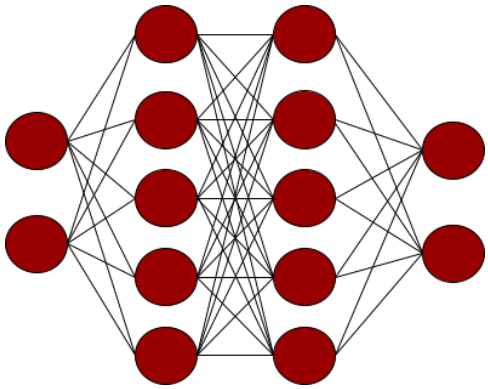
Dropout Regularization

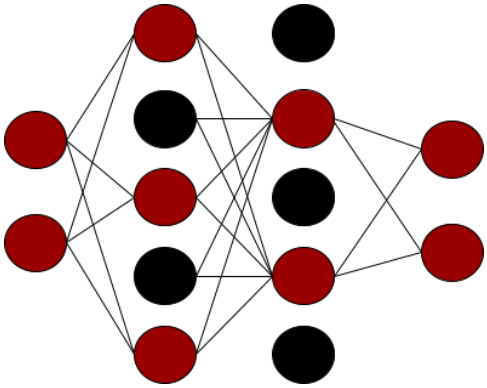
You've been exploring *overfitting*, where a network may become too specialized in a particular type of input data and fare poorly on others. One technique to help overcome this is use of *dropout regularization*.

When a neural network is being trained, each individual neuron will have an effect on neurons in subsequent layers. Over time, particularly in larger networks, some neurons can become overspecialized—and that feeds downstream, potentially causing the network as a whole to become overspecialized and leading to overfitting. Additionally, neighboring neurons can end up with similar weights and biases, and if not monitored this can lead the overall model to become overspecialized to the features activated by those neurons.

For example, consider this neural network, where there are layers of 2, 5, 5, and 2 neurons. The neurons in the middle layers might end up with very similar weights and biases.



While training, if you remove a random number of neurons and connections, and ignore them, their contribution to the neurons in the next layer are temporarily blocked



This reduces the chances of the neurons becoming overspecialized. The network will still learn the same number of parameters, but it should be better at generalization—that is, it should be more resilient to different inputs.

The concept of dropouts was proposed by Nitish Srivastava et al. in their 2014 paper "Dropout: A Simple Way to Prevent Neural Networks from Overfitting".

To implement dropouts in TensorFlow, you can just use a simple Keras layer like this:

```
tf.keras.layers.Dropout(0.2),
```

This will drop out at random the specified percentage of neurons (here, 20%) in the specified layer. Note that it may take some experimentation to find the correct percentage for your network.

For a simple example that demonstrates this, consider the Fashion MNIST classifier you explored earlier.

If you change the network definition to have a lot more layers, like this:

```
model = tf.keras.models.Sequential([
                    tf.keras.layers.Flatten(input_shape=(28,28)),
                    tf.keras.layers.Dense(256, activation=tf.nn.relu),
                    tf.keras.layers.Dense(128, activation=tf.nn.relu),
                    tf.keras.layers.Dense(64, activation=tf.nn.relu),
                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

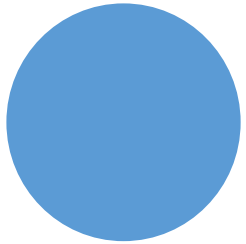
Training this for 20 epochs gave around 94% accuracy on the training set, and about 88.5% on the validation set. This is a sign of potential overfitting.

Introducing dropouts after each dense layer looks like this:

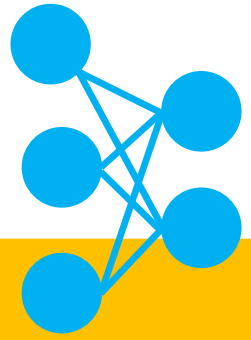
```
model = tf.keras.models.Sequential([
                  tf.keras.layers.Flatten(input_shape=(28,28)),
                  tf.keras.layers.Dense(256, activation=tf.nn.relu),
                  tf.keras.layers.Dropout(0.2),
                  tf.keras.layers.Dense(128, activation=tf.nn.relu),
                  tf.keras.layers.Dropout(0.2),
                  tf.keras.layers.Dense(64, activation=tf.nn.relu),
                  tf.keras.layers.Dropout(0.2),
                  tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

When this network was trained for the same period on the same data, the accuracy on the training set dropped to about 89.5%. The accuracy on the validation set stayed about the same, at 88.3%. These values are much closer to each other; the introduction of dropouts thus not only demonstrated that overfitting was occurring, but also that adding them can help remove it by ensuring that the network isn't overspecializing to the training data.

Keep in mind as you design your neural networks that great results on your training set are not always a good thing. This could be a sign of overfitting. Introducing dropouts can help you remove that problem, so that you can optimize your network in other areas without that false sense of security!

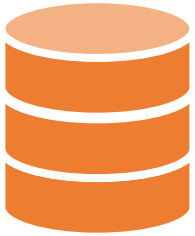


Data acquisition



Design Model **TensorFlow**

Architecture



Training /Validation Data Set

Train **TensorFlow**

Model

Model Quantization **TensorFlow**

Quantized Model

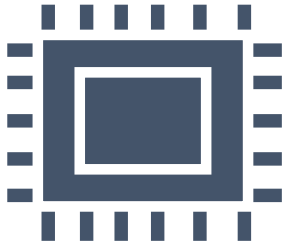
ML space

Embedded Space

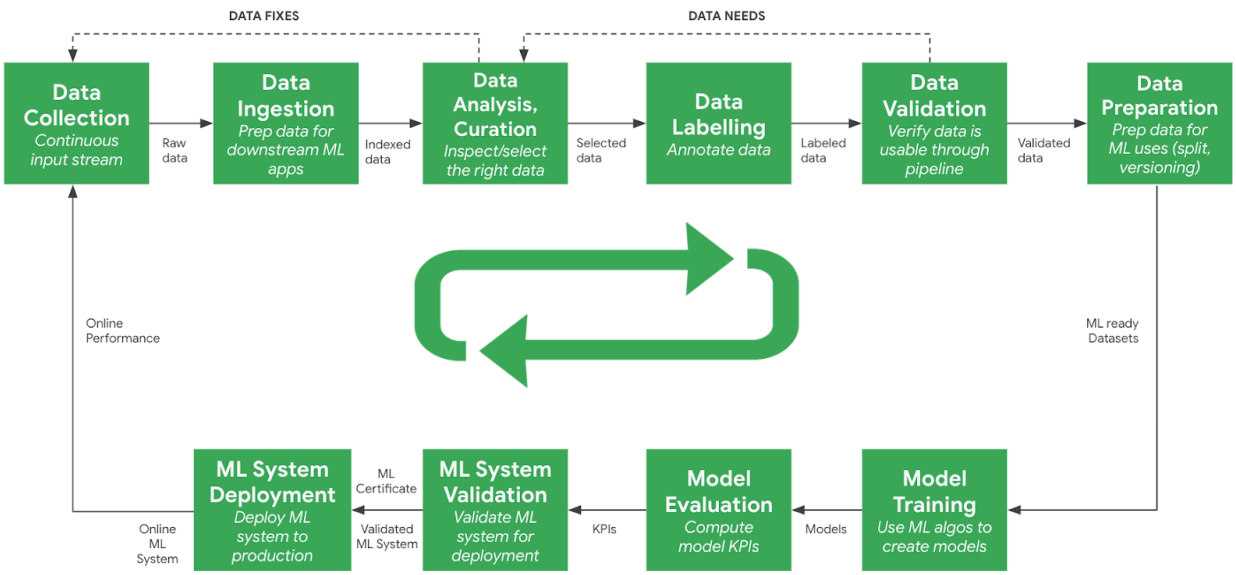
Covert to C- byte array

Quantized Model as C array

Application Development **TFT for Microcontroller**



Machine Learning Lifecycle



The development of a machine learning model follows a multi-step design methodology, commonly referred to as the machine learning lifecycle. Diagrams typically illustrate this lifecycle using a varying set of discrete steps. This methodology is also applicable to tiny machine learning applications, albeit with different requirements to conventional machine learning models. Some of the most important steps in the machine learning lifecycle are as follows:

Design Requirements. The first stage of the machine learning lifecycle is where the application requirements are established. For tiny machine learning applications, hardware constraints (e.g., memory and storage footprint, latency) are often the most pressing requirements. Other requirements can include temporal resolution of data, the number of inferences per second, or minimum model accuracy. This step, whilst being the least well-defined, is the most crucial step, as it sets the scene for the remainder of the lifecycle. The more clearly defined the design criteria are, the easier it is to evaluate the model in later stages. Data requirements, such as the variable of interest and potential feature variables, can also be specified during this stage.

Data Collection/Curation. In this stage, design requirements are known and data begin to be collected that are suspected to aid in predicting the variable of interest. This can often involve active data collection or curation of information from external sources, such as pre-existing datasets (e.g., CIFAR10, ImageNet). This is often one of the most time consuming stages, as it can be difficult and time-consuming to curate a sufficiently large dataset, simultaneously ensuring to minimize dataset bias. No data analysis is

performed during this stage. In Course 2, we will learn the challenges associated with data collection for different types of sensors that serve as inputs for tiny machine learning applications.

Exploratory Data Analysis/Data Preprocessing. During this phase, the collected data is analyzed to determine which features are most informative at predicting the variable of interest. Feature engineering is common at this stage, extracting new information from available data. An example of this would be to extract the day of the week or whether it is a weekend from a time variable. Preprocessing involves ensuring the data follows standard modeling assumptions (e.g., are normally distributed) or manipulating data to meet these assumptions (e.g., log transformations on highly skewed variables). Data imputation is also typically done to appropriately fill in or remove missing data, and invalid or duplicate data is handled accordingly. When we develop our keyword spotting model, we will learn how to pre-process the audio signal.

Model Development. All of the previous stages focused on the design aspects, as well as the procurement and analysis of data. These stages often make up the dominant proportion of the time in the machine learning lifecycle. Once a viable dataset is available, the next stage is creating a suitable machine learning model. There are a plethora of machine learning techniques that have various pros and cons. One of the jobs of a machine learning practitioner is selecting an appropriate model for the task at hand. For example, basic linear regression may be suitable in environments where interpretability is key, but are constrained to the set of linear functions. This may present bias in situations where the distribution of data is highly non-linear in feature space, which can result in poor model accuracy. Conversely, neural networks may offer high performance as they are able to model the non-linear data distribution more effectively, but are less interpretable to the user. Keeping in mind the design requirements is important during this stage to know on what grounds to evaluate different models. One thing we will do in Course 2 is to understand the different model development approaches. Training a neural network from scratch is not always necessary.

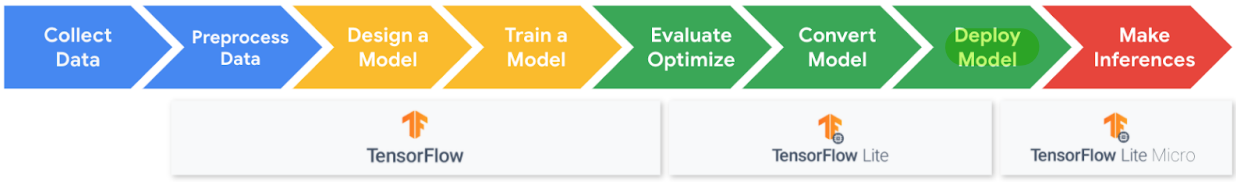
Model Validation. It is commonplace to generate multiple models and to compare their performance on an unseen data set, known as the test set. The presence of a test set helps to ensure that the model has effectively modeled the distribution of data and has not overfit to points in the training set. The design criteria help to determine what metrics should be used in comparing the performance of various models. For example, if accuracy is the dominant metric, the model with the highest accuracy should be chosen. This stage can also be used to determine whether specific features improved or hindered model performance (e.g., by analyzing the relative importance of features or by successively removing variables and retraining the model, known as ablation study).

But there is more to model validation than just looking at the model metrics. We will discover how to think about metrics in a new light, specifically, from an application deployment perspective.

Model Deployment. The last stage of the lifecycle involves the deployment of the model in the production environment. For tiny machine learning applications, this is one of the most important stages. The model size must be reduced sufficiently to fit on the embedded device through various means such as model compression (i.e., model distillation), type conversion (e.g.., changing model weights from floats to integers), and lossless compression (e.g., Huffman encoding). The model must then be compiled into a format compatible with the end device. We will learn about "quantization" and use it to optimize the model size for performance and latency to make sure we are building sufficiently 'tiny' models. We will use TensorFlow's quantization API, as well as understand what's going on behind the scenes.

After all of these stages, continuous monitoring of the model in the production environment is typically necessary to ensure that it is working effectively. If, after all of these steps, the performance of the model is unsatisfactory in the production environment, the model must be updated. This can be done using new design requirements, data, or modeling techniques. This is the essence of the machine learning lifecycle.

ML Workflow / Pipeline



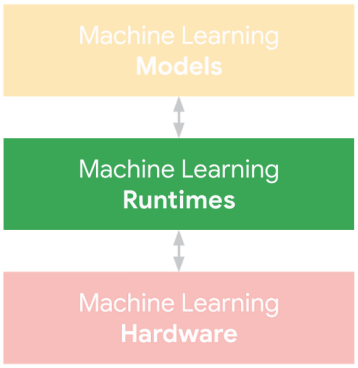
TinyML Application Pipeline

The figure above shows the workflow methodology we will be following for building the tinyML applications in this course. These are broken down into three main stages:

- Step 1: Collect & Preprocess Data
- Step 2: Design and Train a Model
- Step 3: Evaluate, Optimize, Convert and Deploy Model

What's Underneath the Hood?

Often, when learning about machine learning, we forget that models are just a piece of the bigger picture. There is more to machine learning than just the models. These are the underlying ML runtimes and hardware that enable these models to run efficiently. The ML runtimes we will be dealing with are TensorFlow, TensorFlow Lite, and TensorFlow Lite Micro.



One of the goals of our course is to dive into understanding the fundamental differences between machine learning runtimes so that we can deploy the models we prepare efficiently. To this end, it is important to understand the various frameworks we use across all three applications in this course: Keyword Spotting, Visual Wake Works, and Anomaly Detection.

So coming up next is material that highlights the **fundamental differences between TensorFlow and TensorFlow Lite (for microcontrollers)**. You will learn not only the high-order bits but also the mechanics that implement the API calls you learn to make with the code.

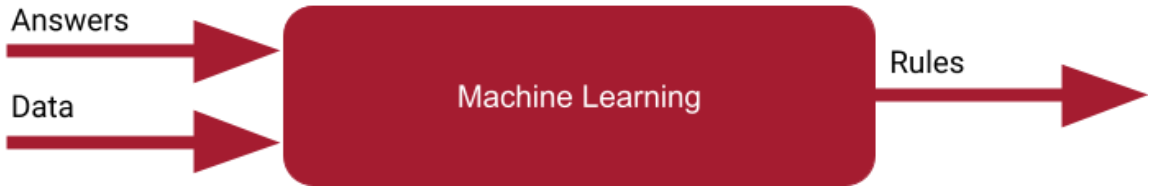
TensorFlow: Where We Left Off

Before we dive into our TinyML applications' underlying mechanics, you must understand the software ecosystem that we will be using. The software that wraps around the models we train is just as important as the models themselves. Imagine working hard to train a tiny machine learning model that occupies only a few kilobytes of memory, only to realize that the TensorFlow framework used to run the model is itself several megabytes large.

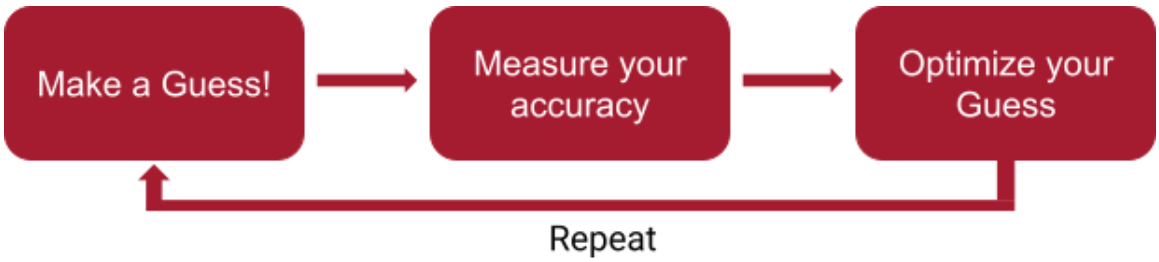
To avoid such oversight, we first introduce the fundamental concepts around TensorFlow's usage versus TensorFlow Lite (and soon TensorFlow Lite Micro). We will be using TensorFlow for training the models. We will be using TensorFlow Lite for evaluation and deployment because it supports the optimizations we need. It has a minimal memory footprint, which is especially essential for deployment in mobile and embedded systems. To this end, in the following two modules, Laurence and I are going to first introduce the concepts, programming interfaces, and the mechanics behind them before diving deep into the applications. Every application reuses these fundamental building blocks.

Recap of the Machine Learning Paradigm

In the previous course you had an introduction to machine learning, exploring how it is a new programming paradigm that changes the programming paradigm. Instead of creating rules explicitly with a programming language, the rules are learned by a neural network.



Using TensorFlow you could create a neural network, compile it and then train it, with the training process, at a high level looking like this:



The neural network would randomly initialize, effectively making a guess to the rules that match the data to the answers, and then over time it would loop through measuring the accuracy and continually optimizing.

An example of this type of code is seen here:

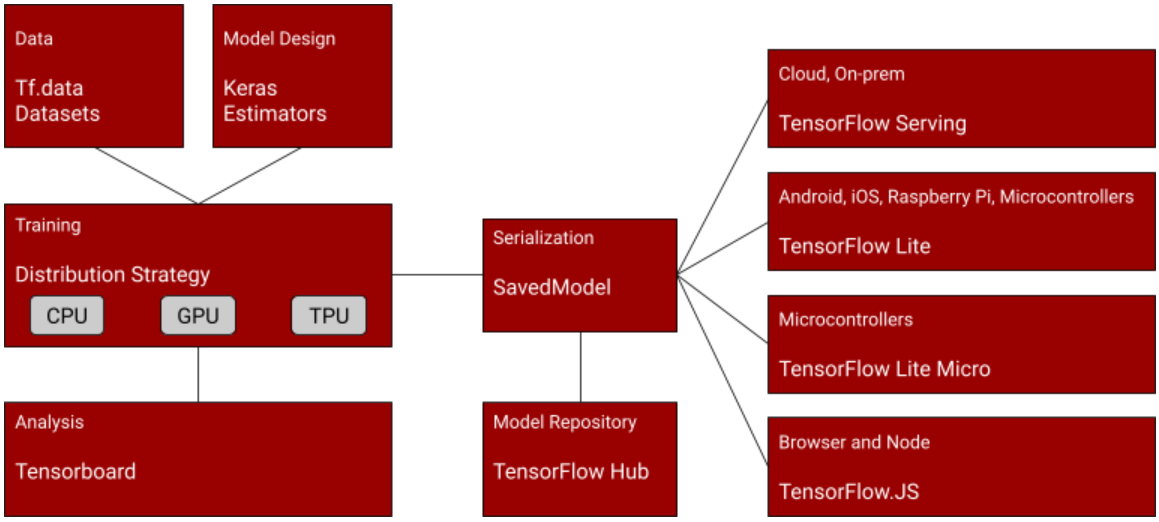
```
import tensorflow as tf
data = tf.keras.datasets.mnist

(training_images, training_labels), (val_images, val_labels) =
data.load_data()
training_images  = training_images / 255.0
val_images = val_images / 255.0

model =
tf.keras.models.Sequential([tf.keras.layers.Flatten(input_shape=(28,28)),
                                    tf.keras.layers.Dense(20,
activation=tf.nn.relu),
                                    tf.keras.layers.Dense(10,
activation=tf.nn.softmax)])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(training_images, training_labels, epochs=20,
          validation_data=(val_images, val_labels))
```

With the bottom line being that instead of creating a program or an app, you train a *model*, and you use this model in future to get an *inference* based on the rules that it learned.

Model Training vs. Deployment

With TensorFlow there are a number of different ways you can deploy these models. As you've been studying this specialization, you've been running inference within the colabs that you were training with. However, you can't share your model with other users that way.

The overall TensorFlow ecosystem can be represented using a diagram like this -- with the left side of the diagram showing the architecture and APIs that can be used for training models.



In the center is the architecture for saving a model, called TensorFlow SavedModel. You can learn more about it at: https://www.tensorflow.org/guide/saved_model

On the right are the ways that models can be deployed.

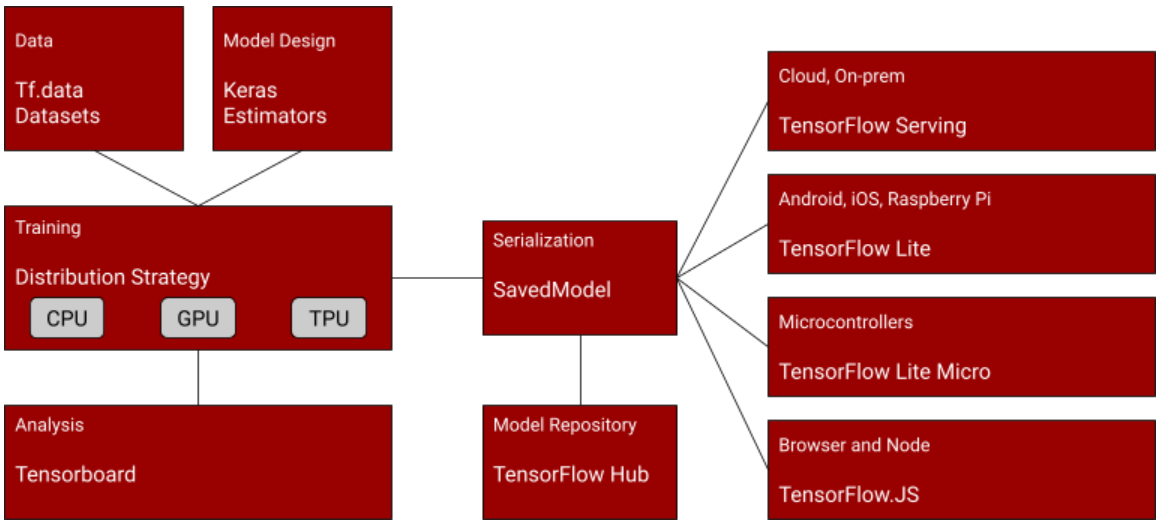
- The standard TensorFlow models that you've been creating this far, trained without any post-training modification can be deployed to Cloud or on-Premises infrastructures via a technology called TensorFLow Serving, which can give you a REST interface to the model so inference can be executed on data that's passed to it, and it returns the results of the inference over HTTP. You can learn more about it at https://www.tensorflow.org/tfx/guide/serving
- TensorFlow Lite is a runtime that is optimized for smaller systems such as Android, iOS and embedded systems that run a variant of Linux, such as a Raspberry Pi. You'll be exploring that over the next few videos. TensorFlow Lite also includes a suite of tools that help you *convert* and *optimize* your model for this runtime. https://www.tensorflow.org/lite
- TensorFlow Lite Micro, which you'll explore later in this course, is built on top of TensorFlow Lite and can be used to shrink your model even further to work on microcontrollers and is a core enabling technology for TinyML. https://www.tensorflow.org/lite/microcontrollers
- TensorFlow.js provides a javascript-based library that can be used both for training models and running inference on them in JavaScript-based environments such as the Web Browsers or Node.js. https://www.tensorflow.org/js

Let's now explore TensorFlow Lite, so you can see how TensorFlow models can be used on smaller devices on our way to eventually deploying TinyML to microcontrollers in Course 3. Over the rest of this lesson we'll be exploring models that will generally run on devices such as Android and iOS phones or tablets. The concepts you learn here will be used as you dig deeper into creating even smaller models to run on tiny devices like microcontrollers.

How to use TFLite Models

Previously you saw how to train a model and how to use TensorFlow's Saved Model APIs to save the model to a common format that can be used in a number of different places.

Recall this architecture diagram:



So, after training your model, you could use code like this to save it out:

```
export_dir = 'saved_model/1'
tf.saved_model.save(model, export_dir)
```

This will create a directory with a number of files and metadata describing your model. To learn more about the SavedModel format, take a little time now to read https://www.tensorflow.org/guide/saved_model, and also check out the colab describing how SavedModel works at https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/saved_model.ipynb, and in particular explore 'The SavedModel format on disk' section in that colab.

Once you had your saved model, you could then use the TensorFlow Lite converter to convert it to TF Lite format:

```
converter = tf.lite.TFLiteConverter.from_saved_model(export_dir)
tflite_model = converter.convert()
```

This, in turn could be written to disk as a single file that fully encapsulates the model and its saved weights:

```
import pathlib
tflite_model_file = pathlib.Path('model.tflite')
tflite_model_file.write_bytes(tflite_model)
```

To use a pre-saved tflite file, you then instantiate a tf.lite.Interpreter, and use the 'model_content' property to specify an existing model:

```
interpreter = tf.lite.Interpreter(model_content=tflite_model)
```

Or, if you don't have the existing model already, and just have a file, you can use the 'model_path' property to have the interpreter load the file from disk:

```
interpreter = tf.lite.Interpreter(model_path=tflite_model_file)
```

Once you've loaded the model you can then start performing inference with it. Do note that to run inference you need to get details of the input and output tensors to the model. You'll then set the value of the input tensor, invoke the model, and then get the value of the output tensor. Your code will typically look like this:

```
# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

to_predict = # Input data in the same shape as what the model expects

interpreter.set_tensor(input_details[0]['index'], to_predict)

tflite_results = interpreter.get_tensor(output_details[0]['index'])
```

...and a large part of the skills in running models on embedded systems is in being able to format your data to the needs of the model. For example, you might be grabbing frames from a camera that has a particular resolution and encoding, but you need to decode them to 224x224 3-channel images to use with a common model called mobilenet. A large part of any engineering for ML systems is performing this conversion.

To learn more about running inference with models using TensorFLow Lite, check out the documentation at:
https://www.tensorflow.org/lite/guide/inference#load_and_run_a_model_in_python

The TinyML Kit

If you haven't already ordered your tinyML Kit, it is [available from Arduino for $49.99 at this link](). The kit is the easiest and most reliable way to obtain all of the parts necessary for this course. If you already have your kit or are planning on purchasing one then you can move onto the next reading.

