*Application Note*

# *Neural-Network Processing Unit (NPU) Guide*

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

**ABSTRACT**

This Neural-network Processing Unit Guide provides a comprehensive framework for deploying machine learning solutions on the F28P55x NPU, specifically designed for automotive and industrial applications. By leveraging this on-chip hardware accelerator, C2000™Ware customers can implement real-time inference for predictive maintenance, anomaly detection, sensor fusion, and advanced control systems while maintaining deterministic performance critical in these domains. Through a practical sine function approximation example, this guide walks engineers through the complete workflow—from architecture design to hardware validation—highlighting the NPU's capabilities despite memory and computational constraints. The documentation addresses quantization techniques essential for effective NPU utilization, compilation procedures using TI's toolchain, and integration strategies within CCS projects. Automotive and industrial customers will gain the practical knowledge needed to develop efficient embedded ML applications that meet the strict timing, power, and reliability requirements of specialized environments.

## Table of Contents

## Trademarks

# 1 Introduction

## 1.1 NPU Definition and Purpose

The F28P55x Neural Processing Unit (NPU) is a dedicated hardware accelerator integrated within TI's C2000 microcontroller architecture, specifically designed to execute neural network computations with high efficiency. This purpose-built silicon enables machine learning inference directly on the embedded device, eliminating the need for external processors or cloud connectivity. As an integral component of the C2000 ecosystem, the NPU works in concert with the main CPU, analog front-end, and control peripherals to enable sophisticated intelligence in real-time control systems. The NPU represents TI's commitment to bringing advanced machine learning capabilities to resource-constrained environments where deterministic performance remains paramount.

## 1.2 Key Capabilities

The F28P55x NPU delivers several critical capabilities that enhance C2000 applications:

- *Accelerated Neural Network Inference:* Hardware-optimized execution of neural network operations that significantly outperforms software implementations on the main CPU, with typical acceleration factors upto 70x depending on the model architectures.
- *Integer-Based Computation:* Specialized for efficient fixed-point arithmetic operations, enabling power-efficient inference processing optimized for embedded constraints.
- *Real-Time Processing:* Deterministic execution that maintains the predictable timing requirements essential for control systems in automotive and industrial applications.
- *Peripheral Integration:* Seamless operation with ADCs, DACs, and other C2000 peripherals enables complete signal processing and control workflows.
- *Parallel Operation:* Ability to perform neural network computations while the main CPU handles other tasks, maximizing system throughput.
- *Automotive/Industrial Focus:* Designed to meet stringent requirements for reliability, temperature range, and long-term availability needed in these demanding domains.

## 1.3 Technical Limitations

While powerful, the F28P55x NPU operates under several constraints that influence application design:

- *Architectural Limitations:* Neural Network topologies like CNNs and MLPs with ReLu activations are better supported compared to complex architectures such as LSTMs or Transformers.
- *Precision Tradeoffs:* Quantization necessary for NPU execution introduces precision loss compared to floating-point implementations, requiring careful training approaches to maintain accuracy.
- *Development Workflow Complexity:* Specific toolchain requirements for model compilation and deployment add additional development steps compared to standard microcontroller programming.

These capabilities and limitations frame the practical application space for the F28P55x NPU in automotive and industrial embedded systems, where balancing computational power with resource constraints is essential for successful implementation.

# 2 Development Flow Overview

The development workflow for F28P55x NPU applications follows a structured process that bridges machine learning model development with embedded system deployment. This section provides a high-level overview of the complete development cycle, which is explored in greater detail throughout subsequent chapters.

## 2.1 Model Development Phase

The NPU development journey begins with model design and training specifically optimized for embedded deployment:

- *Model Architecture Design:* Create neural network architectures that balance application requirements with NPU hardware constraints, typically favoring smaller networks with optimized layer types.
- *Dataset Preparation:* Curate representative training data that covers the full operational range expected in deployment, with particular attention to input normalization strategies compatible with quantization.
- *Quantization-Aware Training:* Implement training procedures that incorporate quantization effects during the training process, enabling the model to learn parameters that perform well under the integer-only constraints of the NPU.
- *Model Validation:* Verify model performance using metrics relevant to the target application, evaluating both accuracy and computational efficiency within the quantized environment.

## 2.2 Model Compilation Phase

Once trained, models must be transformed into a format compatible with the F28P55x NPU:

- *ONNX Export:* Convert trained models to the Open Neural Network Exchange (ONNX) format, which serves as the interchange standard for the compilation toolchain.
- *TI NPU Compiler Configuration:* Define compilation parameters through configuration files that specify target device, optimization strategies, and I/O requirements.
- *Compilation Execution:* Process the ONNX model through TI's Neural Network Compiler (TI MCU NNC) to generate C/C++ artifacts compatible with the NPU hardware.
- *Compilation Output Verification:* Validate the generated header files and libraries to ensure proper conversion, especially for critical aspects like dequantization in regression tasks.

## 2.3 Application Integration Phase

The final phase integrates the compiled model into a complete embedded application:

- *CCS Project Setup:* Establish a Code Composer Studio project incorporating the generated model artifacts alongside application-specific code.
- *Hardware Peripheral Configuration:* Configure necessary peripherals (ADC, DAC, communications interfaces) to provide inputs to and process outputs from the neural network.
- *Application Logic Implementation:* Develop the main application logic that coordinates data flow between peripherals and the NPU, including proper scaling of inputs and outputs.
- *Hardware Testing:* Verify end-to-end functionality on actual F28P55x hardware under realistic operating conditions.
- *Performance Optimization:* Fine-tune the application for optimal balance between inference speed, accuracy, and power consumption.
- *Deployment Packaging:* Prepare the final firmware package for production deployment in automotive or industrial environments.

# 3 Example Model Creation (Python)

This section demonstrates the practical application of NPU development principles through the creation of a sine function approximator. This example serves as a reference implementation that showcases the complete workflow from model design to deployment on the F28P55x NPU.

## 3.1 Model Selection Rationale

The sine function was deliberately chosen as a demonstration example for several compelling reasons:

- *Mathematical Complexity:* Despite the apparent simplicity, the sine function represents a non-linear mathematical relationship that requires neural network modeling capability beyond simple linear approximation.
- *Bounded Output Range:* With outputs constrained between -1 and +1, the sine function presents a well-defined range suitable for quantization strategies.
- *Visual Verification:* While this example uses oscilloscope waveform visualization (ideal for the sine function), other applications would require different verification methods appropriate to their specific tasks, such as confusion matrices for classification, heatmaps for anomaly detection, or error distribution plots for other regression tasks appear in control systems, signal processing, and motion control applications common in automotive and industrial domains.
- *Complete Pipeline Demonstration:* The sine function allows demonstration of a complete analog-to-digital-to-analog signal processing chain with the NPU as the central processing element.

## 3.2 Model Architecture Design

The sine function approximator implements a multilayer perceptron (MLP) with a deliberately constrained architecture for the F28P55x NPU:

```
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)))
```

***Code 1: Sine Approximator Backbone Architecture for Sine_64_Model***

This architecture incorporates several key design decisions specifically for NPU deployment:

- *Input Layer:* Single neuron input that accepts an angle value (mapped to 0-2π radians).
- *Hidden Layers:* Two hidden layers with 64 neurons each, chosen as a balance between accuracy and resource efficiency, though models with up to 128 neurons per layer can fit on the NPU.
- *Activation Functions:* ReLU activations selected for computational efficiency and quantization-friendly characteristics.
- *Output Layer:* Single neuron output that produces the predicted sine value (in range -1 to +1).

