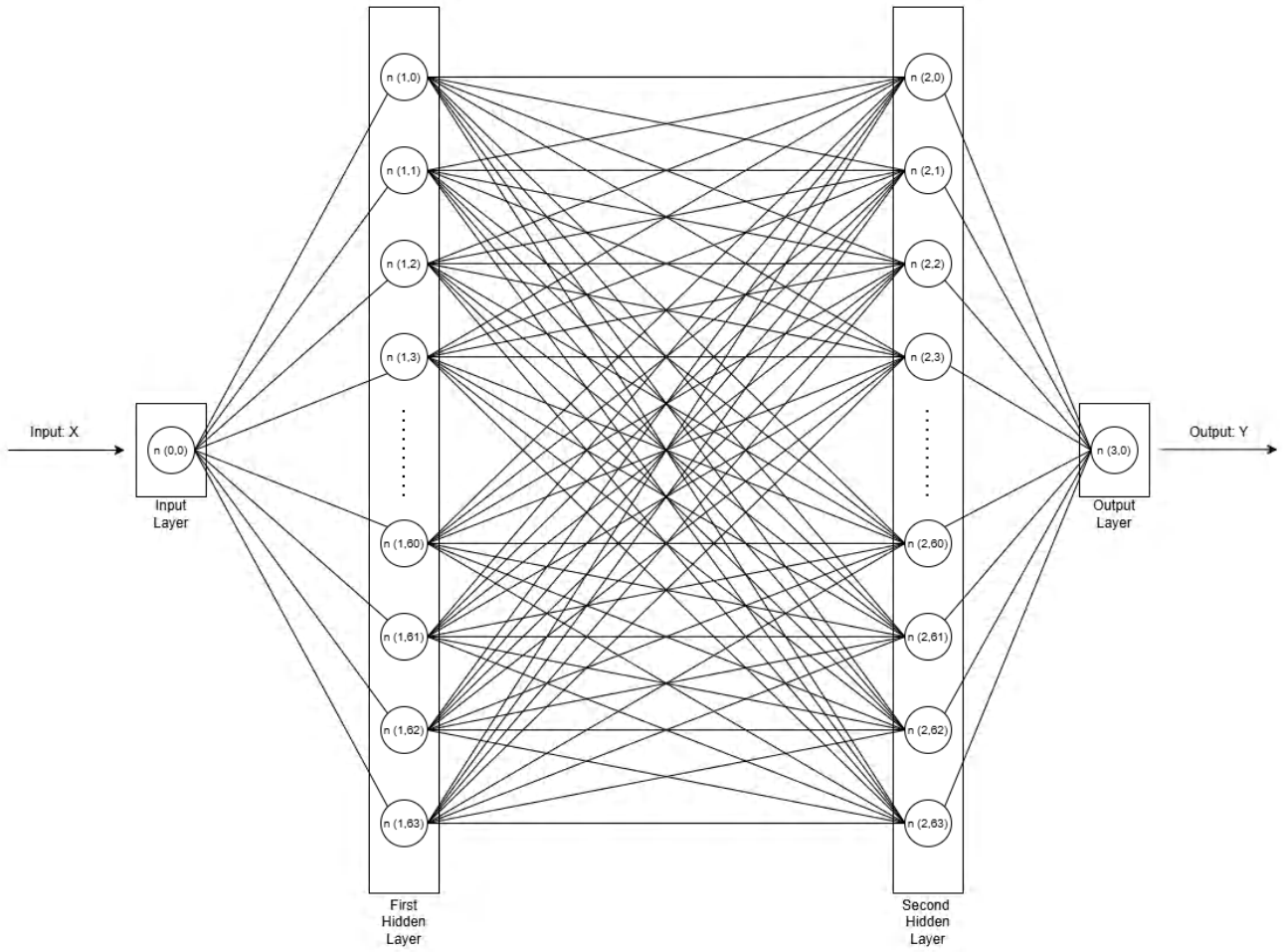


图 3-1. Sine_64_Model 的正弦逼近器核心架构

模型架构规模严格依据 F28P55x NPU 的内存与算力设计。关键约束为总参数量，而非单层神经元数量。我们的架构（层大小为 1×64、64×64 和 64×1）产生了总共 4,353 个参数，在精度与资源利用率之间实现了平衡。虽然该器件支持每层可容纳多达 128 个神经元的模型，但基于精度和延迟考虑因素，选择了 64 神经元配置。没有特征提取器的大型网络（例如每层使用 512 或 1024 个神经元的网络）会超出 NPU 的可用内存。工程师针对该平台设计模型时，需重点关注总参数预算。这种资源受限型模型设计方法广泛应用于各类边缘 AI 实现方案，受内存空间、处理能力及能效预算限制，需精细优化神经网络架构，以确保在既定硬件约束下实现最优性能。

3.3 训练详细信息

模型训练遵循一种结构化方法：

3.3.1 开发环境设置

- *模型设置示例：*
  - 用于神经网络设计和训练的 PyTorch® 框架。
  - 用于量化感知训练的 TINPUTinyMLQATFxModule 包装器（有关量化的更多信息，请参阅第 4 节）。
  - 用于数值计算和可视化的支持库。
- *通用用户设置：*
  - 安装适配目标硬件的 TI 模型优化工具。
  - 确保所选框架版本与 TI 编译工具链的兼容性。
  - 使用虚拟环境（conda、venv）搭建统一环境，避免依赖冲突。
  - 包括特定于应用领域的可视化库（例如，用于传感器的信号绘图、用于计算机视觉的图像显示）。

3.3.2 数据集生成

- *模型数据集详细信息示例：*
  - 创建 100,000 个随机角度值，范围覆盖 0 至 2π 弧度。
  - 计算这些角度对应的正弦值，作为训练目标值。
  - 按 80% 训练数据、20% 验证数据的比例划分训练/测试集。
- *通用用户方法：*
  - 生成或收集跨越整个输入域的数据，以保持稳健的模型性能。
  - 针对传感器类应用，采集涵盖所有工作工况与环境变量的数据。
  - 平衡数据集，以避免训练模型中出现偏差（尤其是在分类任务中）。
  - 依据部署场景的预期输入范围，执行合理归一化。
  - 对分类任务使用分层采样，以保证训练/测试集类别分布一致。
  - 考虑为训练数据添加可控噪声，提升模型在实际部署场景下的稳健性。
  - 对于时间序列数据，在数据集准备阶段应保持正确的序列处理。

3.3.3 模型训练配置

- *模型训练参数示例：*
  - *Batch size：*512 个样本
  - *学习速度：*1e-5（刻意设置为较小值，确保收敛稳定）
  - *优化器：* Adam
  - *损失函数：*均方误差 (MSE)
  - *训练轮数：*160
- *通用用户训练流程：*
  - 根据模型复杂度与可用内存调整批次大小（内存受限环境需调小）。
  - 根据模型大小和收敛行为选择适当的学习速度。
  - *根据任务要求选择优化器：*
    - Adam 用于通用训练场景，收敛速度快
    - 带动量的 SGD，部分场景泛化能力更优
    - RMSprop 用于循环神经网络
  - *按任务需求选择损失函数：*
    - MSE 用于回归问题
    - 交叉熵用于分类任务
    - 自定义损失函数用于专门应用
  - 实施学习速率调度，以提高训练稳定性和最终模型精度。
  - 考虑及早停止，以防止过度拟合，尤其是在训练数据有限的情况下。
  - 监控训练和验证指标，以验证模型是否正确泛化。

3.3.4 量化感知训练流程

- *模型量化过程示例：*
  - 初始模型包装至 TINPUTinyMLQATFxModule，以仿真量化效应。
  - 前向传播过程包含权重和激活量化仿真。
  - 反向传播时，梯度计算将量化效应纳入考量。
  - 定期验证以监控量化模型性能。

```
model = TINPUTinyMLQATFxModule(
      model,
      total_epochs=(int)(MAX_EPOCH/10),
      output_int=False,
      quantization_weight_bitwidth=2)
```

代码 2：使用量化包装器对模型进行包装，以进行微调

- *通用用户量化流程：*
  - *选择量化方法：*
    - 通过 QAT 实现更高的精度（训练耗时更长）

- PTQ 助力加快开发速度（精度可能有所折损）

- *针对目标硬件配置：*
  - 使用硬件专用包装器（例如，TINPUTinyMLQATFxModule）
  - 设置适当的位宽（2/4/8 位）和精度参数
  - 设置微调轮数
- *优化训练流程：*
  - 先浮点训练，再量化微调
  - 与浮点基准模型对比，以评估精度影响
- *缓解量化问题：*
  - 监控激活范围以防止截断
  - 保持回归输出的适当反量化
  - 利用权重/激活直方图来确定分布异常

4 嵌入式平台量化

神经网络量化是将权重和激活项的高精度浮点表示转换为低精度格式（通常为整数）的过程。对于 F28P55x NPU 而言，该转换并非单纯的优化手段，而是核心要求，因为该硬件专为基于整数的计算设计。