That said, we understand that for some learners it may be beneficial, or even necessary, to obtain some or all of these parts via other means. As such, the remainder of this document details a bill of materials for the components you need to complete all of the planned exercises alongside potential vendors and functional alternatives. Note that availability may depend on your location and other shipping dynamics limitations.

tinyML Kit BOM

In the picture above you can see all of the components included in the purpose-designed kit that we have developed with Arduino. Fortunately, there aren't too many parts to begin with and most of them are generally accessible on a global scale, with the exception of Arduino's Tiny Machine Learning Shield, or PCB cradle (in the middle), for the Nano 33 BLE Sense AI-enabled microcontroller board (on the left) that serves to breakout the MCU's IO to connectors that permit easy, reliable connections with the camera module (on the right), as well as other sensing modules via connectors included preemptively for expansion to new areas of

application, calling upon the Grove system for nearly plug-and-play compatibility with a range of transducers, available from SeeedStudio.[1] Fortunately, while the shield can certainly make your life easier and lends a hand in creating reliable connections with off-board sensors / actuators, we can readily replace its function with other off the shelf components (e.g., a breadboard and some wires as listed below).

The table below is a list of parts that are comparable with suggested vendors alongside alternatives. To be clear, the edX course staff will only support the official kit we have developed in partnership with Arduino.

Description	Part Number	Vendors	Alternatives
Nano 33 BLE Sense *with headers* See note (1)	ABX00035	Arduino, Amazon, DigiKey, Newark, RS Components, Arrow, Mouser, Verical, Farnell, element14, Arrow (cn)	NA
USB microB cable, often to type A *or* C	Varied	DigiKey, Amazon, and many others	See note (2)
Breadboard	Varied	DigiKey, Amazon, and many others	See note (3)
Jumper wires 'Male to female'	Varied	DigiKey, Amazon, and many others	See note (4)
Camera sensor with breakout PCB	OV7675	RobotShop, Botland, Uctronics, and others	OV7670 See note (5)

Notes

1. Make sure that you select the variant of the Nano 33 BLE Sense *with headers*, which means that the board will arrive with 0.1" pitch pins pre-soldered, enabling you to quickly seat the microcontroller board into a breadboard without additional equipment.

2. The USB cable specification need only be *data carrying* **and** with connector type micro-B (plug) on one end — the other end will depend on the port available on your computer. Type A is the most common connector type, whereas modern laptops are increasingly featuring Type C connectors. Note that not all USB cables support data exchange and some are intended for power delivery only. Be sure to select a cable with a description that explicitly mentions data transfer, or equivalent.

    a. Type C cables tend to be more expensive than Type A, so it may be comparable or even cheaper to source a USB A to C adaptor, if your computer only features a Type C port. These adaptors are generally available.

[1] You'll want to check module voltage requirements and IO diagrams to verify Grove compatibility

b. For both cables and adaptors, as applicable (see 1a), think carefully about the necessary designation of plug vs. receptacle on each end. For cables, you'll likely want plugs on both ends. For a typical USB A to C adaptor, you'll find a Type A receptacle that gives way to a Type C plug, to be inserted into your PC.

3. Be sure that the breadboard you purchase is both *solderless* and equal to or greater in spatial dimensions than the standard 'half' size (5.5 x 8.5 cm), which we generally recommend for use in this context.

4. Here, 'jumper wires' refer to bundles of adjoined or individual lengths of wire that terminate in one of two possible forms: a 'male' pin (M) or a 'female' receptacle (F). To connect the camera module to our microcontroller board via the breadboard, you'll want to pick up about **25 or more** (individual wires) of the M-F variety. If you intend to go on to connect other off-board sensor modules, it may be helpful to have M-M jumpers on hand. Jumper wires can be purchased in various lengths. We suggest between 3 to 6 inches, here.

5. In searching for the OV7675 camera module, you may come across vendors selling the sensor itself, with only a ribbon cable breakout. Be sure to pick up a PCB breakout module that terminates in a 2x20 array of pins, like in the suggested link. If, for some reason, you can't obtain the OV7675 module, we have previously had success in using the related OV7670 module, but note, as above, that the edX course staff will only officially support the official kit we have developed in partnership with Arduino, including the requisite code for the OV7675 modules. Any adjustments to said code to call on the OV7670 module technically fall outside of the officially supported scope. With that said, we have noted in the course the required changes (fortunately there are very few) as we recognize that many students are finding it hard to source the OV7675.

C++ for Python Users

If you are comfortable with C/C++, please feel free to move on to the next reading. If you are new to C++, we hope this introductory material will be helpful for you.

Python, the language you have been using in all of your Colabs, is a dynamically-typed, “high-level” language that is interpreted at runtime. C++ (also written Cpp), on the other hand, is a statically-typed, “low-level” language that is pre-compiled before running, allowing for very compact code.

The good news is that since we are using the Arduino platform, we won’t have to deal with much of the complications of C++ as that is taken care of for us by the many libraries and board files we will be able to leverage (more on that soon). We’ll then just have to pay attention to the changes in syntax which are more cosmetic than functional. That is, all of the main loops and conditional statements (e.g., `for, if else`) remain the same functionally!

In order to make sure you feel comfortable we’ve put together a set of short appendix items and have collected some other good resources to walk you through the main changes between Python and C++ for Arudino which you can find below. As you go through the course if you have good ideas for other introductory material, changes to the current material, or additional resources, please let us know and we’d be very excited to add them to the list for future learners!

For more information take a look at our short appendix document that covers:

- Data Types
- Scope, Paraenthese, and Semicolons
- Functions
- Libraries, Header Files, `#include`
- Other General Syntax Points

Additional resources:

- The C++ Language Tutor
- The Google C++ Style Guide
- The Arduino Standard Library Language Reference

Setting up your Hardware

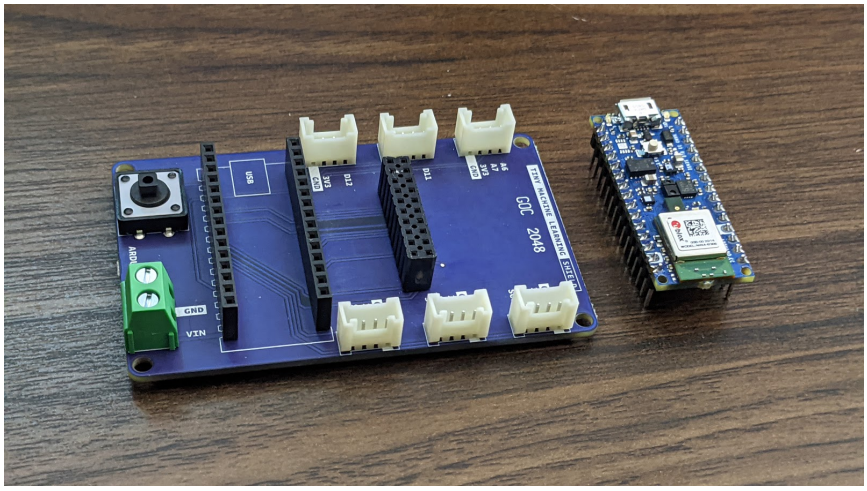
This reading will walk you through setting up your hardware whether you purchased the TinyML Kit or your own off-the-shelf parts.

Screencast of Brian walking through this reading goes here on the edX course

Setting up the TinyML Kit

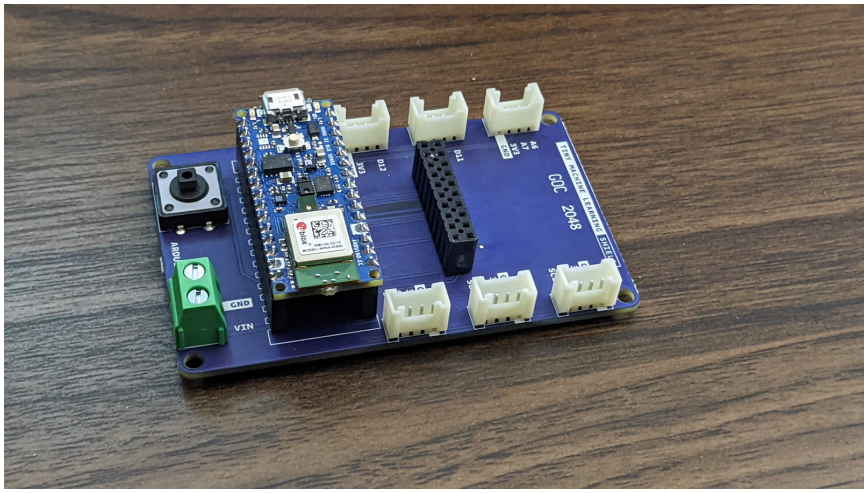
Certainly the easiest and most reliable way to obtain all of the parts necessary for this course is to procure the purpose-designed kit that we have developed with Arduino, available at this link for $49.99. If you ordered the official kit, then getting setup will be quick and easy. We outline the required steps below:

1. Slot the Nano 33 BLE Sense board into the Tiny Machine Learning Shield

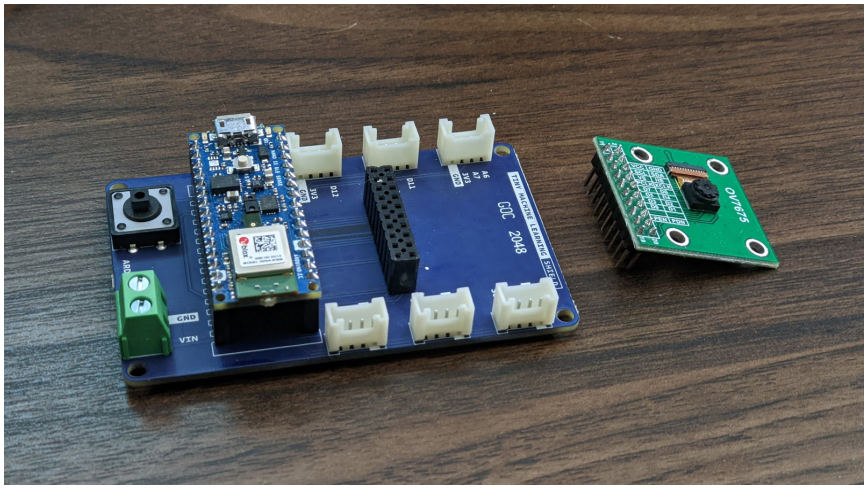


You'll want to target the pair of spatially separated 1x15 female headers. Carefully align the pins of the microcontroller board with the headers below and then gently push down

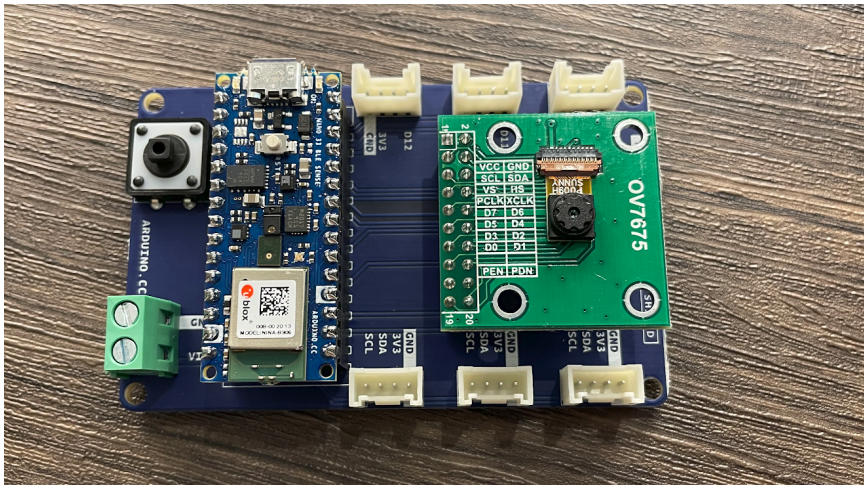
until the board is seated flush against the top of each header. The downward facing pins should no longer be visible. As best you can, avoid touching the components atop the board to prevent inadvertently damaging the surface mount devices. Pay attention to the orientation of the board so that the indication of the USB port on the PCB silkscreen matches the physical port on the board itself.



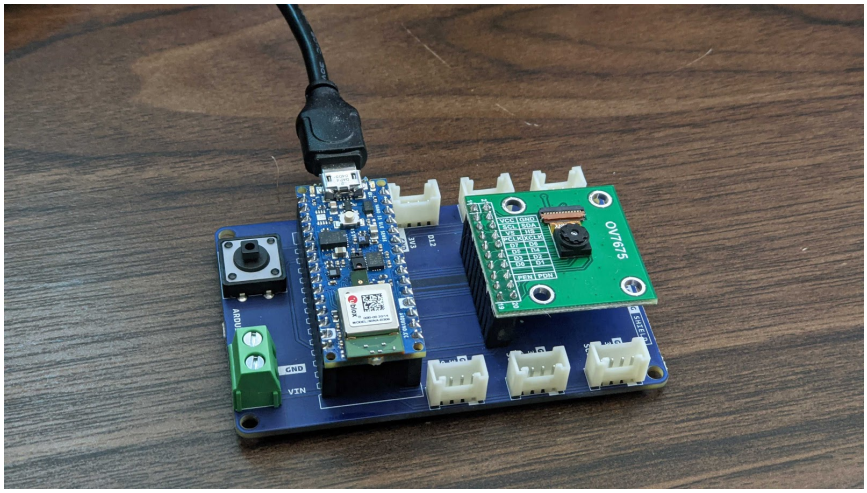
2. Slot the OV7675 camera module into the shield using the same technique



You'll want to target the 2x10 female header. Carefully align the pins of the camera module with the headers and then gently push down until the board is seated flush against the top of each header. The downward facing pins should no longer be visible. As best you can, avoid touching the camera module atop the board to prevent inadvertent damage. Pay attention to the orientation of the camera module so that the camera sensor is to the right of the header array (as shown), further from the microcontroller board than the header array.



3. Finally, use the provided USB cable (type-A to microB) to connect the Nano 33 BLE Sense development board to your machine. If your PC only features type-C USB ports, you will need to obtain an adaptor.



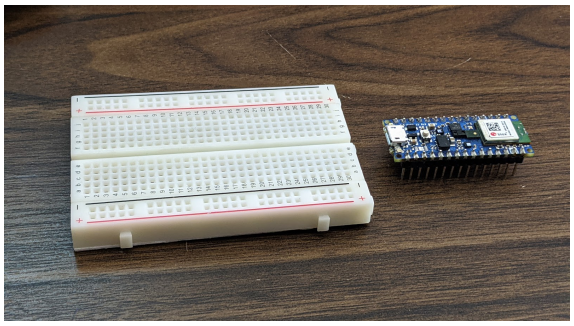
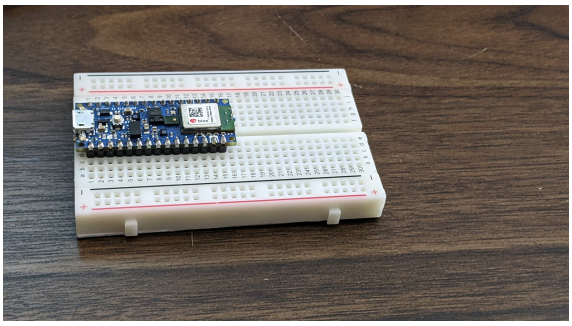
Note that if you are *only* calling upon hardware found on the Nano 33 BLE Sense development board (say the MCU and IMU), you could forgo connecting it to the Tiny Machine Learning Shield. If you need to remove either the Nano board or the camera module from the shield, grip each side of whichever board and pull back with a *gentle* rocking motion (back and forth) to work the pins out from the headers below.

And that's it! If you purchased the official kit you can move onto the next reading!

Wiring Up Off-the-Shelf Parts

While we generally encourage all to consider Arduino's Tiny Machine Learning kit as the primary and recommended hardware option, we understand that in certain circumstances, it may be beneficial, or even necessary, to obtain some or all of these parts via other means. If you have bought off-the-shelf components using our guide, we outline the steps you are likely to take in getting setup, below.

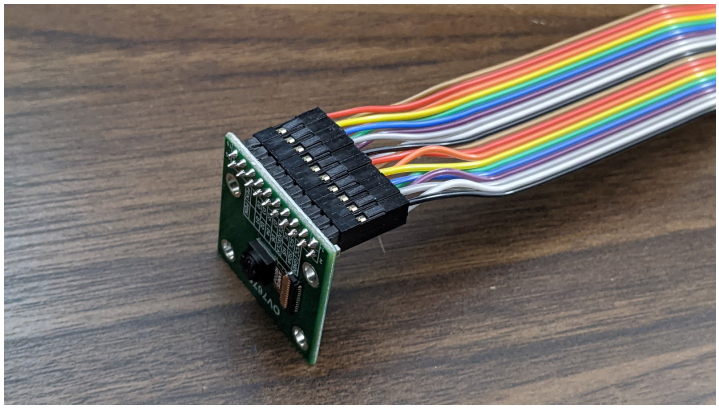
1. Slot the Nano 33 BLE Sense board into a solderless breadboard.

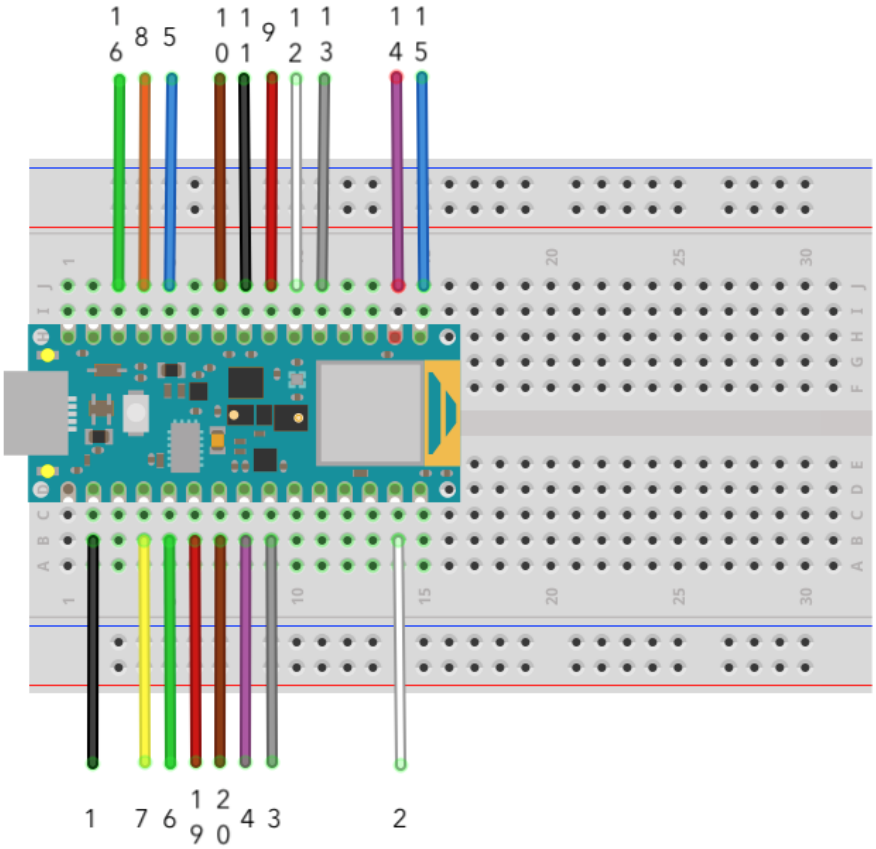
   If you aren't familiar with how breadboards can be used as a substrate for connecting various electrical components, you can take a moment to review the mechanism, here. As shown in the pictures above, we've targeted the left-hand side of the breadboard and seated the microcontroller board into the solderless breadboard along its length, with each of the 1x15 arrays of male pins on opposite sides of the 'trough' that runs through the middle of the breadboard. Given the board's width, one side will have three rows of receptacles beside it along the board while the other will have two. Which side is of no consequence. Note also that we've positioned the USB port at the end of the breadboard, to facilitate connections to your PC. So, altogether, you'll want to carefully align the pins of the microcontroller board with the receptacles below and then gently push down until the board is seated flush against the breadboard. The downward facing pins should not be visible. Avoid touching the components atop the board to prevent inadvertently damaging the surface mount devices.

2. With female-to-male jumper wire, use the following Fritzing (wiring) diagram, pinout diagrams, and connection table to link the OV7675 camera module to the microcontroller board via the solderless breadboard.

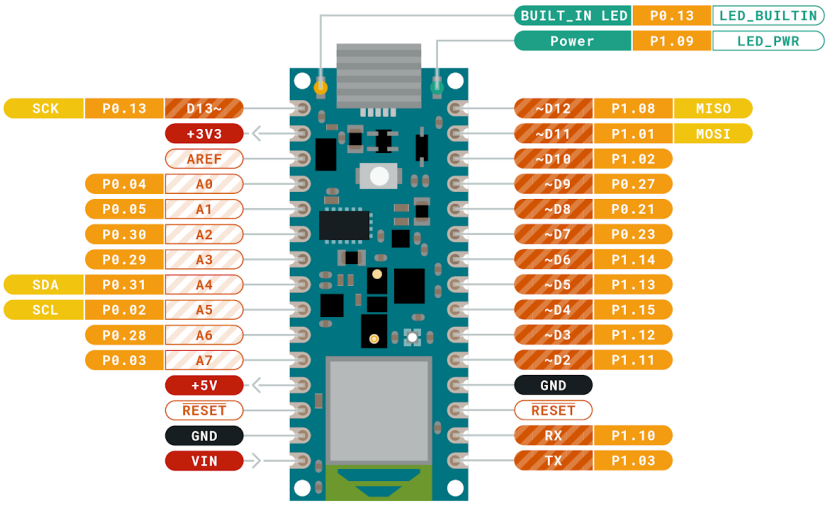
   You can start by isolating (tearing away, as applicable) 20 adjoined female-to-male jumper wires as shown in the first photo above. For clarity, keep the desired 20 wires connected together, but tear away any number above this. The sequence of colors is of no consequence. Next connect the female side of this wire assembly to the male pins of the camera module in an alternating pattern. To keep track of what each color represents, it may be easiest to have the left or right-most color in your wire assembly connect to pin 1, where the next color in your sequence connects to 2, and so on. That way, the camera module's pinout is now encoded by the sequence of colors.



3. Our next step will be to connect the camera module pinout (1-20) to specific pins on the microcontroller board via the solderless breadboard. Below we've mapped these OV7675 module pin numbers onto a fritzing (wiring) diagram for the Arduino Nano 33 BLE sense (assuming it is placed into a breadboard):



In case also helpful, we have also included the full pinout (designation) for the Nano 33 BLE Sense development board below as well as a table explaining the full pin connections needed from the OV7675 to the Arduino Nano 33 BLE Sense:



Description	Camera Module Pin	Microcontroller Board Pin
VCC / 3.3V	1	3.3V
GND	2	GND
SIOC / SCL	3	SCL / A5
SIOD / SDA	4	SDA / A4
VSYNC / VS	5	D8
HREF / HS	6	A1
PCLK	7	A0
XCLK	8	D9
D7	9	D4
D6	10	D6
D5	11	D5
D4	12	D3
D3	13	D2
D2	14	D0/RX
D1 (may be labeled D0)	15	D1/TX
D0 (may be labeled D1)[1]	16	D10
NC	17	--
NC	18	--
PEN / RST	19	A2
PWDN / PDN	20	A3

[1]Be careful, the silkscreen label on some OV7675 camera modules mistakenly swaps D0 and D1!

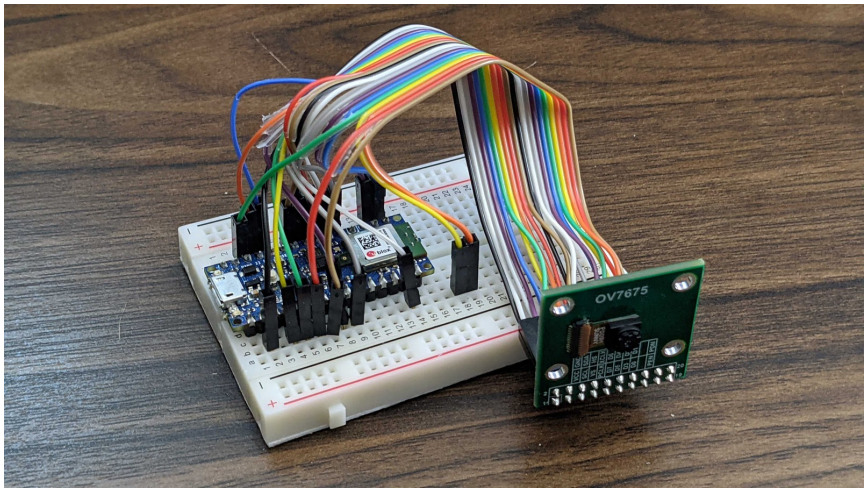
So, for example, the black wire shown in the top right of the image of the set of wires connected to the OV7675 above, is connected to the camera module's first pin (pin #1). We can directly see on the fritzing diagram that pin #1 from the OV7675 should be connected to the second to the top left pin on the Arduino. We can also see that according to the table above, we need to connect that to the Arduino's 3.3V pin, which we can find in the pinout diagram above as being the second pin on the top left.

Another example: the second wire (white) should then connect to either the second pin from the bottom on the left (as we marked on the fritzing diagram), or the fourth pin from the bottom on the right hand side of the MCU board, as either spot[2] will provide a connection to GND. Continue down the table (or around the fritzing diagram) until all pins have been connected, outside of pins 17 and 18, which are left unconnected. We choose to include wires for these pins, even though they serve no purpose, because their presence in connecting to the camera module makes the entire cable assembly physically more robust, which is favorable in maintaining a solid connection.

We want to underline that you should take your time in this process and verify each connection as you go, because most (nearly all) of the issues reported for this camera module stem from a wire that is connected to the wrong pin - perhaps only one location off or so. Precision here for the entire table is required for functionality.

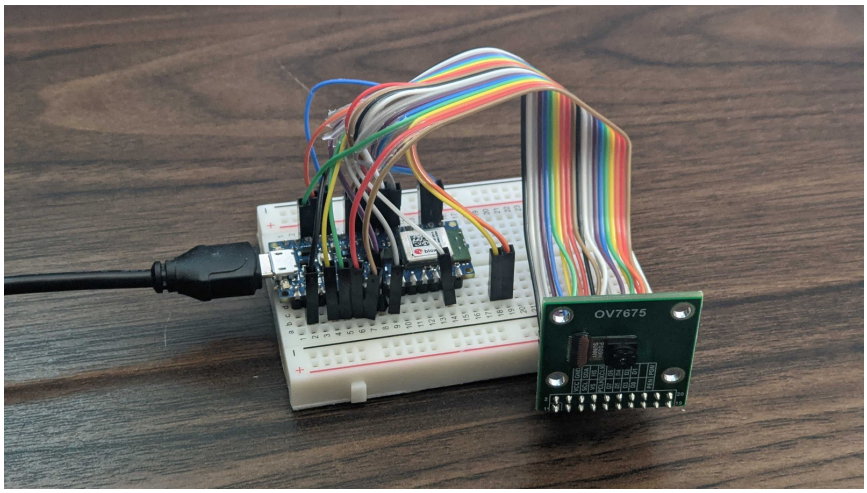
If you're interested, the communication between the controller and the camera module is standardized as a Serial Camera Control Bus (SCCB) as well as a Camera Parallel Interface (CPI) or, equivalently, Digital Video Port (DVP).