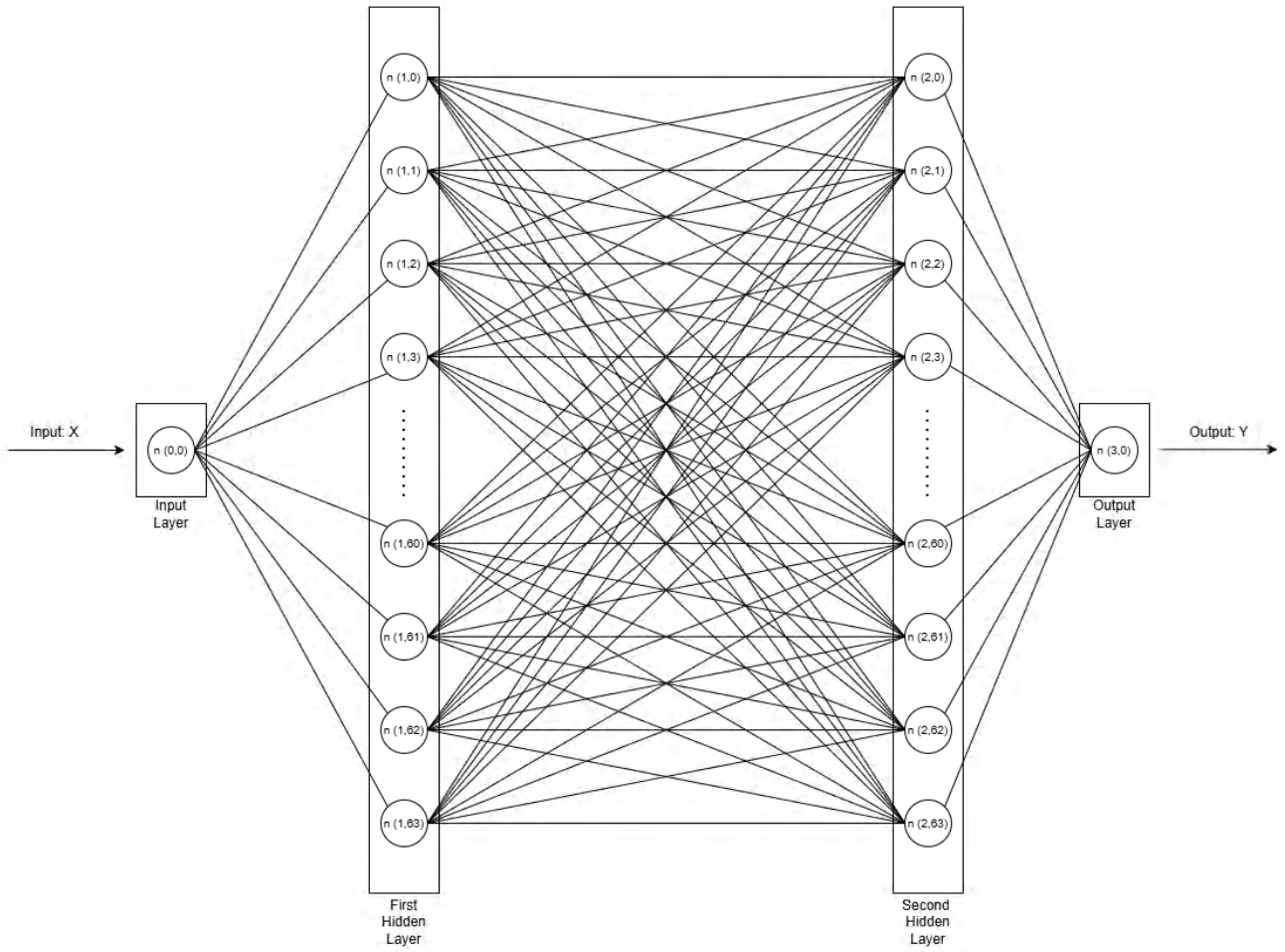



**Figure 3-1. Sine Approximator Backbone Architecture for Sine_64_Model**

The model architecture is deliberately sized based on the F28P55x NPU's memory and computational capabilities. The key constraint is the total parameter count, not a specific limit on neurons per layer. Our architecture (with layers sized as 1×64, 64×64, and 64×1) results in 4,353 total parameters, which represents a balance between accuracy and resource utilization. While models with up to 128 neurons per layer can fit on the device, the 64-neuron configuration was selected based on accuracy and latency considerations. Larger networks without feature extractor (such as those using 512 or 1024 neurons per layer) exceed the NPU's available memory. Engineers need to focus on the total parameter budget when designing models for this platform. This resource-constrained model design approach is common across all edge AI implementations, where limited memory, processing power, and energy budgets necessitate careful optimization of neural network architectures to achieve the best possible performance within the available hardware constraints.

## 3.3 Training Details

The model training follows a structured approach:

### *3.3.1 Development Environment Setup*

- *Example Model Setup:*
  - PyTorch® framework for neural network design and training.
  - TINPUTinyMLQATFxModule wrapper for quantization-aware training (more information on quantization is given in Section 4).
  - Supporting libraries for numerical computation and visualization.
- *Generic User Setup:*
  - Install TI's model optimization tools appropriate for your target hardware.
  - Maintain compatibility between your chosen framework version and TI's compilation toolchain.

  - Set up a consistent environment using virtual environments (conda, venv) to avoid dependency conflicts.
  - Include visualization libraries specific to your application domain (for example, signal plotting for sensors, image display for computer vision).

### 3.3.2 Dataset Generation

- *Example Model Dataset Details:*
  - Creation of 100,000 randomly generated angle values between 0 and 2π radians.
  - Calculation of corresponding sine values as training targets.
  - Train/test split with 80% training data and 20% validation data.
- *Generic User Approach:*
  - Generate or collect data that spans your full input domain to maintain robust model performance.
  - For sensor applications, capture data across all operating conditions and environmental variables.
  - Balance your dataset to avoid bias in trained models (especially for classification tasks).
  - Apply appropriate normalization based on the expected input range of your deployment scenario.
  - Use stratified sampling for classification tasks to maintain class distribution in train/test splits.
  - Consider adding controlled noise to training data to improve model robustness for real-world deployment.
  - For time-series data, maintain proper sequence handling during dataset preparation.

### 3.3.3 Model Training Configuration

- *Example Model Training Parameters:*
  - *Batch size:* 512 samples
  - *Learning rate:* 1e-5 (deliberately small for stable convergence)
  - *Optimizer:* Adam
  - *Loss function:* Mean Squared Error (MSE)
  - *Training epochs:* 160
- *Generic User Training Flow:*
  - Adjust batch size based on your model complexity and available memory (smaller for memory-constrained environments).
  - Select an appropriate learning rate based on your model size and convergence behavior.
  - *Choose optimizer based on your task requirements:*
    - Adam for general-purpose training and fast convergence
    - SGD with momentum for potentially better generalization in some cases
    - RMSprop for recurrent neural networks
  - *Select loss function appropriate to your task:*
    - MSE for regression problems
    - Cross-entropy for classification tasks
    - Custom loss functions for specialized applications
  - Implement learning rate scheduling to improve training stability and final model accuracy.
  - Consider early stopping to prevent overfitting, especially with limited training data.
  - Monitor both training and validation metrics to verify proper generalization.

### 3.3.4 Quantization-Aware Training Process

- *Example Model Quantization Process:*
  - Initial model wrapped in TINPUTinyMLQATFxModule to simulate quantization effects.
  - Forward passes include weight and activation quantization simulation.
  - Gradients account for quantization effects during backpropagation.
  - Regular validation to monitor quantized model performance.

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

**Code 2: Wrap model with quantization wrapper for fine tunning**

- *Generic User Quantization Flow:*
  - *Select quantization approach:*

    - QAT for higher accuracy (longer training)
    - PTQ for faster development (potential accuracy tradeoff)
- *Configure for target hardware:*
    - Use hardware-specific wrappers (for example, TINPUTinyMLQATFxModule)
    - Set appropriate bit-width (2/4/8 bits) and precision parameters
    - Set the number of epochs for finetuning
- *Optimize training process:*
    - Train initially with floating-point, then fine-tune with quantization
    - Benchmark against float baseline to assess accuracy impact
- *Mitigate quantization issues:*
    - Monitor activation ranges to prevent clipping
    - Maintain proper dequantization for regression outputs
    - Use weight/activation histograms to identify distribution problems

# 4 Quantization for Embedded Platform

Neural network quantization is the process of converting high-precision floating-point representations of weights and activations to lower-precision formats, typically integers. For the F28P55x NPU, this conversion is not merely an optimization but a fundamental requirement, as the hardware is designed specifically for integer-based computation.

In the context of our sine function approximator, quantization translates the continuous mathematical relationship into a form that the NPU can efficiently process while preserving the essential characteristics of the sine wave.

## 4.1 Quantization Approaches: QAT versus PTQ

Two fundamental approaches exist for quantizing neural networks: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). Understanding the differences between these approaches is crucial for selecting the appropriate strategy for F28P55x NPU deployment.

### *4.1.1 Post-Training Quantization (PTQ)*

PTQ applies quantization to a model that has already been trained using floating-point precision. This approach offers simplicity but can sacrifice accuracy, particularly for smaller models or precision-sensitive tasks.

*Key Characteristics:*

- *Process:* Train model normally → Calibrate quantization parameters → Convert to quantized format.
- *Calibration:* Uses a representative dataset to determine scaling factors.
- *Development Speed:* Faster development cycle (no retraining required).
- *Accuracy Impact:* Typically, higher accuracy loss compared to QAT.

*Advantages:*

- Simpler workflow with existing trained models.
- No need to modify training procedures.
- Faster deployment path.
- Lower computational requirements for development.

*Limitations:*

- Can result in significant accuracy degradation.
- Less control over quantization effects.
- Particularly challenging for regression tasks like our sine function.
- Limited ability to compensate for quantization artifacts.

### *4.1.2 Quantization-Aware Training (QAT)*

QAT incorporates quantization effects during the training process, allowing the network to learn parameters that perform best under quantized conditions. This approach generally preserves accuracy better but requires more development effort.

*Key Characteristics:*

- *Process:* Initial training → Insert quantization operations → Continue training → Convert to quantized format.
- *Simulation:* Simulates quantization effects during both forward and backward passes.
- *Development Speed:* Longer development cycle (requires additional training).

*Advantages:*

- Better preservation of model accuracy.
- Network learns to compensate for quantization effects.
- Particularly valuable for precision-sensitive applications.
- More predictable performance on quantized hardware.

*Limitations:*

- More complex development workflow.
- Requires additional training computation.
- Longer time-to-deployment.

- Requires careful hyperparameter tuning.

For our sine function approximator, we selected QAT due to the accuracy preservation in regression tasks, which was critical for maintaining the smooth sine wave characteristics across the entire input domain.

## 4.2 Quantization Frameworks and Wrapper Modules

The TI toolchain provides specialized wrapper modules that encapsulate different quantization approaches for various deployment targets. Understanding these wrappers is essential for selecting the appropriate quantization strategy for your application.