对于我们的正弦函数逼近器而言，量化将连续的数学关系转换为 NPU 可高效处理的形式，同时保留正弦波的核心特征。

4.1 量化方法：QAT 与 PTQ

量化神经网络有两种基本方法：训练后量化 (PTQ) 和量化感知训练 (QAT)。了解这些方法之间的差异对于为 F28P55x NPU 部署选择合适的策略至关重要。

4.1.1 训练后量化 (PTQ)

PTQ 针对已完成浮点精度训练的模型执行量化。该方法流程简洁，但可能会牺牲精度，尤其是对于较小的模型或精度敏感的任务。

主要特性：

- *过程：*常规训练模型 → 校准量化参数 → 转换为量化格式。
- *校准功能：*使用具有代表性的数据集来确定缩放因子。
- *开发速度：*加快开发周期（无需重新训练）。
- *精度影响：*通常，与 QAT 相比，精度损失更高。

优势：

- 使用现有经过训练的模型简化工作流程。
- 无需修改训练流程。
- 部署路径更短。
- 开发阶段计算资源需求更低。

限制：

- 可能导致精度显著下降。
- 对量化效应的可控性低。
- 对正弦函数逼近这类回归任务挑战极大。
- 补偿量化失真的能力有限。

4.1.2 量化感知训练 (QAT)

QAT 在训练过程中纳入量化效应，使网络学习到在量化条件下表现最优的参数。该方法通常能更好地保持模型精度，但需要投入更多的开发精力。

主要特性：

- *过程：*初始训练 → 插入量化操作 → 继续训练 → 转换为量化格式。
- *仿真：*在正向传播和反向传播过程中均模拟量化效应。
- *开发速度：*开发周期更长（需要额外的训练）。

优势：

- 模型精度保留效果更佳。
- 网络可学习补偿量化效应。
- 对精度敏感型应用尤为重要。
- 在量化硬件上的性能表现更可预测。

限制：

- 开发工作流程更加复杂。
- 需要额外的训练计算。
- 部署时间更长。
- 需精细调优超参数。

针对本项目的正弦函数逼近器，我们选择 QAT 方案的原因是其在回归任务中能够保持精度，这对于在整个输入域中保持正弦波的平滑特性至关重要。

4.2 量化框架和包装器模块

TI 工具链提供专用的包装器模块，这些模块封装各种部署目标的不同量化方法。要为应用选择合适的量化策略，了解这些包装器至关重要。

TI 的量化框架提供四个不同的包装器模块，它们由两个关键维度组合而成：

- *目标硬件：*
  - *通用：*针对 CPU 执行进行了优化（标准整数运算）
  - *TINPU：*专门针对 TI 的神经处理单元硬件进行了优化
- *量化方法：*
  - *QAT（量化感知训练）：*在训练期间包含量化效应
  - *PTQ（训练后量化）：*训练完成后应用量化

这样就形成了一个包含四种包装器方案的矩阵：

表 4-1. 量化封装器

目标/方法	QAT	PTQ
通用 CPU	GenericTinyMLQATFxModule	GenericTinyMLPTQFxModule
TI NPU	TINPUTinyMLQATFxModule	TINPUTinyMLPTQFxModule

4.2.1 用于 CPU 量化的通用包装器

通用包装器（GenericTinyMLQATFxModule 和 GenericTinyMLPTQFxModule）面向基于 CPU 的执行，并使用 PyTorch 原生量化 API：

```
from tinyml_torchmodelopt.quantization import GenericTinyMLQATFxModule
# or
from tinyml_torchmodelopt.quantization import GenericTinyMLPTQFxModule
```

代码 3：通用量化 API

这些包装器适用于：

- 在无 NPU 加速的 C2000/ARM MCU 上部署。
- 在 NPU 专用优化前测试量化效应。
- 性能要求不太严格的应用。
- 模型原型设计和初始验证。

通用包装器利用标准 PyTorch 量化 API，并通过一个统一的接口简化了应用，该接口只需对现有模型进行极少的代码更改。

4.2.2 用于 NPU 硬件加速的 TINPU 包装器

TINPU 包装器（TINPUTinyMLQATFxModule 和 TINPUTinyMLPTQFxModule）专门面向 TI 的神经处理单元硬件加速器：

```
from tinyml_torchmodelopt.quantization import TINPUTinyMLQATFxModule
# or
from tinyml_torchmodelopt.quantization import TINPUTinyMLPTQFxModule
```

代码 4：TI NPU 专用量化 API

这些包装器对于以下情况至关重要：

- 在 F28P55x 及其他带 NPU 加速的 TI 器件上部署。
- 最大程度提升 TI 硬件的性能。
- 需要实时推理的应用。

- 保持与 NPU 编译工具的兼容性。

TINPU 包装器融入了 TI NPU 硬件的特定约束，确保模型不仅能从通用量化中获益，还能针对 NPU 架构的执行进行专属优化。

5 验证模型

正弦函数逼近器模型经过了一个两阶段验证流程，旨在优化其在 F28P55x NPU 硬件上的表现。本节将详细介绍验证方法、量化感知训练方案，以及证明模型有效性的最终性能指标。

5.1 两阶段训练策略

训练过程采用了精心设计的两阶段方法，以最大限度地提高量化受限 NPU 硬件的性能：

5.1.1 初始训练阶段

- 模型使用常规浮点权重完成训练。
- 重点学习底层的正弦函数数学关系。
- 建立基础的模式识别能力。
- 在引入量化约束前，构建坚实的模型基础。

5.1.2 微调阶段

- 模型使用 F28P55x 专用的量化权重完成训练。
- 模拟 NPU 实际的纯整数运算。
- 针对量化执行场景对权重进行专属优化。
- *Quant_epoch* 通常设置为 *max(5,float_epoch/10)*。

该策略兼顾两者优势：完全浮点精度的初始训练可以实现可靠的特征提取，而量化权重的定向微调专门针对部署硬件优化性能。

5.2 训练阶段比较

表 5-1 对比常规浮点训练最后一轮与量化感知微调最后一轮的性能指标：

表 5-1. 量化过程前后的指标对比

Metric	浮点训练 160 轮后指标	QAT 微调 16 轮后指标	变化 (%)
验证损失	0.00181	0.00105	提高了 42%
验证 MAE	0.02031	0.0215	下降 6%
验证 R^2 分数	0.9963	0.9989	提高了 0.26%

标准浮点模型在传统环境中性能优异，但在 NPU 纯整数运算约束下，精度大幅衰减。QAT 过程可模拟整个训练过程中的量化效应，使神经网络能够相应地调整参数。此优化验证了在部署到 F28P55x NPU 时，该模型在利用硬件专用的神经网络加速功能的同时保持计算完整性。

在验证损失 (-42.0%) 和 R^2 评分 (+0.26%) 方面观察到的提升表明，QAT 可以在为 NPU 部署准备模型的同时增强特定的性能指标。MAE 小幅增加 (+5.9%)，这正是量化过程中的内在权衡。这些结果验证了量化感知训练方法在资源受限的嵌入式硬件上实现出色性能的有效性。

5.3 验证结果和指标

为保持性能评估的客观性，验证过程采用了一套简单的方法：

- *验证数据集：*从生成数据中抽取 20% 不参与训练。
- *多维指标：*采用互补的评估指标进行全面评估。
- *量化仿真：*采用与目标硬件一致的量化方案开展验证。

经过量化感知训练的 Sine_64 模型达到了良好性能指标，符合简单 ML 模型的预期，并证明了其在 NPU 部署中的有效性：

- 验证损失通过均方误差 (MSE) 进行衡量。均方误差衡量的是预测值 ($\hat{y}_i$) 与实际值 (y_i) 之间差值平方的平均值。数值越低，模型性能越好。

$$MSE = 1/n\sum_{0}^{n}(\mathrm{yi} - \hat{\mathrm{y}}\mathrm{i})^2 \quad (1)$$

- 平均绝对误差衡量的是预测值与实际值之间绝对差值的平均值。与 MSE 不同，该指标不会对误差求平方，对异常值不敏感。

$$MAE = 1/n\sum_0^n |yi - \hat{y}i| \quad (2)$$

- R^2 表示模型相对于使用均值作为预测器，对数据方差的解释程度。值范围为 0 到 1，其中 1 表示完美预测。验证 R^2 分数反映了模型对给定数据方差的解释能力。

$$R^2 = 1 - \left(\sum_0^n (yi - \hat{y}i)^2\right) / \left(\sum_0^n (yi - \bar{y})^2\right) \quad (3)$$

Variables：

- y_i = 第 i 个样本的实际（真实）正弦值
- $\hat{y}_i$ = 神经网络对第 i 个样本的正弦预测值
- $\bar{y}$ = 验证数据集中所有实际正弦值的均值

表 5-2. 训练验证指标

Metric	Metric	技术意义
验证损失	0.00038	均方误差极低，表明预测精度高
验证 MAE	0.01305	整个正弦范围（-1 至 +1）内的平均绝对误差约 1.3%
验证 R^2 分数	0.9993	决定系数接近完美，表明模型可解释 99.93% 的正弦值方差