When you are done wiring up the camera your setup should like something like the following:



[2]Note that GND, or ground, is the only example where a pin designation is repeated.

4. Finally, use the provided USB cable (type-A to microB) to connect the Nano 33 BLE Sense development board to your machine. If your PC only features type-C USB ports, you will need to obtain an adaptor.



It's probably apparent why we are so excited that Arduino has developed the Tiny Machine Learning Shield for our course kits! :)

If you've sourced your jumper wires from SparkFun, you may be able to call on the same sequence of colors as we have above.

Changes for the OV7670 camera module

While the course staff will not officially support the OV7670 camera module, it exists as a viable alternative if the OV7675 is unavailable in your region. Fortunately, there is an Arduino blog post concerning using the OV7670 module for tinyML.

IMPORTANT NOTE: The connection table in the Arduino blog post is out of date, you should use the same connection table as for the OV7675 listed above. The one change is that the OV7670 only has 18 pins. Therefore you need to omit the two NC pins between D0/D1 and RST/PWDN.

Setting up your Software

In this reading, we will walk through setting up the software you'll need for this course, the Arduino integrated development environment (IDE). We will be using the Arduino IDE to program your microcontroller.

Introduction

What is Arduino?

Arduino describes itself as "an open-source electronics platform based on easy-to-use hardware and software." In large part, this description is fitting, as the company designs and sells a collection of microcontroller development boards that simplify deployment of embedded hardware alongside a software framework that abstracts away all but the most relevant considerations for your application. Perhaps what's under-represented by this description is the role that the surrounding community plays in enabling many plug-and-play experiences given the number and quality of auxiliary hardware modules, support libraries, and tutorials that have and continue to be produced within Arduino's ecosystem.

One thing we'd like to acknowledge here is that Arduino's mission of creating easy-to-use hardware and software has the necessary tradeoff of limiting the feature set of its development environment to the essentials.

What is an Integrated Development Environment (IDE)?

As is true for all forms of programming, an integrated development environment, or IDE, is an application with a feature set that facilitates software development generally or within a particular niche. The Arduino Desktop IDE is highly specific in the sense that it is intended to facilitate software development for a specific set of microcontroller boards, in C++.

Within the niche of IDEs for embedded software, there is a noticeable dichotomy between light-weight applications, (like Arduino's IDE) that minimize functionality and abstract away details in the name of simplicity and full-featured IDEs (like Keil µVision and Visual Studio Code) that exist to support industry professionals.

If you're interested in finding a happy medium to use in future embedded projects, our staff have had good experience using VS Code with the hardware specific extension PlatformIO, that together enable nice-to-have features like line completion, reference tracking, et cetera, without all of the complexity that more advanced IDEs introduce. Having said this, this particular combination of VS Code + PlatformIO will not be officially supported within this course.

What is the Arduino IDE?

As you might expect given the description above, the Arduino IDE is a lightweight development environment with features that permit you to very quickly manipulate microcontroller development boards. While there is a cloud-based offering (the Arduino Create Web Editor) as well as a so-called Arduino Pro IDE that is in development (specifically in the alpha stage at the time of writing), we are going to use the standard Arduino Desktop IDE in this course.

Downloading and Installing the Arduino IDE

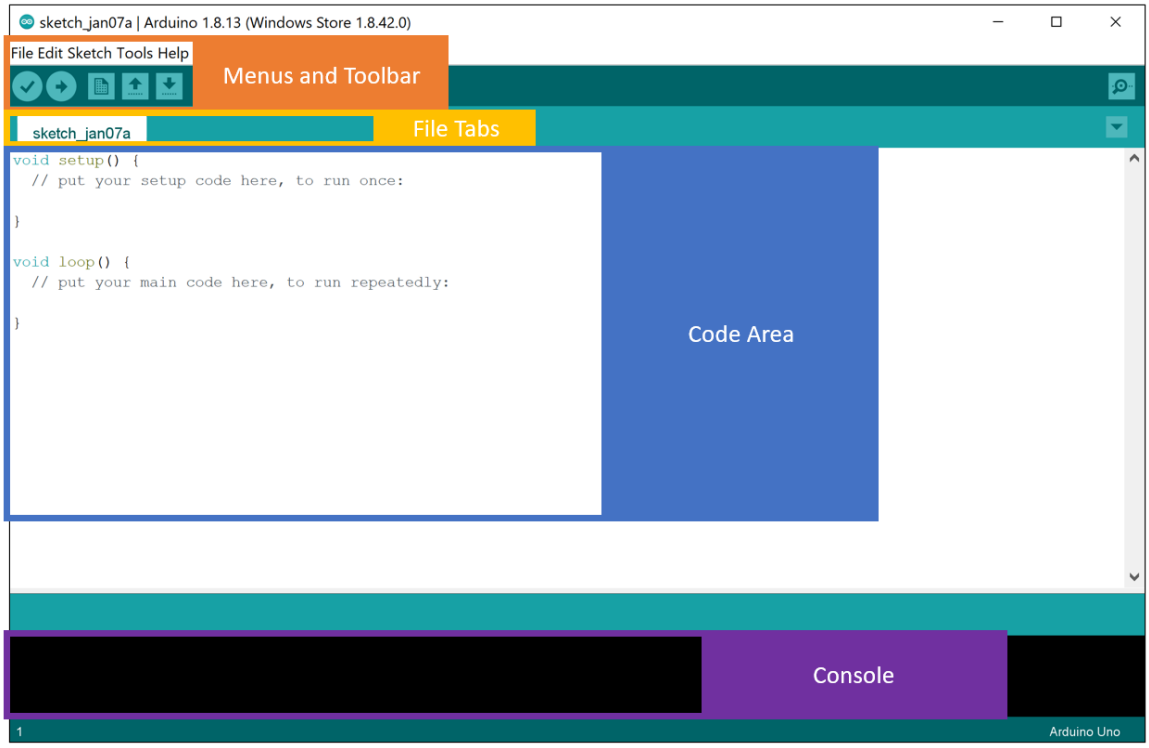
Note: Our staff will only officially support the latest release of the Arduino Desktop IDE (v1.8.13). If you have an older version of the Desktop IDE, we recommend that you update your software.

1. Navigate to arduino.cc and at the top of the page select Software and click Downloads
2. Click on the download link appropriate for your machine
3. There is no obligation to contribute, you can click 'Just Download' to proceed

Operating specific installation instructions vary, so if what follows isn't self evident, you can find guidance, by OS, at the following links:

- Windows
- Mac OS X
- Linux

A Quick Tour of the IDE



When you first open the IDE you will see a screen that looks something like the above screenshot.

Up at the top of the IDE you will find the menus and the toolbar which we will explore as we work through the rest of this document and when we run the tests later in this section.

Below that you will find the file tabs. For now there will only be one file open that is most likely named sketch_<date>, however later in the course when we work through more complicated examples you will see all of the various files that make up the project across the file tab area. In “Arduino speak,” a sketch is a simple project/application.

In the middle of the screen you will see the large code area. Each sketch can consist of many files and use many libraries but at its core each sketch is made of two main functions, “setup” and “loop”, which you can see pre-populated in the code area. We’ll explore what those functions are used for through the Blink example in a future video. For now, let's continue orienting ourselves with and setting up the IDE.

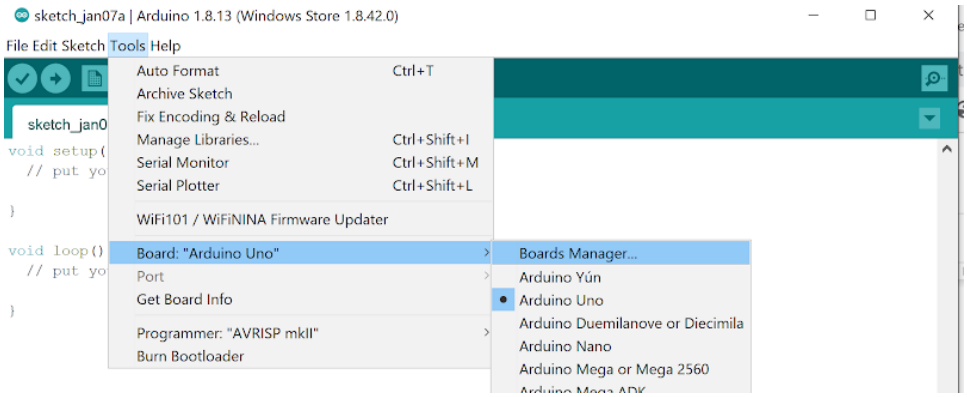
Finally, at the bottom of the screen you’ll find the console. This is where you will see debug and error messages that result from compiling your C++ sketch and uploading it to your Arduino.

Installing the Board Files for the Nano 33 BLE Sense

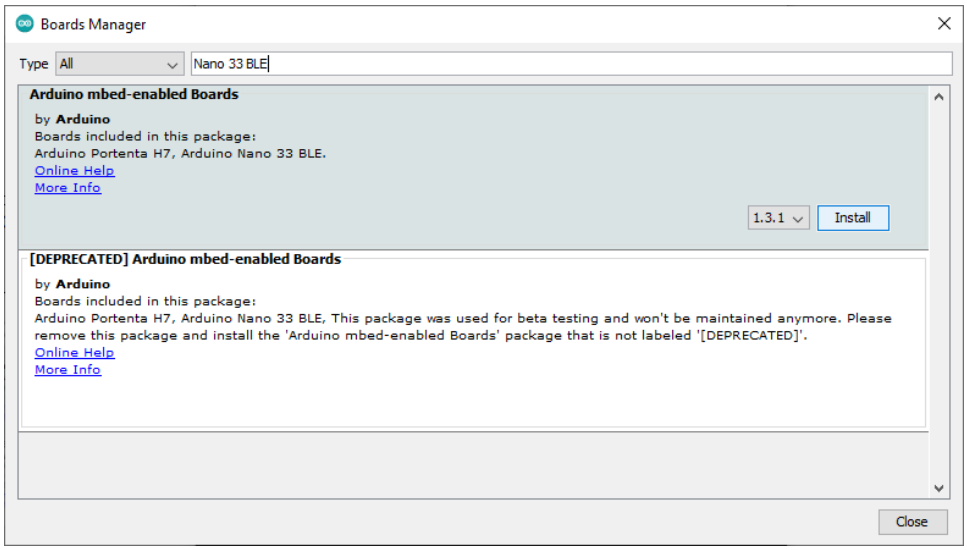
One of the primary advantages that the Arduino ecosystem affords is the portability of code you write for one or another board within their line-up or even in porting code to affiliate boards. This is made possible by the support files organized in the Boards Manager, which coordinates a download and installation of files that detail the Arduino functions (sometimes 'core') that are defined for that particular board (which is how hardware differences between boards are abstracted) as well as compiler or linker details specific to the given board.

To install the board files you will need for your Arduino Nano 33 BLE Sense please do the following:

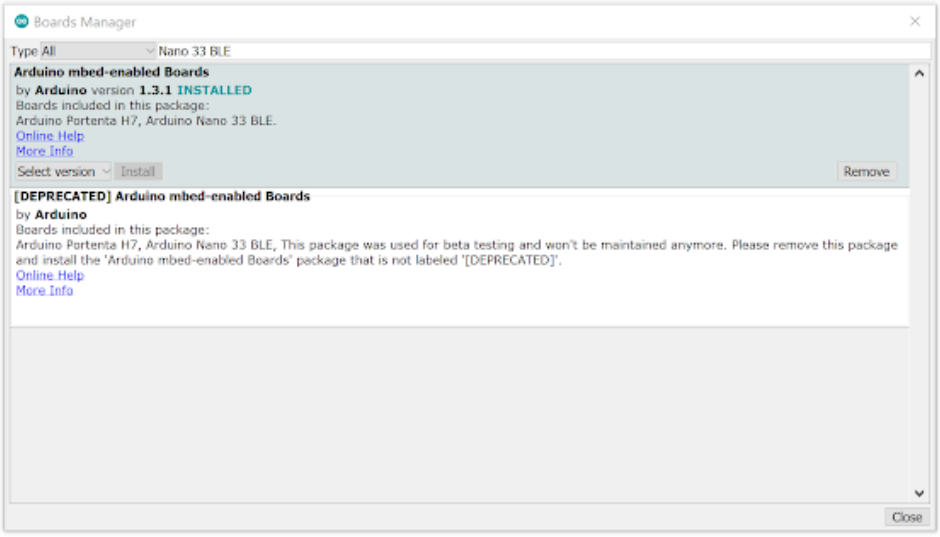
1. Open the Boards Manager, which you can find via the Tools drop-down menu. Navigate, as follows: `Tools → Board → Boards Manager`. Note that the Board may be set to "Arduino UNO" by default



2. In the Boards Manager dialog box, use the search bar at the top right to search for “Nano 33 BLE,” which should bring up two results. We’re interested in the first result (as shown), named “Arduino mbed-enabled Boards,” (without the DEPRECATED tag). Make sure **Version 1.3.1** is selected and then click “Install.” As the install process progresses you will see a blue completion bar work its way across the bottom of the Board Manager window. Be patient, you may need to install USB drivers which requires you to approve an administrator privileges popup which can take a couple minutes to appear.



After you have successfully installed the board if you exit and re-open the Board Manager and search again for “Nano 33 BLE” you will now see a green INSTALLED next to the library and the option to “Remove” the library or install a different version.

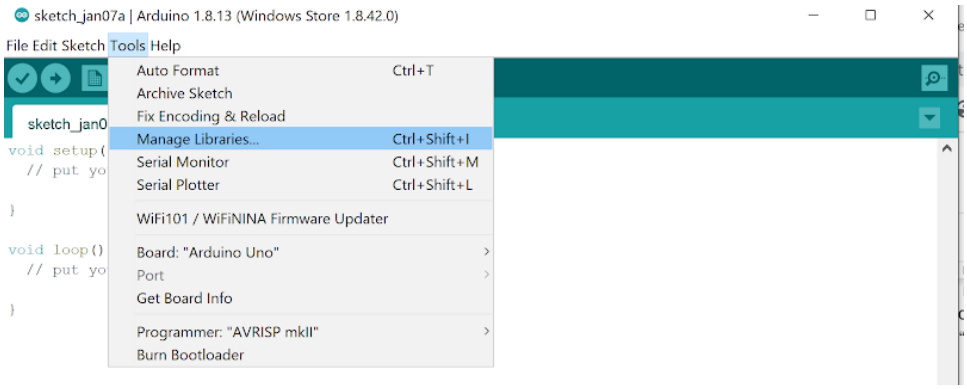


Installing the Libraries needed for this Course

Another advantage of the Arduino ecosystem is the availability of a wide array of libraries for performing various tasks, such as interfacing with a sensor module or manipulating data using common algorithms. There are many libraries that can be accessed from within the Library Manager in the Arduino IDE as described below. Check here for a complete list.

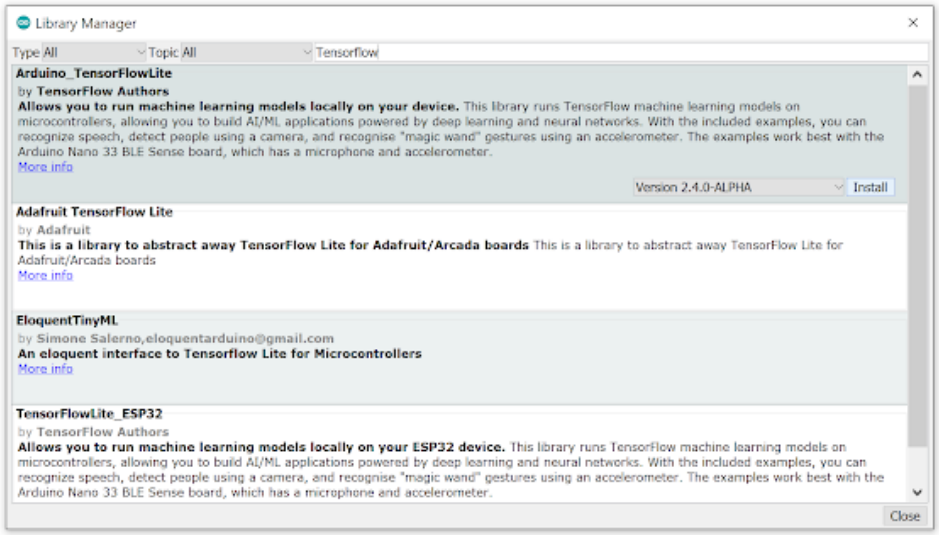
For this course we are going to need four libraries. To install the libraries please do the following and **make sure to install the version specified in the reading below or the tinyML applications will not work**:

1. Open the Library Manager, which you can find via the Tools drop-down menu. Navigate, as follows: `Tools → Manage Libraries.`



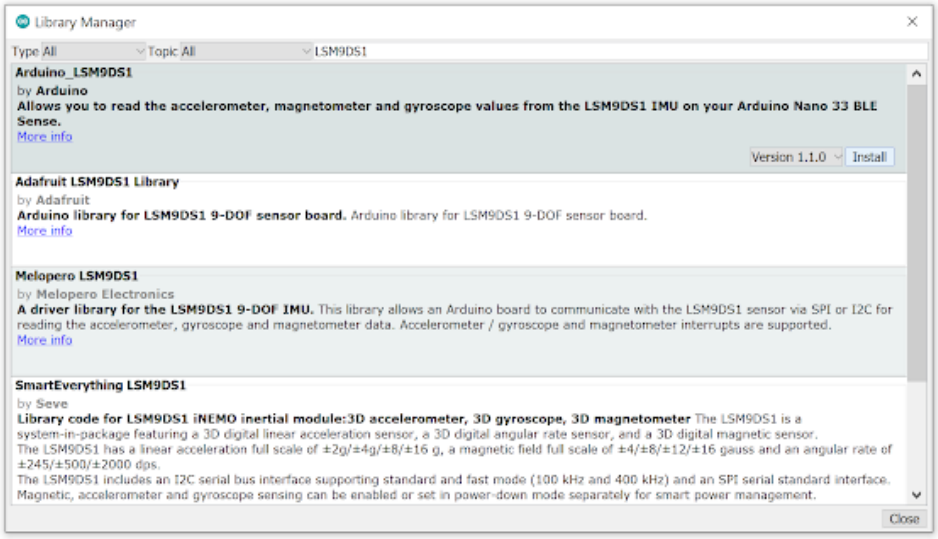
2. Then, much like for the Boards Manager, in the Library Manager dialog box, use the search bar at the top right to search for the following libraries, one at a time. Note that like with the Board manager a blue completion bar will appear across the bottom of the Library Manager window.

    a. The Tensorflow Lite Micro Library
        i. Search Term: Tensorflow
        ii. Library Name: Arduino_TensorFlowLite
        iii. Version: 2.4.0-ALPHA

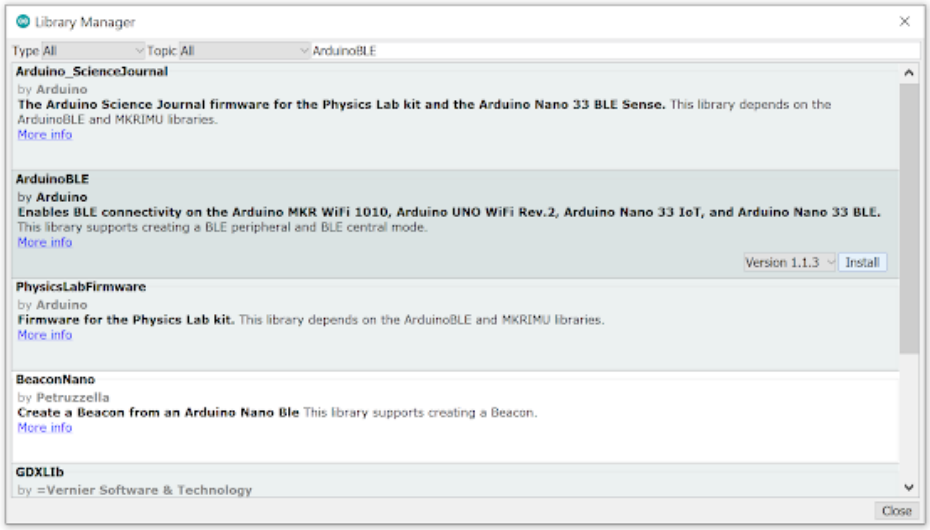


    b. The Harvard_TinyMLx Library we put together for this course!
        i. Search Term: TinyMLx
        ii. Library Name: Harvard_TinyMLx
        iii. Version: 1.0.1

c. The library that supports the accelerometer, magnetometer, and gyroscope on the Nano 33 BLE sense
    i. Search Term: LSM9DS1
    ii. Library Name: Arduino_LSM9DS1
    iii. Version: 1.1.0



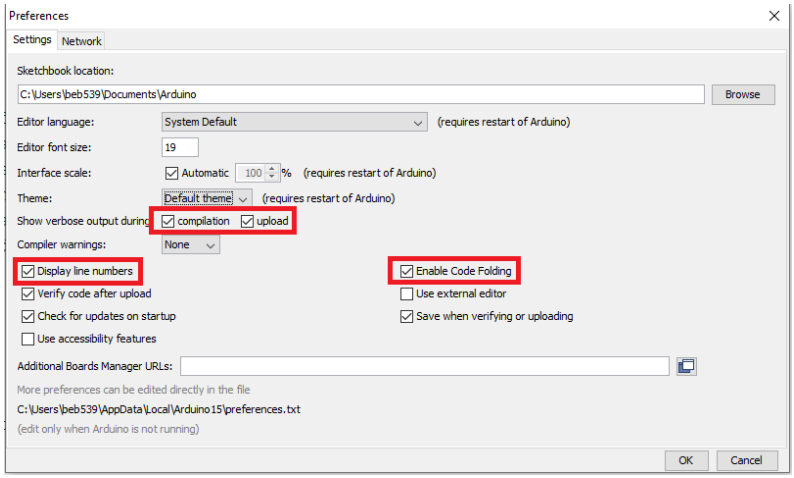
d. ArduinoBLE
    i. Search Term: ArduinoBLE
    ii. Library Name: ArduinoBLE
    iii. Version: 1.1.3



Setting your Preferences

You can adjust the preferences set for the Arduino Desktop IDE via the File drop-down menu, `File → Preferences`. There are a few preferences that we recommend enabling to make the Arduino IDE a little easier to use, namely:

1. Show verbose output during: compilation and upload
2. Enable code folding
3. Display line numbers



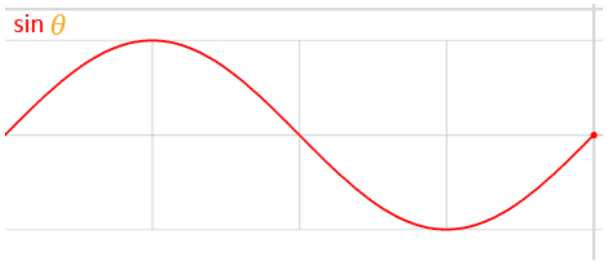
<Alt-text: Screenshot of the preferences window showing how to enable code folding, display line numbers, and show verbose output as described in the reading.>

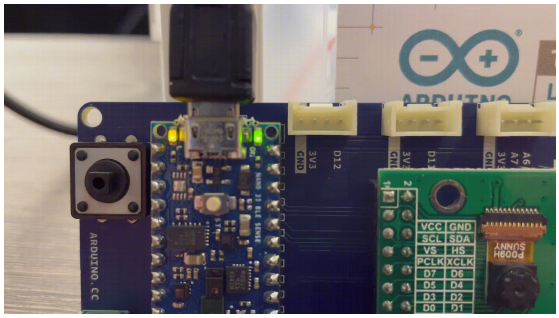
Of final note if you don't like the default theme for the Arduino Desktop IDE there is a nice tutorial for a dark theme you can find here. Also if you would like to learn more about the IDE, check out Arduino's documentation.

And that's it! Your Arduino IDE should be all configured for this course. Now that you have all of the necessary board files and libraries installed it's time to explore more of the features of the IDE available under the "Tools" menu and start to test out your Arduino by deploying the Blink example!

Testing the TFLM Installation

In this reading, we are going to walk through deploying the 'Hello World' example provided in the TensorFlow Lite for Microcontrollers (TFLM) library. The Hello World application runs a neural network model that can take a value, x, and predict its sine, y. Visually, the sine wave is a pleasant curve that runs smoothly from –1 to 1 and back. This makes it perfect for controlling a visually pleasing light show! We'll be using the output of our model to power a smoother blink of your Arduino LED.



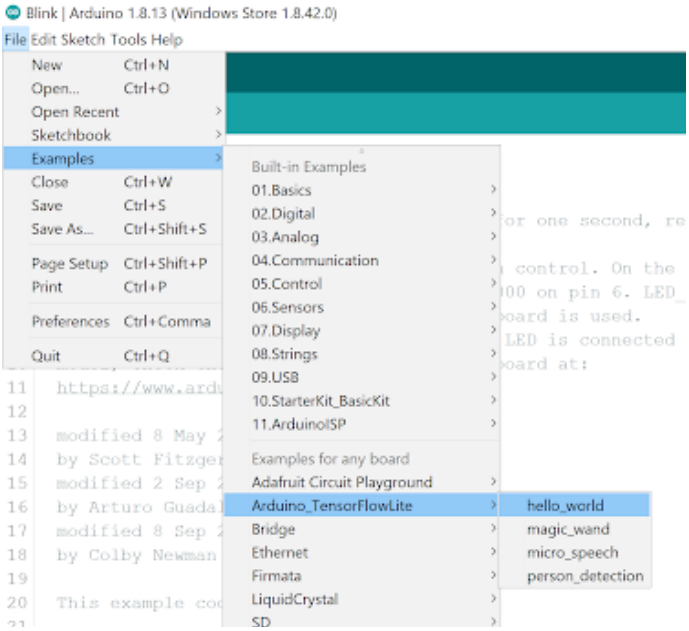


While the result won't differ greatly from what we've accomplished with standard Arduino Blink sketch we just deployed (in either case, an LED will be turning on and off), in this instance we are going to be deploying a model. Previously, we have learned about how neural networks can learn to model patterns they see in training data to go on to make predictions. Here, the model we deploy will call on maps an input (x) to an output y = sin(x). While this is admittedly contrived, it will allow us to quickly evaluate if the TFLM stack is functional on the hardware by using this output y to control the intensity of light emitted by the on-board LED. As promised, below you'll find instructions, screenshots, and videos walking you through the process!