TI's quantization framework offers four distinct wrapper modules, created by combining two key dimensions:

- *Target Hardware:*
  - *Generic:* Optimized for CPU execution (standard integer operations)
  - *TINPU:* Optimized specifically for TI's Neural Processing Unit hardware
- *Quantization Approach:*
  - *QAT (Quantization-Aware Training):* Incorporates quantization effects during training
  - *PTQ (Post-Training Quantization):* Applies quantization after training completes

This creates a matrix of four wrapper options:

**Table 4-1. Quantization Wrappers**

| Target/Approach | QAT | PTQ |
|---|---|---|
| **Generic CPU** | GenericTinyMLQATFxModule | GenericTinyMLPTQFxModule |
| **TI NPU** | TINPUTinyMLQATFxModule | TINPUTinyMLPTQFxModule |

### *4.2.1 Generic Wrappers for CPU Quantization*

The Generic wrappers (GenericTinyMLQATFxModule and GenericTinyMLPTQFxModule) target CPU-based execution and utilize PyTorch's native quantization APIs:

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

***Code 3: Generic Quantization APIs***

These wrappers are good for:

- Deployment on C2000/ARM MCUs without NPU acceleration.
- Testing quantization effects before NPU-specific optimization.
- Applications with less stringent performance requirements.
- Model prototyping and initial verification.

The Generic wrappers utilize standard PyTorch quantization APIs but simplify the application through a consistent interface that requires minimal code changes to existing models.

### *4.2.2 TINPU Wrappers for NPU Hardware Acceleration*

The TINPU wrappers (TINPUTinyMLQATFxModule and TINPUTinyMLPTQFxModule) specifically target TI's Neural Processing Unit hardware accelerator:

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

***Code 4: TI NPU Specific Quantization APIs***

These wrappers are essential for:

- Deployment on F28P55x and other TI devices with NPU acceleration.
- Maximizing performance on TI hardware.
- Applications requiring real-time inference.

- Maintaining compatibility with NPU compilation tools.

The TINPU wrappers incorporate specific constraints of the TI NPU hardware, maintaining that models not only benefit from quantization in general but are specifically optimized for execution on the NPU architecture.

# 5 Validating the Model

The sine function approximator model underwent a two-phase validation process specifically designed to optimize performance on the F28P55x NPU hardware. This section details the validation methodology, quantization-aware training approach, and the resulting performance metrics that demonstrate the model's effectiveness.

## 5.1 Two-Phase Training Strategy

The training process employed a deliberate two-phase approach to maximize performance on the quantization-constrained NPU hardware:

### *5.1.1 Initial Training Phase*

- Model trained with normal floating-point weights.
- Focus on learning the underlying mathematical sine function.
- Establishing fundamental pattern recognition capabilities.
- Building a strong foundation before quantization constraints.

### *5.1.2 Fine-Tuning Phase*

- Model trained with F28P55x-specific quantized weights.
- Simulating the actual integer-only operations of the NPU.
- Optimizing weights specifically for quantized execution.
- *Quant_epoch* is normally set as *max(5,float_epoch/10)*.

This strategic approach leverages the best of both worlds: initial training with full floating-point precision establishes robust feature extraction, while targeted fine-tuning with quantized weights optimizes performance specifically for the deployment hardware.

## 5.2 Training Phase Comparison

Table 5-1 compares the performance metrics from the last epoch of normal floating-point training with those from the final epoch after quantization-aware fine-tuning:

**Table 5-1. Comparison of Metrics before and after Quantization Process**

| Metric | Float Training metrics after 160 epochs | QAT Fine-tuning metrics after 16 epochs | Change (%) |
|---|---|---|---|
| Validation Loss | 0.00181 | 0.00105 | Improved by 42% |
| Validation MAE | 0.02031 | 0.0215 | Degraded by 6% |
| Validation $R^2$ Score | 0.9963 | 0.9989 | Improved by 0.26% |

Standard floating-point models, despite exhibiting excellent performance in conventional environments, experience substantial accuracy degradation when constrained to the integer-only operations of the NPU. The QAT procedure simulates quantization effects throughout the training process, enabling the neural network to adapt parameters accordingly. This optimization verifies that when deployed to the F28P55x NPU, the model maintains computational integrity while leveraging the hardware's specialized neural network acceleration capabilities.

The observed improvements in Validation Loss (-42.0%) and $R^2$ Score (+0.26%) demonstrate that QAT can enhance specific performance metrics while simultaneously preparing the model for NPU deployment. The modest increase in MAE (+5.9%) illustrates the inherent trade-offs in the quantization process. These results validate the effectiveness of the quantization-aware training methodology for achieving optimal performance on resource constrained embedded hardware.

## 5.3 Validation Results and Metrics

The validation process employed a simple methodology to maintain objective performance assessment:

- *Validation Dataset:* 20% of the generated data held out from training.
- *Multiple Metrics:* Comprehensive assessment using complementary evaluation metrics.
- *Quantization Simulation:* Validation performed using the same quantization scheme as the target hardware.

The quantization-aware trained Sine_64 model achieved good performance metrics, as expected for a simple ML model, that validate its effectiveness for NPU deployment:

- Validation Loss is measure by Mean Squared Error (MSE). Mean Squared Error measures the average of squared differences between predicted values ($\hat{y}_i$) and actual values ($y_i$). Lower values indicate better model performance.

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

- Mean Absolute Error measures the average absolute difference between predicted and actual values. Unlike MSE, this does not square the errors, which is less sensitive to outliers.

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

- $R^2$ indicates how well the model explains the variance in the data compared to using the mean as a predictor. Values range from 0 to 1, with 1 representing perfect prediction. A validation $R^2$ score indicates the model explanation of variance of given data.

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

*Variables:*

- $y_i$ = Actual (true) sine value for the $i^{th}$ sample
- $\hat{y}_i$ = Predicted sine value from the neural network for the $i^{th}$ sample
- $\bar{y}$ = Mean of all actual sine values in the validation dataset

**Table 5-2. Training Validation Metrics**

| Metric | Metric | Significance |
|---|---|---|
| Validation Loss | 0.00038 | Extremely low mean squared error indicating high prediction accuracy |
| Validation MAE | 0.01305 | Mean absolute error of ~1.3% across the sine range (-1 to +1) |
| Validation $R^2$ Score | 0.9993 | Nearly perfect coefficient of determination, indicating the model explains 99.93% of the variance in sine values |

These metrics demonstrate several key achievements:

- **High Precision:** Despite quantization constraints, the model achieves sub-percent accuracy.
- **Consistent Performance:** Strong $R^2$ score indicates reliable prediction across the entire input range.
- **Quantization Resilience**: Minimal performance degradation despite integer-only operations.
- **Deployment Readiness:** Metrics validate the model's suitability for NPU implementation.

# 6 Testing the Model

After successful training and validation, comprehensive testing of the sine function approximator model is essential to verify performance in practical applications. This section details the testing methodologies, inference procedures, and results assessment techniques used to evaluate the model's readiness for NPU deployment.

## 6.1 Inference Setup and Methodology

Our sine function testing methodology uses a dedicated inference notebook with ONNX Runtime to validate the model before hardware deployment. The framework implements an encapsulated testing class providing a clean interface for predictions while maintaining consistent data handling.

The testing methodology follows a systematic approach:

- *Model Loading:* The ONNX model is loaded using the ONNX Runtime, providing access to the same model that eventually is compiled for the NPU.
- *Input Preparation:* Test points are generated across the entire input domain from 0 to 2π radians.
- *Reference Comparison:* Each prediction is compared against the actual mathematical sine function.

This testing approach verifies that the exported ONNX model, particularly the version with dequantization layers, correctly approximates the sine function across the entire range, serving as a critical quality gate before proceeding to hardware implementation.

### *6.1.1 Generic User Testing Approach*

- *Model Validation:* Use appropriate frameworks to test your exported model.
- *Representative Test Data:* Create test datasets that reflect your deployment conditions and edge cases.
- *Domain-Specific Metrics:* Select evaluation metrics that match your application requirements:
  - *Classification:* Accuracy, precision, recall, F1-score.
  - *Regression:* MAE, MSE, $R^2$ score.
  - *Signal processing:* SNR, cross-correlation, frequency response.
  - *Vision:* IoU, mAP for object detection, SSIM for image quality.
- *Performance Benchmarking:* Measure inference speed, memory usage, and power consumption.
- *Comparison Baselines:* Benchmark against reference algorithms where applicable.
- *Environmental Testing:* For critical applications, test across temperature ranges, voltage variations, or other environmental factors.

This validation serves as a critical quality gate before proceeding to hardware implementation, maintaining that your quantized model meets application requirements before investing in hardware deployment.

## 6.2 Testing Results and Visual Analysis

### *6.2.1 Visual Performance Assessment*

A key component of the testing process is the visual comparison between predicted and actual sine values. The notebook generates a comprehensive plot that overlays the model's predictions against the true sine function across the entire input domain.

This visualization immediately reveals the quality of the model's approximation. In the sine function example, the predicted curve closely follows the actual sine curve, with minimal visible deviation.

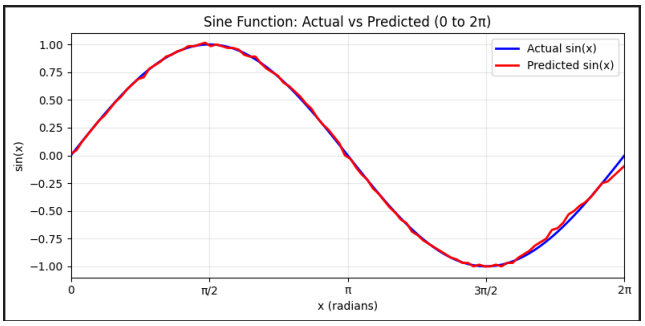



**Figure 6-1. Neural Network Sine Function Approximation Performance in Python**

This graph, denoted in Figure 6-1, demonstrates the performance of the trained neural network in approximating the sine function across the full input domain. The blue line represents the true mathematical sine function, while the red line shows the predictions generated by the quantization-aware neural network model when ran as an ONNX file. This graph demonstrates the performance of the trained neural network in approximating the sine function across the full input domain.