上述指标验证了以下核心成果：

- **高精度：**虽受量化约束，但该模型仍实现了百分之一以内的精度。
- **稳定的性能：**优异的 R^2 分数表明模型在全输入范围内预测结果可靠。
- **量化鲁棒性**：纯整数运算下，模型性能几乎无衰减。
- **部署就绪性：**指标验证模型对于 NPU 实施的适用性。

6 测试模型

模型训练与验证成功后，对正弦函数逼近器模型进行全面测试至关重要，以此验证其在实际应用中的性能。本节详细介绍了评估模型是否具备 NPU 部署条件所采用的测试方法、推理程序和结果评估技术。

6.1 推理设置和方法

我们的正弦函数测试方法使用带 ONNX Runtime 的专用推理笔记本在硬件部署之前验证模型。该框架实现了一个封装测试类，为预测提供了一个干净的接口，同时保证数据处理一致性。

测试方法遵循以下系统化步骤：

- *模型加载：*该 ONNX 模型通过 ONNX Runtime 加载，以便访问最终将编译部署至 NPU 的同一版模型文件。
- *输入准备：*测试点覆盖 0 至 2π 弧度的整个输入域。
- *基准比较：*将每个预测与实际的数学正弦函数进行比较。

这种测试方法验证了导出的 ONNX 模型（尤其是带反量化层的版本）在全值域内准确逼近正弦函数，是硬件部署前的关键质量验证环节。

6.1.1 通用用户测试方法

- *模型验证：*使用适当的框架来测试导出的模型。
- *代表性测试数据：*创建反映部署条件和边缘案例的测试数据集。
- *领域专用指标：*选择符合应用要求的评估指标：
  - *分类：*准确率、精确率、召回率、F1 分数。
  - *回归分析：*MAE、MSE、R^2 分数。
  - *信号处理：*SNR、交叉相关、频率响应。
  - *愿景:* 目标检测的交并比 (IoU)、平均精度均值 (mAP)，用于图像质量的结构相似性 (SSIM)。
- *性能基准测试：*测量推理速度、内存使用情况和功耗。
- *比较基准：*适用时对照参考算法进行基准测试。
- *环境测试：*对于关键应用，请在温度范围、电压变化或其他环境因素下进行测试。

此验证是进入硬件实施前的关键质量关卡，确保量化模型在投入硬件部署前满足应用要求。

6.2 测试结果和可视化分析

6.2.1 可视化性能评估

测试关键环节为预测正弦值与真实值的可视化对比。笔记本端生成综合图表，叠加展示全输入域内模型预测值与真实正弦函数。

该可视化立即显示模型的逼近效果。在正弦函数示例中，预测曲线非常接近实际正弦曲线，可见偏差极小。

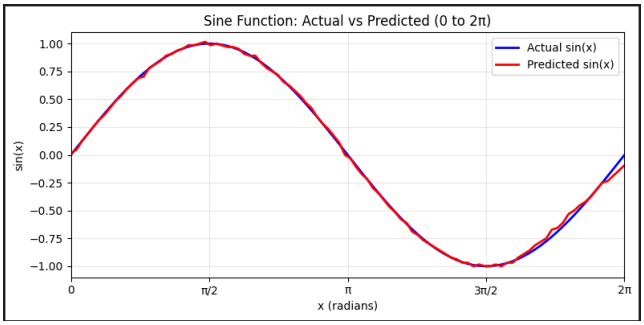


图 6-1. Python 中的神经网络正弦函数逼近性能

该图（如图 6-1 中所示）展示了训练后神经网络在全输入域内的正弦函数逼近性能。蓝色曲线代表真实的数学正弦函数，而红色曲线则显示了量化感知神经网络模型作为 ONNX 文件运行时生成的预测值。该图展示训练后神经网络在全输入域内的正弦函数逼近性能。

6.3 定量性能指标

除可视化检查外，测试框架还会计算综合误差指标，以精确量化预测准确性。模型性能可通过测试过程中获取的以下误差指标进行精准量化：

- *平均绝对误差 (MAE)：*0.013827
- *最大误差：*0.093750
- *最小误差：*0.000126

这些量化结果补充了图表提供的可视化评估，从数值层面证实，尽管受量化训练的约束，该模型仍具备优异的逼近能力。可视化验证与数值验证相结合，可确保在硬件实现前，模型性能具备充分的可靠性。

7 将模型迁移至 TI MCU (C2000 – F28P55x)【入门级】

将 ONNX 模型部署到 F28P55x 可通过两种不同的方式实现，分别适配不同用户的需求和专业水平。

对于追求简洁高效设计的用户，入门级方案提供了配置要求极少的简化流程。该方法屏蔽了诸多技术细节，同时仍能交付一个可运行的实现方案。

用户可以遵循*适用于 MCU 的 TI 神经网络编译器用户指南 — 2.1.0.LTS* 的第 1 节和第 2 节的说明，其中提供了获取部署所需编译后库文件和头文件的分步操作指引。

8 将模型迁移至 TI MCU (C2000 – F28P55x)【开发人员级别】

该开发者方案适用于需要全面掌控编译流程并追求与现有系统进行更深层次整合的工程师。尽管涉及的步骤更多，但这种方法提供了对部署流程每个步骤的完整可见性，并支持对编译参数进行广泛的自定义。以下各节将详细介绍该流程，助力开发者针对特定应用需求优化神经网络的实现方案。

本节会详细介绍将数学模型转换为可集成到 Code Composer Studio 工程中的硬件兼容文件的完整工作流程。

8.1 编译前提条件

启动模型编译流程前，必须正确配置若干专用工具与环境。该准备阶段对编译与部署的成功至关重要。

8.1.1 必需的 TI 软件组件

编译工具链依赖于若干 TI 专属资源库和工具：

- *tinyml-modelmaker*
  - *用途：*为模型开发和编译提供统一的接口。
  - *功能：*协调编译流程并与其他组件集成。
- *tinyml- modeloptimization*
  - *用途：*模型量化和优化工具套件。
  - *功能：*提供包装器和工具，专门用于为 TI-NPU 部署做模型预处理。
- *C2000Ware SDK*
  - *用途：*面向 C2000 系列微控制器的平台专属驱动、库及工具。
  - *功能：*为 F28P55x 平台提供核心驱动与硬件抽象层。
- tinyml-modelzoo
  - *用途*：提供针对嵌入式系统优化的精选神经网络模型合集。
  - 功能：是通过 tinyml-modelmaker 方法编译模型的必备组件。

8.1.2 环境设置过程

编译环境的搭建遵循结构化流程：

- *克隆所需代码仓库：*
  - git clone https://github.com/TexasInstruments/tinyml-tensorlab/tree/main/tinyml-modelmaker
  - git clone https://github.com/TexasInstruments/tinyml-tensorlab/tree/main/tinyml-modeloptimization
  - git clone https://github.com/TexasInstruments/tinyml-tensorlab/tree/main/tinyml-modelzoo
  - *运行初始化脚本：*
    - *进入* ***tinyml-modelmaker*** *代码仓库目录*
    - 执行 *setup_all.sh* 以编排完整的环境设置
    - 执行 *setup_c2000ware.sh* 以配置 C2000Ware SDK
    - 执行 *setup_cg_tools.sh* 以配置 TI 代码生成工具
- *Python 环境设置：*
  - 建议使用专用 Python 虚拟环境来避免依赖冲突。
  - 所需的关键 Python 包包括：
    - ONNX 和 ONNX Runtime
    - TVM (TI MCU NNC) 框架
    - NumPy 和相关数值库

8.2 配置文件设置

编译过程由 *config.yaml 文件* 控制，该文件定义了将 ONNX 模型转换为 NPU 兼容代码的关键参数。这种配置方法在保持编译流程一致性的同时，提供了灵活性。

8.2.1 配置文件结构

编译正弦函数模型的典型 config.yaml 文件包含以下核心配置段：

```
common:  # The common section can be plainly copied as it is
    target_module: 'timeseries'
    task_type: 'generic_timeseries_regression'
    target_device: 'F28P55'

dataset:
    dataset_name: Sine # Can be anything, used for directory name
    enable: False      # Please note the 'False'. Since model is already trained.
training:
enable: False  # Please note the 'False'. Since model is already trained.     model_name:
'SineModel_1e-05_160_64_Dequant'   # Can be anything, used for
                                                        directory name
    output_int: False # We need the model to dequantize the values after
                              running on NPU and return float value
compilation:
    enable: True
    model_path: "path//to//SineModel_1e-05_160_64_Dequant.onnx"
```

代码 5：配置文件示例

各配置段在编译流程中承担特定作用：

- *常用设置：*定义基本任务类型和目标硬件。
  - *task_type* 标识机器学习任务类别。
  - *target_device* 启用 F28P55x 专属优化。
- *数据集设置：*控制数据集处理行为。
  - 使用预训练模型时通常禁用。
- *训练设置：*配置训练参数。
  - *output_int：*将此值设置为 False，对于回归任务至关重要，用于验证浮点输出是否正确。
  - 预训练模型通常禁用训练。
- *编译设置：*控制模型转换。
  - *enable：*设为 True 会激活编译阶段。
  - *model_path* 指定带反量化层的 ONNX 模型的位置。