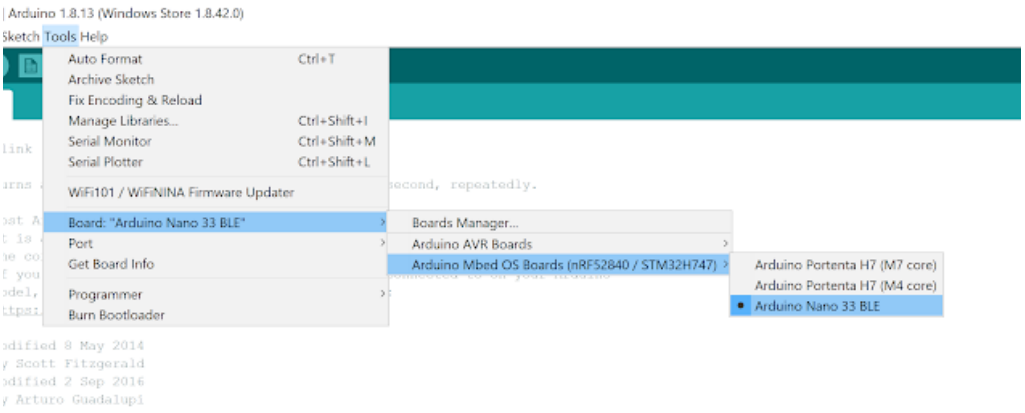
Screencast of Brian walking through this section goes here on the edX course

Preparing for Deployment

1. Use a USB cable to connect the Arduino Nano 33 BLE Sense to your machine.
2. Open the hello_world.ino sketch, which you can find via the File drop-down menu. Navigate, as follows: `File → Examples → Arduino_TensorFlowLite → hello_world`.

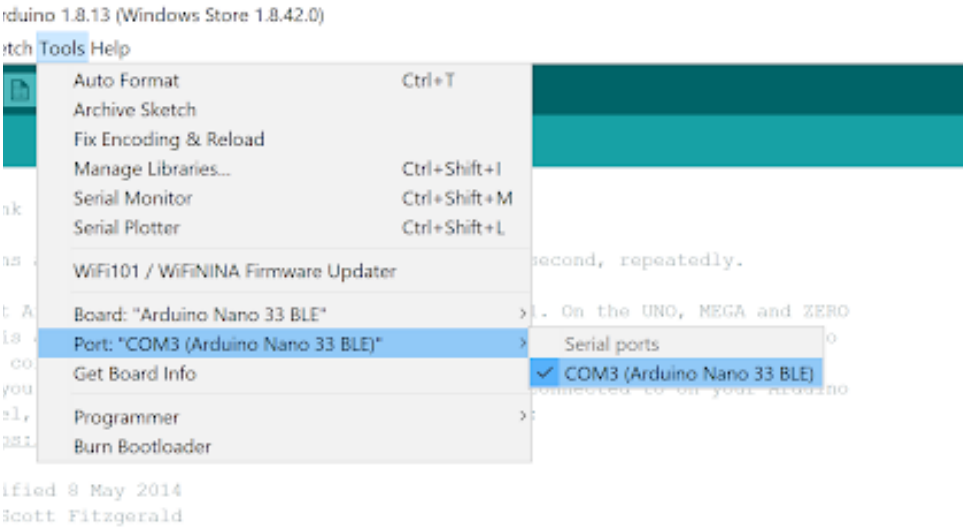


3. Just like with the Arduino Blink example (and as we will do for every deployment) Use the Tools drop down menu to select appropriate Port and Board.
    a. Select the Arduino Nano 33 BLE as the board by going to `Tools → Board: <Current Board Name> → Arduino Mbed OS Boards (nRF52840) → Arduino Nano 33 BLE.` Note that on different operating systems the exact name of the board may vary but/and it should include the word Nano at a minimum. If you do not see that as an option then please go back to Setting up the Software and make sure you have installed the necessary board files.

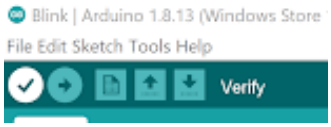


b. Then select the USB Port associated with your board. This will appear differently on Windows, macOS, Linux but will likely indicate ‘Arduino Nano 33 BLE” in parenthesis. You can select this by going to `Tools → Port: <Current Port (Board on Port)> → <TBD Based on OS> (Arduino Nano 33 BLE).` Where <TBD Based on OS> is most likely to come from the list below where <#> indicates some integer number

   i. Windows → `COM<#>`
   ii. macOS → `/dev/cu.usbmodem<#>`
   iii. Linux → `ttyUSB<#>` or `ttyACM<#>`



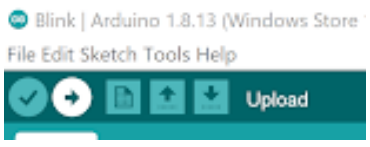
4. As always it is best practice to then verify that the code is valid. Note that this may take a while the first time you run this as all of TFLM is being compiled (it will compile much faster in the future). Also, you may, or may not, see a series of compiling warnings (but not errors) as the code compiles, this is entirely normal either way and is the result of TFLM being bleeding edge software.



Deploying (Uploading) the Sketch

Once we know that the code at hand is valid, we can flash it to the MCU:

1. Use the rightward arrow next to the ‘compile’ checkmark to upload / flash the code.

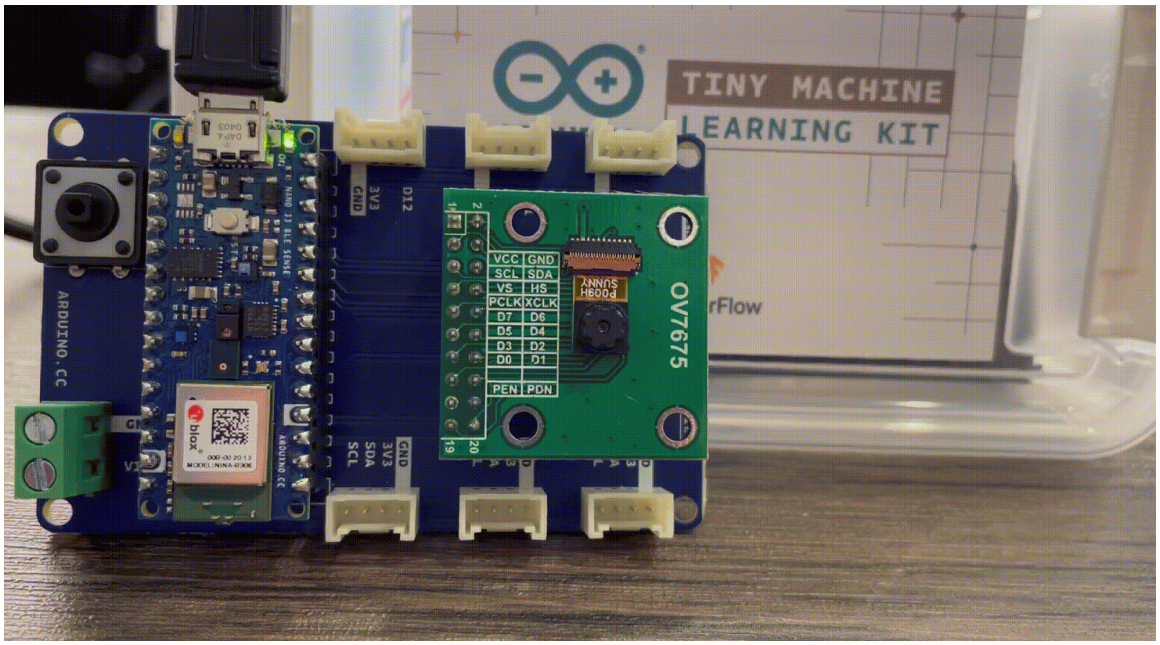


You'll know the upload is complete when you red text in the console at the bottom of the IDE that shows 100% upload of the code and a statement that says something like "Done in <#.#> seconds."

If you skipped the verification step do note that this may take a while the first time you run this as all of TFLM is being compiled (it will compile much faster in the future). Also, you may, or may not, see a series of compiling warnings (but not errors) as the code compiles, this is entirely normal either way and is the result of TFLM being bleeding edge software.

If you receive an error you will see an orange error bar appear and a red error message in the console (as shown below). Don't worry -- there are many common reasons this may have occurred. To help you debug please check out our FAQ appendix with answers to the most common errors!

2. At this point, you'll want to look to the board itself. The orange LED opposite the green LED power indicator about the USB port should now be *fading* at the rate set by the sinusoidal model. Note that the default rate is quite fast so it should almost appear to be blinking very quickly. However, if you look closely you'll notice that it is fading in and out vs blinking.



Congratulations! At this point, if all has gone well, you can be sure that your embedded microcontroller is configured properly to run TensorFlow Micro.

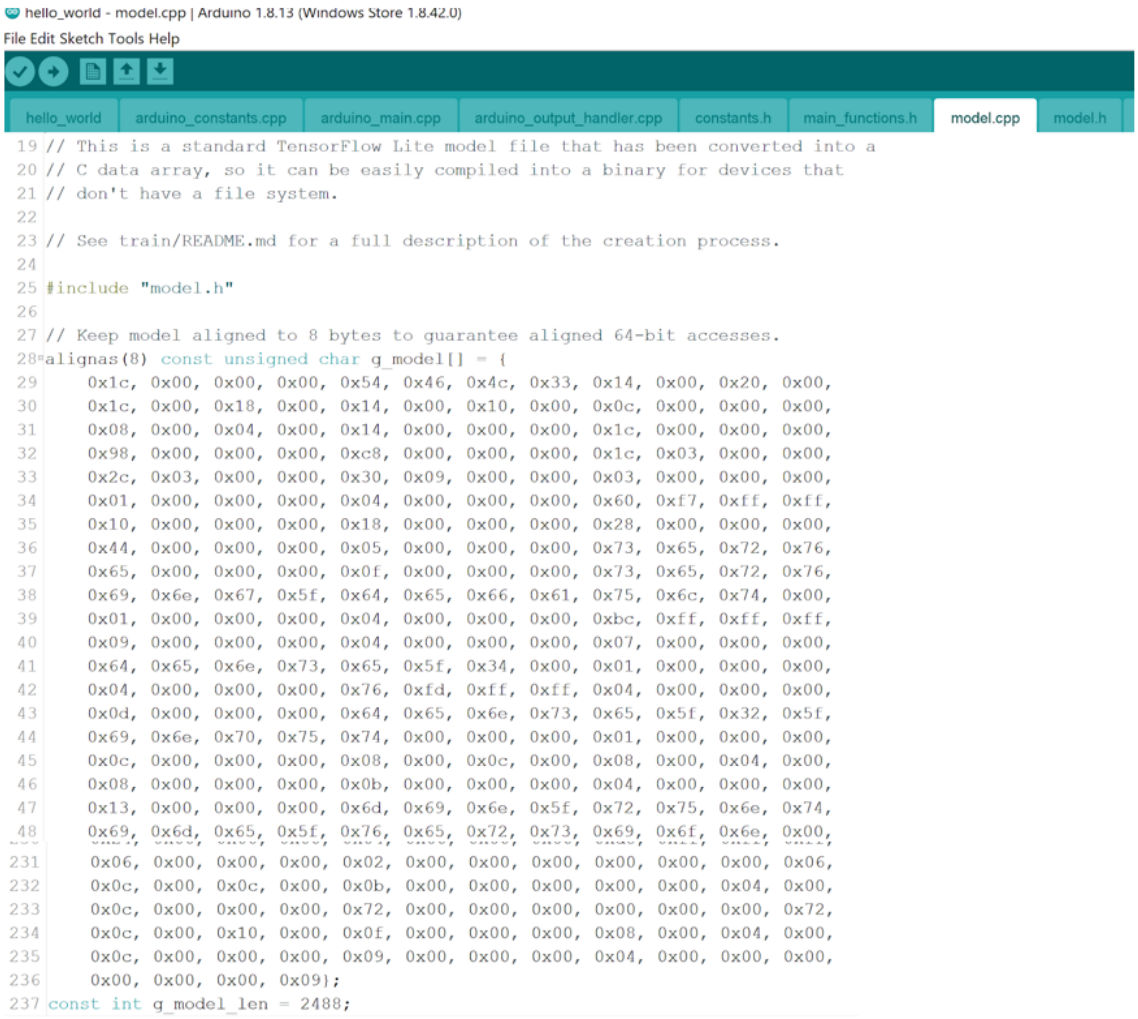
Understanding the TFLM Hello World Example

The first thing you will notice when looking at this example is that there is a lot more code in this TFLM version of blink! However, at a high level the code is structured the same. There is still a `setup()` function that runs once to initialize things and a `loop()` function that runs continuously to blink the LED.

As far as the high level differences go, in the main hello_world file, before either of those functions, there are a set of `#include` lines and global variable definitions. The `#include` lines are used to (in Python speak) import the various libraries we need for this example. In this case it is the TFLM library and some other helper functions. The global variables are defined outside of the `setup()`and `loop()` functions so that there are references to variables that can be used in both the `setup()` function as well as in every loop of the `loop()` funcion. For example, the global `tensor_arena` memory is used to initialize and store our neural network once and then is used to execute the neural network every loop iteration. This is much more efficient than re-initializing the neural network in each loop iteration.

One other high level difference is that you'll notice that there are a series of tabs across the top of the IDE window. These tabs each represent different files that make up this hello_world project. Each holds some helper functions or definitions that our main hello_world file uses.

For example, the model.cpp file contains a large array which contains the binary data for the neural network model (in fact it is simply the binary version of the TensorFlow Lite file) as well as an integer which tells TFLM how long that array is. We'll explore how that binary file is generated later in the course!



Getting back to the main hello_world file, as mentioned above, the `setup()` function has a decent amount of code in it but at a high level it does exactly what you would expect, it initializes our neural network.

The `loop()` function also has a decent amount of code in it but again at a high level all it does is keep track of a counter over time for the x-value and uses the neural network to compute the approximate y-value of y = sin(x) and then set the brightness of the LED according to that approximate y value.

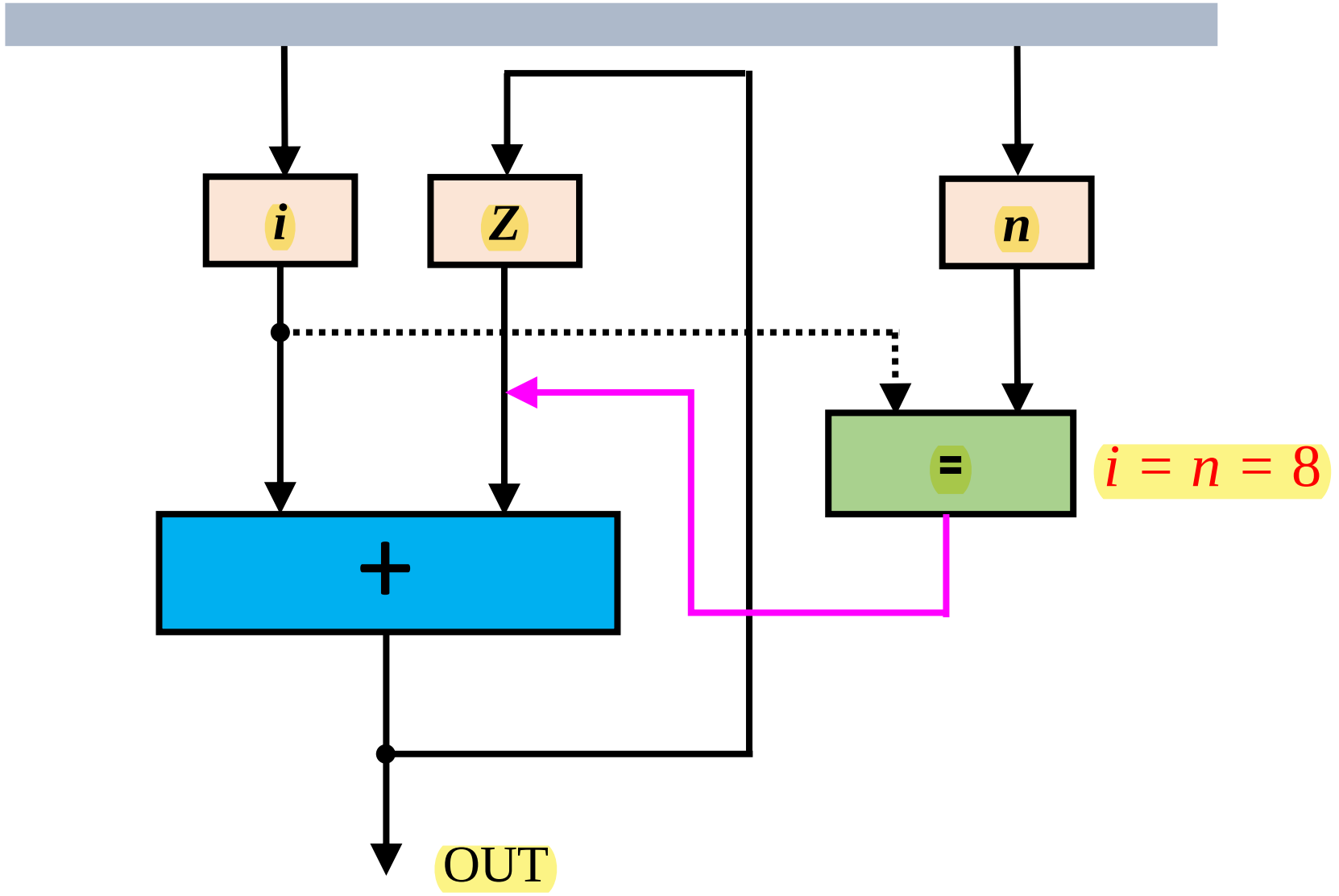
Now there are a lot of details that go into making all of those helper functions work, and excitingly, Pete Warden, the Technical lead of TensorFlow Mobile and Embedded team at Google will dig deeper into many of these topics in more detail later on. But before that you'll need to run one more set of sketches to test your sensors!

AI/ML Architectures

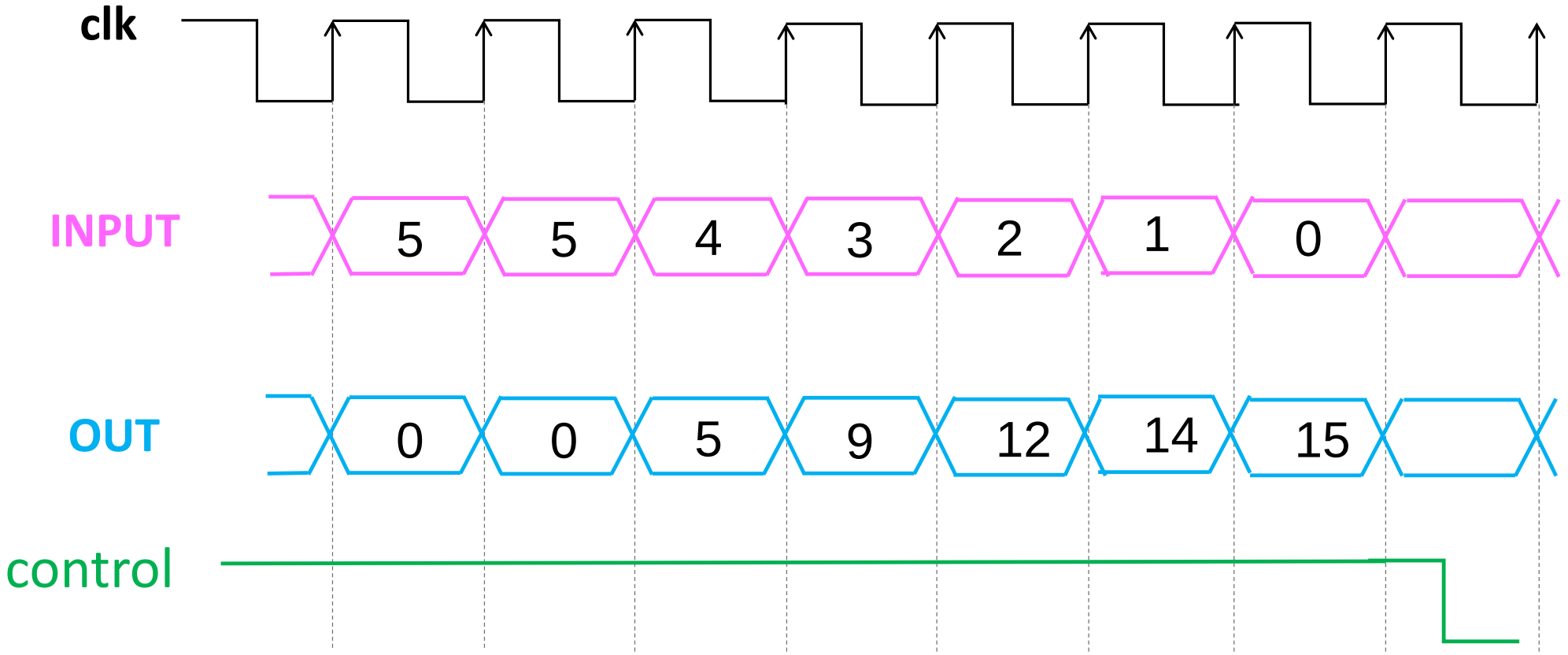
Compute $Z = \sum_{i=1}^{n=8} X(i)$

```
INPUT n      # Here n = 8
Z = 0
    FOR i = 1 to n
    Z = Z + 1
    NEXT i
OUTPUT Z
```

Architecture Computes $Z = \sum_{i=1}^{n=8} X(i)$



Timing Analysis



AI Model Computes

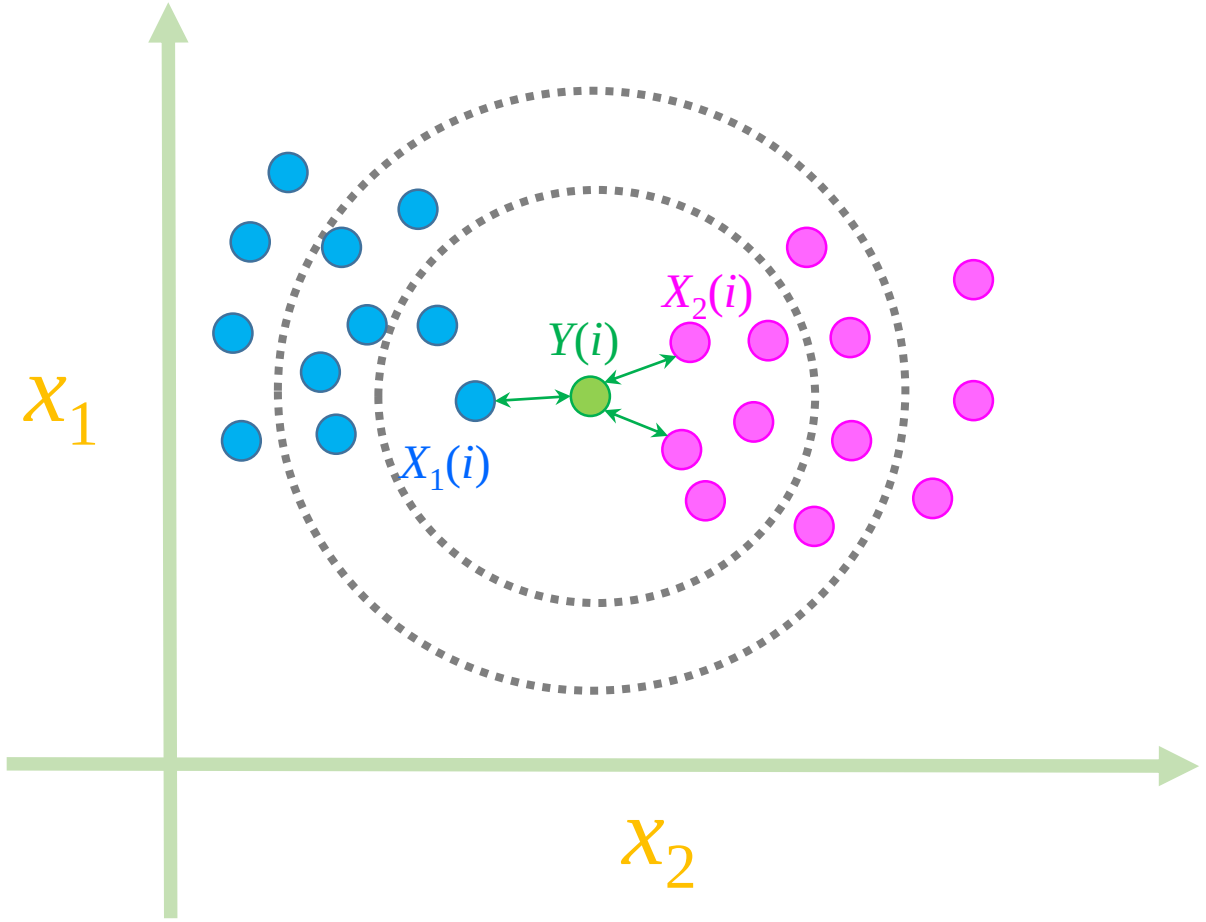
Manhattan Distance: $Z = \sum_{i=1}^{n} \left| X(i) - Y(i) \right|$

