Deep Learning on the Edge

Dan Kacenjar

Email: dan@kacenjar.com

Git Hub:

https://github.com/kacenjar/codemash2020

Description:

Machine learning on microcontrollers? You bet! With the introduction of TensorFlow Lite, machine learning inferencing has moved to the edge... as in edge computing! Cloud-based servers or internet connections are no longer required. Machine learning can occur on the microcontroller, directly on the hardware, without network latency, or even any network connection. Let's explore how machine learning inferencing works on microcontrollers and look at some live examples.

Requirements – what we need to know

- Embedded development / Microcontroller
- C++ (make files, compiler, linker, etc.)
- ML concepts/theory
- Python
- Associated Python ML libraries (TensorFlow, Numpy, etc.)

This can all be a bit intimidating...

But...
We have some tools/examples that make this all very accessible to anyone here.

Tools

- Hardware
  - Sparkfun Edge board (created specifically for experimentation with TinyML
  - Arduino Nano BLE Sense
  - Adadfruit EdgeBadge

- Software
  - Arduino IDE (cloud based or local installation)
  - Google Colab for ML model development and training

What is ML? (Machine Learning)

- Machine learning is a technique for using computers to predict things based on past observations.
  - Imagine we have a factory with a machine that makes widgets. At times, the machine will break down and it's very expensive to repair.
  - If we were able to collect data about the machine during operation, we might be able to predict when it's about to break down and stop operation before damage occurs. (Examples of relevant data – Temperature, vibration, production rate)

Traditional programming

- In traditional programming, we design and algorithm which takes an input, applies various rules (business logic), and generates an output.
- The above algorithm is defined/designed by the programmer and carried out explicitly through lines of code.

Machine Learning Programming

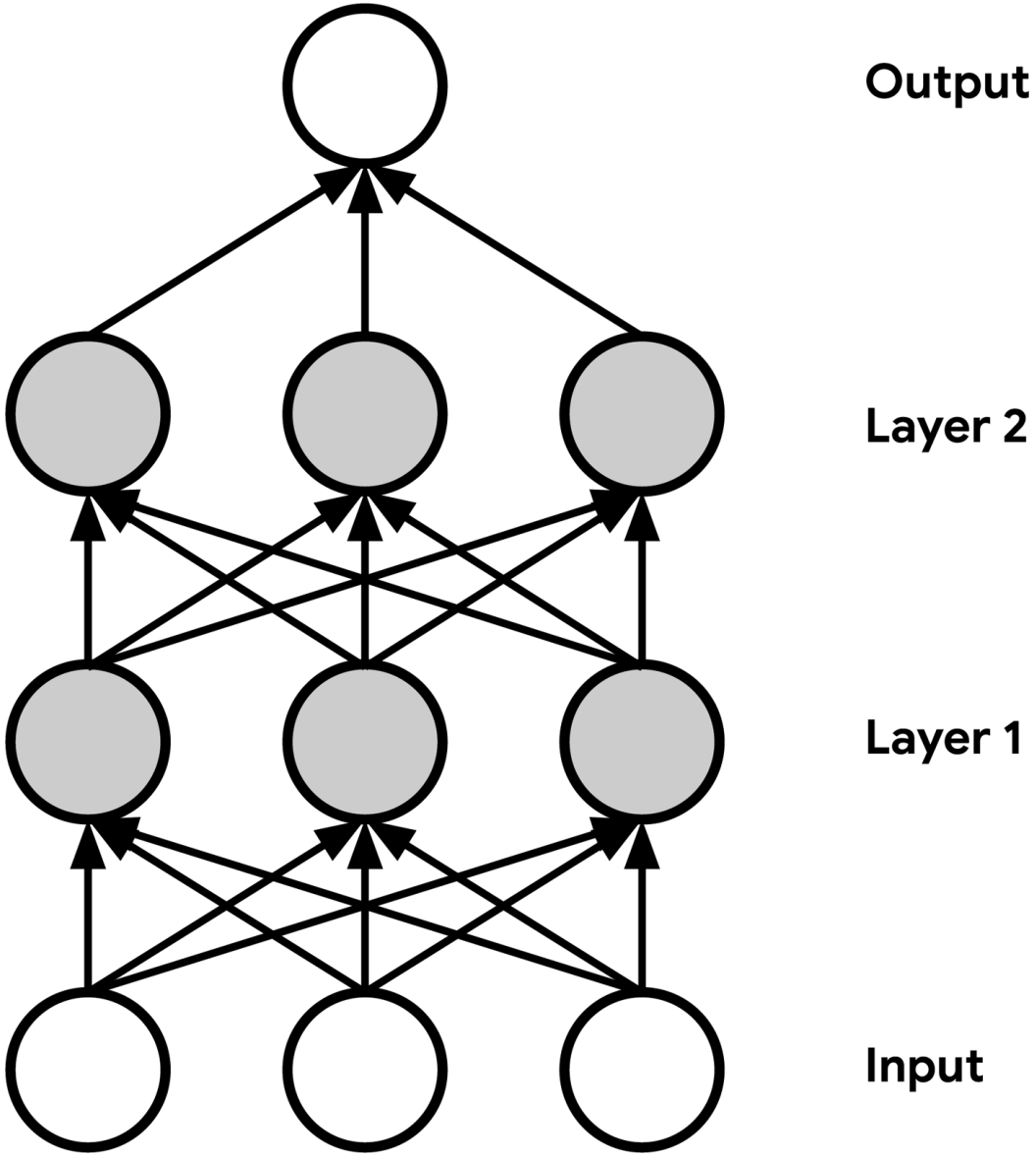
- Feed data into a special kind of algorithm and let the algorithm discover the rules (business logic)
- This means we can create programs that make predictions based on complex data without us having to understand all the complexities ourselves.
- The algorithm builds a ***model*** of the system based on the data we feed it. (This is called ***training***).
- We run data through the model to make predictions, in a process call ***inference***.