8.2.1.1 需要反量化标志的模型

*output_int：*False 设置对于以下情况至关重要：

- *回归模型：*任何预测连续值的模型，例如：
  - 时序预测（温度、压力、电压预测）
  - 控制系统建模（PID 系数估算）
  - 函数逼近（如我们的正弦示例）
  - 信号滤波应用
  - 传感器校准模型
- *高度依赖归一化的模型：*输出缩放显著的应用：
  - 输出归一化到特定范围的模型
  - 具有校准输出的传感器融合算法
  - 物理量估算（力、扭矩等）
- *多输出模型：*部分输出需要浮点精度的场景：
  - 分类/回归综合模型
  - 姿态估计（角度需要浮点）
  - 坐标回归（目标定位）

如果没有此标志，这些模型将生成量化整数输出，不适用于需要连续值范围或精确小数输出的应用。

8.2.2 回归模型专属配置

正弦函数逼近等回归任务需要特殊处理，以确保将 NPU 输出的纯整数结果正确反量化回浮点值。

8.2.2.1 输出反量化标志

- *output_int*：对于回归任务，必须将该参数设置为 False。
- 这会指示编译器在生成的代码中保留反量化运算。
- 如果没有此标志，模型会生成不适合正弦逼近的整数输出。

8.2.2.2 编译器常量修改

除了配置文件，编译器还需进行修改以支持浮点输出。

- 找到并打开 *tinyml-modelmaker/ai_modules/timeseries/constant.py*。
- 滚动浏览该文件，直至找到 *COMPILATION_C28_HARD_TINPU* 字典：

```
COMPILATION_C28_HARD_TINPU= dict(
      target= "c, ti-npu type=hard skip_normalize=true output_int=true",
      target_c_mcpu='c28',
      cross_compiler=CL2000_CROSS_COMPILER,
)
```

代码 6：非回归模型的参考常量

- 这个现有常量是专门为分类任务而配置的，其中：
  - *type=hard* 指定硬件 NPU 加速（而非 CPU）
  - *skip_normalize=true* 可绕过归一化/缩放
  - *output_int=true* 会强制输出整数
- 复制该常量以创建新常量 *COMPILATION_C28_HARD_TINPU_TEMP*，并做如下关键修改：

```
COMPILATION_C28_HARD_TINPU_TEMP = dict(
      target= "c, ti-npu type=hard skip_normalize=false output_int=false",
      target_c_mcpu='c28',
      cross_compiler=CL2000_CROSS_COMPILER,
)
```

代码 7：添加额外常量来处理回归模型

- 修改以下参数以适应回归模型：
- *type=hard*：
  - 保持不变。这会将编译定向到 NPU 硬件加速器，而不是使用 CPU 进行神经网络计算的*软*方式。
- *skip_normalize=false*：
  - 示例应用为演示用简单模型，无需归一化。
- *output_int=false*：
  - 编译器会生成浮点输出而非整数。
  - 保留回归任务所需的小数精度。
  - 允许表示值的完整连续范围。
- *target_c_mcpu='c28'*：保持不变 — 指定为 C28x 架构生成代码。
- *cross_compiler=CL2000_CROSS_COMPILER*：保持不变。定义要使用的编译器工具链。

8.2.2.3 编译字典更新

必须修改默认编译字典才能使用自定义常量：

- 将所使用的常量 — *COMPILATION_C28_HARD_TINPU* 更改为 *COMPILATION_C28_HARD_TINPU_TEMP*。

```
TASK_TYPE_GENERIC_TS_REGRESSION: {
      COMPILATION_DEFAULT: dict(
```

```
-  compilation=dict(**COMPILATION_C28_HARD_TINPU,
   cross_compiler_options=CROSS_COMPILER_OPTIONS_F28P55, )
+  compilation=dict(**COMPILATION_C28_HARD_TINPU_TEMP,
   cross_compiler_options=CROSS_COMPILER_OPTIONS_F28P55, )
 ),
```

代码 8：为适应回归模型所做的更改

8.3 编译处理流程

将 ONNX 模型转换为 F28P55x NPU 兼容代码的过程，涉及由 TI MCU 神经网络编译器 (TI MCU NNC) 执行的多阶段流程。本节会详细介绍逐步编译的工作流程以及内部转换过程。

8.3.1 启动编译

正确设置配置文件后，即可使用 TinyML modelmaker 脚本启动编译：

```
cd tinyml-modelzoo
/<path/to/run_tinyml_modelzoo.sh> <path/to/config_file>
```

代码 9：将 ONNX 文件编译为嵌入式兼容文件的命令

对于正弦函数示例，该命令为：

```
./run_tinyml_modelzoo.sh    ./config.yaml
```

代码 10：触发编译的命令示例

此命令会触发完整的编译流水线，将 ONNX 模型转换为针对 F28P55x NPU 优化的 C/C++ 代码。

8.3.2 编译阶段

编译器通过三个主要阶段处理模型：

- *模型验证*
  - 编译器会检查 ONNX 模型是否可以在 NPU 上运行。
  - 确认所有运算均受支持（如我们正弦模型中的线性层和 ReLU）。
  - 验证模型是否符合 NPU 内存限制。
- *模型变换*
  - 将 ONNX 模型转换为针对 NPU 优化的格式。
  - 应用量化参数以支持基于整数的计算。
  - 对于我们的正弦函数逼近器这类回归模型，保留反量化信息。
- *Code Generation*
  - 创建可在 F28P55x 上运行的 C/C++ 代码
  - 生成两个主文件：
    - 带有函数声明的头文件 (tvmgen_default.h)
    - 包含已编译模型的库文件 (mod.a)

8.3.3 需要注意的常见问题

尽管编译器已处理大部分复杂逻辑，但需关注以下常见消息：

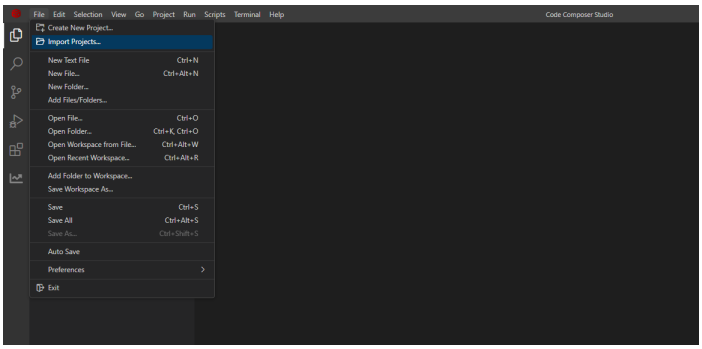
- *不支持的运算：*模型包含 NPU 无法执行的运算。
- *超出内存约束：*您的模型对于边缘器件来说太大了。
- *反量化错误：*对于回归模型，输出反量化环节可能需要重点关注。

9 MCU 工程设置

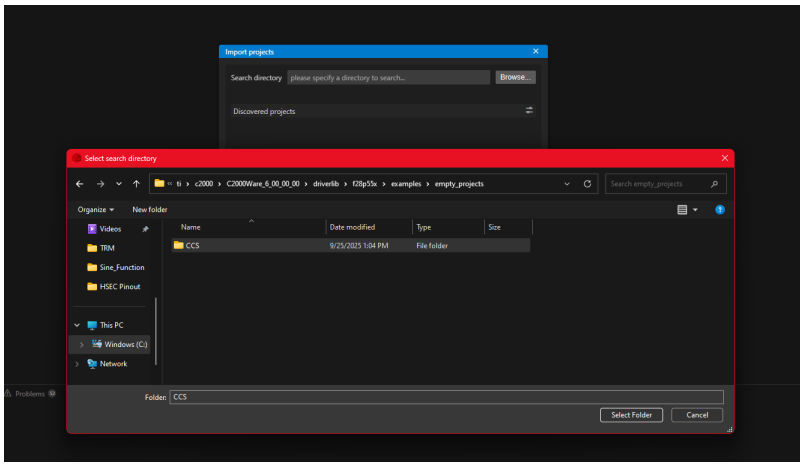
神经网络模型成功编译为 NPU 兼容文件后，下一关键阶段是将这些组件集成到 Code Composer Studio（CCS）工程中，以便在 F28P55x 微控制器上执行。本节将指导工程师建立工程结构、配置开发环境以及实现充分发挥 NPU 能力所需的应用框架。