Euclidean Distance: $Z = \sum_{i=1}^{n} \left| X(i) - Y(i) \right|^2$

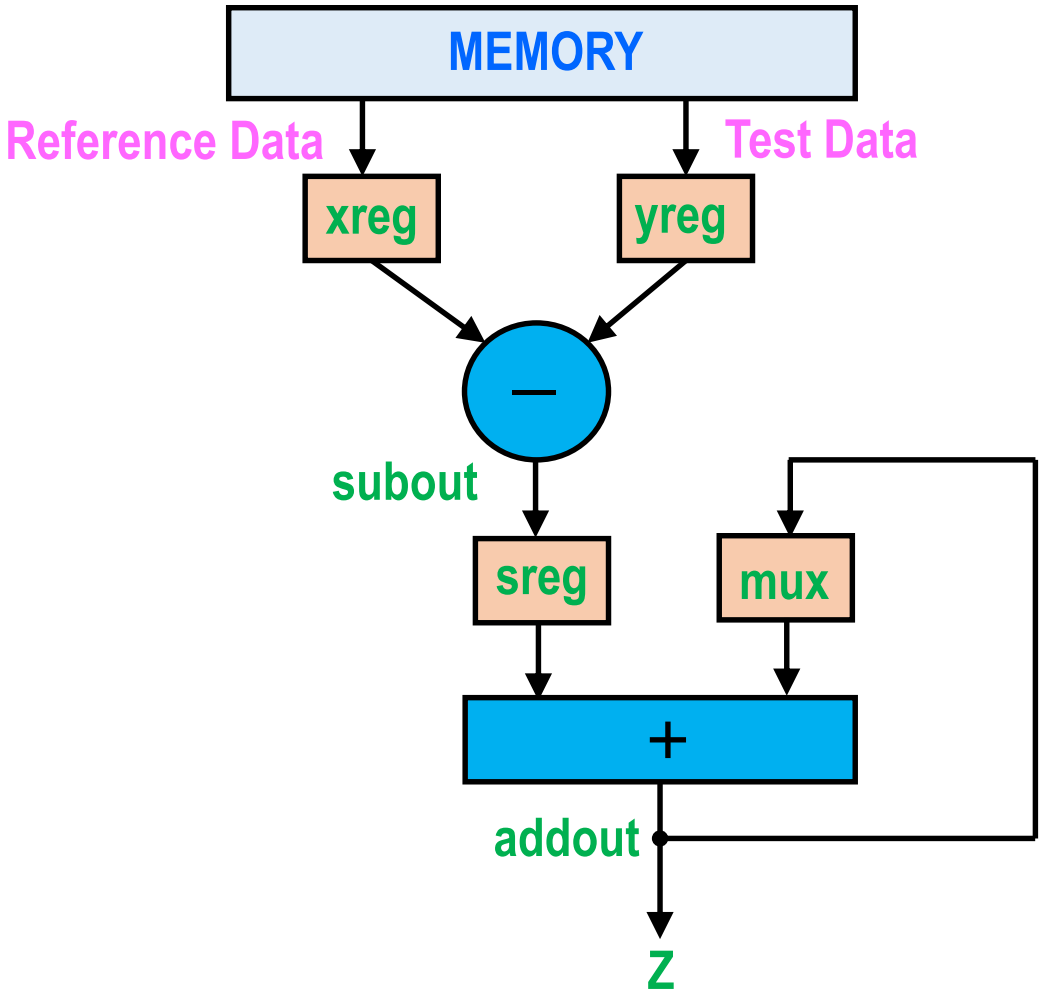
Support Vector Machine (SVM): $Z = \exp\left(-\gamma \left| X(i) - Y(i) \right|^2 \right)$

Neural Networks: $Z = \frac{1}{1 + \exp\left[-w(i) \bullet x(i) \right]}$

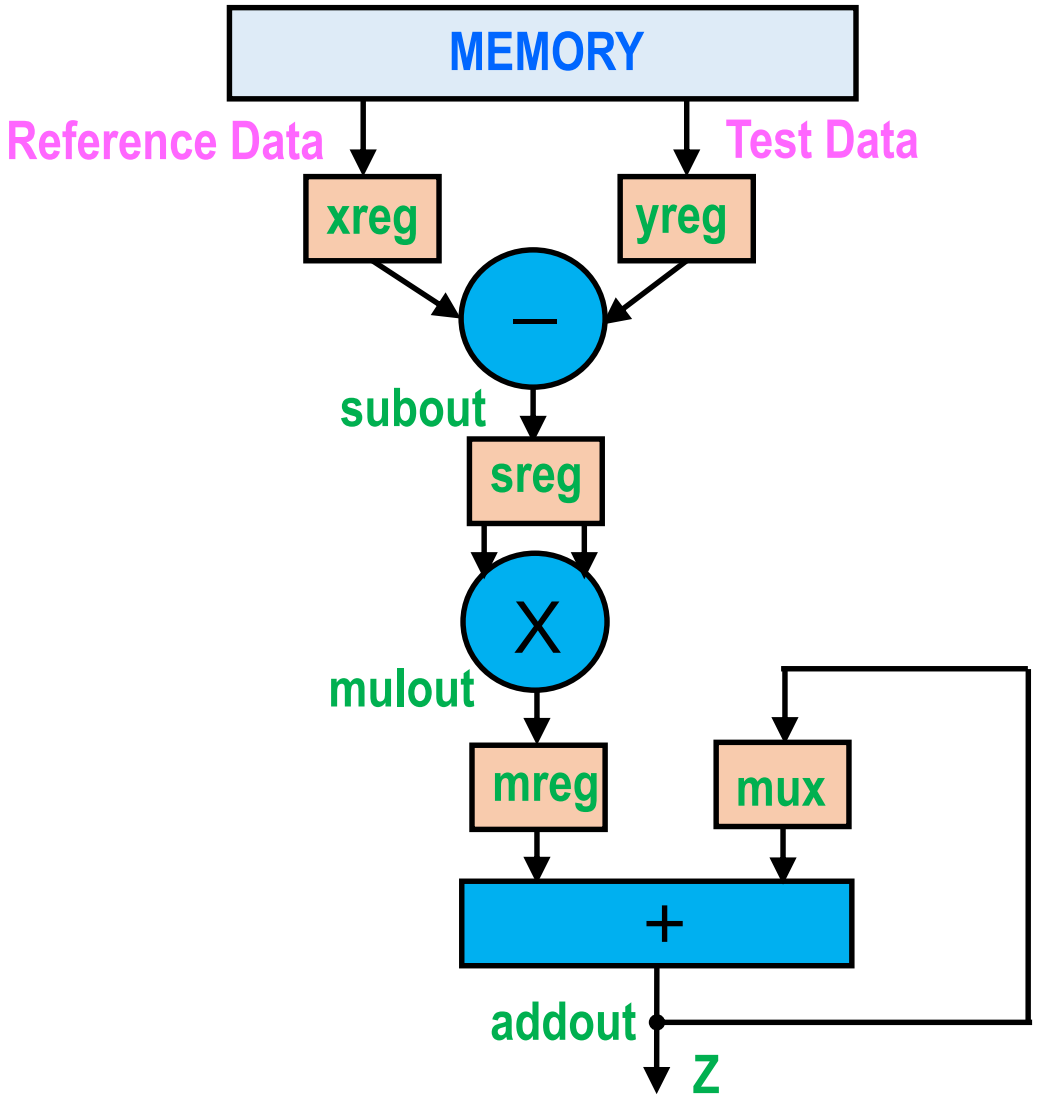
Manhattan Distance: $Z = \sum_{i=1}^{n} |X(i) - Y(i)|$



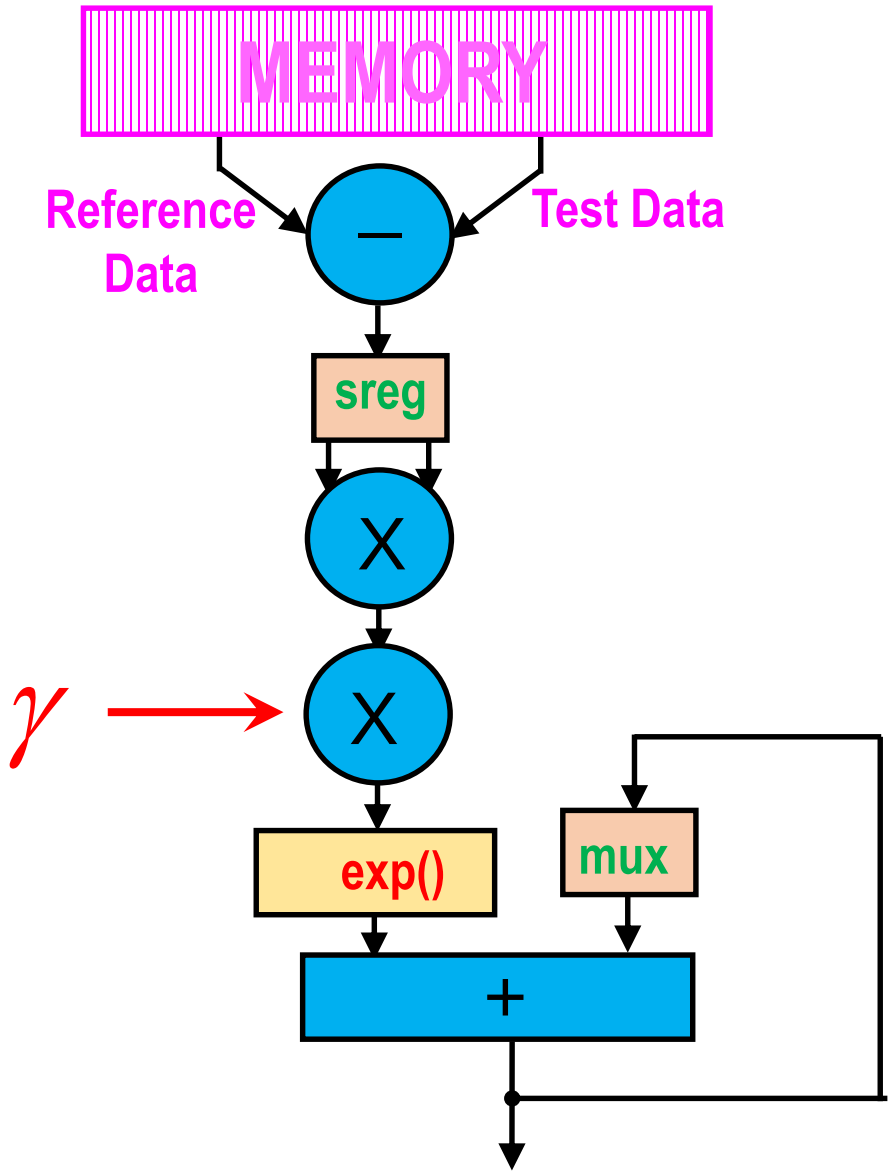
Manhattan Distance: $Z = \sum_{i=1}^{n} |X(i) - Y(i)|$



Euclidean Distance: $Z = \sum_{i=1}^{n} |X(i) - Y(i)|^2$

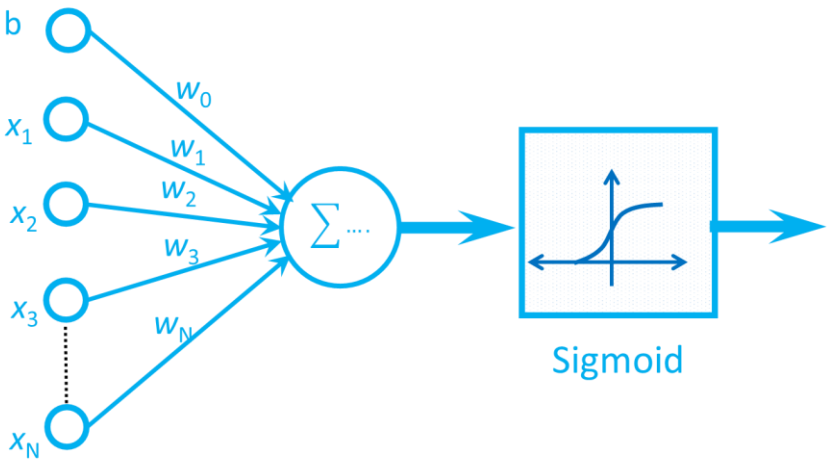


SVM: $Z = \exp\left(-\gamma \left| X(i) - Y(i) \right|^2\right)$



Neural Networks (NN)

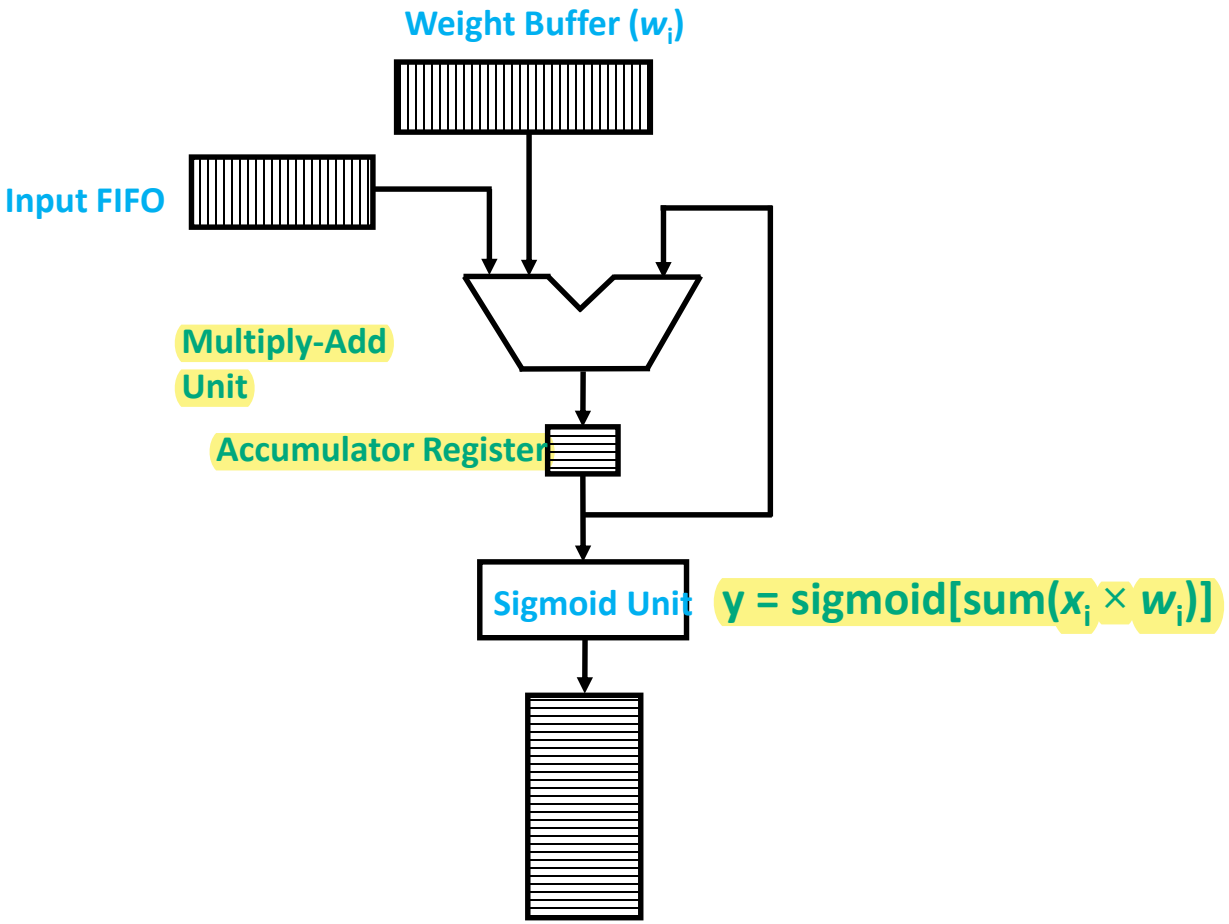
Sigmoid function: $Z = \dfrac{1}{1+\exp\left[-w(i)\bullet x(i)\right]}$



Steps:

1. Neural Computing Unit
2. Approximate with Maclaurin series
3. Cascading XOR, XNOR and MUX gates
4. Processing Element (PE) matrix

Neural Processing Unit (NPU)



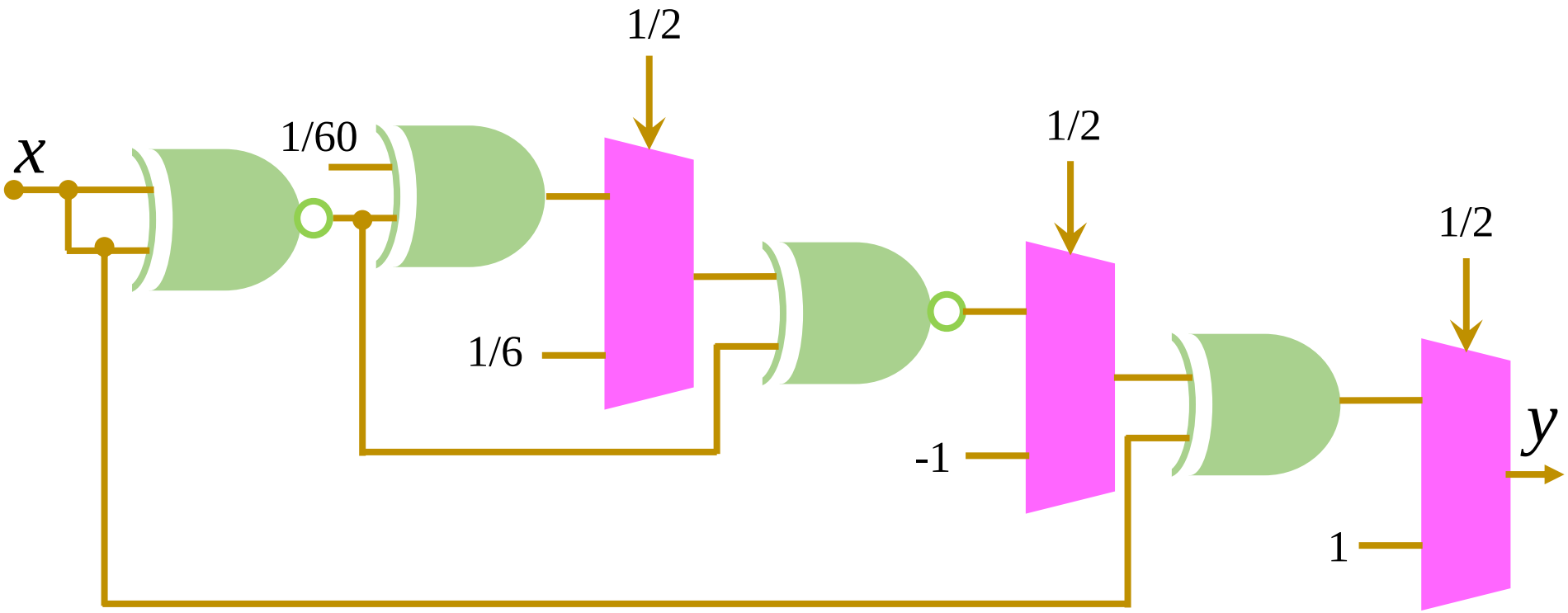
Sigmoid Function

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

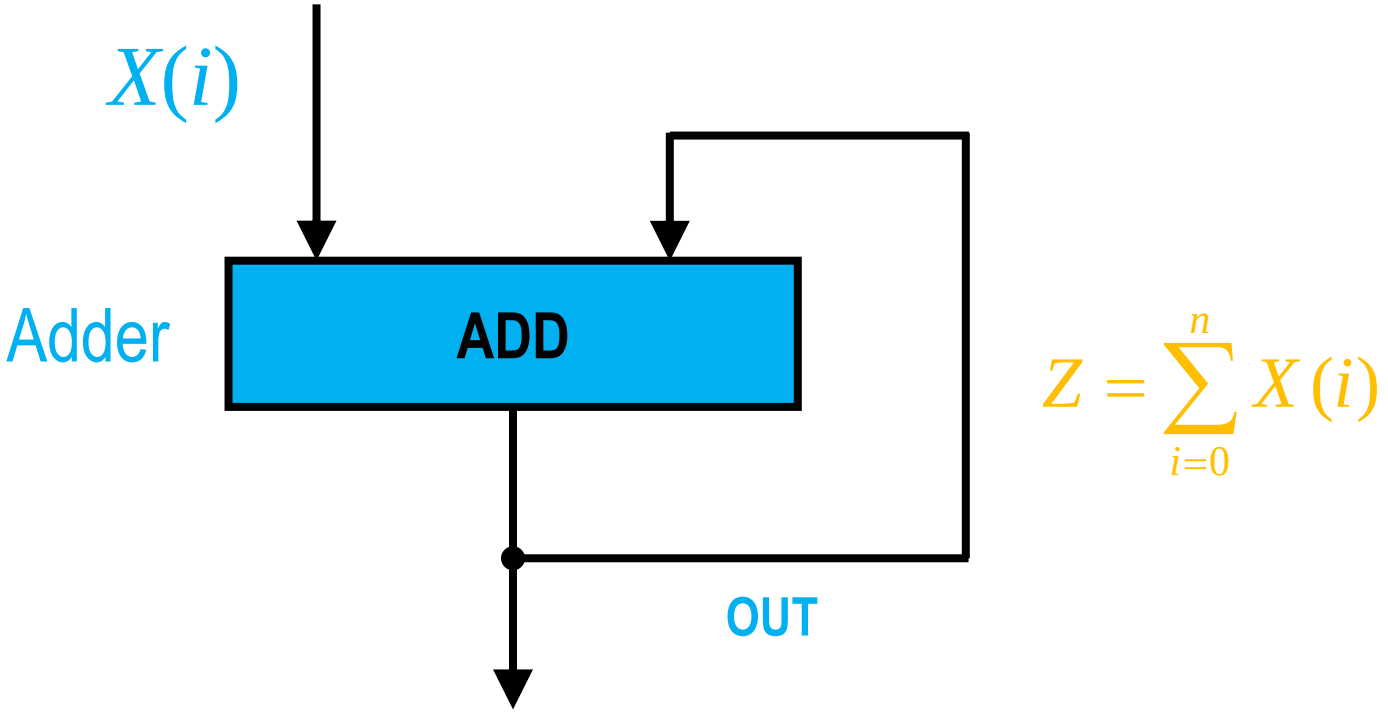
$$\approx \frac{1}{2} + \frac{x}{4} + \frac{x^3}{48} + \frac{x^5}{480}$$

$$= \frac{1}{2} - \frac{1}{2}x\left(\frac{1}{2}\left(-1 + x^2\frac{1}{2}\left(\frac{1}{6} - \frac{x^2}{60}\right)\right)\right)$$

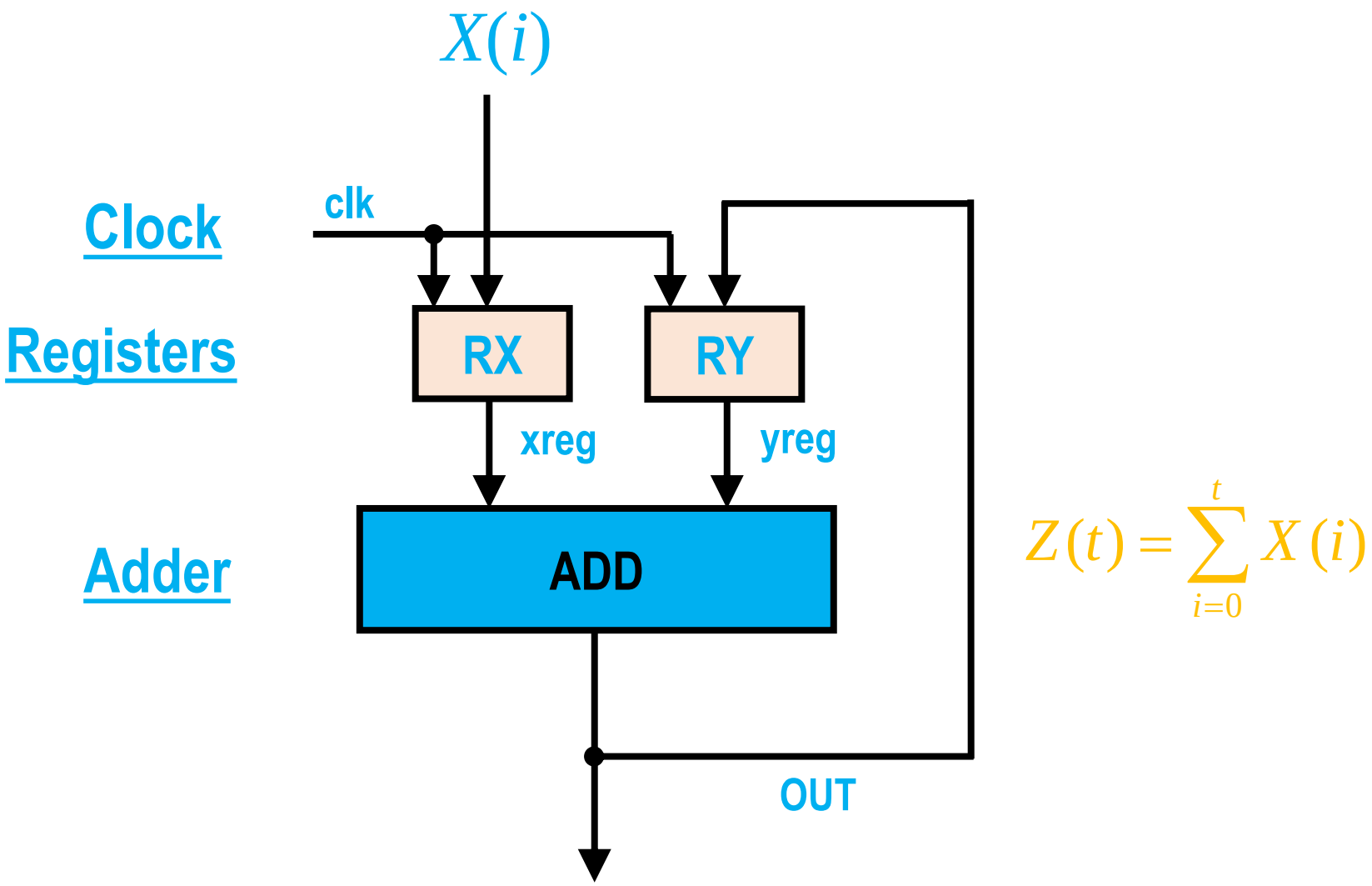
Sigmoid Unit



Architecture Computes $Z = \sum_{i=0}^{n} X(i)$



Architecture Computes $Z(t) = \sum_{i=0}^{t} X(i)$



Not secure | neuralnetworksanddeeplearning.com

Apps Facebook Gmail Neural networks an... Search results for 'P... AWS IoT Rule Tutor... Android Basics: Use... GitHub FooPlot | Online gr... Mohammad S. Sadr... ee1001_2015sa:star...

Neural Networks and Deep Learning

Neural Networks and Deep Learning is a free online book. The book will teach you about:

- Neural networks, a beautiful biologically-inspired programming paradigm which enables a computer to learn from observational data
- Deep learning, a powerful set of techniques for learning in neural networks

Neural networks and deep learning currently provide the best solutions to many problems in image recognition, speech recognition, and natural language processing. This book will teach you many of the core concepts behind neural networks and deep learning.

For more details about the approach taken in the book, see here. Or you can jump directly to Chapter 1 and get started.