Deep Learning

- A simplified version of how the brain works
- A network of simulated neurons (arrays of numbers) is trained to model the relationship between various inputs and outputs
- We use different architectures of simulated neurons for different tasks (i.e. some are better at image or sound recognition, while others are better at predicting the next value in a sequence.

Deep Learning Workflow

1. Decide on a goal
2. Collect a dataset
3. Design model architecture
4. Train the model (Stop when it converges. We look at loss and accuracy)
5. Convert the model
6. Run inference
7. Evaluate/Troubleshoot

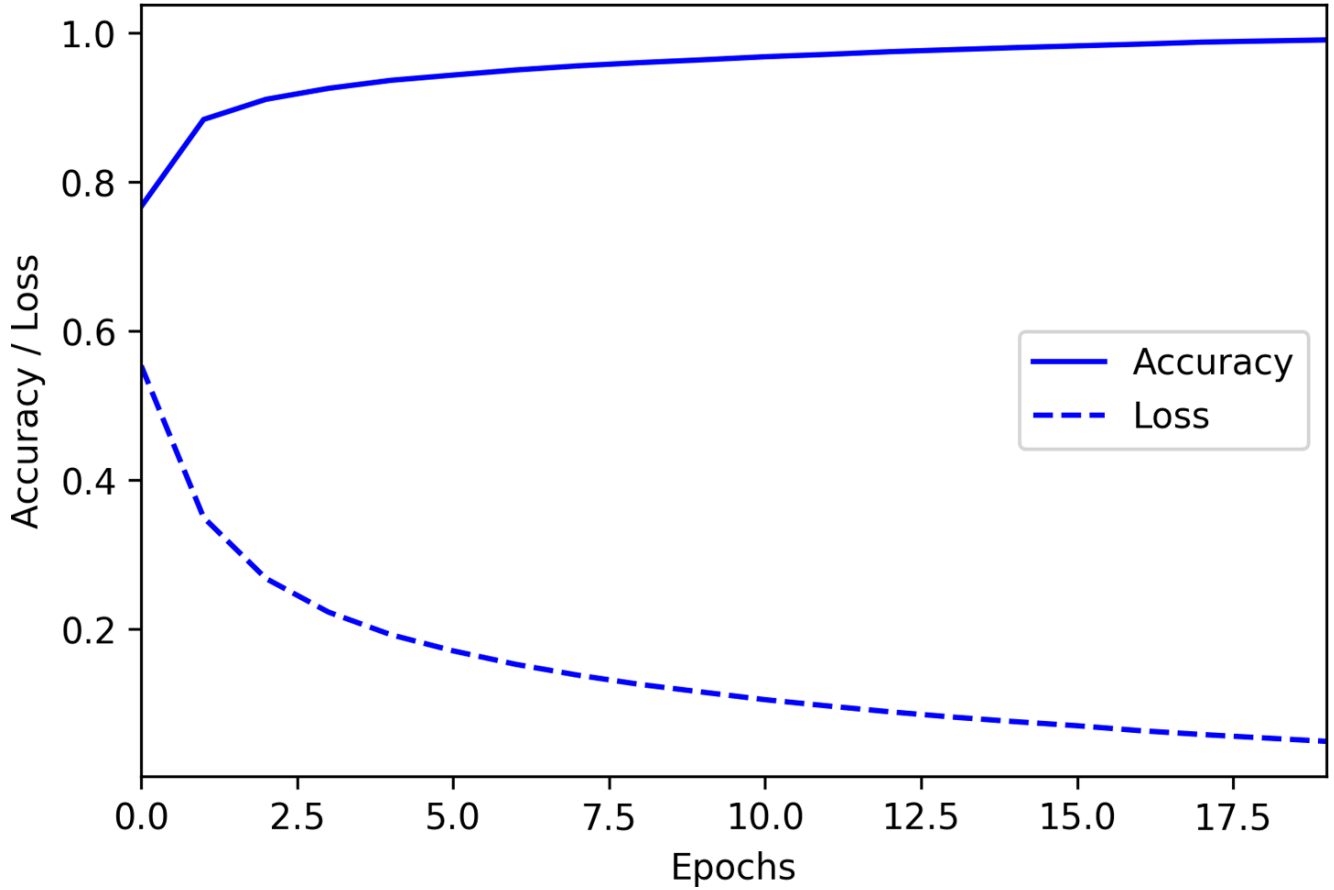


Neural Network Model

Convergence

- When our model's performance stops improving, and we're making accurate predictions, it is said to have ***converged***.
- To improve performance, we can change model architecture, and adjust values used to setup model and training process
- Examples include
  - Number of epochs to run (iterations)
  - Number of neurons in each layer

Loss and Accuracy



No guarantees

- May not be able to achieve good enough accuracy
- May not have enough information in dataset to make accurate predictions
- Some problems can be solved, even with state-of-the-art ML
- However, some models may be useful even if not 100% accurate