9.1 为 NPU 应用创建 CCS 工程

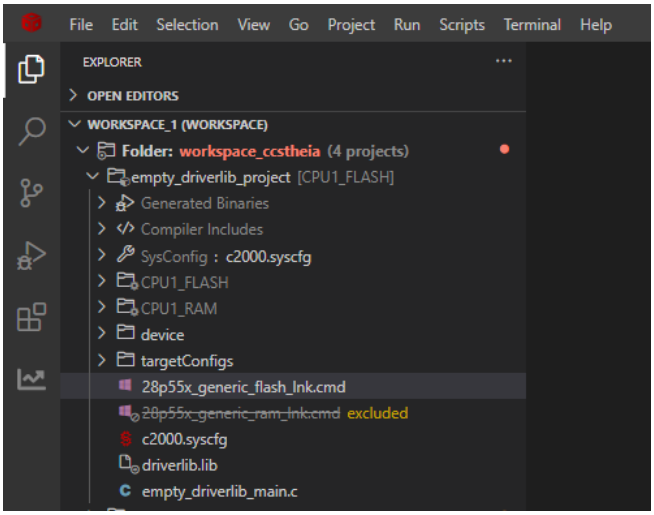
- *第 1 步：*打开 CCS，点击 *File* 和 *Import Projects*。



- **第 2 步：**从 C2000Ware SDK 导入 *empty_project*。



- *第 3 步：*打开已导入工程中的 *f28p55x_generic_flash_lnk.cmd* 文件



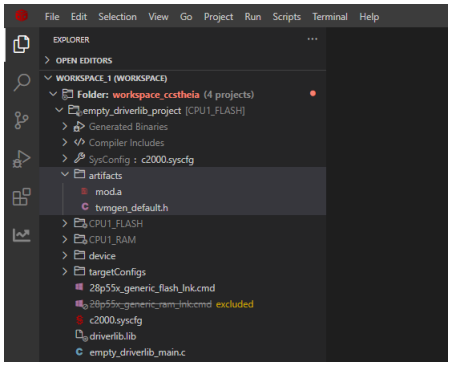
- *第 4 步：*在 FLASH 中添加 .rodata.tvm 段，并将 .bss.noinit.tvm 段放置到全局共享 SRAM 中，以便硬件 NPU 可以访问。例如，F28P55x MCU 器件上的 RAMGS0、RAMGS1、RAMGS2 或 RAMGS3。

```
SECTIONS
{
    codestart       : > BEGIN
    .text           : >> FLASH_BANK0 | FLASH_BANK1, ALIGN(8)
    .cinit          : > FLASH_BANK0, ALIGN(8)
    .switch         : > FLASH_BANK0, ALIGN(8)
    .reset          : > RESET, TYPE = DSECT /* not used, */

    .stack          : > RAMLS3
#if defined(__TI_EABI__)
    .bss            : >> RAMLS47 | RAMLS89 // RAMLS5
    .bss:output     : > RAMLS2 //RAMLS3
    .init_array     : > FLASH_BANK0, ALIGN(8)
    .const          : > FLASH_BANK0, ALIGN(8)
    .data           : > RAMLS89  //RAMLS5 | RAMLS6  //RAMLS2
    .sysmem         : > RAMLS2 //RAMLS4
#else
    .pinit          : > FLASH_BANK0, ALIGN(8)
    .ebss           : > RAMLS2 //RAMLS5 | RAMLS6
    .econst         : > FLASH_BANK0, ALIGN(8)
    .esysmem        : > RAMLS2 //RAMLS5
#endif

    .rodata.tvm     : > FLASH_BANK0
    .bss.noinit.tvm  : > RAMGS012
```

- *第 5 步：*在工程中创建一个名为 ***artifacts*** 的文件夹，并将编译后的库和头文件放入该文件夹。



- *第 6 步：*在应用中引入该头文件，然后继续编写使用 NPU 运行 ML 模型的应用代码。使用下一节中详述的 NPU 接口。

empty_driverlib_main.c

workspace_ccstheia > empty_driverlib_project > empty_driverlib_main.c > ...

```
// Included Files
//
#include "driverlib.h"
#include "device.h"
#include "board.h"
#include "c2000ware_libraries.h"
#include "tvmgen_default.h"

//
// Main
//
void main(void)
{

    //
    // Initialize device clock and peripherals
    //
    Device_init();

    //
    // Disable pin locks and enable internal pull-ups.
    //
    Device_initGPIO();

    //
    // Initialize PIE and clear PIE registers. Disables CPU interrupts.
    //
    Interrupt_initModule();

    //
    // Initialize the PIE vector table with pointers to the shell Interrupt
    // Service Routines (ISR).
    //
    Interrupt_initVectorTable();

    //
    // PinMux and Peripheral Initialization
    //
    Board_init();

    //
    // C2000Ware Library initialization
    //
    C2000Ware_libraries_init();

    //
    // Enable Global Interrupt (INTM) and real time interrupt (DBGM)
    //
    EINT;
    ERTM;

    while(1)
    {

    }
}
```

9.2 了解 NPU 接口

头文件 (tvmgen_default.h) 提供关键结构和函数，作为应用程序与 NPU 之间的接口。了解这些组件对于成功集成神经网络至关重要。

9.2.1 主要接口组件

头文件定义若干基本元素：

- *输入和输出结构*：这些结构提供了在神经网络中进行数据往来传递的机制。

```
struct tvmgen_default_inputs {
    void* input;  // Points to input data (float for sine case)
};
struct tvmgen_default_outputs {
    void* output;  // Points to output buffer for results
};
```

代码 11：编译过程中生成的输入和输出结构

- *初始化函数：*该函数必须在应用启动时调用一次，以配置 NPU 硬件。

```
void TI_NPU_init();
```

代码 12：NPU 初始化 API

- *执行函数：*此函数会触发 NPU 上的神经网络执行。

```
  int32_t tvmgen_default_run(
    struct tvmgen_default_inputs* inputs,
    struct tvmgen_default_outputs* outputs
);
```

代码 13：用于触发 NPU 的 API

- *完成标志：*此标志指示神经网络处理何时完成。

```
extern volatile uint8_t tvmgen_default_finished;
```

代码 14：指示 NPU 处理结束的标志

9.2.2 基本使用模式

使用 NPU 的典型流程包括：

- *一次性初始化：*在系统启动期间调用 TI_NPU_init()。
- *输入准备：*创建输入结构并把指向数据的指针填入其中。
- *输出分配：*创建输出结构并在其中填充指向接收结果的变量的指针。
- *模型执行：*使用输入和输出结构调用 tvmgen_default_run()。
- *完成情况监控：*监控 tvmgen_default_finished 标志，以确定处理何时完成。
- *结果使用：*设置完成标志后，输出数据即可在应用中使用。

10 在嵌入式环境中测试模型

模型训练与转换成功后，在 F28P55x 微控制器上直接对正弦函数逼近器进行全面测试至关重要，以此验证其在实际部署环境中的性能。本节详细介绍了在目标硬件上运行神经网络模型所采用的测试方法及得到的结果。

10.1 可视化性能评估

测试过程的一个关键环节是对正弦波的预测值与真实值进行可视化对比。采用 CCS 绘图工具绘制整个输入域的神经网络预测值与真实正弦函数曲线。

图 10-1 展示了 NPU 生成的正弦近似和真实正弦值的可视化对比：

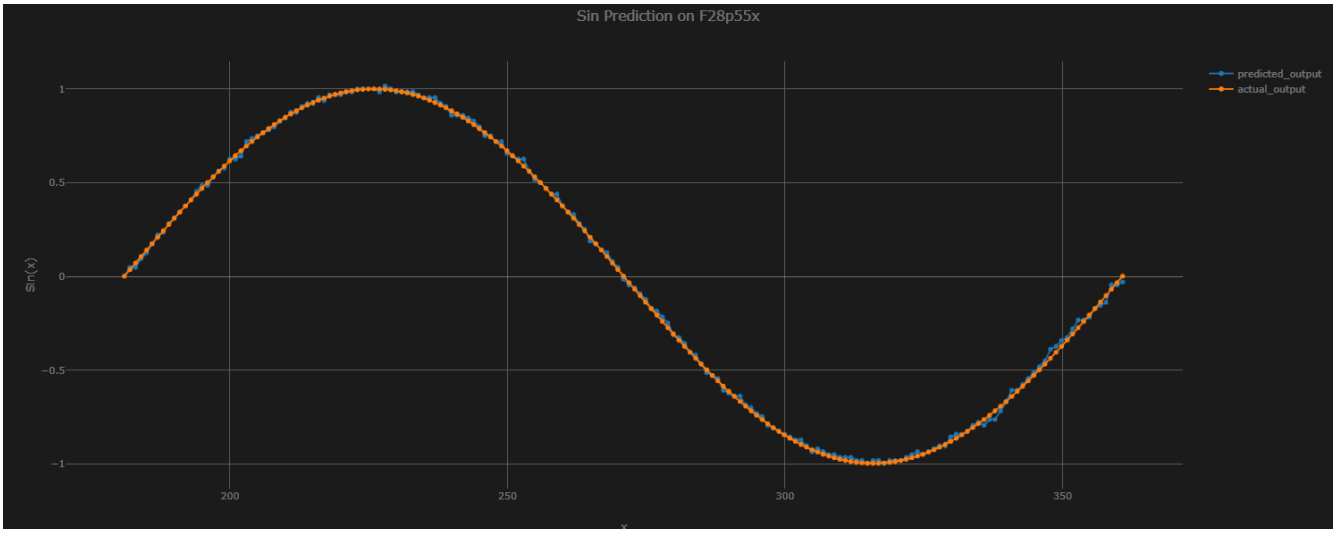


图 10-1. F28p55x 上的神经网络正弦函数逼近性能

该可视化演示了模型在实际硬件上的以下核心性能：

- *稳定的性能：*该模型在整个波形中保持高精度，包括在极值（峰值和波谷）和过零点等关键点。
- *平稳过渡：*尽管 NPU 的计算具有量化性质，但逼近曲线在整个过程中保持平滑过渡，没有可见的不连续性或伪影。
- *全周期覆盖：*测试评估了从 0 到 360 度的完整正弦函数周期，确认波形在所有相位阶段的性能一致性。