## 6.3 Quantitative Performance Metrics

Beyond visual inspection, the testing framework calculates comprehensive error metrics to quantify prediction accuracy with precision. The model's performance can be precisely quantified through the following error metrics obtained during testing:

- *Mean Absolute Error (MAE):* 0.013827
- *Maximum Error:* 0.093750
- *Minimum Error:* 0.000126

These quantitative results complement the visual assessment provided by the graph, offering numerical confirmation of the model's excellent approximation capabilities despite the constraints of Quantized training. The combination of visual and numerical validation provides complete confidence in the model's performance prior to hardware implementation.

# 7 Moving the Model to TI MCU (C2000 – F28P55x) [Beginner Level]

Deploying an ONNX model to the F28P55x can be accomplished through two distinct pathways, tailored to different user needs and expertise levels.

For those seeking a streamlined, efficient design, the beginner approach offers a simplified workflow with minimal configuration requirements. This method abstracts many technical complexities while delivering a functional implementation.

Users can follow Sections 1 and 2 of the *TI Neural Network Compiler for MCUs User's Guide - 2.1.0.LTS*, which provide step-by-step instructions for obtaining the compiled library and header files necessary for deployment.

# 8 Moving the Model to TI MCU (C2000 – F28P55x) [Developer Level]

The developer approach caters to engineers who require comprehensive control over the compilation process and deeper integration with existing systems. While more involved, this method provides complete visibility into each step of the deployment pipeline and allows for extensive customization of the compilation parameters. The following sections detail this workflow, empowering developers to optimize neural network implementations for specific application requirements.

This section details the complete workflow for transforming the mathematical model into hardware-compatible files that can be integrated into a Code Composer Studio project.

## 8.1 Compilation Prerequisites

Before initiating the model compilation process, several specialized tools and environments must be properly configured. This preparatory phase is critical for successful compilation and deployment.

### *8.1.1 Required TI Software Components*

The compilation toolchain relies on several TI-specific repositories and tools:

- *tinyml-modelmaker*
  - *Purpose:* Provides a consistent interface for model development and compilation.
  - *Function:* Orchestrates the compilation workflow and integrates with other components.
- *tinyml- modeloptimization*
  - *Purpose:* Model quantization and optimization toolkit.
  - *Function:* Provides wrappers and tools for preparing models specifically for TI-NPU deployment.
- *C2000Ware SDK*
  - *Purpose:* Platform-specific drivers, libraries and tools for C2000 microcontrollers.
  - *Function:* Provides essential drivers and hardware abstraction for the F28P55x platform.
- tinyml-modelzoo
  - *Purpose*: Provides a curated collection of neural network models optimized for embedded systems.
  - Function: Necessary for model compilation via tinyml-modelmaker method.

### *8.1.2 Environment Setup Process*

Setting up the compilation environment follows a structured approach:

- *Clone Required Repositories:*
  - 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
  - *Run Initialization Script:*
    - *cd into **tinyml-modelmaker** repo*
    - Execute *setup_all.sh* which orchestrates the complete environment setup
    - Execute *setup_c2000ware.sh* which configures the C2000Ware SDK
    - Execute *setup_cg_tools.sh* which sets up TI Code Generation Tools
- *Python Environment Setup:*
  - A dedicated Python virtual environment is recommended to avoid dependency conflicts.
  - Key Python packages required include:
    - ONNX and ONNX Runtime
    - TVM (TI MCU NNC) framework
    - NumPy and related numerical libraries

## 8.2 Configuration File Setup

The compilation process is controlled by a *config.yaml file* that defines critical parameters for transforming the ONNX model into NPU-compatible code. This configuration approach provides flexibility while maintaining consistency in the compilation workflow.

### 8.2.1 Configuration File Structure

A typical config.yaml file for compiling the sine function model includes the following key sections:

```
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"
```

***Code 5: Example Configuration File***

Each configuration section serves a specific purpose in guiding the compilation process:

- *Common Settings:* Define fundamental task type and target hardware.
  - *task_type* identifies the machine learning task category.
  - *target_device* activates F28P55x-specific optimizations.
- *Dataset Settings:* Control dataset processing behavior.
  - Typically disabled when using pre-trained models.
- *Training Settings:* Configure training parameters.
  - *output_int:* Setting this as False is critical for regression tasks to verify proper floating-point outputs.
  - Training is typically disabled for pre-trained models.
- *Compilation Settings:* Control model transformation.
  - *enable:* True activates the compilation phase.
  - *model_path* specifies the location of the ONNX model with dequantization layers.

#### 8.2.1.1 Models Requiring Dequantization Flag

The *output_int:* False setting is essential for:

- *Regression Models:* Any model predicting continuous values, such as:
  - Time-series forecasting (temperature, pressure, voltage predictions)
  - Control system modeling (PID coefficient estimation)
  - Function approximation (like our sine example)
  - Signal filtering applications
  - Sensor calibration models
- *Normalization-Heavy Models:* Applications where output scaling is significant:
  - Models with outputs normalized to specific ranges
  - Sensor fusion algorithms with calibrated outputs
  - Physical quantity estimation (force, torque, etc.)
- *Multi-Output Models:* When some outputs require floating-point precision:
  - Combined classification/regression models
  - Pose estimation (angles require floating-point)
  - Coordinate regression (object localization)

Without this flag, these models produce quantized integer outputs unsuitable for applications requiring continuous value ranges or precise fractional outputs.

### 8.2.2 Special Configuration for Regression Models

Regression tasks like sine function approximation require special handling to maintain proper dequantization of the integer-only NPU outputs back to floating-point values.

#### 8.2.2.1 Output Dequantization Flag

- The *output_int:* False setting is mandatory for regression tasks.
- This instructs the compiler to preserve dequantization operations in the generated code.
- Without this flag, the model produces integer outputs unsuitable for sine approximation.

#### 8.2.2.2 Compiler Constants Modification

In addition to the configuration file, the compiler requires modification to support floating-point outputs.

- Locate and open *tinyml-modelmaker/ai_modules/timeseries/constant.py*.
- Scroll through the file until you find the *COMPILATION_C28_HARD_TINPU* dictionary:

```
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,
)
```

***Code 6: Reference Constant for Non- Regression Models***

- This existing constant is configured for classification tasks, where:
  - *type=hard* specifies hardware NPU acceleration (versus CPU)
  - *skip_normalize=true* bypasses normalization/scaling
  - *output_int=true* forces integer outputs
- Create a new constant named *COMPILATION_C28_HARD_TINPU_TEMP* by duplicating the existing one with critical modifications:

```
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,
)
```

***Code 7: Extra Constant added to handle Regression Models***

- The following parameters are modified to accommodate a Regression Model:
- *type=hard:*
  - Kept unchanged. This directs compilation to target the NPU hardware accelerator, rather than *soft* which would use the CPU for neural network calculations.
- *skip_normalize=false*:
  - The example application didn't need normalization since it is a simple model created as a demo.
- *output_int=false:*
  - The compiler produces floating-point outputs rather than integers.
  - Preserves decimal precision essential for regression tasks.
  - Allows representing the full continuous range of values.
- *target_c_mcpu='c28'*: Kept unchanged - Specifies generation of code for the C28x architecture.
- *cross_compiler=CL2000_CROSS_COMPILER:* Kept unchanged. Defines which compiler toolchain to use.

#### 8.2.2.3 Compilation Dictionary Update

The default compilation dictionary must be modified to use the custom constants:

- Change the constant being used- *COMPILATION_C28_HARD_TINPU* to *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, )
),
```

***Code 8: Changes made to accommodate regression models***

## 8.3 Compilation Process Flow

The transformation of an ONNX model into F28P55x NPU-compatible code involves a multi-stage process executed by the TI MCU Neural Network Compiler (TI MCU NNC). This section details the step-by-step compilation workflow and internal transformations.

### *8.3.1 Launching the Compilation*

Once the configuration file is properly set up, compilation is initiated using the TinyML modelmaker script:

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

***Code 9: Command to compile ONNX file to embedded compatible file***

For the sine function example, the command is:

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

***Code 10: Example command to trigger compilation***

This command triggers the complete compilation pipeline that transforms the ONNX model into C/C++ code optimized for the F28P55x NPU.

### *8.3.2 Compilation Phases*

The compiler processes the model through three main phases:

- *Model Validation*
  - The compiler checks if the ONNX model can run on the NPU.
  - Confirms all operations are supported (like the linear layers and ReLU in our sine model).
  - Verifies the model fits within the NPU memory limits.
- *Model Transformation*
  - Converts the ONNX model into a format optimized for the NPU.
  - Applies quantization parameters for integer-based computation.
  - For regression models like our sine approximator, preserves dequantization information.
- *Code Generation*
  - Creates C/C++ code that can run on the F28P55x
  - Generates two main files:
    - Header file (tvmgen_default.h) with function declarations
    - Library file (mod.a) containing the compiled model

### *8.3.3 Common Issues to Watch For*

While the compiler handles most of the complexity, watch for these common messages:

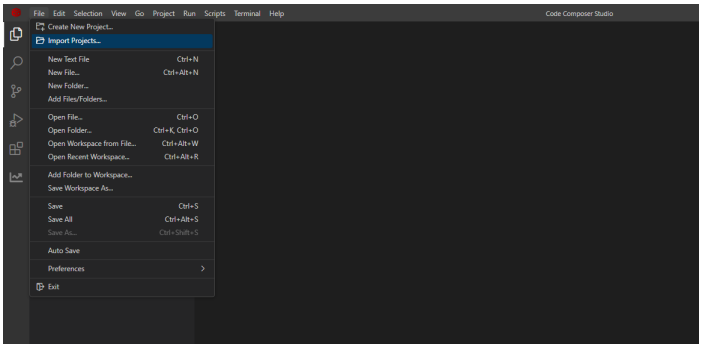
- *Unsupported operation:* Your model contains an operation the NPU can't run.
- *Memory constraint exceeded:* Your model is too large for the edge device.
- *Dequantization error:* For regression models, output dequantization may need attention.

# 9 Setting up the MCU Project

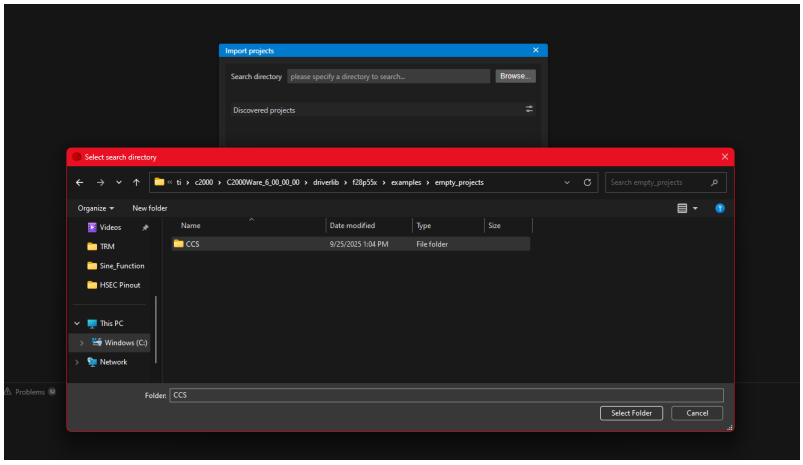
After successfully compiling the neural network model into NPU-compatible files, the next critical phase involves integrating these components into a Code Composer Studio (CCS) project for execution on the F28P55x microcontroller. This section guides engineers through establishing the project structure, configuring the development environment, and implementing the application framework necessary to leverage the NPU's capabilities.

## 9.1 Creating a CCS Project for NPU Applications

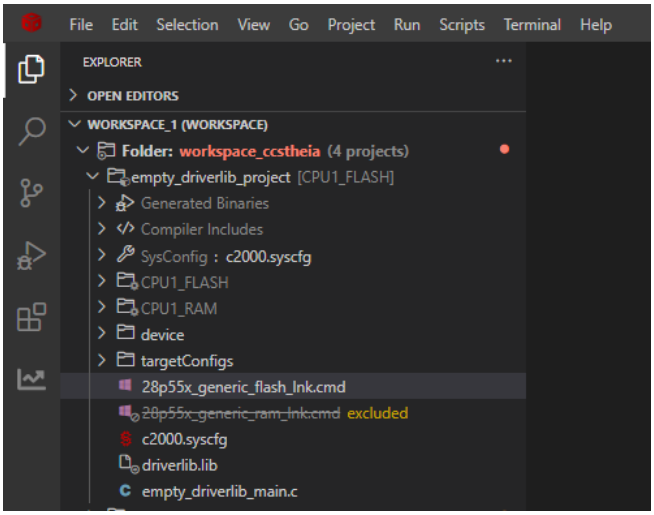
- *Step 1:* Open CCS, click on *File* and *Import Projects*.



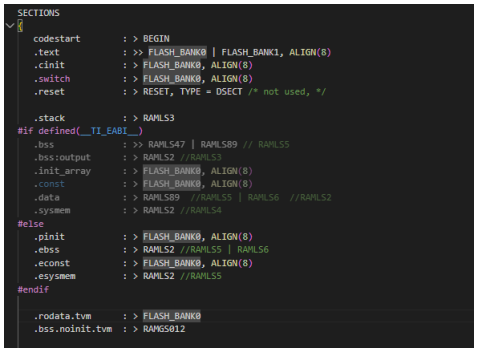

- **Step 2:** Import the *empty_project* from C2000Ware SDK.



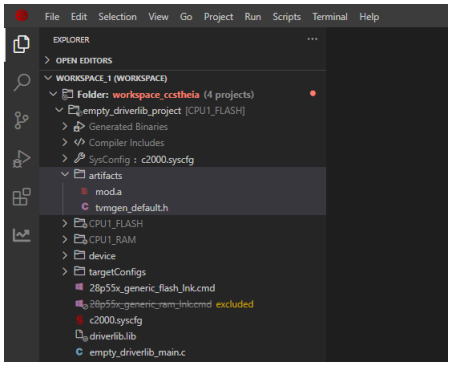

- *Step 3:* Open the *f28p55x_generic_flash_lnk.cmd* file in the imported project



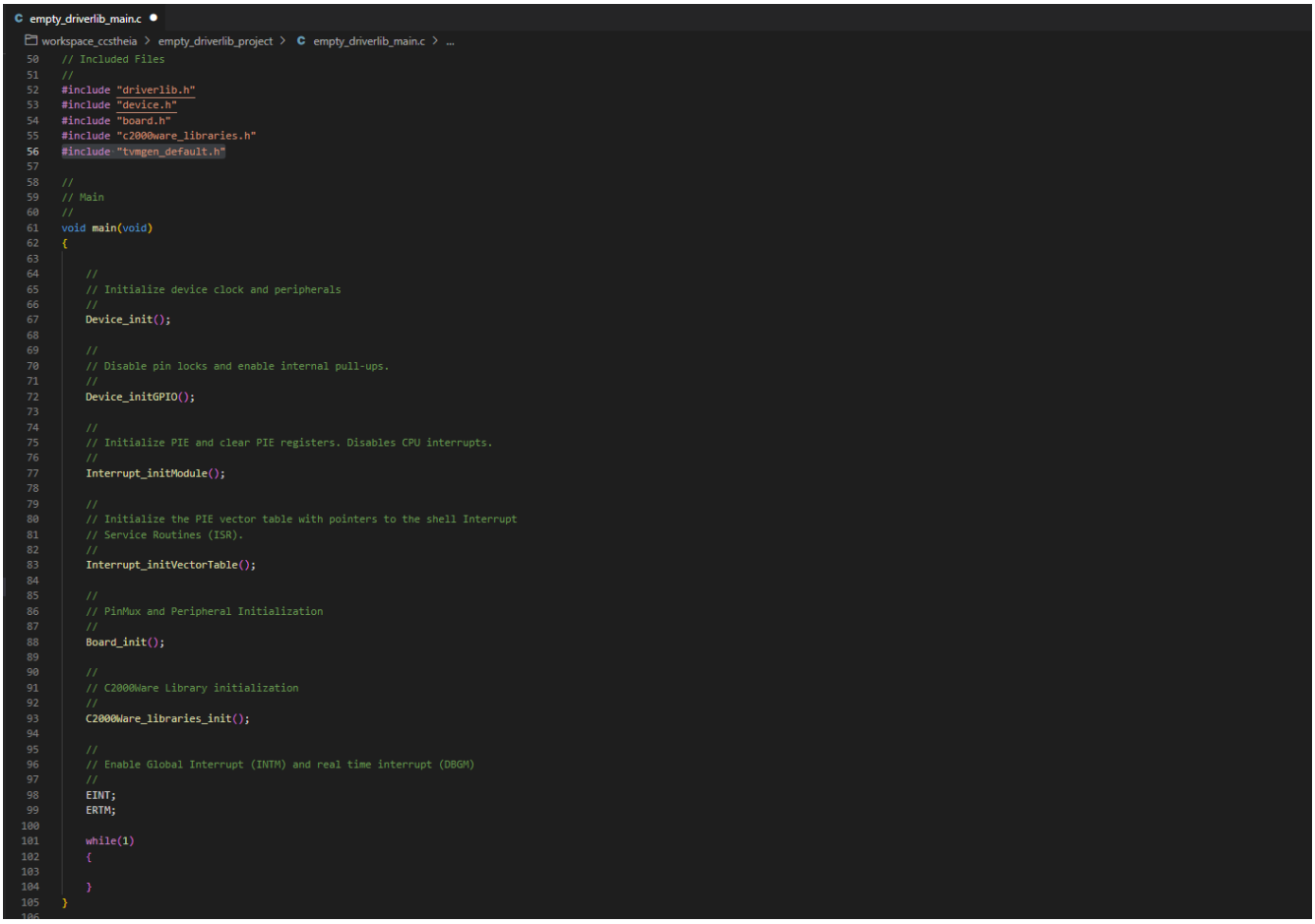
- *Step 4:* Add .rodata.tvm section in FLASH, and place the .bss.noinit.tvm section in global shared SRAM so that hardware NPU can access it. For example, RAMGS0, RAMGS1, RAMGS2, or RAMGS3 on an F28P55x MCU device.




- *Step 5:* Create a folder within the project called *artifacts* and place the compiled library and header file in the folder.




- *Step 6:* Include the header file in the application and proceed to write an application that uses the ML model running on the NPU. Use the NPU interface detailed in the next section.




## 9.2 Understanding the NPU Interface

The header file (tvmgen_default.h) provides critical structures and functions that serve as the interface between your application and the NPU. Understanding these components is essential for successful neural network integration.

### *9.2.1 Key Interface Components*

The header file defines several essential elements:

- *Input and Output Structures*: These structures provide the mechanism to pass data to and from the neural network.

```
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
};
```

***Code 11: Input and Output structures generated during the compilation process***

- *Initialization Function:* This function must be called once at application startup to configure the NPU hardware.

```
void TI_NPU_init();
```

***Code 12: NPU initialization API***

- *Execution Function:* This function triggers neural network execution on the NPU.

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

***Code 13: API to trigger NPU***

- *Completion Flag:* This flag indicates when neural network processing is complete.