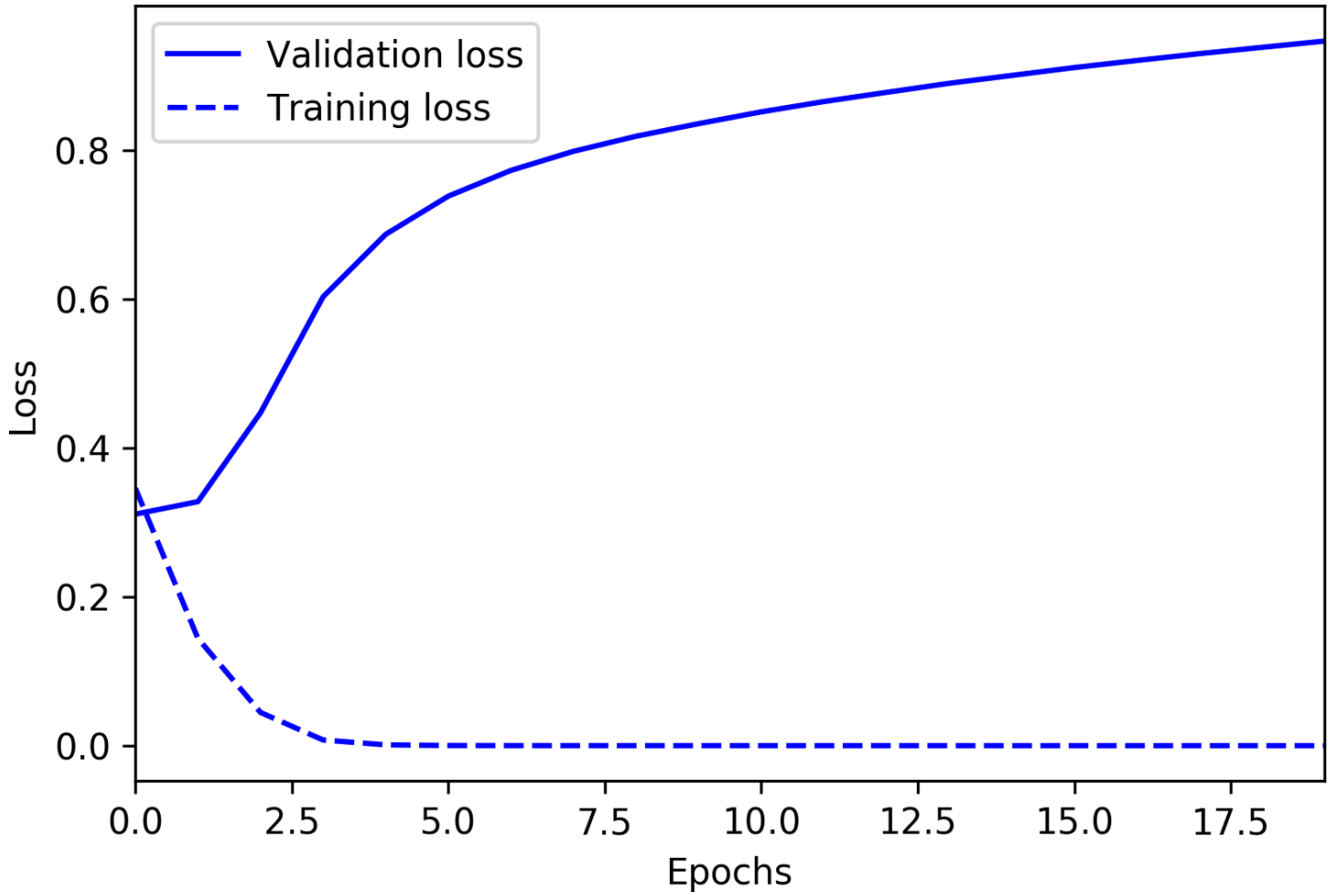
Overfitting and Underfitting

- Two most common reasons a model fails to converge
- A neural network learns to fit behavior to the pattern it recognizes. If a model is correctly fit, it will produce the correct output for a given set of inputs
- ***Underfit*** – It hasn't been able to learn enough to make good predictions
- ***Overfit*** – It's learned it's training data too well and is not able to accurately generalize it's learning to data it hasn't previously seen.

Training, Validation, and Testing

- We typically take our dataset and split it out for Training, Validation, and Testing using a 60%, 20%, and 20% split, respectively.
- While we're training (running our training data in our model), we also validate at a specified interval. (We're using data that wasn't used as part of the training to determine accuracy and loss of our model).
- This helps us determine whether our model is overfitting.

Overfitting



TinyML

- Running neural network models at an energy cost of below 1mW
  - By consensus amongst academia and industry experts

- What does this mean in real terms?
  - A device running on a coin cell battery that has a lifetime of one year.