可视化结果提供令人信服的证据，模型成功从训练阶段迁移至 F28P55x 硬件，且精度下降极小。预测值和实际值几乎难以分辨的覆盖，展示了量化感知训练方法在整个编译过程中保持模型保真度的有效性。

10.2 定量性能指标

除可视化检查外，计算全面的量化指标，客观评估模型在 F28P55x NPU 上的性能。量化评估产生了以下关键性能指标：

- *平均绝对误差 (MAE)：*0.01214
- *最大误差：*0.0973
- *延迟 (ms)：*0.214
- R^2 *分数：*0.9997

这些指标可对精度与性能进行全面评估。误差指标量化模型的逼近精度，而时序测量结果则体现出 NPU 硬件相较主 CPU 上软件执行方案的效率优势。这种精度和速度的组合能够支撑汽车和工业控制系统中普遍存在的对时序和精度要求严苛的实时应用场景。

11 NPU 在实时信号链中的集成

尽管使用预定义数据集进行的初始模型验证能为神经网络性能提供有价值的指导依据，但 NPU 应用的真正考验在于处理实时数据的能力。本节将探讨如何实现完整的实时信号处理链，该链通过 F28P55x NPU 上的神经网络推理将实时模拟输入转换为有意义的输出。

11.1 应用方框图

图 11-1 展示了为验证 F28P55x 微控制器上神经处理单元 (NPU) 的实现而搭建的完整仿真环境图。该架构图展示了从生成到处理再到可视化的完整信号流，演示了神经网络模型如何将锯齿波实时转换为正弦波。

- *信号生成：*最左侧部分显示了配置为 DAC 的 CMPSS 模块，该模块生成振幅范围为 0-360 单位的锯齿波形。该波形以上方框图中的锯齿状图案呈现，作为系统的输入激励信号。
- *信号采集：*锯齿波馈入 ADC（模数转换器）模块中，该模块将模拟信号数字化。输出保持相同的数值范围 (0-360)，但现在处于数字域中。
- *神经网络处理：*图中央部分展示了包含 NPU 和正弦模型的 F28P55x 微控制器。输入值（标记为“x”）传递到正弦模型，该模型计算相应的正弦值。
- *输出生成：*从 -1 到 1（正弦值的自然范围）的神经网络输出被定向到第二个 DAC，以便转换回模拟域。
- *可视化：*最右侧部分显示了示波器上的最终输出，呈现出平滑的正弦波图案，直观地确认了从锯齿波到正弦波的成功转换。

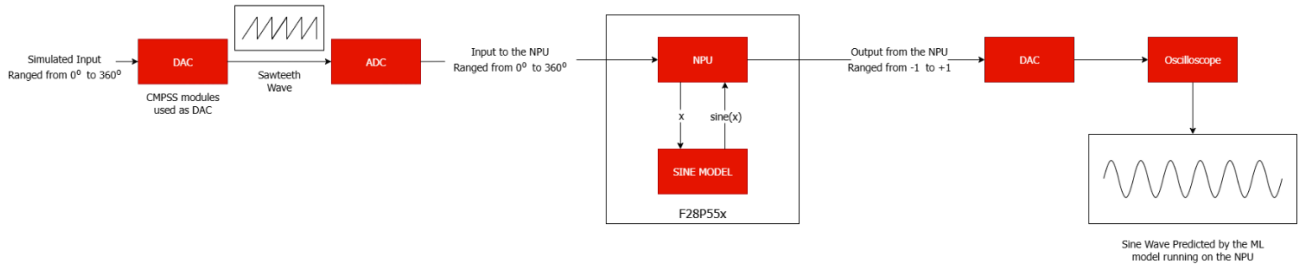


图 11-1. 应用程序端的数据流

11.2 应用代码实现

以下代码演示了如何实现完整的信号处理流水线，通过集成 CMPSS、ADC、NPU 和 DAC 元件，使用神经网络模型将锯齿波转换为正弦波：

```
while(1){
   for(i = 0; i < MAX_ADC_VALUE; i++)
   {
      // Set the CMPSS low DAC value to generate sawtooth waveform
      CMPSS_setDACValueLow(CMPSS1_BASE, i);
      // Trigger ADC conversion to read back the analog signal
      ADC_forceSOC(myADC0_BASE, myADC0_SOC0);
      // Read the result from the ADC
      adcResult = ADC_readResult(myADC0_RESULT_BASE, myADC0_SOC0);
      // Scale ADC value to phase angle in range [0, 2π)
      input = (float)adcResult * (2.0f * M_PI / (MAX_ADC_VALUE + 1));
      // Prepare input/output structures for the neural network
      struct tvmgen_default_inputs inputs = { (void*)input_arr };
      struct tvmgen_default_outputs outputs = { output_arr };
      // Execute the neural network model to predict sine value
      tvmgen_default_run(&inputs, &outputs);
      // For NPU mode, wait until the neural network processing is complete
      #if defined(TVMGEN_DEFAULT_TI_NPU)
         while (!tvmgen_default_finished);
      #endif
      // Scale the sine output from [-1, 1] to DAC range [0, 4095]
      dacValue = (uint16_t)((output + 1.0f) * (MAX_DAC_VALUE / 2.0f));
      // Output the predicted sine wave to the DAC
      DAC_setShadowValue(myDAC0_BASE, dacValue);
      // Delay to control the wave frequency
      DEVICE_DELAY_US(WAVE_DELAY_US);  // Delay between samples
```

```
    }
}
```

代码 15：在 F28P55x 微控制器上实现基于神经网络的实时正弦波生成的主应用代码

11.3 所使用的硬件组件

仿真环境基于 F28P55x 平台以下核心硬件组件搭建：

- *CMPSS DAC（比较器子系统）：*该模块虽常规用于比较器功能，此处复用为信号发生器。CMPSS 模块内置 12 位 DAC，通过程序递增生成锯齿波。这种方法展示了 F28P55x 外设在生成测试信号方面的灵活性，无需依赖外部硬件。
- *ADC 模块：*F28P55x 集成的 12 位模数转换器对生成波形进行采样。在实际应用中，这通常连接到外部传感器或信号源，但在该仿真环境中，它对内部生成的波形进行采样，以创建完整的信号路径。
- *NPU 硬件加速器：*与单独使用主 CPU 相比，F28P55x 内置的专用神经处理单元能够以更高的性能和效率执行正弦模型计算。该图展示了 NPU 如何集成到处理链中、接收数字输入并生成计算输出。“正弦模型”代表已训练、编译并部署到 NPU 的神经网络模型。该模型包含将输入值转换为相应正弦值所需的权重和结构。
- *缓冲 DAC：*一条独立的 DAC 通道将神经网络的浮点输出转换回模拟信号，以便观察。该 DAC 与 CMPSS DAC 在典型使用模式和性能特性上均有所不同。