```
extern volatile uint8_t tvmgen_default_finished;
```

***Code 14: Flag to denote end of NPU processing***

### *9.2.2 Basic Usage Pattern*

The typical workflow for using the NPU involves:

- *One-time Initialization:* Call TI_NPU_init() during system startup.
- *Input Preparation:* Create and populate the input structure with pointers to your data.
- *Output Allocation:* Create the output structure with pointers to variables that receive results.
- *Model Execution:* Call tvmgen_default_run() with the input and output structures.
- *Completion Monitoring:* Monitor the tvmgen_default_finished flag to determine when processing is complete.
- *Result Utilization:* Once the completion flag is set, the output data is ready for use in your application.

# 10 Testing the Model in the Embedded Environment

After successful training and model conversion, comprehensive testing of the sine function approximator directly on the F28P55x microcontroller is essential to verify performance in the actual deployment environment. This section details the testing methodology and results obtained from running the neural network model on the target hardware.

## 10.1 Visual Performance Assessment

A critical component of the testing process is the visual comparison between predicted and actual sine values. The CCS graphing tool was used to plot both the neural network predictions and the true sine function across the entire input domain.

Figure 10-1 presents the visual comparison between the NPU-generated sine approximation and the true sine function values:

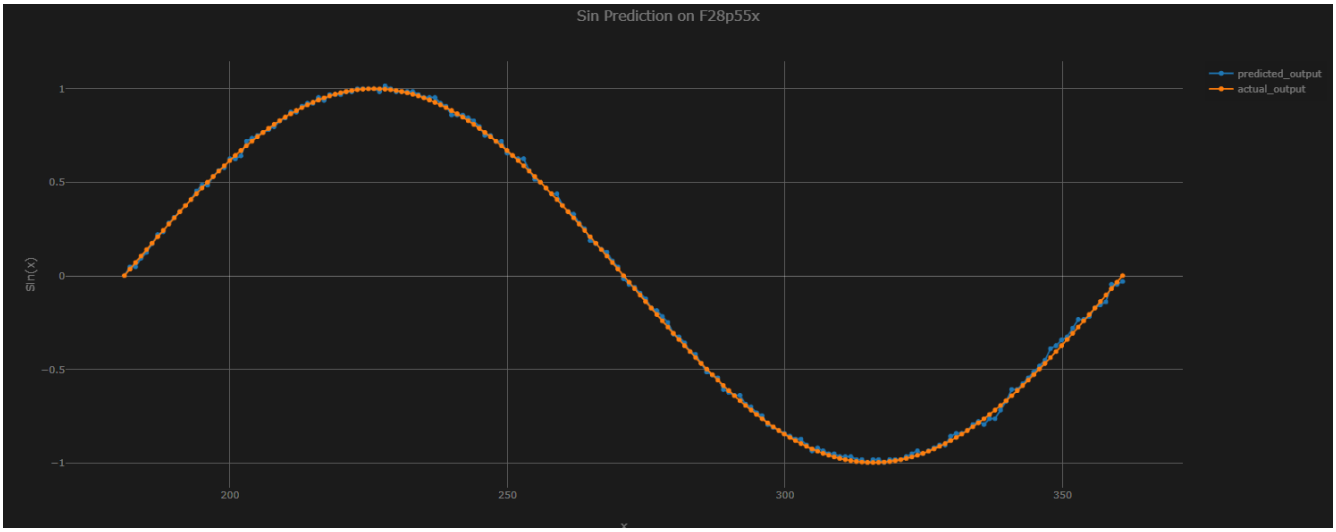



**Figure 10-1. Neural Network Sine Function Approximation Performance on F28p55x**

The visualization demonstrates several key aspects of the model's performance on the actual hardware:

- *Consistent Performance:* The model maintains high accuracy throughout the entire waveform, including at critical points such as extrema (peaks and troughs) and zero-crossings.
- *Smooth Transitions:* Despite the quantized nature of the NPU's computations, the approximation maintains smooth transitions throughout the curve, with no visible discontinuities or artifacts.
- *Complete Cycle Coverage:* The test evaluates a complete cycle of the sine function from 0 to 360 degrees, confirming consistent performance across all phases of the waveform.

The visual results provide compelling evidence that the model successfully transfers from training to the F28P55x hardware with minimal degradation in accuracy. The nearly indistinguishable overlay of predicted and actual values demonstrates the effectiveness of the quantization-aware training approach in preserving model fidelity through the compilation process.

## 10.2 Quantitative Performance Metrics

Beyond visual inspection, comprehensive quantitative metrics were calculated to provide objective assessment of the model's performance on the F28P55x NPU hardware. The quantitative evaluation generated the following key performance metrics:

- *Mean Absolute Error (MAE):* 0.01214
- *Maximum Error:* 0.0973
- *Latency (ms):* 0.214
- $R^2$ *Score:* 0.9997

These metrics provide a comprehensive evaluation of both accuracy and performance. The error metrics quantify the precision of the model's approximation, while the timing measurements demonstrate the efficiency of the NPU hardware compared to software-based execution on the main CPU. This combination of accuracy and speed enables real-time applications with stringent timing and precision requirements common in automotive and industrial control systems.

# 11 NPU Integration in a Real-Time Signal Chain

While initial model validation using pre-defined datasets provides valuable insights into neural network performance, the true test of an NPU application lies in the ability to process real-time data. This section explores the implementation of a complete real-time signal processing chain that transforms live analog inputs into meaningful outputs through neural network inference on the F28P55x NPU.

## 11.1 Application Block Diagram

Figure 11-1 presents a comprehensive diagram of the simulation environment created for validating the Neural Processing Unit (NPU) implementation on the F28P55x microcontroller. This architectural diagram illustrates the complete signal flow from generation through processing to visualization, demonstrating how the neural network model transforms a sawtooth wave into a sine wave in real-time.

- *Signal Generation:* The leftmost section shows the CMPSS module configured as a DAC, generating a sawtooth waveform with an amplitude range of 0-360 units. This waveform, visually represented by the zigzag pattern in the box above, serves as the input stimulus for the system.
- *Signal Acquisition:* The sawtooth wave feeds into an ADC (Analog-to-Digital Converter) block, which digitizes the analog signal. The output maintains the same numerical range (0-360) but is now in the digital domain.
- *Neural Network Processing:* The central portion of the diagram shows the F28P55x microcontroller containing the NPU and the sine model. The input value (labeled as 'x') is passed to the sine model, which computes the corresponding sine value.
- *Output Generation:* The neural network output, ranging from -1 to 1 (the natural range of sine values), is directed to a second DAC for conversion back to the analog domain.
- *Visualization:* The rightmost section shows the final output being displayed on an oscilloscope, revealing a smooth sine wave pattern that visually confirms the successful transformation from sawtooth to sinusoidal waveform.

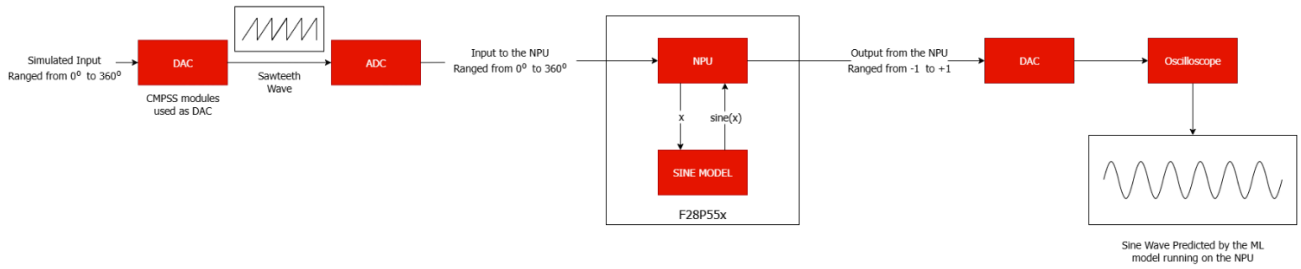



**Figure 11-1. Flow of data on application side**

## 11.2 Application Code Implementation

The following code demonstrates how to implement the complete signal processing pipeline, integrating the CMPSS, ADC, NPU, and DAC components to transform a sawtooth wave into a sine wave using the neural network model:

```
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
    }
}
```

***Code 15: Main Application Code for Real-Time Neural Network-Based Sine Wave Generation on F28P55x***

## 11.3 Hardware Components Utilized

The simulation environment leverages several key hardware components of the F28P55x platform:

- *CMPSS DAC (Comparator Subsystem):* Although typically used for comparator functions, this block is repurposed as a signal generator. The CMPSS modules include internal 12-bit DACs that are programmatically incremented to create the sawtooth pattern. This approach demonstrates the flexibility of the F28P55x peripherals for test signal generation without requiring external hardware.
- *ADC Module:* The F28P55x's integrated 12-bit Analog-to-Digital Converter samples the generated waveform. In real-world applications, this typically connects to external sensors or signal sources, but in this simulation environment, it samples the internally generated waveform to create a complete signal path.
- *NPU Hardware Accelerator:* The dedicated Neural Processing Unit within the F28P55x executes the sine model computations with higher performance and efficiency than is possible using the main CPU alone. The diagram shows how the NPU is integrated into the processing chain, receiving digital inputs and producing computed outputs. 'Sine Model' represents the neural network model that has been trained, compiled, and deployed to the NPU. The model contains the weights and structure necessary to transform input values to corresponding sine values.
- *Buffered DAC:* A separate DAC channel converts the neural network's floating-point outputs back to analog signals for observation. This DAC differs from the CMPSS DAC in the typical usage pattern and in performance characteristics.

## 11.4 Hardware Validation Results

The oscilloscope capture shown in Figure 11-2 provides definitive validation of the entire NPU application, demonstrating the successful real-time transformation from sawtooth to sine wave through neural network inference on the F28P55x platform.

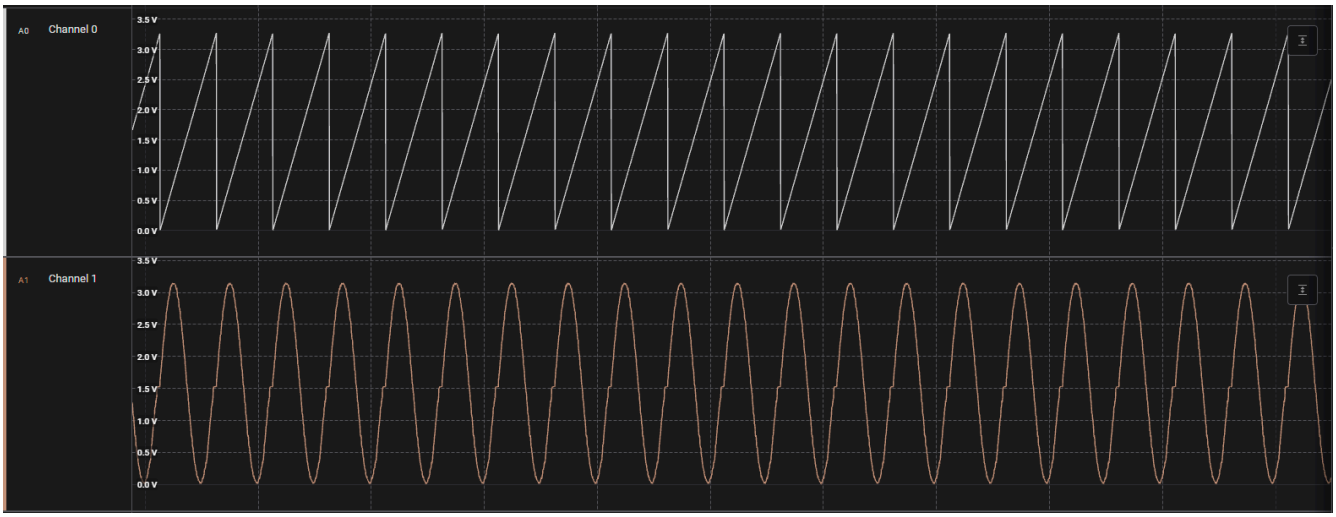



**Figure 11-2. Application Validation on Oscilloscope**

### *11.4.1 Input Signal Characteristics*

The top trace (Channel 0) displays the sawtooth waveform generated by the CMPSS DAC:

- Clean, linear ramps from 0V to 3.3V with sharp resets.
- Consistent period and amplitude across all cycles, indicating stable signal generation.
- The sharp transitions at the reset points provide challenging test conditions for the neural network, as these represent discontinuities in the input function.

This linear ramp input serves as the ideal test signal, as it systematically sweeps through the entire input domain of the neural network in each cycle, providing comprehensive coverage of the model's operational range.

### *11.4.2 Neural Network Output Analysis*

The bottom trace (Channel 1) shows the sine waveform produced by the neural network and output through the DAC:

- Smooth sinusoidal waveform with amplitude ranging from 0V to 3.3V.
- Consistent amplitude and frequency across all visible cycles.
- Minimal distortion at the peaks and troughs, indicating excellent approximation at critical points.
- Proper phase relationship with the input sawtooth wave, with each sawtooth cycle corresponding to exactly one sine cycle.
- Slight visible imperfections at some zero-crossing is due to linearity constraints of CMP1_DACL effectively using only a 10-bit accurate source.

The quality of the sine output is remarkable, displaying smooth transitions and well-formed waveforms that closely match the theoretical sine function. The consistency across multiple cycles demonstrates that the neural network is producing reliable, repeatable results in continuous operation.

# 12 Key Design Decisions and Impact

## 12.1 NPU Handling of Numbers

The f28p55x NPU is fundamentally designed for integer-based processing, which presents challenges when working with both negative values and floating-point data. However, with appropriate modifications to the neural network architecture and compilation process, the NPU can effectively handle these data types. This document provides comprehensive designs for addressing these limitations.

### *12.1.1 Integer-Only Architecture*

- The NPU hardware is optimized exclusively for integer arithmetic operations.
- Native support exists only for unsigned integer calculations.
- This core design choice maximizes processing efficiency for embedded applications but requires special handling for other data types.

### *12.1.2 Working with Negative and Floating-Point Values*

- *Quantization Process:* Both negative values and floating-point data require quantization to map them to the NPU's integer representation
- *Automatic Conversion Layers:* The compilation process automatically inserts quantization layers for inputs and dequantization layers for outputs
- *Configuration Requirements:* The compilation pipeline must be explicitly configured to preserve the full range of values:

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

***Code 16: Flag to enable dequantized float output***

- *Special Compiler Flags:* For applications requiring floating-point or negative outputs (like our sine function), additional compilation flags are needed:

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

***Code 17: Flag to disable normalization***

- *Processing Overhead:* The additional quantization/dequantization layers add minor computational overhead.

## 12.2 Supported Neural Network Layers and Constraints

The F28p55x NPU supports a specific set of neural network layer types with particular constraints. Understanding these capabilities is essential for designing models that can be successfully compiled and deployed on this hardware.

### *12.2.1 Supported Layer Types*

#### 12.2.1.1 Convolution Layers

- *First Convolution (FCONV):* Supports single-channel input feature maps (not depth-wise, not point-wise).
- *Generic Convolution (GCONV):* Handles multi-channel inputs in multiples of 4 (not depth-wise, not point-wise).
- *Depth-Wise Convolution (DWCONV):* Applies filters to individual input channels.
- *Point-Wise Convolution (PWCONV):* Implements 1×1 convolution for channel-wise mixing.
- *Point-Wise Convolution with Residual (PWCONVRES):* Includes skip connections for residual learning.
- *Transposed Convolution (TCONV):* Supports upsampling operations.

#### 12.2.1.2 Other Core Layers

- *Fully-Connected (FC):* Supports dense/linear operations.
- *Average Pooling (AVGPOOL):* Downsamples using averaging.
- *Max Pooling (MAXPOOL):* Downsamples by selecting maximum values.

#### 12.2.1.3 Flexibilities

- Input, output, and residual tensor data (feature maps) are 8-bit integers, signed or unsigned.
- Weights can be 2, 4, or 8-bit and are always signed.

- Further combinations of 4-bit data and 4-bit weights will be supported in the future.
- Only a batch size of 1 is supported for inference.
- The number of groups in the convolution layers is always 1, except for the DWCONV layer. Grouped convolution will be supported in the future.
- The number of input/output channels should be a multiple of 4, except for FCONV input.
- The TCONV layer's strides must be the same as the kernel size.
- There is no limit on the number of layers in the model.
- A layer's input can have different sign-ness and bit-width than the layer's output.
- Layers can have mixed precision, for example, one layer may use 8-bit weights, while another layer uses 2-bit weights.

## 12.3 Model Complexity and Size Limitations

The experimentation revealed significant constraints on model complexity when targeting the f28p55x NPU. Understanding these limitations is essential for developing effective neural network applications on this platform.

### *12.3.1 Memory Constraints and Model Size*

- *PC versus Embedded Development:* Our initial models with 1024 and 512 neurons per layer were 2.5-4.2MB in size. While these models ran successfully on PC, the models completely failed to fit on the F28p55x.
- *Successful Architecture:* We ultimately settled on a much smaller model with 64 neurons per hidden layer as the best balance between accuracy and performance:
  - *Input layer:* 1×64 weights + 64 bias = 128 parameters
  - *Hidden layer:* 64×64 weights + 64 bias = 4,160 parameters
  - *Output layer:* 64×1 weights + 1 bias = 65 parameters
  - *Total:* 4,353 parameters (dramatically smaller than larger architectures)
- *Physical Constraints:* As with all edge computing devices, the F28P55x is constrained by limited on-chip resources, making model size optimization a critical consideration when developing neural network designs for this platform.

### *12.3.2 Optimization Process and Performance Trade-offs*

- *Progressive Size Reduction:* We systematically tested different model sizes and found that while models with up to 128 neurons per layer can fit on the device, the 64-neuron configuration emerged as the best balance point for accuracy and latency.
- *Memory versus. Accuracy:* Memory constraints became our primary design consideration, forcing us to work backward from hardware limitations rather than forward from accuracy goals.
- *Implementation Considerations:* The final model not only fit on the device but also:
  - Compiled reliably with the NPU toolchain
  - Maintained adequate precision for sine wave approximation
  - Delivered consistent inference times below 1ms

# 13 Benchmarks

To quantify the performance trade-offs involved in deploying neural networks on the f28p55x NPU, we conducted comprehensive benchmarking across different model configurations, deployment platforms, and optimization approaches. These benchmarks provide valuable insights into the practical considerations when selecting model architectures for embedded applications.

## 13.1 Model Performance Comparison

The performance evaluation employs the following key metrics:

- *Latency (ms):* The time required to process a single inference, measured in milliseconds. Lower values indicate faster response times critical for real-time applications.
- *Throughput (samples/sec):* The number of inferences that can be processed per second. Higher values indicate better processing capacity, particularly important for streaming data applications.
- *Mean Absolute Error (MAE):* The average of the absolute differences between predictions and actual values. Lower values indicate higher prediction accuracy.
- *$R^2$ Score:* Coefficient of determination, measuring how well the model fits the data. Values closer to 1.0 indicate better predictive performance, with 1.0 representing perfect prediction.
- *Maximum Error:* The largest absolute difference between any prediction and the corresponding actual value. Lower values indicate better worst-case performance.