- This constrains us to embedded devices. Specifically, we'll be looking at 32bit Cortex-M class microcontrollers.

- No OS, no dynamic memory allocation.

Convert the Model

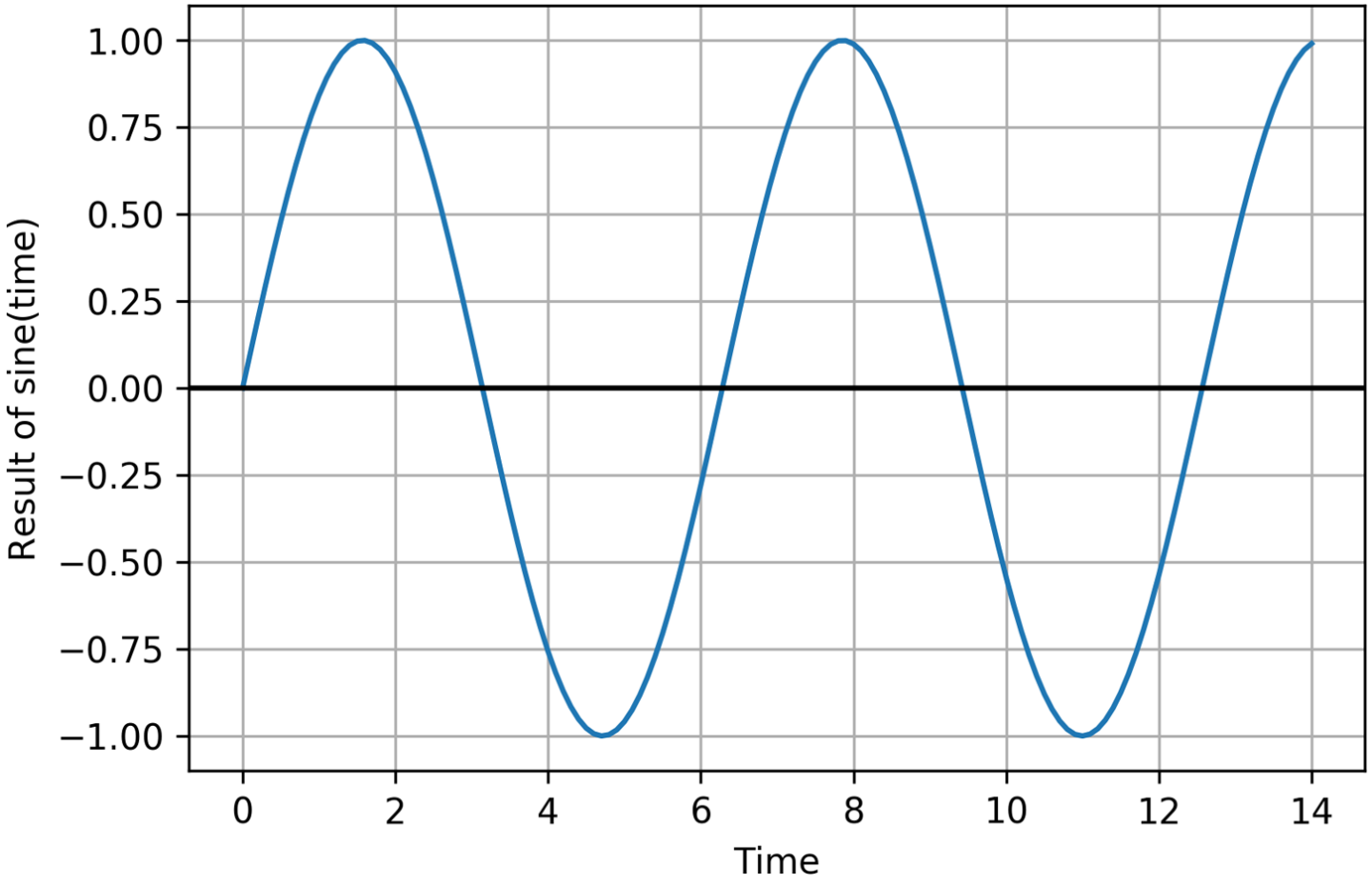
- Tensorflow models are essentially instructions that tell the interpreter how to transform data to produce an output.
- The Tensorflow interpreter is designed to run on powerful desktop computers and servers.
- But we're using microcontrollers, therefore, we need a different interpreter that's designed for microcontrollers (Tensorflow Lite for Microcontrollers)
- We must convert our model to Tensorflow Lite format using a set of tools included with Tensorflow.