11.4 硬件验证结果

图 11-2 中所示的示波器捕获结果为整个 NPU 应用提供了确凿的验证，证明通过 F28P55x 平台上的神经网络推理，已成功实现从锯齿波到正弦波的实时转换。

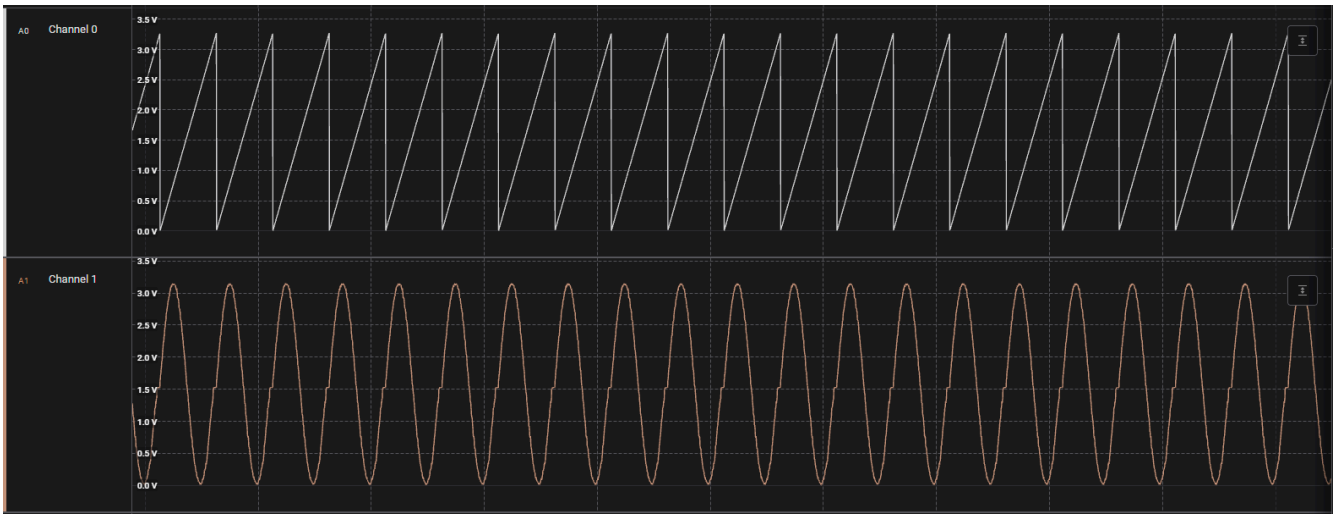


图 11-2. 示波器上的应用验证

11.4.1 输入信号特性

上方迹线（通道 0）显示由 CMPSS DAC 生成的锯齿波形：

- 波形为 0V 至 3.3V 的干净线性斜坡，伴随陡峭的复位沿。
- 在所有周期内具有一致的周期和幅度，表明信号生成稳定。
- 复位点处的陡峭跳变给神经网络带来了挑战性的测试条件，因为这类跳变代表输入函数的不连续点。

该线性斜坡输入是理想的测试信号，因为它会在每个周期中系统地遍历神经网络的完整输入域，全面覆盖模型的工作范围。

11.4.2 神经网络输出分析

下方迹线（通道 1）显示了神经网络产生并通过 DAC 输出的正弦波形：

- 平滑正弦波形，幅值范围 0V 至 3.3V。
- 所有可见周期内幅值与频率保持一致。

- 波峰与波谷处失真极小，表明在临界点具有良好的逼近效果。
- 与输入锯齿波相位关系匹配，每个锯齿波周期对应一个完整正弦波周期。
- 部分过零点处存在轻微可见瑕疵，是由于 CMP1_DACL 的线性度限制所致，而这种限制仅有效使用 10 位精度源。

正弦波输出质量优异，过渡平滑、波形规整，与理论正弦函数高度吻合。多周期一致性验证了神经网络在连续运行时能输出稳定、可复现的结果。

12 关键设计决策和影响

12.1 NPU 数值处理

f28p55x NPU 本质上专为基于整数的运算设计，这给负值与浮点数据的处理带来了挑战。但是，通过适当修改神经网络架构和编译过程，NPU 可有效处理这些数据类型。本文档提供了应对这些局限性的全面设计。

12.1.1 纯整数架构

- NPU 硬件专为整数算术运算进行了优化。
- 仅原生支持无符号整数运算。
- 这一核心设计选择最大限度地提高了嵌入式应用的处理效率，但对其他数据类型的处理提出了特殊要求。

12.1.2 负值与浮点值处理

- *量化过程：*负值和浮点数据都需要量化，以便映射到 NPU 的整数表示
- *自动转换层：*编译过程会自动为输入插入量化层，为输出插入反量化层
- *配置要求：*编译流水线必须显式配置，以保留数值的完整范围：

```
output_int: False  # Enables float output from integer NPU processing
```

代码 16：用于启用反量化浮点输出的标志

- *特殊编译器标志：*对于需要浮点或负输出的应用（如我们的正弦函数），需要额外的编译标志：

```
skip_normalize=false output_int=false
```

代码 17：用于禁用归一化的标志

- *处理开销：*额外的量化/反量化层会增加少量计算开销。

12.2 支持的神经网络层和约束

F28p55x NPU 支持特定类型的神经网络层，且这些层存在特定约束。了解这些功能是设计可成功编译并部署至该硬件的模型的关键。

12.2.1 支持的层类型

12.2.1.1 卷积层

- *第一卷积 (FCONV)：*支持单通道输入特征图（非深度卷积、非逐点卷积）。
- *通用卷积 (GCONV)：*处理通道数为 4 的倍数的多通道输入（非深度卷积、非逐点卷积）。
- *深度卷积 (DWCONV)：*对单个输入通道应用滤波器。
- *逐点卷积 (PWCONV)：*实现 1×1 卷积，用于通道间特征融合。
- *带残差连接的逐点卷积 (PWCONVRES)：*包含用于残差学习的跳跃连接。
- *转置卷积 (TCONV)：*支持上采样操作。

12.2.1.2 其他核心层

- *全连接 (FC)：*支持全连接/线性运算。
- *平均池化 (AVGPOOL)：*通过取平均值实现下采样。
- *最大池化 (MAXPOOL)：*通过选取最大值实现下采样。

12.2.1.3 灵活性

- 输入、输出和残差张量数据（特征图）均为 8 位整数，可为有符号或无符号。
- 权重可以是 2、4 或 8 位，且均为有符号。
- 未来将支持 4 位数据与 4 位权重的更多组合。
- 推理时仅支持批量大小为 1。
- 卷积层中的组数始终为 1，DWCONV 层除外。未来将支持分组卷积。
- 输入/输出通道的数量应为 4 的倍数，FCONV 输入除外。
- TCONV 层的步长必须与内核大小相同。
- 模型层数无限制。

- 层的输入可以与层的输出具有不同的符号类型和位宽。
- 层可以具有混合精度，例如，某层可以使用 8 位权重，而另一层使用 2 位权重。

12.3 模型复杂度和大小限制

实验表明，针对 f28p55x NPU 开发时，模型复杂度存在显著约束。了解这些限制对于在此平台上开发有效的神经网络应用至关重要。

12.3.1 内存约束和模型大小

- *PC 与嵌入式开发：*我们初始的模型每层 1024 和 512 个神经元，大小为 2.5-4.2MB。虽然这些模型在 PC 端成功运行，但完全无法安装在 F28p55x 上。
- *成功的架构：*最终，我们选择了一个小得多的模型，每个隐藏层有 64 个神经元，以在精度与性能间取得最优平衡：
  - *输入层：*1×64 权重 + 64 偏置 = 128 个参数
  - *隐藏层：*64×64 权重 + 64 偏置 = 4,160 个参数
  - *输出层：*64×1 权重 + 1 偏置 = 65 个参数
  - *总计：*4,353 个参数（远小于大架构模型）
- *物理约束：*与所有边缘计算器件一样，F28P55x 受限于有限的片上资源，因此在为该平台开发神经网络设计时，模型大小优化是一个关键考量因素。

12.3.2 优化流程和性能权衡

- *逐步缩减尺寸：*我们系统地测试了不同的模型规模，发现尽管每层 128 神经元的模型可放入器件，但 64 神经元配置在精度与延迟间达到最佳平衡。
- *内存与精度：*内存约束成为主要设计考量，迫使我们从硬件限制反向推导，而非从精度目标正向设计。
- *实现注意事项：*最终模型不仅能装进器件，还具备以下特点：
  - 可以通过 NPU 工具链稳定编译
  - 对正弦波逼近保持足够的精度
  - 推理时间稳定低于 1ms

13 基准测试

为了量化在 f28p55x NPU 上部署神经网络所涉及的性能权衡，我们针对不同的模型配置、部署平台和优化方法进行了全面的基准测试。这些基准测试为嵌入式应用选择模型架构时的实际考量提供了有价值的指导依据。

13.1 模型性能比较

性能评估采用以下关键指标：

- *延迟 (ms)：*处理单次推理所需的时间，以毫秒为单位。数值越低，响应越快，这对于实时应用至关重要。
- *吞吐量（样本/秒）：*每秒可处理的推理次数。数值越高，处理能力越强，对流式数据应用尤其重要。
- *平均绝对误差 (MAE)：*预测值与实际值之间绝对差异的平均值。数值越低，预测精度越高。
- *R^2 分数：*决定系数，衡量模型与数据的拟合程度。值越接近 1.0，预测性能越好，1.0 代表完全精准预测。
- *最大误差：*所有预测值与相应实际值之间的最大绝对差。值越低表示最坏情况下的性能越好。

对于每种神经元配置，评估了三种模型变体：

- *参考 Python 模型：*标准实现，仅可在 PC 上运行。
- *量化感知 ONNX 模型：*在 PC 上基于量化感知训练并以 ONNX 格式进行验证。
- *部署的 f28p55x 模型：*ONNX 模型经编译并部署在 f28p55x 硬件上。

13.1.1 128 神经元模型

表 13-1. Sine_128_Model 的基准测试

指标	F28p55x CPU	F28p55x NPU
延迟 [ms]	1.012	0.7116
样本/秒	987	1405
MAE	0.0015	0.0097
R2 分数	0.9999	0.9996
最大误差	0.01583	0.04407