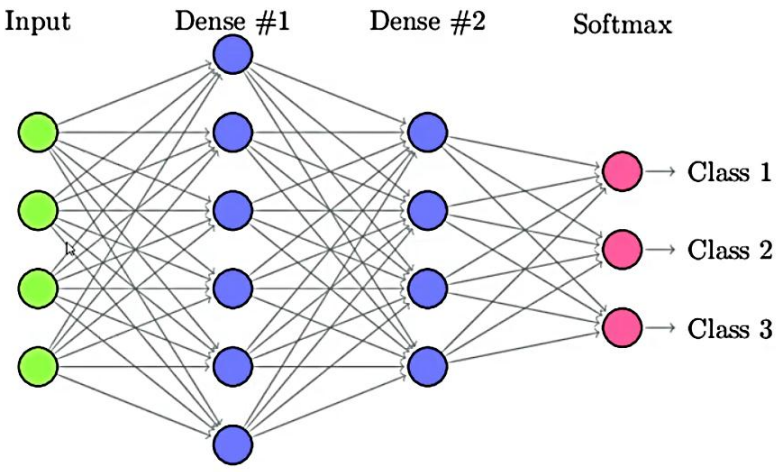
Neural Networks and Deep Learning
What this book is about
On the exercises and problems
▶ Using neural nets to recognize handwritten digits
▶ How the backpropagation algorithm works
▶ Improving the way neural networks learn
▶ A visual proof that neural nets can compute any function
▶ Why are deep neural networks hard to train?
▶ Deep learning
Appendix: Is there a *simple* algorithm for intelligence?
Acknowledgements
Frequently Asked Questions

If you benefit from the book, please make a small donation. I suggest $5, but you can choose the amount.

Donate

Alternately, you can make a

Fully Connected Neural Network



Ref: C. Pelletier et.al, Temporal Convolutional Neural Network for the Classification of Satellite Image Time Serie

Training

- In most cases will be using pre-trained networks
  - Training requires lot of hardware
  - Algorithms are not very hardware friendly
  - Most networks are trained only once → waste of resources used for training
  - Most applications, training is not the time critical path
- So we will train in computers with all sophistication and find the weight and bias values and directly use them during hardware design

Number representation

- The input to the NN (pixel values, sound samples) are generally positive
- The weights are biases can be positive or negative
- But generally all these numbers have fractional part
- They can be represented and processed through two representation → IEEE **floating point representation** (32 bits called single precision, or 64 bits called double precision) and **fixed point representation**

Number representation

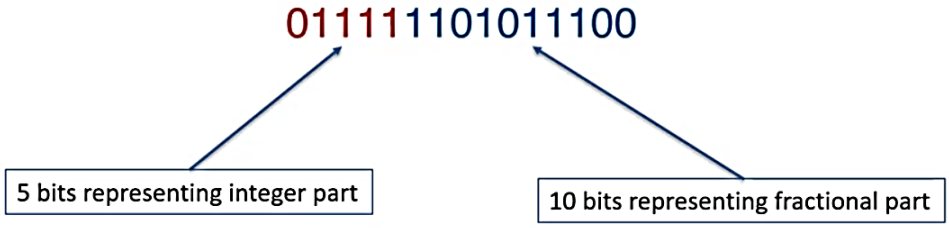
- Floating point representation helps in representing large numbers and they may provide better precision, but their implementation and manipulation is difficult
- Again they consume so much resources, you won't be able to implement more than a few tens of neuron on a platform like Zedboard
- So we will be going with **fixed point** representation
- Since the input values used in NNs are generally normalized (between 0 to 1 or -1 to 1), there won't be an issue of not able to represent large numbers

Number representation

- There may be slight degradation in accuracy but if no overflow occurs, a 32-bit fixed-point representation will give better performance than 32-bit floating-point representation
- This representation is highly flexible and can be parameterized depending upon the target application
- In this representation you will have to specify the total number of bits, the number of bits representing the integer part and the number of bits representing the fractional part

Number representation

- Eg: 15-bit fixed number representation. 5 bits representing integer part, 10 bits representing fractional part
- Representation of 15.84



- So the position of the decimal point is always fixed, it is after 5 bits. Hence the name fixed point representation

Number representation

- But .11011011100 is 0.83984375 not 0.84. Thus it introduces an error of 0.00015625.
- That is less than 1.9×10^{-4} % but if the error accumulates, will severely affect the performance.
- So instead of 15 bit representation, if we decide to use 10 bit representation with 5 bits for integer and 5 bits for fractional, 15.84 will be 0111111010 which is actually 15.8125. Here error is 0.0275 which is significantly higher than the previous one
- So by fixing the number of bits to represent, we do a tradeoff between accuracy and resource utilization

Number representation

- Now as mentioned before, weights and biases can be positive or negative
- Here also you can choose between two representations, either **signed-magnitude representation** or **2's complement representation**
- In signed magnitude representation, the MSB represents the sign. 0 if positive, 1 if negative
- Then as usual n bits will represent integer part and m bits will represent fractional part
- Eg: 0111111101011100 represents +15.83984375

  and 1111111101011100 represents -15.83984375 for

15 bit fixed point representation with 1 sign, 4 integers

Number representation

- Eg: 0111111101011100 represents +15.83984375

  and 1000000010100100 represents -15.83984375
- To find the 2's complement, you find 1's complement of the positive number (you can forget about the position of decimal point) and add 1 to it
- For addition (subtraction), 2's complement representation is very efficient since the result will be also in 2's complement representation
- But for multiplication, it doesn't directly work in that way. For multiplication, sign magnitude might be more efficient.

Activation Functions

- Many neural networks use non-linear functions such as sigmoid ($1/1+e^{-x}$) or hyperbolic tangent ($e^{x}-e^{-x}/ e^{x}+e^{-x}$) as activation functions
- Again building digital circuits generating these functions are very challenging and they will be very resource intensive
- Hence, we generally pre-calculate their values (since we will be aware of range of x, the input) and store in a ROM
- These ROMs will be also called Look Up Tables (LUTs). Don't confuse with FPGA basic building block LUT. These LUT ROMs are built either using block RAMs or distributed RAMs (FFs and LUTs)

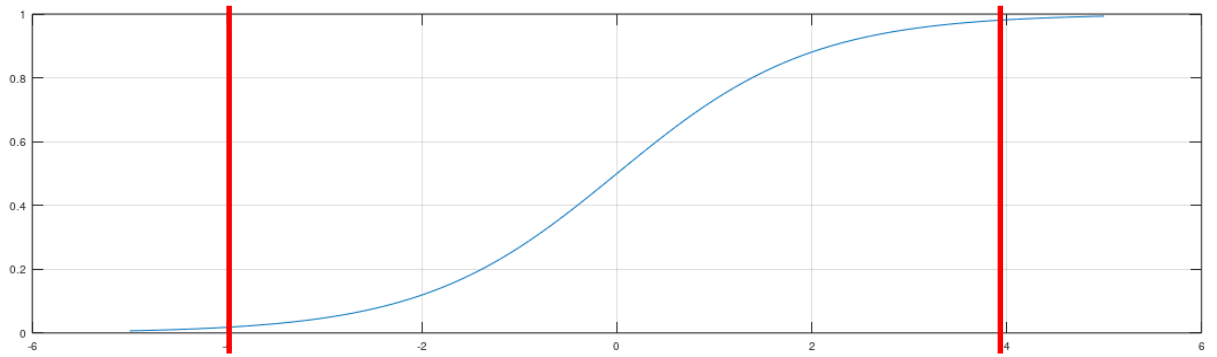
Sigmoid Function

Function of neuron

- z is factors times input values plus bias

  $z = w1 * x1 + w2 * x2 + w3 * x3 + bias$

- Sigmoid: $h = \frac{1}{1 + e^{-z}}$



Implementation

- Function table in ROM
- Values between ±4
- Limitation of values outside this range

Fixed-point implementation

- Values have factor of 2^13 = 8K (8192)
- Value ±4 corresponds to ±32K

Approach

5 binary places for w1, w2, w3
→ shift parameter by 5 bit

input x1, x2, x3 have 8 bit, i.e. 0 to 255,
but correspond to values of 0 to 1
→ must be considered for bias
→ shift bias by another 8 bit

FPGA Implementation of Sigmoid Function

- Sum is limited to ±4 and shifted to positive range

  0 ≤ limit(sum+4) < 8

- Factor of 8K

  0 ≤ sumAdress < 64K

Word width of ROM

- sumAdress has 16 bit
- ROM uses 14 bit

  address => sumAdress(15 downto 2)

- Other word width can be chosen
- Design FPGA_generate uses 12 bit

```
address => sumAdress(15 downto 4),
```

neuron.vhd in FPGA_plain

```
process
begin
    wait until rising_edge(clk);

    -- sum of input with factors and bias
    sum <= (w1 * x1 + w2 * x2 + w3 * x3 + bias);

    -- limiting and invoking ROM for sigmoid
    if (sum < -32768) then
        sumAdress <= (others => '0');
    elsif (sum > 32767) then
        sumAdress <= (others => '1');
    else
        sumAdress <= std_logic_vector(to_unsigned(sum + 32768, 16));
    end if;
end process;

sigmoid : entity work.sigmoid_IP
    port map (clock   => clk,
              address => sumAdress(15 downto 2),
              q       => afterActivation);

    -- format conversion
    output <= to_integer(unsigned(afterActivation));
```

Definition of ROM Values

- ROM implemented as IP (Intellectual Property)
- Values defined in MIF (Memory Initialization File)

Definition of fixed-point values

- Input values corresponds to ±4 as discussed
- Output values correspond to range 0 … 1 with 8 bit word width
  - Range of 0 … 255
  - Same accuracy as input values x1, x2, x3

sigmoid_14_bit.mif

```
DEPTH = 16384;
WIDTH = 8;
ADDRESS_RADIX = DEC;
DATA_RADIX = DEC;
CONTENT
BEGIN
0 : 2;
1 : 2;
2 : 2;
3 : 2;
4 : 2;
5 : 2;
…
8000 : 121;
…
```

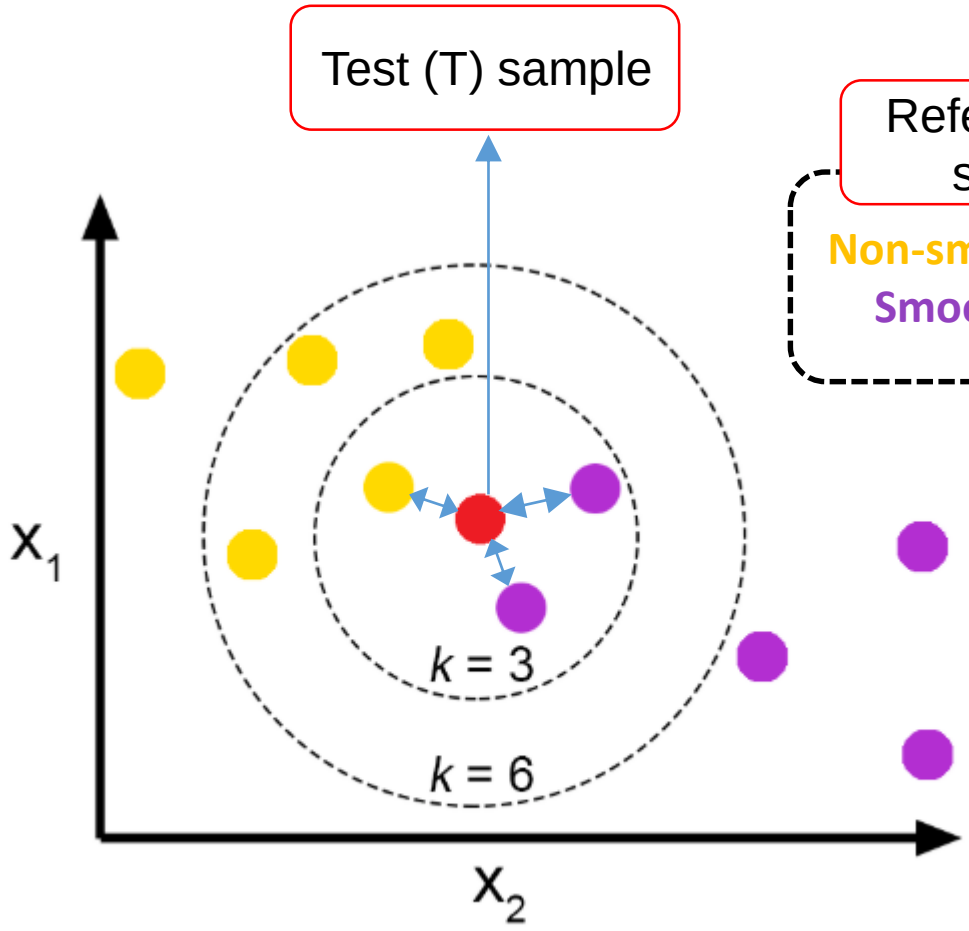
sigmoid_12_bit.mif

```
DEPTH = 4096;
WIDTH = 8;
ADDRESS_RADIX = DEC;
DATA_RADIX = DEC;
CONTENT
BEGIN
0 : 2;
1 : 2;
2 : 2;
…
2000 : 121;
…
```

ROM values for different word width

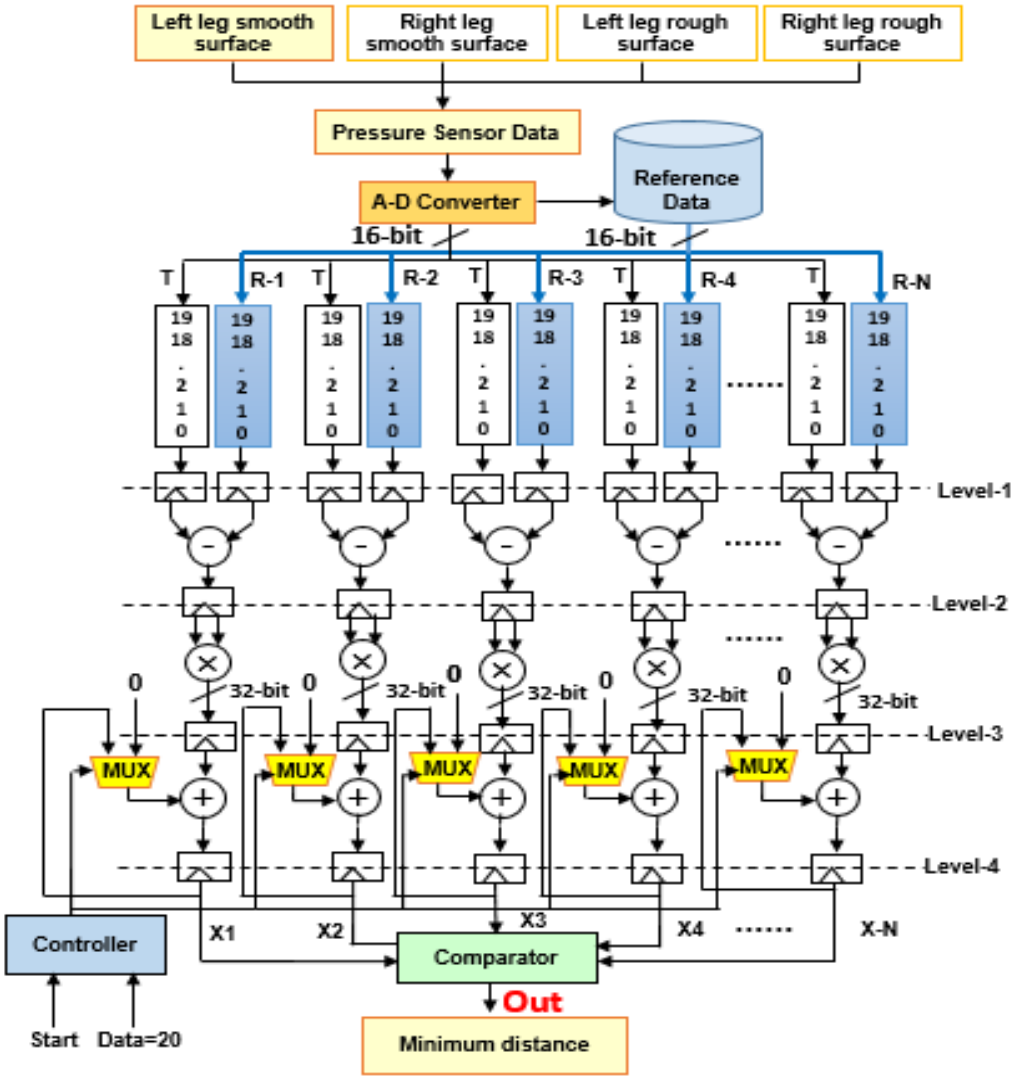
- 12 bit input values correspond to 14 bit values divided by 4

Nearest Neighbor Search (NNS)

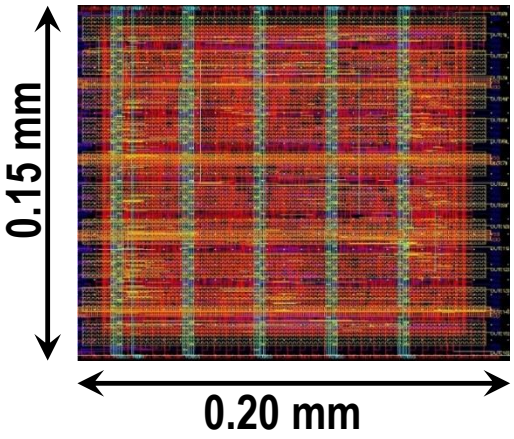


$$E_D = \sqrt{\sum_{j=1}^{n} (R_j - T_j)^2}$$

NNS Circuit and Layout

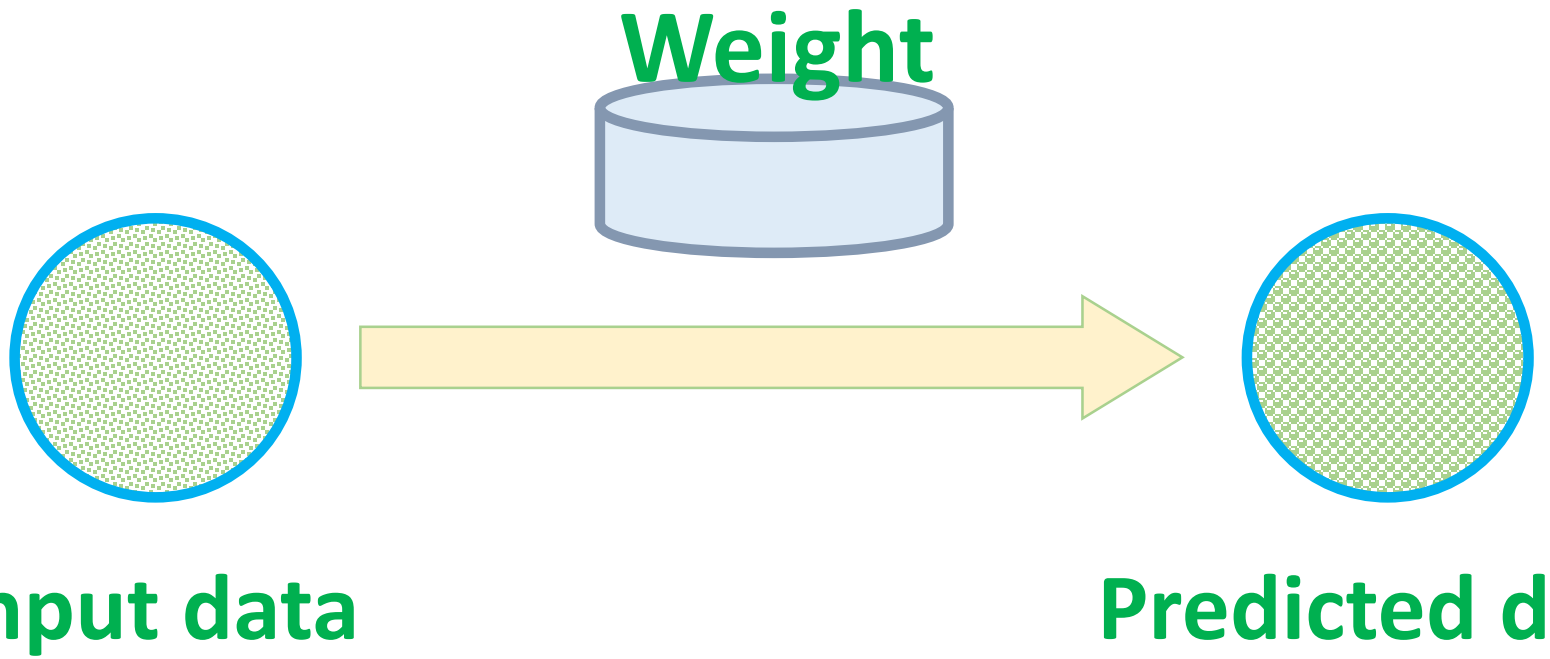


Cadence Virtuoso

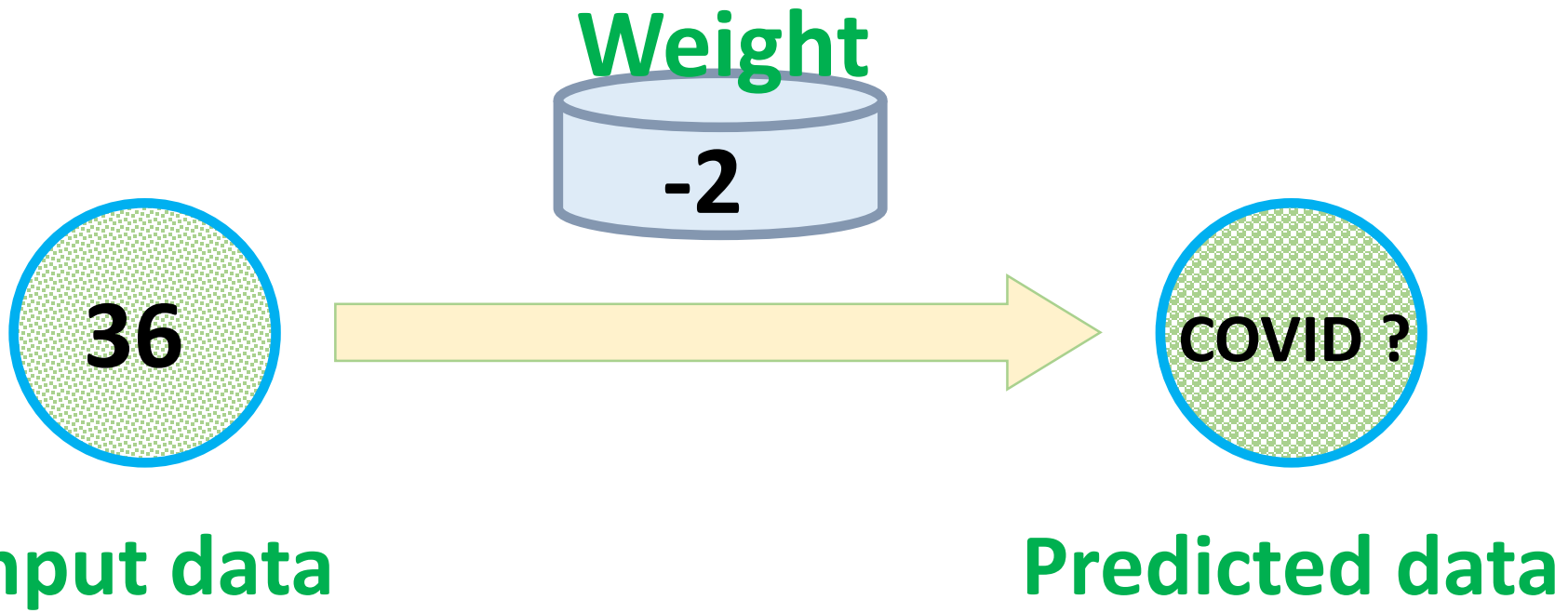


Neural Networks (NNs) in C

Single input single output Neural Network



Example: A Neural Network to predict whether a person is affected by COVID-19 or Healthy given a particular temperature



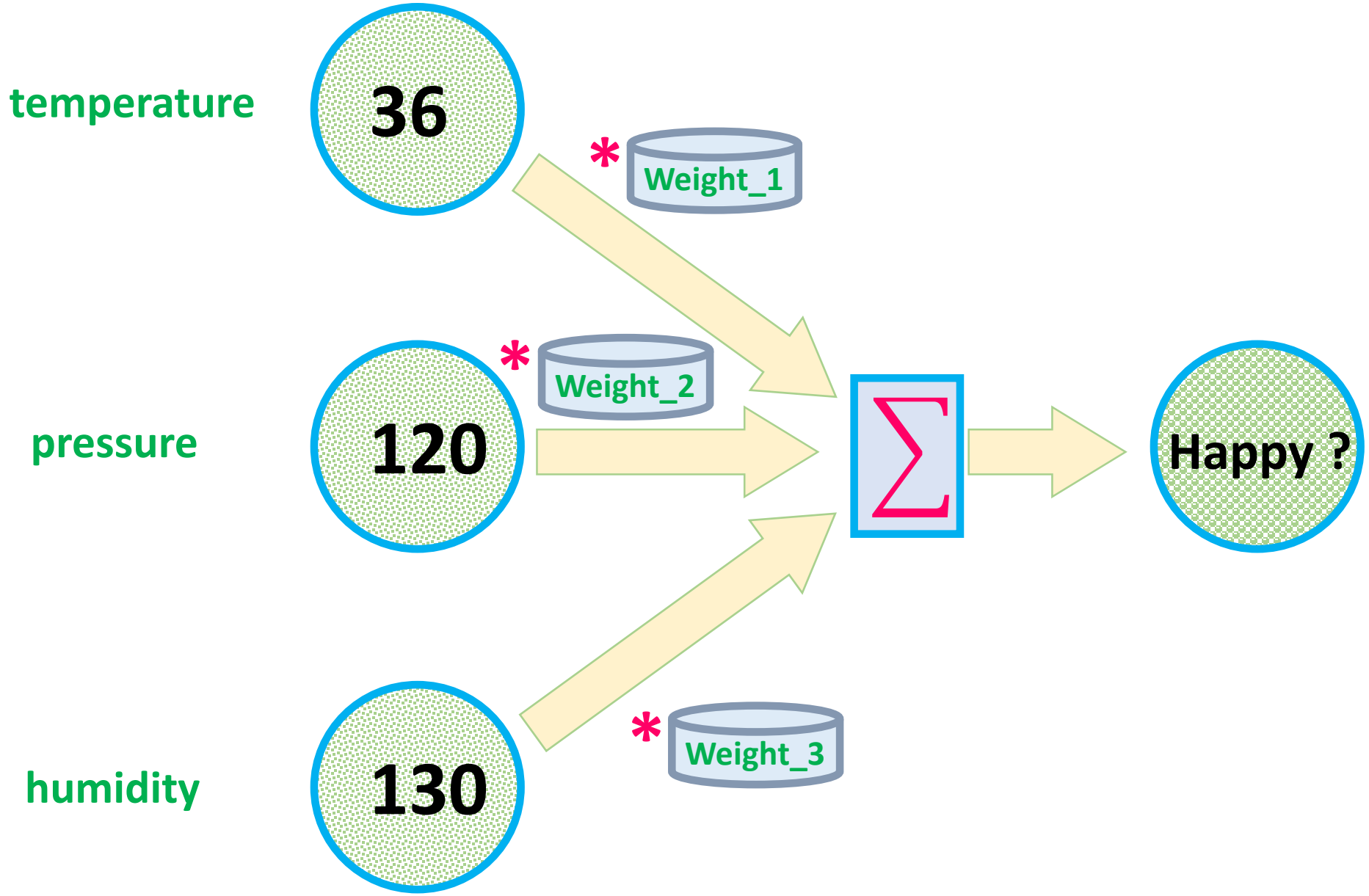
Simple Neural Network model

```
int  simple_nn(input, weight){
      predicted_data = input * weight
      return predicted_value
 }
```