Run Inference

- We now take our converted model, our Tensorflow Lite for Microcontrollers model and run it on our microcontroller to make predictions. (This part is done in C++, prior was in Python)
- We need to write some code that takes raw input data from sensors and transforms it into the form that our model was trained on. We then pass this on to our model and run inference. (This is where the rubber meets the road!)

“Hello World” of TinyML

- Goal is to train model to take in a value, x, and predict it’s sine, y.



How do we implement “Hello World”?

1. Obtain a dataset.
2. Train deep learning model.
3. Evaluate the model’s performance.
4. Convert the model to run on the microcontroller
5. Write code to perform inference on microcontroller.
6. Build the code into a binary.
7. Deploy the binary to microcontroller.

Machine Learning Toolchain

- Python and Jupyter Notebooks
- Google Colaboratory
- TensorFlow and Keras (high level ML abstraction)

“Hello World” Jupyter Notebook

- Clear output
- Import dependencies
- Generate data
- Data is too clean, so we’ll add some noise to simulate real-world conditions
- Split our data for training (60%), validation (20%), and testing (20%)
  - It’s important that the data we use for validation and testing is not the same data we used for training.

Define a Basic Model

- Our model will take in an input value (x) and use it to predict a numeric output value [sin(x)]
  - This type of problem is called a regression
- We'll design a simple neural network using the Keras high-level TensorFlow API
- The neural network will be a Sequential model, meaning each layer of neurons is stacked up on top of the next

Training

- x_train, y_train – training data
- epochs – how many times our entire training set will be run (be careful not to overfit)
- batch_size – how many pieces of training data to feed in before checking accuracy and updating its weights and biases. Too small and it will take too long to train. Too large and it will have less ability to generalize to new data. Compromise is to use something between 1 and number of datapoints, 600.
- validation_data – data run through during training to compare predictions with expected values.

Training and Validation

- Look at training and validation loss
- Graph history for greater insight
- Graph predictions for insight into model performance