Three model variants were evaluated for each neuron configuration:

- *Reference Python Model:* Standard implementation that can only run on PC.
- *Quantization-Aware ONNX Model:* Trained with quantization awareness and validated as ONNX on PC.
- *Deployed f28p55x Model:* The ONNX model compiled and deployed on the f28p55x hardware.

### *13.1.1 128 - Neuron Model*

**Table 13-1. Benchmark for Sine_128_Model**

| Metrics | F28p55x CPU | F28p55x NPU |
|---|---|---|
| Latency [ms] | 1.012 | 0.7116 |
| Samples/sec | 987 | 1405 |
| MAE | 0.0015 | 0.0097 |
| R2 Score | 0.9999 | 0.9996 |
| Max Error | 0.01583 | 0.04407 |

### *13.1.2 64 - Neuron Model*

**Table 13-2. Benchmark for Sine_64_Model**

| Metric | F28p55x CPU | F28p55x NPU |
|---|---|---|
| Latency [ms] | 0.2706 | 0.2146 |
| Samples/sec | 3695 | 4659 |
| MAE | 0.017525 | 0.012144 |
| R2 Score | 0.997 | 0.9993 |
| Max Error | 0.19085 | 0.0979 |

### *13.1.3 16 - Neuron Model*

**Table 13-3. Benchmark for Sine_16_Model**

| Metric | F28p55x CPU | F28p55x NPU |
|---|---|---|
| Latency [ms] | 0.0223 | 0.0252 |
| Samples/sec | 44643 | 39557 |
| MAE | 0.1588 | 0.02029 |
| R2 Score | 0.88812 | 0.8451 |
| Max Error | 0.86379 | 0.9618 |

### 13.1.4 Reference Benchmark

For comparison, a large-scale reference model was tested on PC to establish an accuracy baseline:

**Table 13-4. Benchmark for 1024 Neuron Model (Reference Model)**

| Metric | Reference Python Model (1024 Neurons) |
|---|---|
| Latency [ms] | 0.149 |
| Samples/sec | 7188 |
| MAE | 0.002171 |
| R2 Score | 1 |
| Max Error | 0.0054 |

This reference model achieved near-perfect accuracy but exceeded the f28p55x's memory capacity by a significant margin (3.5-4.2MB).

## 13.2 Performance Analysis

The performance evaluation employs several key metrics to quantify model performance across different configurations:

### 13.2.1 Model Selection Trade-offs

The 128-neuron model represents the highest accuracy configuration for this application. This model achieves impressive precision with an $R^2$ score of 0.9996 and a mean absolute error (MAE) of only 0.0097 when running on the NPU. However, this accuracy comes at a significant cost in terms of processing speed, with latency of 0.7116ms and throughput of only 1,405 samples/sec.

The 64-neuron model balances accuracy with hardware constraints. This model maintains excellent accuracy ($R^2$>0.99) while significantly improving processing speed. When running on the NPU, the 64-neuron model delivers 4,659 samples/sec—more than three times the throughput of the 128-neuron configuration—with only a minimal reduction in prediction quality.

Smaller models offer dramatically improved throughput but at substantial accuracy cost. The 16-neuron model executes at 39,557 samples/sec on the NPU—28 times faster than the 128-neuron model—but the $R^2$ score drops to 0.8451, representing a significant decline in prediction quality that is unacceptable for many precision-critical applications.

### 13.2.2 CPU versus NPU Performance

The comparison between CPU and NPU execution reveals important insights that should guide implementation decisions:

*NPU Advantages for Complex Models:* For larger models, the NPU delivers significant performance improvements. The 128-neuron model runs 29.7% faster on the NPU than on the CPU (0.7116ms vs 1.012ms latency), while the 64-neuron model shows a 20.7% latency reduction. This advantage stems from the NPU's specialized architecture for parallel neural network computations.

*CPU Advantage for Simple Models:* Interestingly, for very small models like the 16-neuron configuration, the CPU actually outperforms the NPU. The CPU achieves 44,643 samples/sec compared to the NPU's 39,557 samples/sec—a 12.9% performance advantage. This counterintuitive result stems from the overhead associated with transferring data to and from the NPU. For the 16-neuron model, the computational workload is so minimal that the CPU can process it directly within its native execution environment, avoiding multiple data transfer steps. With such a small model, the CPU completes the entire inference in a single execution context without the memory transfer penalties that the NPU incurs. Every NPU inference requires setting up DMA transfers, configuring the accelerator, waiting for completion, and retrieving results—operations that collectively consume more time than the actual neural network computation for this lightweight model. Essentially, when the model is this small, the "cost" of using the specialized hardware exceeds its computational benefit.

## 13.3 Pipeline Stage Timing Measurements

The timing for each stage of the signal processing pipeline was measured using hardware timers on the F28P55x. Measurements were conducted with precise instrumentation to understand the contribution of each component to overall system latency.

**Table 13-5. Pipeline Stage Timing for Different Components**

| Pipeline Stage | Timing (ms) |
|---|---|
| Software Write Delay for CMPSS DAC Module | 0.0006 |
| Hardware Delay for ADC Conversion | 0.00084 |
| NPU Processing (64-neuron model) | 0.21 |
| Software Write Delay for Buffer DAC | 0.000593 |
| Total Pipeline Latency | 0.212 |

These precise measurements reveal that:

- Peripheral Operations: The CMPSS module, ADC conversion, and DAC output operations are extremely fast, each taking less than one microsecond.
- NPU Dominance: The neural network inference time completely dominates the pipeline, accounting for 99.04% of the total latency. With the 64-neuron model, the NPU processing time is approximately 210 times longer than all other pipeline components combined.
- Fixed Costs: The combined time for all peripheral operations (CMPSS, ADC, DAC) is just 0.002033ms, representing less than 1% of the total pipeline latency.

# 14 Summary

The F28P55x Neural Processing Unit (NPU) represents a significant advancement in embedded machine learning capabilities for automotive and industrial applications, enabling on-device inference without compromising the deterministic performance essential in these domains. This guide has provided a comprehensive exploration of the NPU's capabilities, constraints, and implementation methodology through a practical sine function approximation example.

## 14.1 Key Capabilities and Constraints

The NPU delivers hardware-accelerated neural network execution that significantly outperforms software implementations on the main CPU, with performance improvements of 20-30% for larger models. This integer-based computation engine enables real-time processing with deterministic execution while maintaining seamless integration with other C2000 peripherals.

However, these capabilities come with important constraints, including limited memory that restricts model complexity, architectural preferences for specific network topologies, and precision tradeoffs introduced by quantization. Our benchmarking revealed that while the NPU excels with larger models, very small neural networks

Smaller model such as the 16 neurons model can actually run more efficiently on the CPU due to the overhead of data transfers to and from the NPU.

## 14.2 Development Workflow

The implementation process follows a structured workflow encompassing model development, compilation, and application integration. The methodology begins with quantization-aware training that prepares models for the NPU's integer-only processing, followed by compilation through TI's Neural Network Compiler to generate hardware-compatible artifacts. These components are then integrated into a CCS project with appropriate peripheral configuration to create a complete signal processing pipeline.

## 14.3 Model Design Considerations

Through systematic experimentation, this guide has demonstrated that successful NPU implementations require careful architecture design that balances accuracy requirements with hardware constraints. The sine function example revealed critical insights about model sizing:

- The optimal 64-neuron architecture achieved excellent accuracy ($R^2 > 0.99$) while fitting within memory constraints.
- Larger models (128 neurons) offered marginally better accuracy at the cost of significantly reduced throughput, while still benefiting from NPU acceleration compared to CPU execution.
- Smaller models (8-16 neurons) provided higher throughput but at significant accuracy cost.

## 14.4 Implementation Challenges and Solutions

This guide addressed several practical challenges encountered during NPU implementation:

- *Negative and Floating-Point Values:* Using proper dequantization techniques and compilation settings to handle sine values in the range [-1, 1].
- *Neural Network Layer Support:* Designing models that leverage supported layer types while avoiding unsupported operations.
- *Memory Limitations:* Systematically reducing model size to fit within hardware constraints.

## 14.5 Broader Applications

While demonstrated through a sine function approximator, the techniques and approaches in this guide extend to a variety of automotive and industrial applications, including:

- Predictive maintenance through vibration or acoustic signal analysis.
- Anomaly detection in sensor data streams.
- Advanced control systems with neural network-based modeling.
- Sensor fusion for enhanced perception and decision making.

The F28P55x NPU brings machine learning directly into embedded control systems, creating intelligent applications that work independently at the edge while maintaining the reliability needed in automotive and industrial settings. By running neural networks right on the microcontroller, systems can make complex decisions locally without sending data elsewhere. This approach keeps response times predictable – crucial for safety systems – while using less power than traditional solutions. Applications like predictive maintenance, sensor fusion, and anomaly detection become practical even in harsh environments where cloud connectivity isn't reliable. The NPU strikes a balance between advanced capabilities and the strict operational requirements of critical control systems, allowing intelligence to be added without sacrificing the deterministic behavior that these applications demand.

## 15 References


- Texas Instruments, TMS320F28P55x Real-Time Microcontrollers Datasheet, PDF
- Texas Instruments, EdgeAI Studio, Webpage
- Texas Instruments, Tinyml-Tensorlab, Repository
- Texas Instruments, TI Neural Network Compiler for MCUs User's Guide, Webpage
- Texas Instruments, C2000Ware for C2000 MCUs, C2000 SDK, Webpage
- Texas Instruments, F28P55X LaunchPad development kit, Product details, Webpage

# IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.



Last updated 10/2025