Prediction

```
temperature [] = {30, 35, 36, 38, 39, 40, 41}
first_predicted_data [0] = simple_nn (temperature [0], -2)
```

Multiple inputs single output Neural Network



Neural Network model

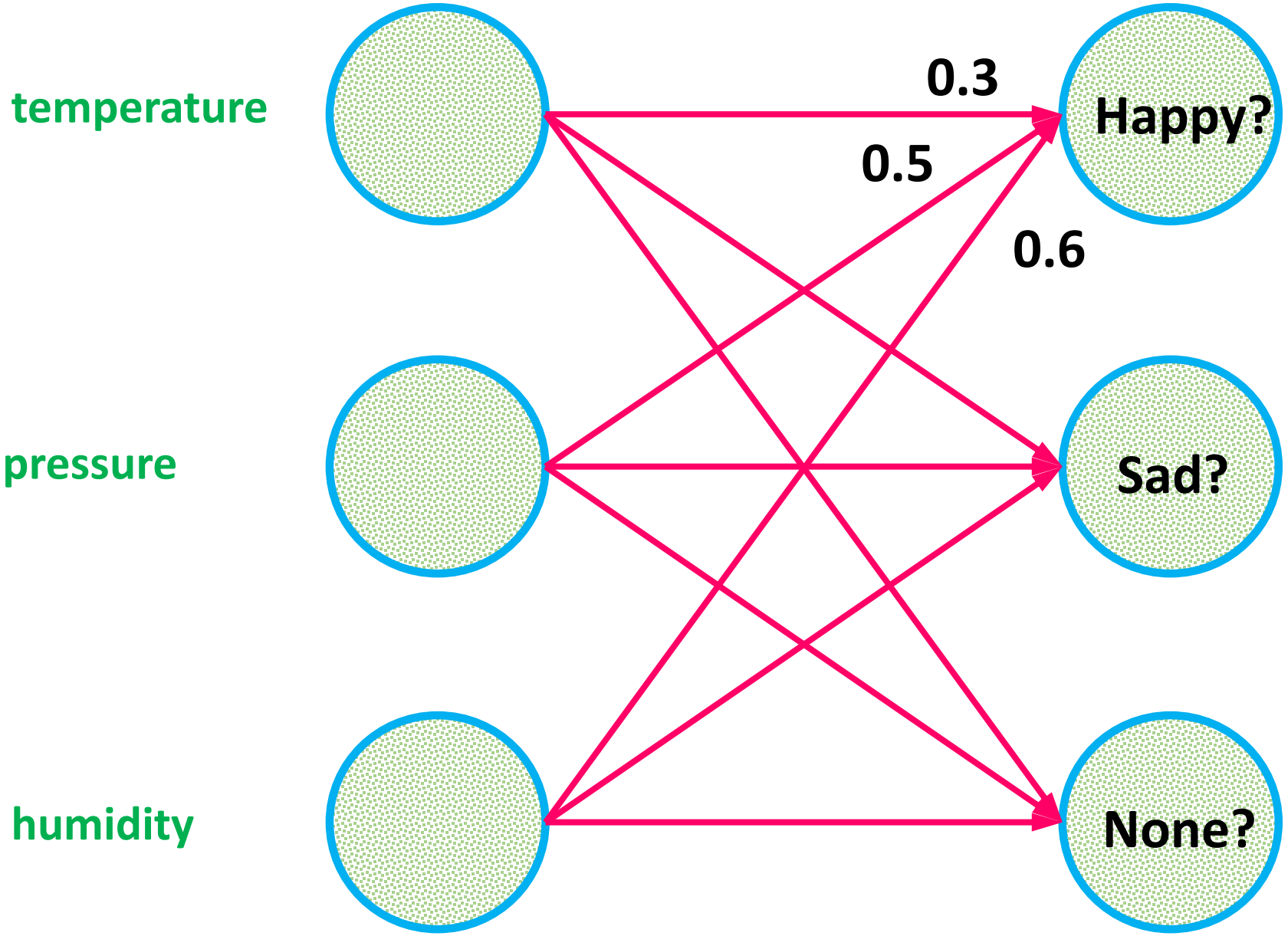
```
int weighted_sum(input, weight){
    for(int i=0;i<INPUT_LEN; i++)
            output += input[i]*weight[i];
             return output;  }
```

```
int multiple_inputs_single_output_nn(input, weight){
          predicted_data =  weighted_sum(input, weight);
          return predicted_data; }
```

Inputs

```
temperature []    =    {35, 36, 37, 38, 39, 40};
pressure []         =    {110, 120, 130, 140, 150, 160};
humidity []         =    {130, 140, 150, 160, 170, 180};
```

Multiple inputs Multiple outputs Neural Network



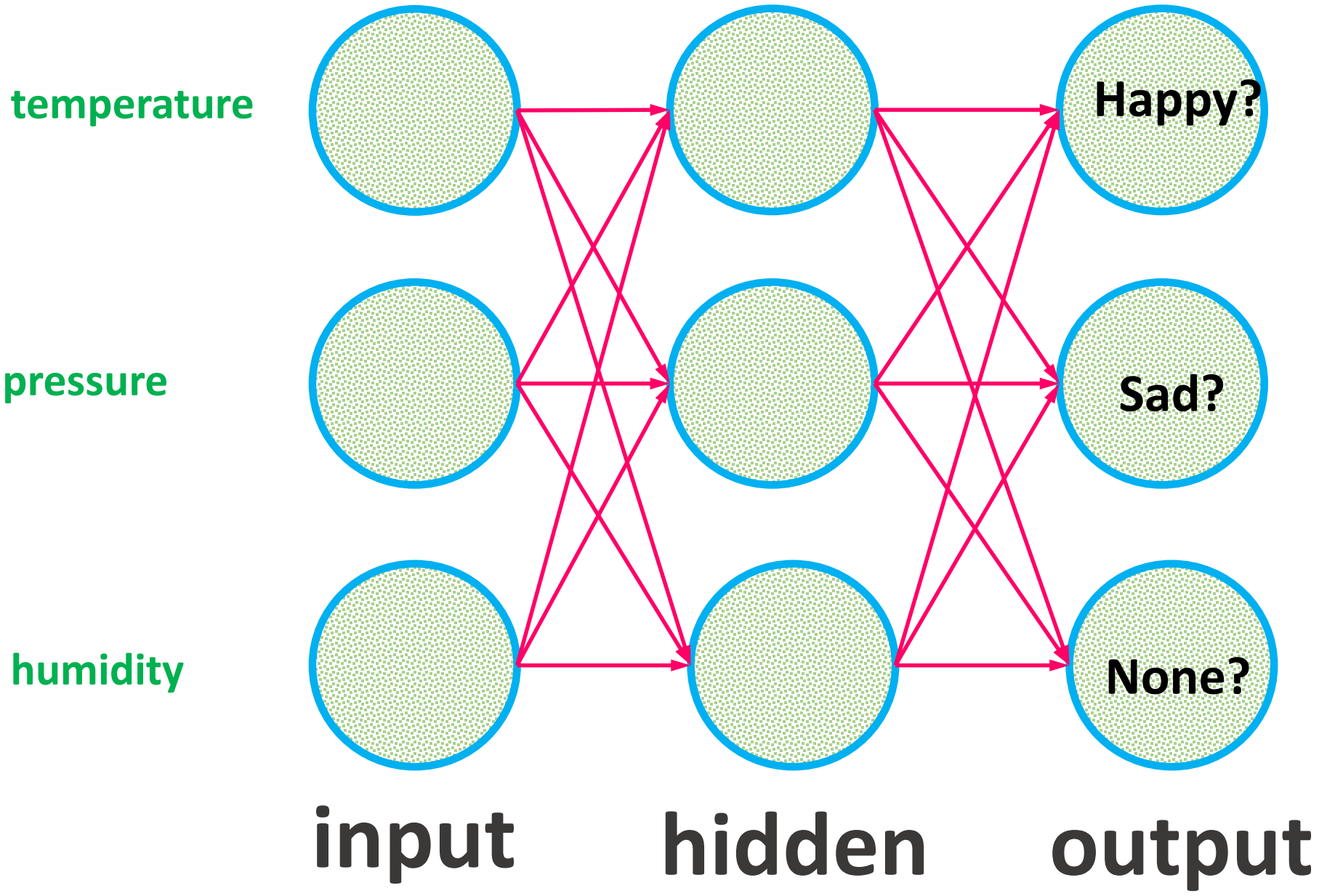
Neural Network model

```
void vector_matrix_multiply(input, output_vector,  matrix){
        for(int k=0;k<LEN; k++){
          for(int i =0;i<LEN; i++){
                    output_vector[k] += input[i] *matrix[k][i]; }
        }
}
```

Inputs

```
                  temp   press  humidity
Weights [] [] = { {-2.0,   9.5,   2.01},      //Happy ?
                  {-0.8,   7.2,   6.3 },      //Sad?
                  {-0.5,   0.45, 0.9}         // none ?
```

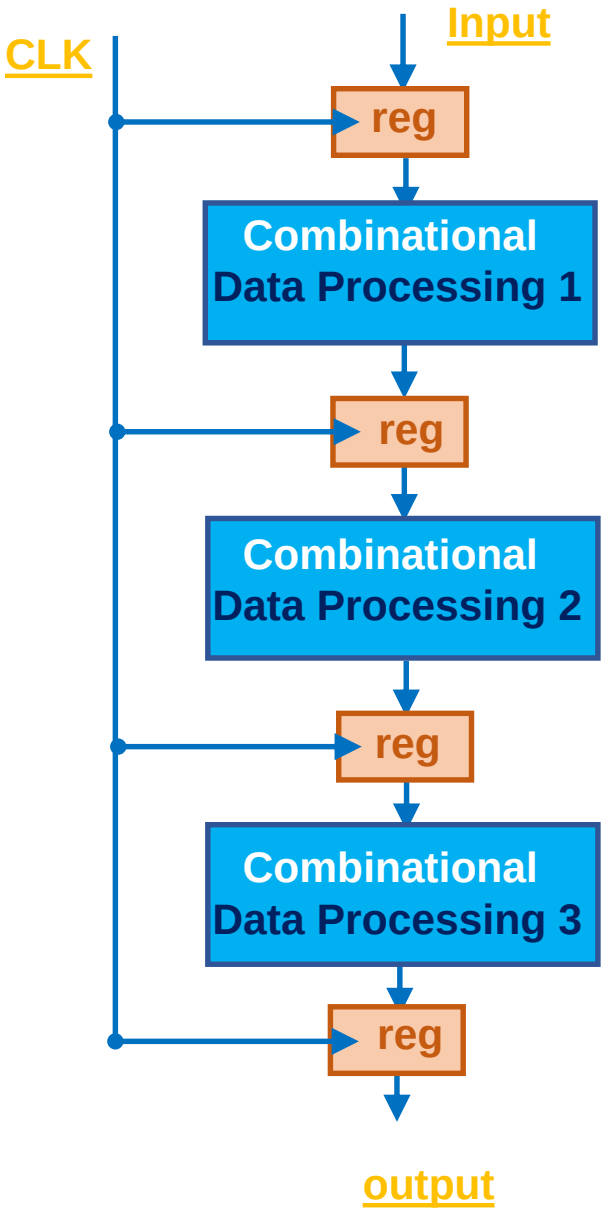
Hidden Layer Neural Network



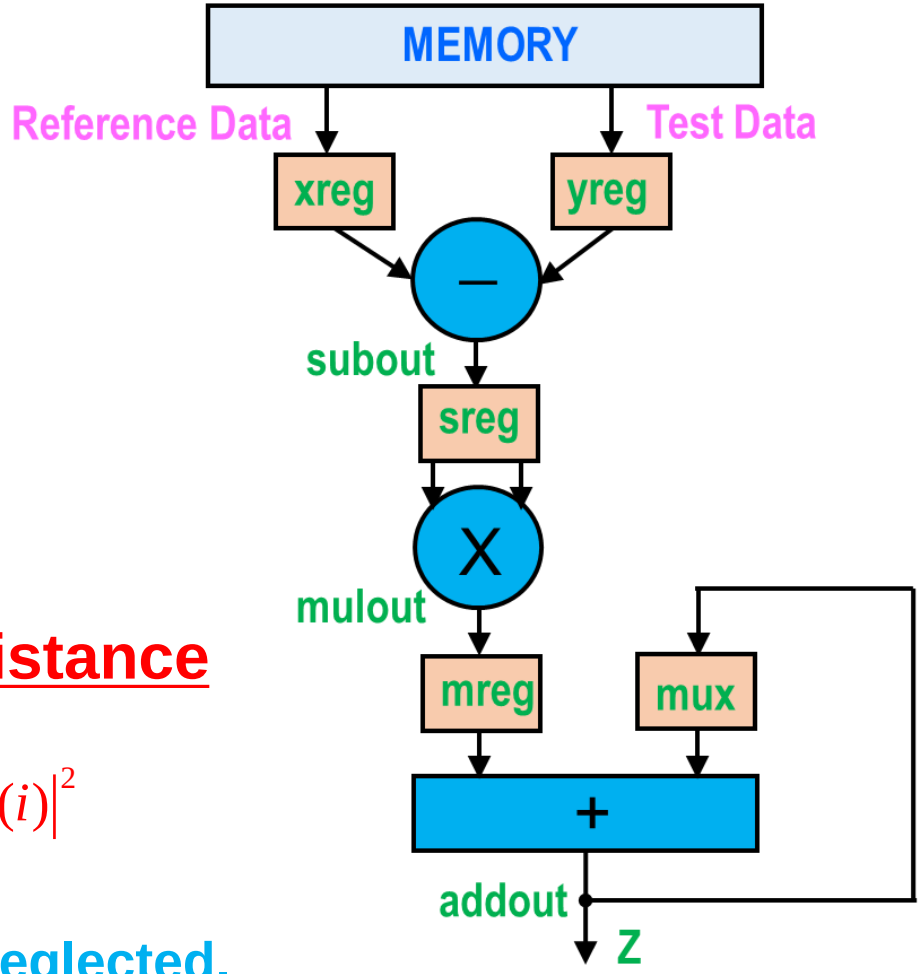
High-Speed: *mS* or less for single task

Pipeline of Neural Networks (NNs)

Sequential Data Processing



Euclidean Distance Search

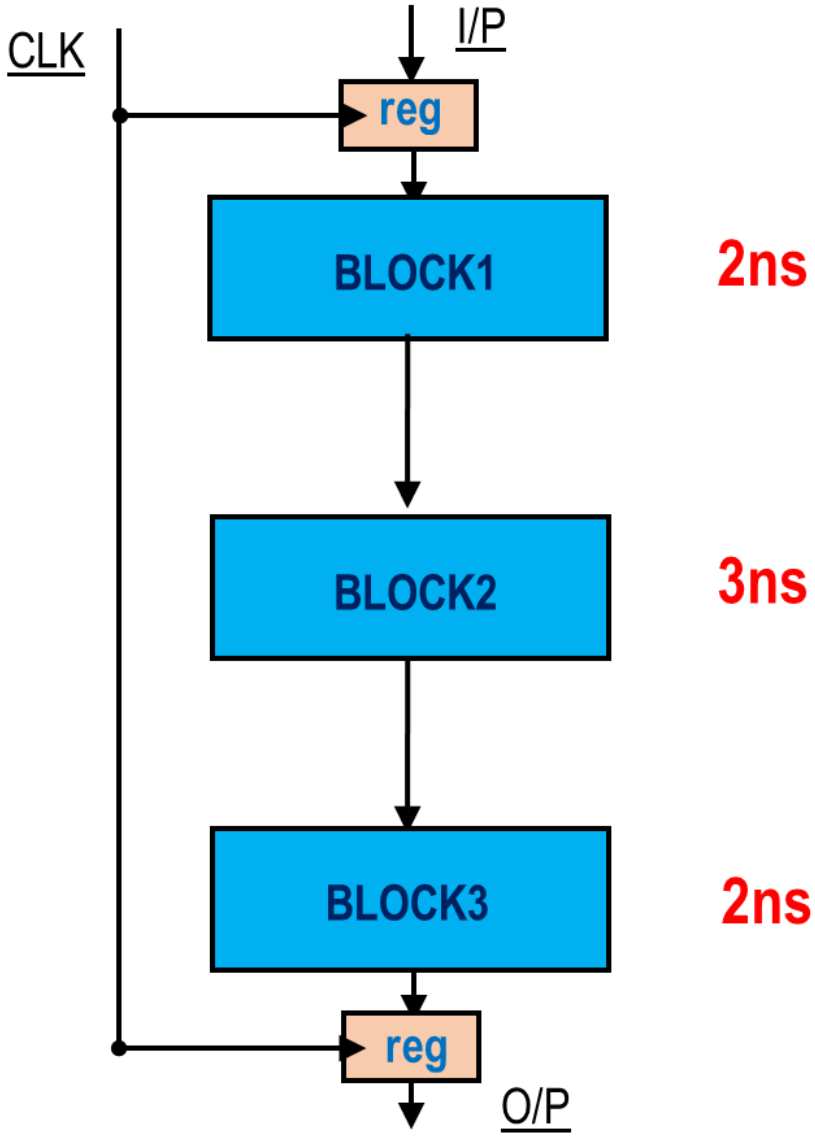


Euclidean Distance

$$Z = \sum_{i=1}^{n} \left| X(i) - Y(i) \right|^2$$

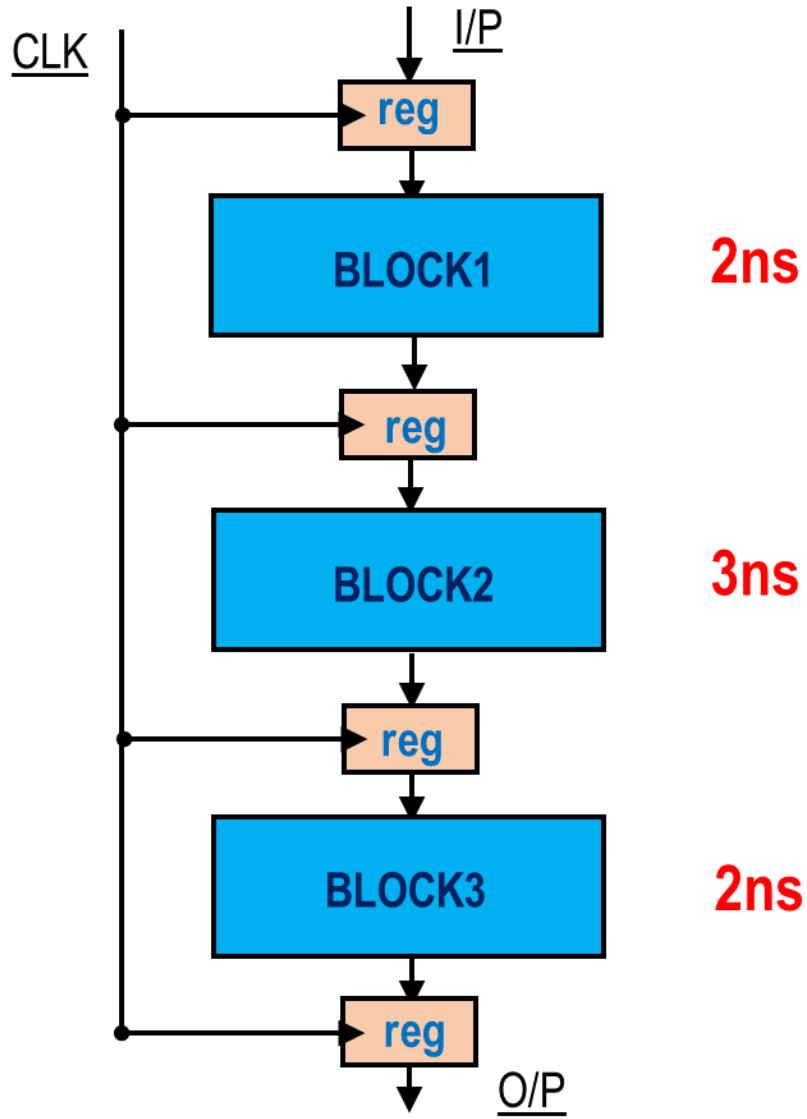
Square root is neglected.

Block Level Representation



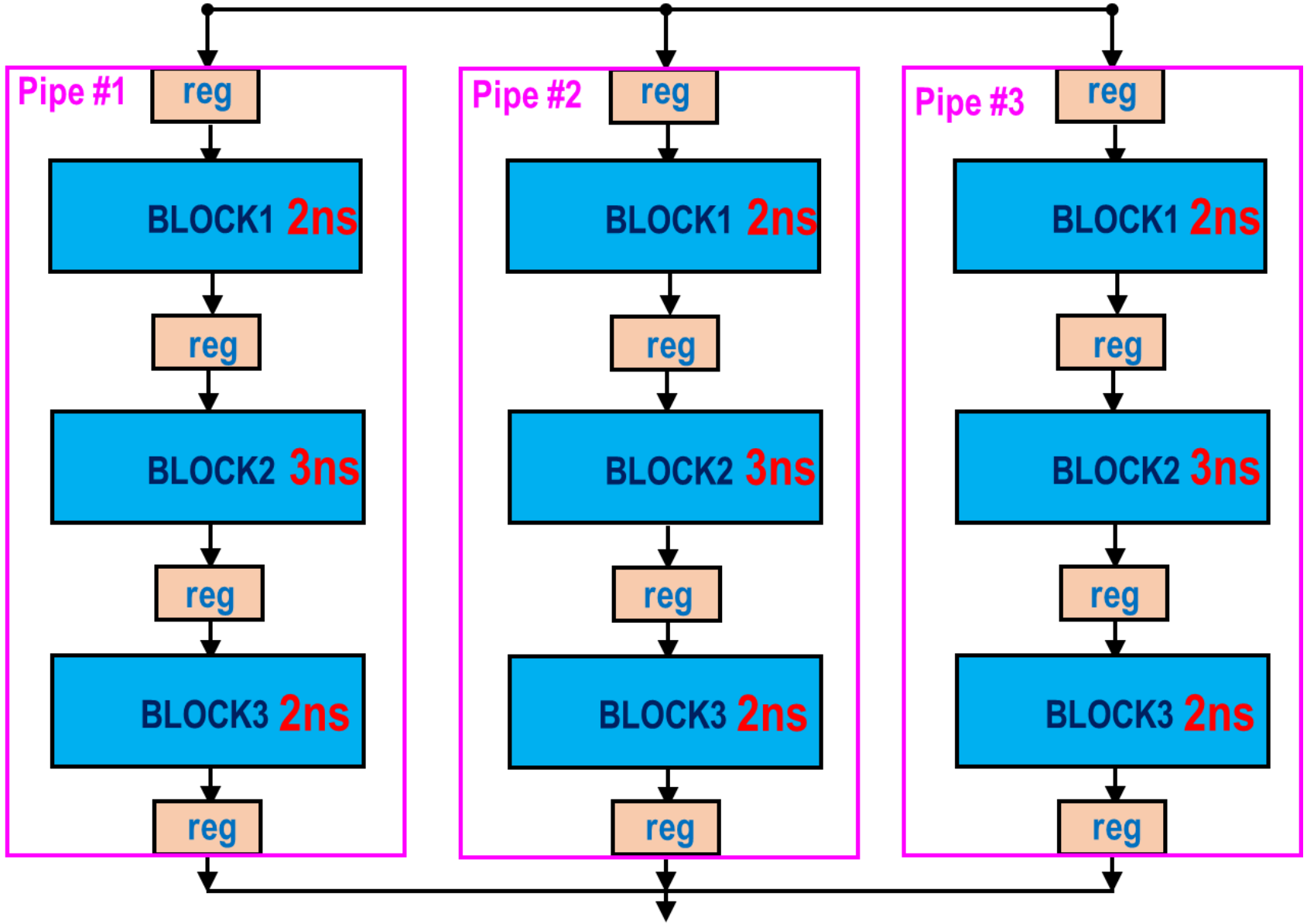
- Total: 7 ns
- Throughput: 1 output / 7ns
- Latency: 7ns