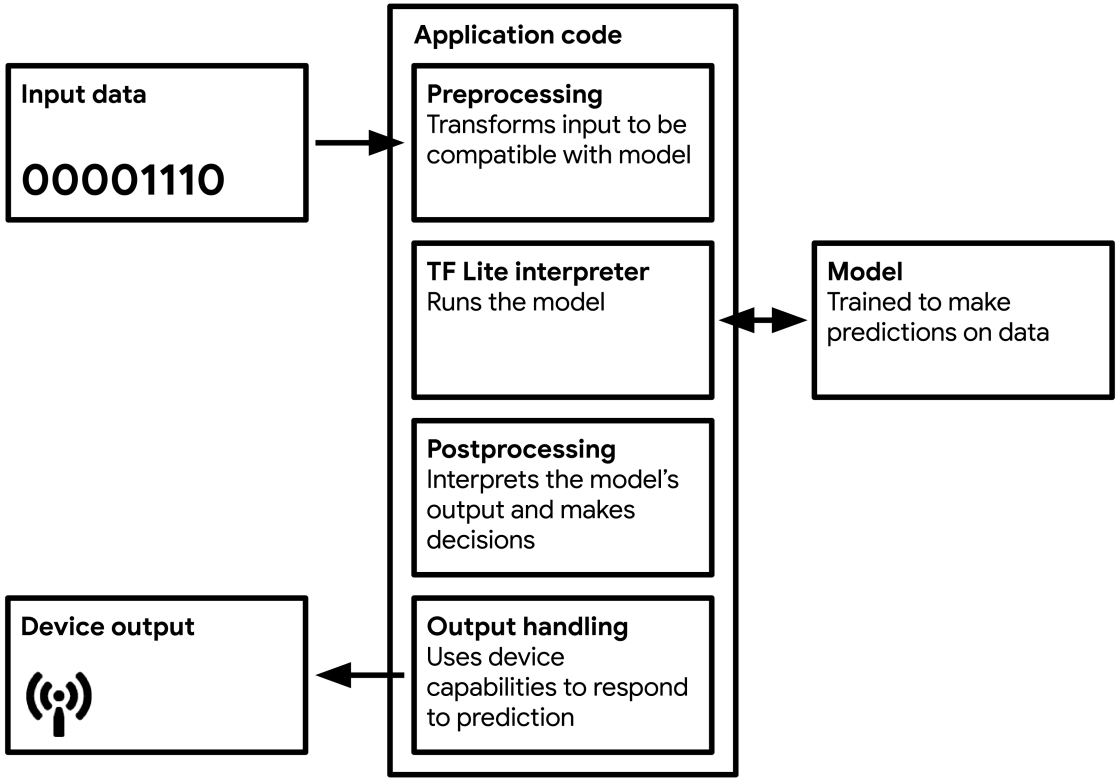
Make required adjustments

- Our model was too small to learn the complexity of our data
- An easy way to make neural network bigger is to add another layer of neurons

Convert Model

- After we validated our model, we need to convert it to a TensorFlow Lite model
- We use quantization to make our model even smaller
- We then convert to a C file to deploy to our microcontroller

Build our Inference application



Hello World Test

- Why would we want to test our model on local dev machine?
- How do we go about doing that?
- Let's take a quick look at the code (You'll find this directly applicable to the "real" application).

- **We should try to write the code that preprocesses inputs, runs inference with the model, and processes any outputs in a set of tests before trying to get it working on-device.**

Hello World Application Structure

- constants.h and constants.cc
  - Various constants that are important for defining the program's behavior.
- main.cc
  - The entry point of the program which runs when first when the application is deployed to the device.
- main_functions.h and main_functions.cc
  - A pair of files which define a setup() function, which performs all initialization for our application, and a loop() function which contains the program's core logic and is designed to be called repeatedly in a loop. These functions are called by main.cc when the program starts

Application Structure Continued…

- output_handler.h and output_handler.cc
  - A pair of files that define a function we can use to display an output each time inference is run. This defaults to the screen, but we can override this to do different things.

- sine_model_data.h and sine_model_data.cc
  - These define an array of data representing our model, as exported using xxd.

Deploy our application using Arduino IDE

- We’re going to jump over to the Arduino IDE and look at the Hello World example