13.1.2 64 神经元模型

表 13-2. Sine_64_Model 的基准测试

Metric	F28p55x CPU	F28p55x NPU
延迟 [ms]	0.2706	0.2146
样本/秒	3695	4659
MAE	0.017525	0.012144
R2 分数	0.997	0.9993
最大误差	0.19085	0.0979

13.1.3 16 神经元模型

表 13-3. Sine_16_Model 的基准测试

Metric	F28p55x CPU	F28p55x NPU
延迟 [ms]	0.0223	0.0252
样本/秒	44643	39557
MAE	0.1588	0.02029
R2 分数	0.88812	0.8451
最大误差	0.86379	0.9618

13.1.4 参考基准测试

为了进行比较，我们在 PC 端测试了大规模参考模型以建立精度基准：

表 13-4. 1024 神经元模型基准测试（参考模型）

Metric	参考 Python 模型（1024 神经元）
延迟 [ms]	0.149
样本/秒	7188
MAE	0.002171
R2 分数	1
最大误差	0.0054

该参考模型实现了近乎完美的精度，但其内存容量远超 f28p55x 的承受范围 (3.5-4.2MB)。

13.2 性能分析

性能评估采用若干核心指标，量化不同配置下的模型性能：

13.2.1 模型选择权衡

128 神经元模型代表了此应用的最高精度配置。该模型在 NPU 上运行时，R^2 分数为 0.9996，平均绝对误差 (MAE) 仅为 0.0097，精度优异。不过，就处理速度而言，这种精度会产生很大的代价，延迟为 0.7116ms，吞吐量仅为每秒 1,405 样本/秒。

64 神经元模型在精度与硬件约束间取得平衡。该模型保持了出色的精度 (R^2>0.99)，同时显著提高了处理速度。在 NPU 上运行时，64 神经元模型达到 4,659 样本/秒，是 128 神经元配置的吞吐量的三倍以上，而预测精度的下降则微乎其微。

较小的模型可显著提高吞吐量，但精度会大幅下降。16 神经元模型在 NPU 上以 39,557 样本/秒的速度执行 — 比 128 神经元模型快 28 倍 — 但 R^2 分数下降至 0.8451，这表明预测质量显著下降，对于许多对精度要求严苛的应用来说是不可接受的。

13.2.2 CPU 与 NPU 性能对比

CPU 与 NPU 执行之间的比较为实施决策提供重要指导依据：

*复杂模型的 NPU 优势：*对较大模型，NPU 可提供显著性能提升。128 神经元模型在 NPU 上的运行速度比在 CPU 上快 29.7%（延迟从 1.012ms 缩减至 0.7116ms），而 64 神经元模型则显示延迟缩减 20.7%。这一优势源于 NPU 专为神经网络并行计算设计的专用架构。

*CPU 对于简单模型的优势：*有趣的是，对于像 16 神经元这类极小模型，CPU 的性能实际上优于 NPU。相较于 NPU 的 39,557 个样本/秒，CPU 达到 44,643 个样本/秒，性能领先 12.9%。这种不合常理的结果源于与 NPU 之间的数据传输开销。16 神经元模型的计算工作量极少，以至于 CPU 可以直接在其原生执行环境中进行处理，避免了多个数据传输步骤。借助这样的小型模型，CPU 可以在单一执行上下文中完成整个推理，而不会产生 NPU 所需的内存传输开销。每次 NPU 推理都需要设置 DMA 传输、配置加速器、等待完成并检索结果 — 对于这种轻量级模型，这些操作总共消耗的时间超过了实际的神经网络计算时间。本质上，当模型如此之小时，专用硬件的使用“成本”超过其计算收益。

13.3 流水线级时序测量

信号处理流水线每一级的时序通过 F28P55x 上的硬件计时器进行测量。我们使用精密仪器进行了测量，以了解每个环节对总体系统延迟的影响。

表 13-5. 不同环节的流水线级时序

流水线阶段	时间 (ms)
CMPSS DAC 模块的软件写入延迟	0.0006
ADC 转换的硬件延迟	0.00084
NPU 处理（64 神经元模型）	0.21
缓冲器 DAC 的软件写入延迟	0.000593
总流水线延迟	0.212

这些精确的测量结果表明：

- 外设操作：CMPSS 模块、ADC 转换和 DAC 输出操作极快，均小于 1 微秒。
- NPU 主导：神经网络推理时间在流水线中完全占主导，占总延迟的 99.04%。在 64 神经元模型中，NPU 处理时间约为所有其他流水线环节总和的 210 倍。
- 固定成本：所有外设操作（CMPSS、ADC、DAC）的总时间仅为 0.002033ms，占流水线总延迟不足 1%。

14 总结

F28P55x 神经处理单元 (NPU) 代表了适用于汽车和工业应用的嵌入式机器学习能力的重大进步，可在不影响这些领域所需确定性性能的情况下实现器件上推理。本指南以实用的正弦函数逼近示例为载体，全面剖析了 NPU 的能力、约束及实现方法。

14.1 关键功能和约束

该 NPU 提供硬件加速的神经网络执行能力，性能显著优于主 CPU 上的软件实现方案，针对大型模型可带来 20-30% 的性能提升。这款基于整数运算的引擎可实现具有确定性行为的实时处理，同时与其他 C2000 外设保持无缝集成。

但这些能力也伴随重要约束，包括内存有限导致模型复杂度受限、架构对特定网络拓扑有偏好、量化带来精度折损。我们的基准测试表明，尽管 NPU 在大模型上性能优势显著，但极小神经网络

诸如 16 神经元模型等小型模型，因来往 NPU 的数据传输开销，在 CPU 上运行效率反而更高。

14.2 开发工作流程

实现流程遵循结构化工作流，涵盖模型开发、编译及应用集成。该方法首先进行量化感知训练，以使模型适配 NPU 的纯整数处理，随后通过 TI 神经网络编译器进行编译，生成硬件兼容的编译产物。随后将这些组件集成到 CCS 工程中，并配置相应外设，构建完整的信号处理流水线。

14.3 模型设计注意事项

通过系统性实验，本指南表明，成功的 NPU 落地实现需要精心的架构设计，以平衡精度要求与硬件约束。正弦函数示例揭示了有关模型大小调整的关键指导依据：

- 最优的 64 神经元架构在满足内存约束的前提下，实现了极佳的精度 ($R^2 > 0.99$)。
- 较大的模型（128 神经元）仅带来小幅精度提升，却导致吞吐量显著下降，但相较于 CPU 执行，仍能受益于 NPU 加速。
- 较小的模型（8-16 神经元）吞吐量更高，但精度损失显著。

14.4 实现挑战和解决方案

本指南解决了 NPU 实现过程中遇到的若干实际挑战：

- *负值与浮点值：*使用恰当的反量化技术与编译配置，处理取值范围为 [-1, 1] 的正弦值。
- *神经网络层支持：*设计模型时，选用受支持的层类型，同时规避不支持的运算。
- *内存限制：*系统性缩减模型尺寸，以适配硬件约束。

14.5 更广泛的应用

尽管以正弦函数逼近器为例，本指南的技术方法可拓展至各种汽车和工业应用，包括：

- 通过振动或声学信号分析进行预测性维护。
- 传感器数据流中的异常检测。
- 具备基于神经网络的建模功能的高级控制系统。
- 传感器融合，增强感知和决策能力。

F28P55x NPU 将机器学习直接引入到嵌入式控制系统中，构建在边缘独立运行的智能应用，同时保持汽车和工业环境中所需的可靠性。通过直接在微控制器上运行神经网络，系统可本地完成复杂决策，无需传输数据至其他地方。该方案响应时间可预测，对安全系统至关重要，且功耗低于传统方案。即使在云连接不可靠的恶劣环境中，预测性维护、传感器融合、异常检测等应用也可落地。NPU 在先进功能与关键控制系统严苛的运行要求之间取得了平衡，能够在不牺牲这类应用所需确定性行为的前提下赋予系统智能。

15 参考资料

- 德州仪器 (TI)，TMS320F28P55x 实时微控制器数据表，PDF
- 德州仪器 (TI)，EdgeAI Studio 网页
- 德州仪器 (TI)，Tinyml-Tensorlab，代码仓库
- 德州仪器 (TI)，TI MCU 神经网络编译器用户指南，网页
- 德州仪器 (TI)，面向 C2000 MCU 的 C2000Ware，C2000 SDK，网页
- 德州仪器 (TI)，F28P55X LaunchPad 开发套件，产品详情，网页

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、与某特定用途的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他安全、安保法规或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。对于因您对这些资源的使用而对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，您将全额赔偿，TI 对此概不负责。

TI 提供的产品受 TI 销售条款)、TI 通用质量指南 或 ti.com 上其他适用条款或 TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。 除非德州仪器 (TI) 明确将某产品指定为定制产品或客户特定产品，否则其产品均为按确定价格收入目录的标准通用器件。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

最后更新日期：2025 年 10 月