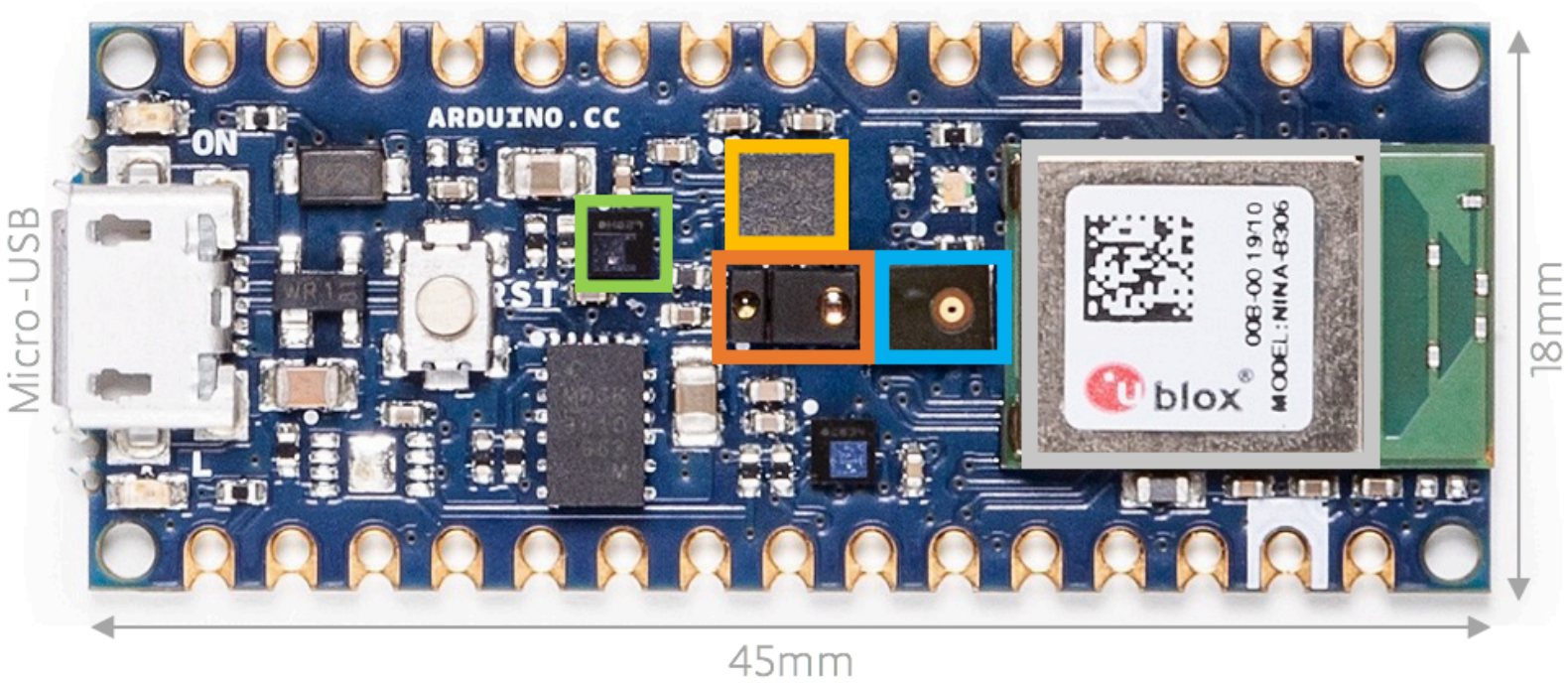
Demo Time!

Micro Speech

- Captures audio with a microphone in order to detect the words "yes" and "no"
- I’ve already pre-programmed this application for the demo

Nano BLE Sense

ARDUINO NANO 33 BLE SENSE



- Color, brightness, proximity and gesture sensor
- Digital microphone
- Motion, vibration and orientation sensor
- Temperature, humidity and pressure sensor
- Arm Cortex-M4 microcontroller and BLE module

Gesture Recognition

- Capture IMU (Inertial Measurement Unit) data for gestures
- Import data captured into Gesture Colab to train classification model using Tensorflow
- Convert model to TensorFlow Lite model
- Deploy to Arduino Nano BLE Sense

Gesture Example

Fruit Identification

- Capture color information data of various fruit objects
- Import color data captured into Fruit Identification Colab to train classification model using TensorFlow
- Convert model to TensorFlow Lite model
- Deploy to Arduino BLE Sense

Fruit Identification Example

References

- Books
  - TinyML (Author: Pete Warden, Publisher: O'Reilly)
  - Programming Machine Learning (Author: Paolo Perrotta, Publisher: The Pragmatic Programmer)
- Website
  - TensorFlow Lite for Microcontrollers
  - TensorFlow Lite for Microcontrollers – Examples
  - TensorFlow GitHub