Pipelining

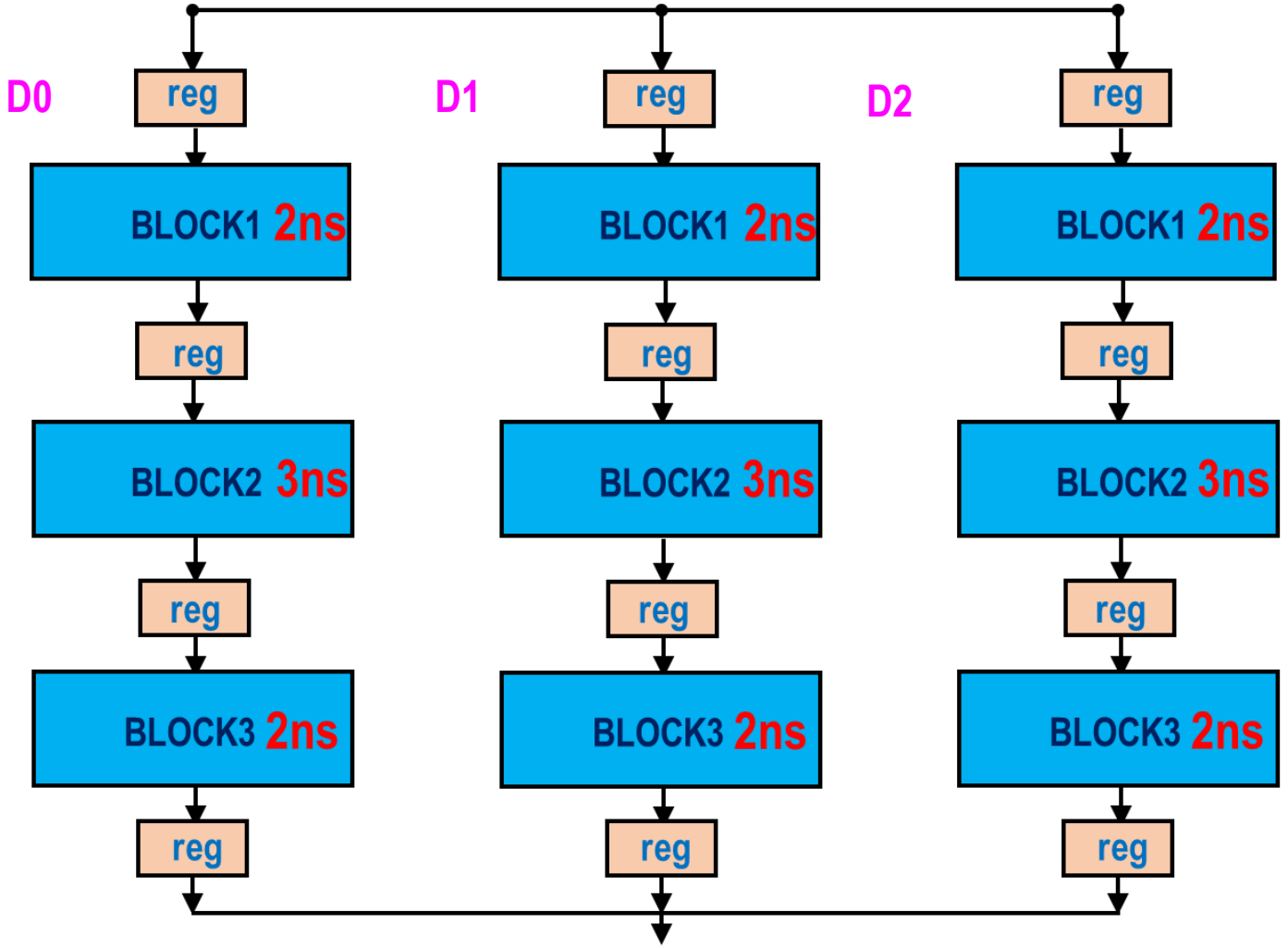


- Throughput: 1 output / 3ns
- Latency: 9ns

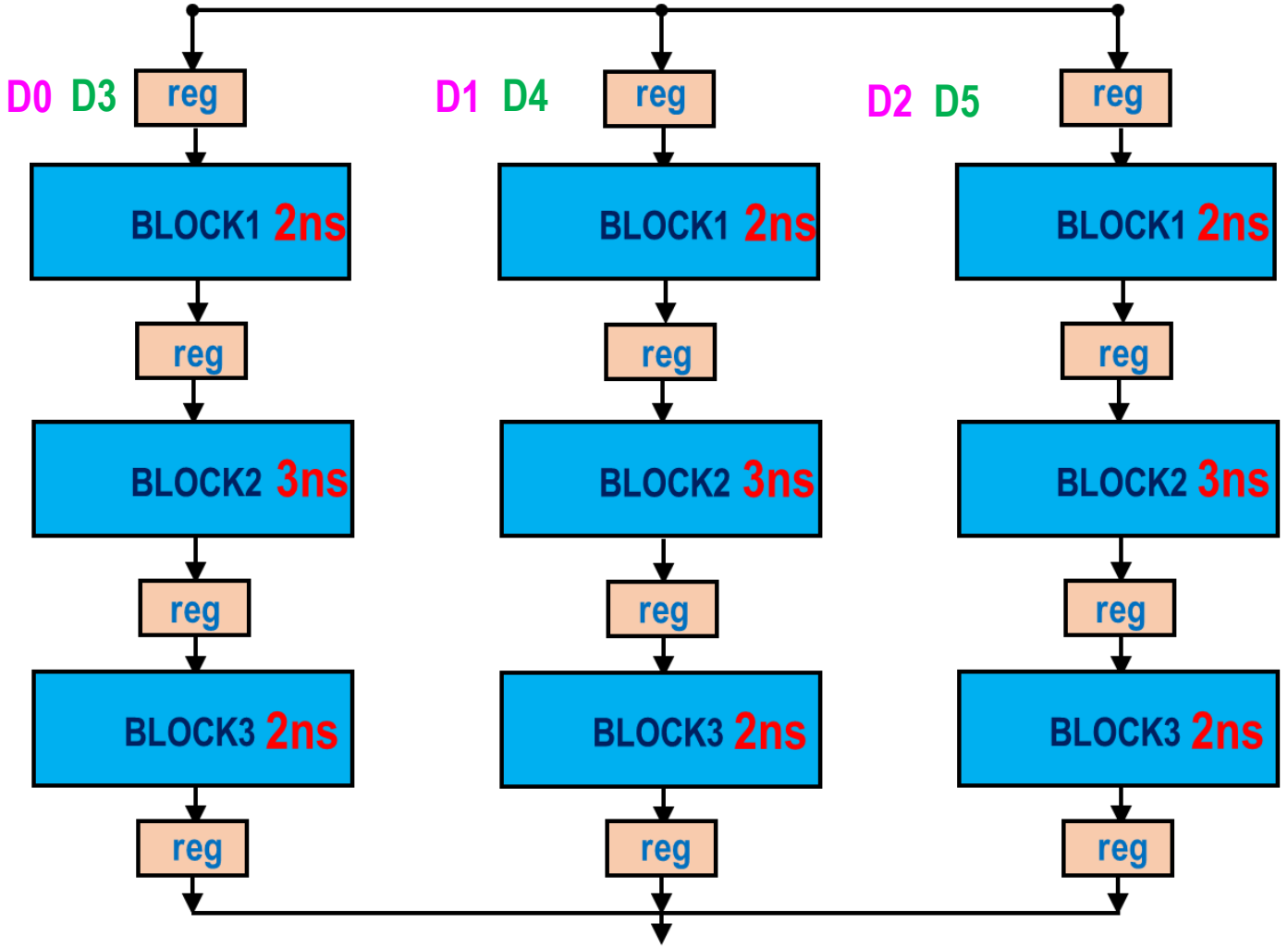
Parallel and Pipelining



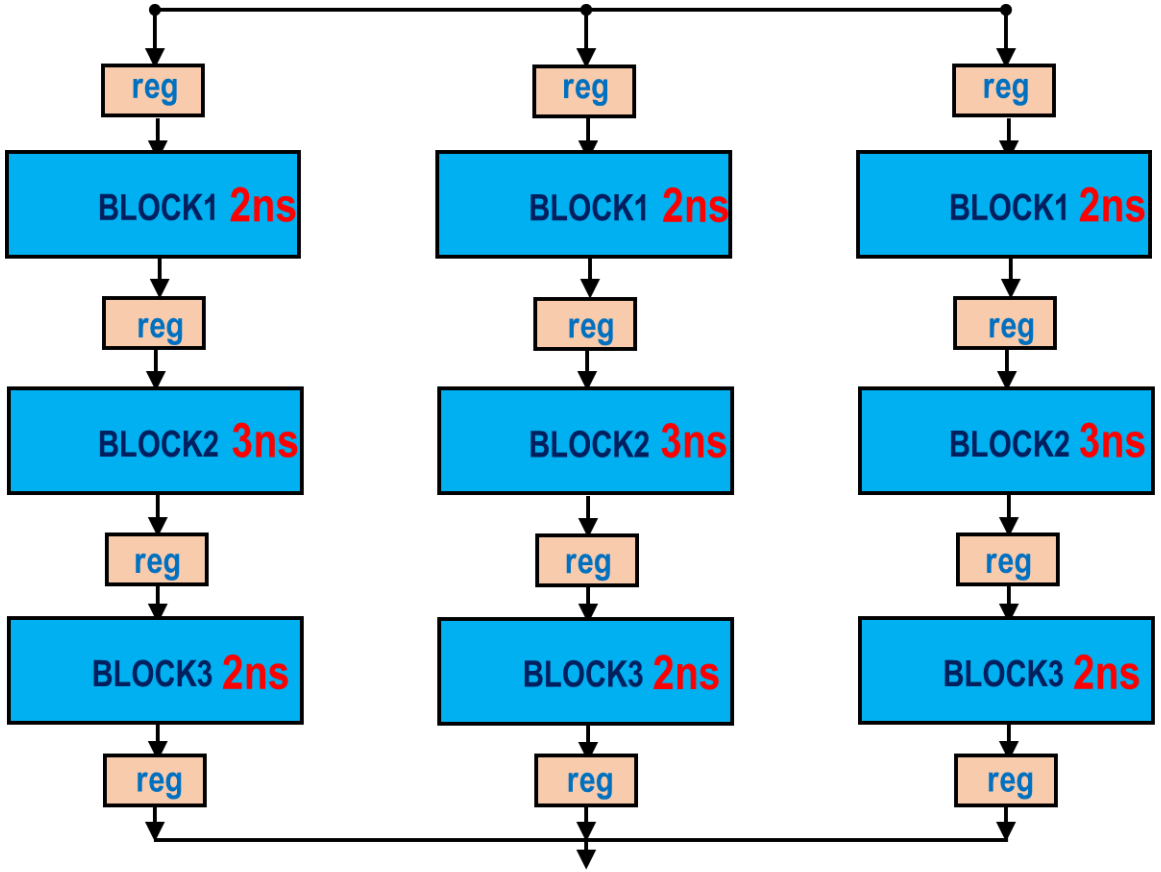
Parallel and Pipelining



Parallel and Pipelining



Parallel and Pipelining



Utilization

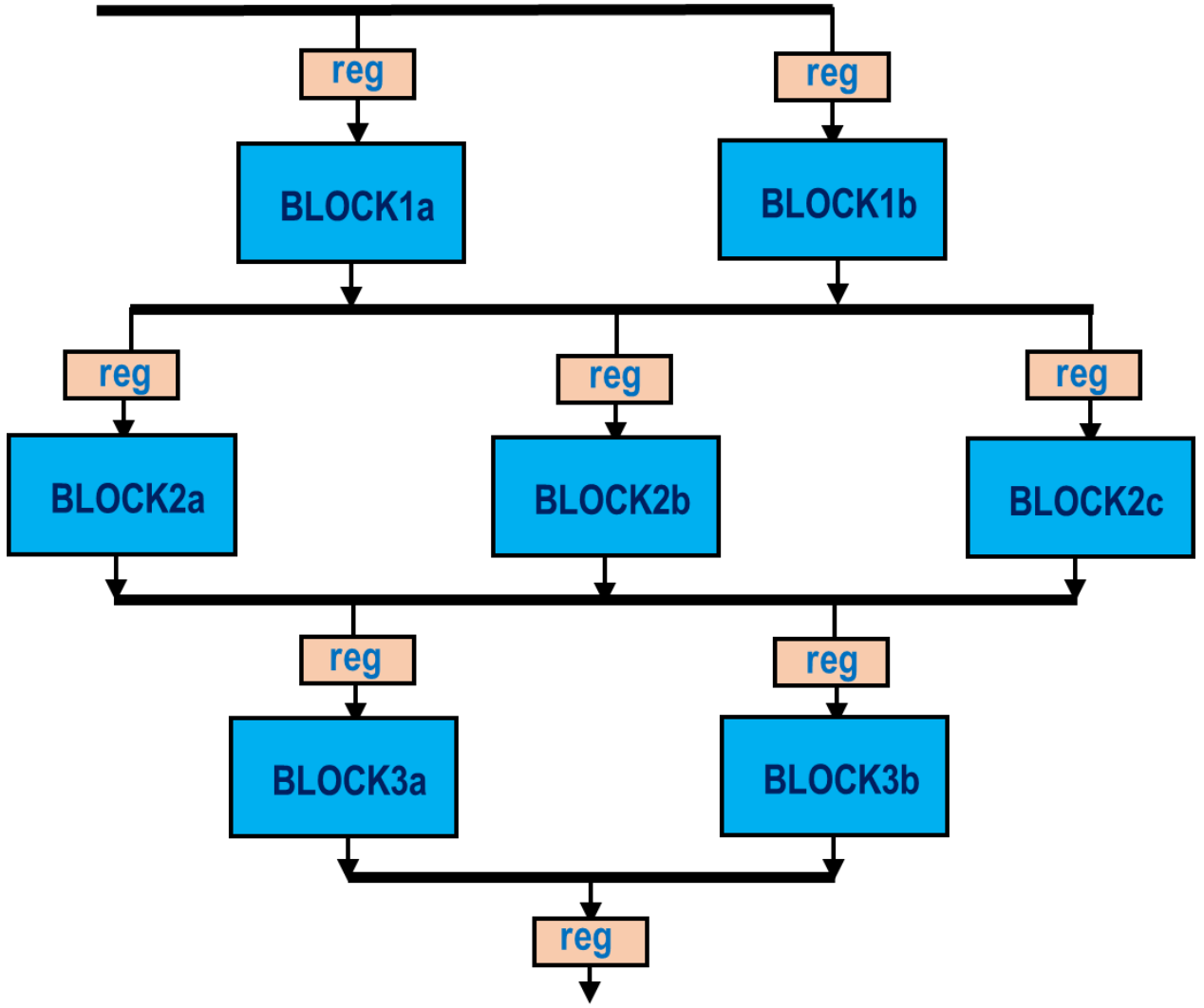
66.67%

100%

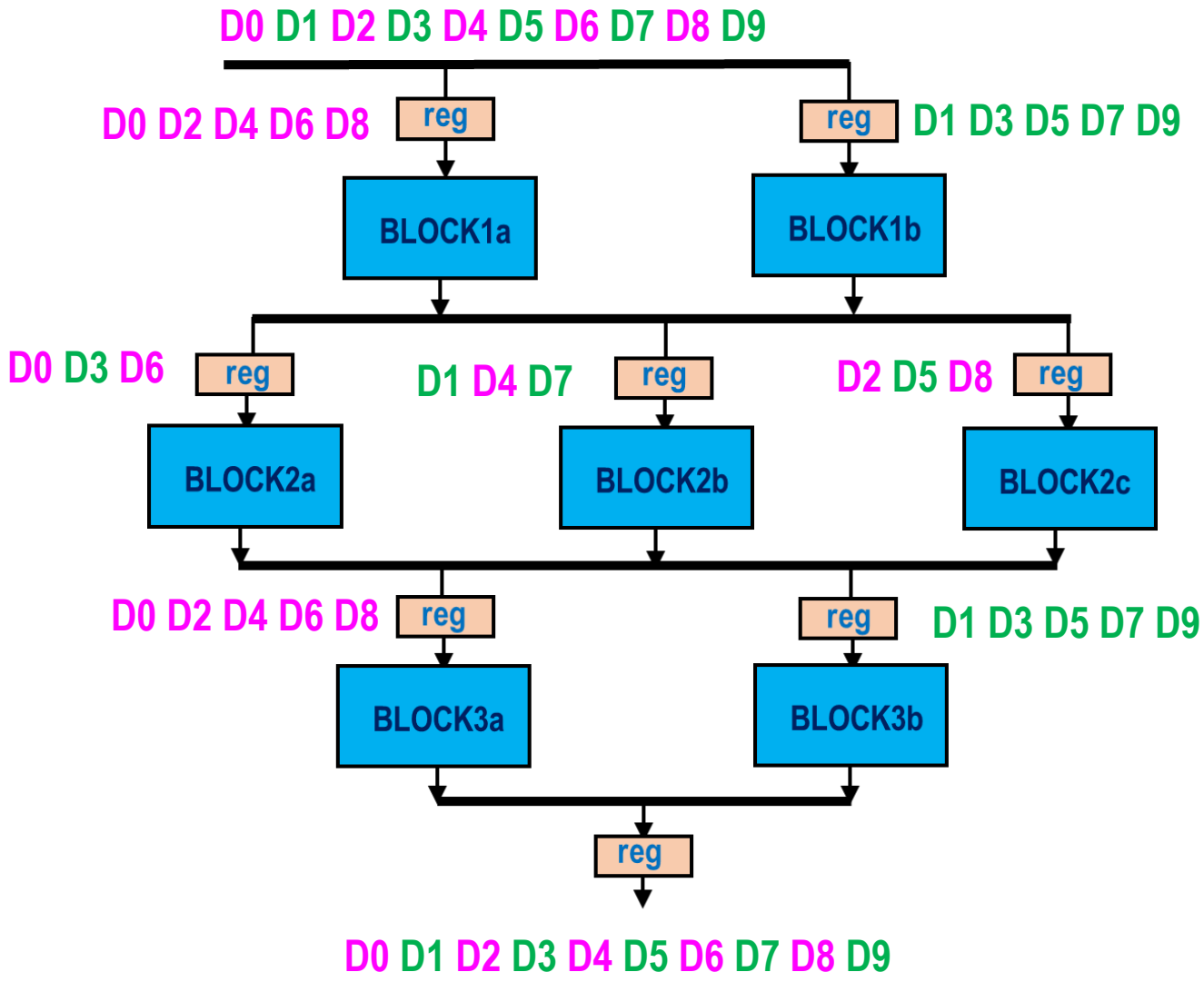
66.67%

- Throughput: 1 output / 1ns
- Latency: 9ns

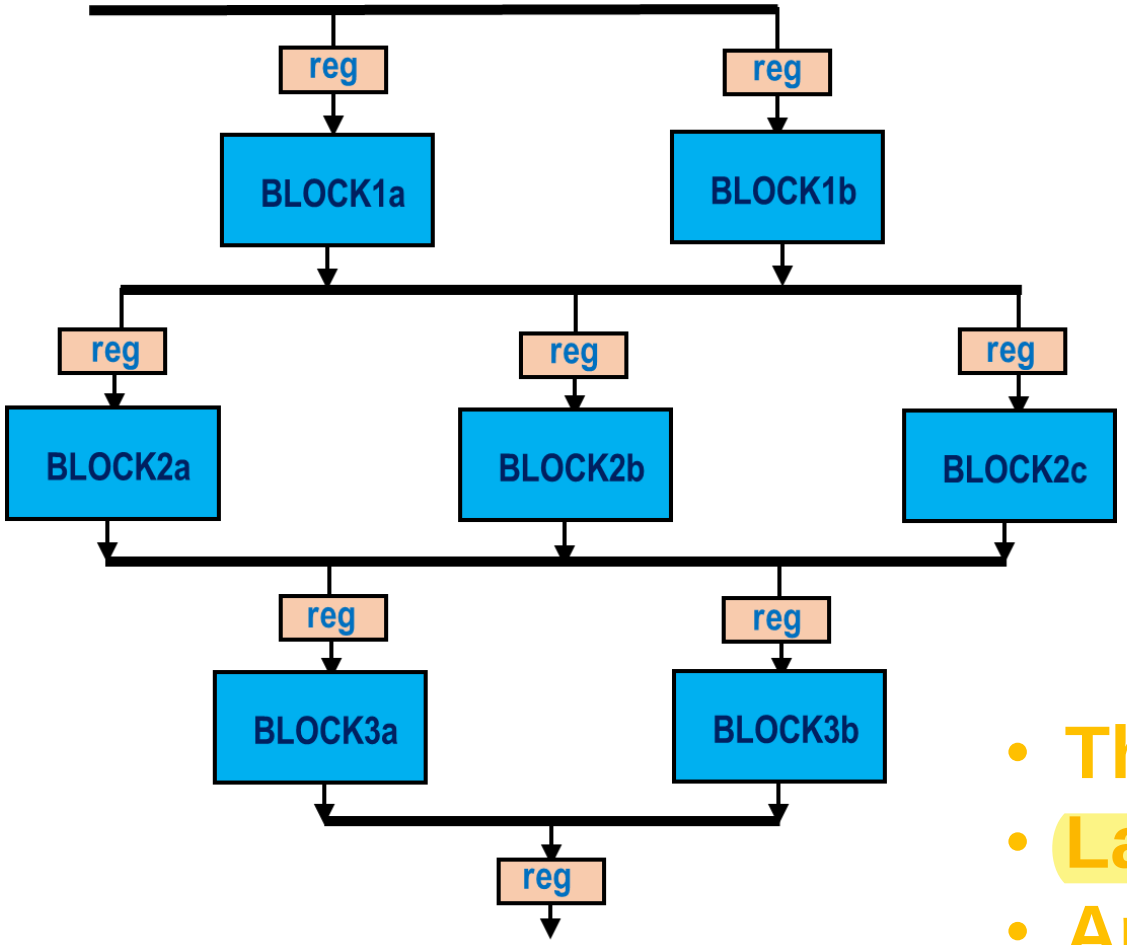
Combining Parallel and Pipelined Structures



Combining Parallel and Pipelined Structures



Combining Parallel and Pipelined Structures



- Throughput: 1 output / 1ns
- Latency: 7ns
- Area: Reduced
- Power: Reduced

EL530-Introduction to Embedded Artificial Intelligence

Credit Structure (L-T-P-Cr): 3-0-2-4
Semester: Autumn'2023

Instructor: Dr. Tapas Kumar Maiti

Course Contents

- Fundamentals of Embedded AI
- Microcontroller, FPGA, Power, and Energy
- Summary of AI Models
- Naive Bayes Model, Laplace Smoothing
- Linear regression, Cost function
- Logistic Regression
- Simple Neural Networks
- Neural Network from Regression to Classification
- Limitations of the DNN, Convolution, Pooling
- From DNN to CNN
- Avoiding Over fitting

Course Contents

- Training a Neural Network for Arduino in TensorFlow
- Deploying a TensorFlow Lite Model to Arduino
- Edge Impulse, Data Collection
- Model Training in Edge Impulse
- Deploy Model from Edge Impulse to Arduino
- Nearest Neighbor Classification Model
- Simple Nearest Neighbor Classification in Verilog/FPGA
- Training the Neural Network for Weight and Bias Calculations
- VLSI Architecture for Neural Networks
- Activation Functions: ReLU, Sigmoid Function, and Softmax

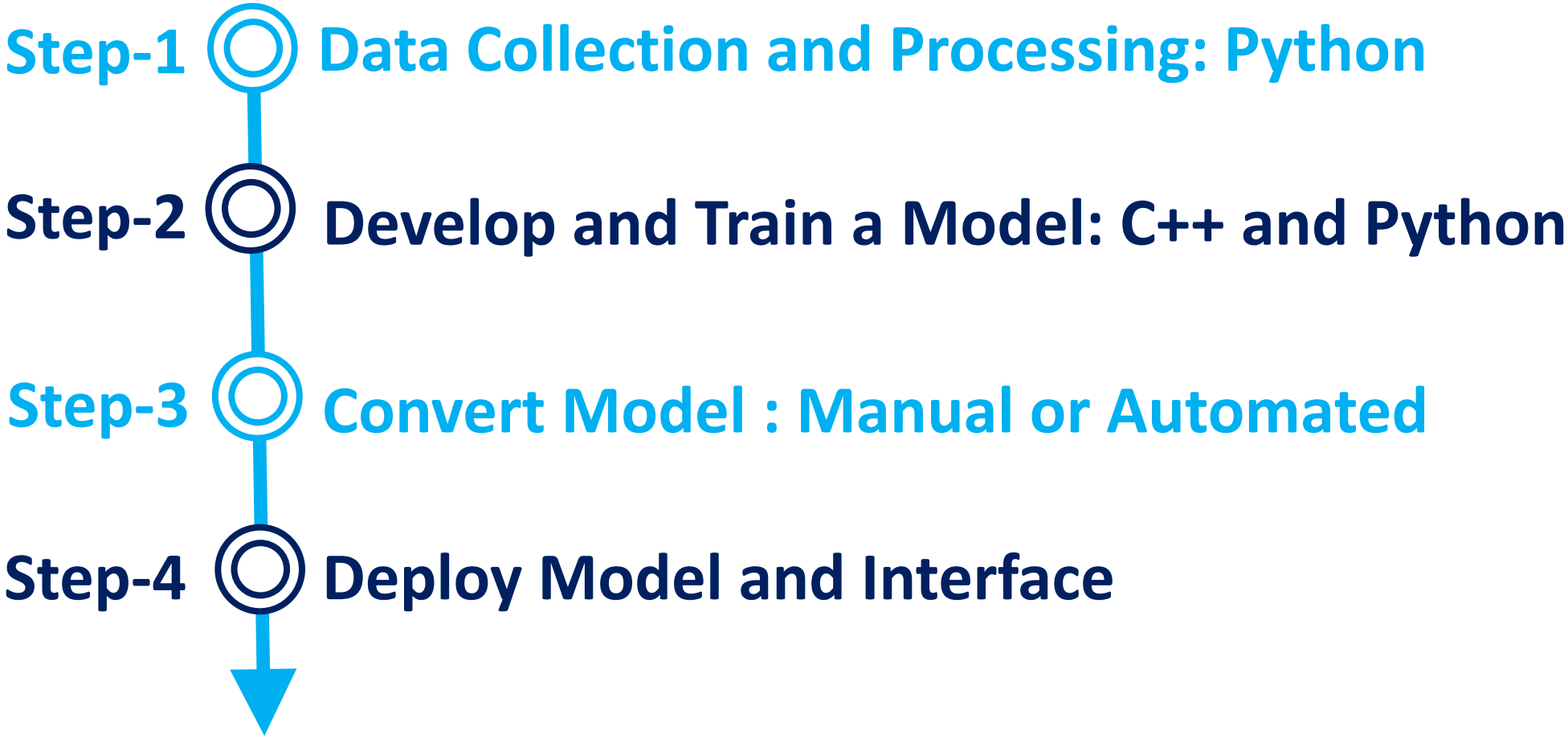
Course Contents

- Designing the Layers
- Hardware Implementation and Verification
- Generating weights, biases and test data from TensorFlow
- Automatic Generation of Neural Networks for FPGAs
- Neural Networks in C from scratch: Weather Prediction
- Single-In-Single-Out (SISO) NNs
- Multiple-In-Single-Out (MISO) NNs
- Multiple-In-Multiple-Out (MIMO) NNs
- Multiple-In-Multiple-Out (MIMO) NNs with a Hidden Layer
- Pipeline of Neural Nets

Evaluation Scheme

- Second In-Sem Exam: 30%
- End Sem-Exam: 30%
- Labs : 40%

AI Deployment



Date: 17/10/2023

AI on FPGA

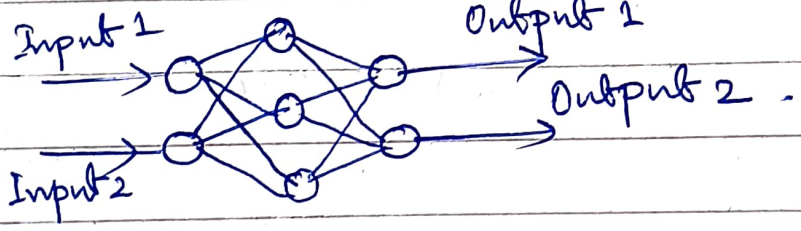
AI model → NN

Model implementation → Verilog / python.

Why AI on FPGA → Speed. ×100.

Implementation of AI model at ckt level.

Fully connected NN



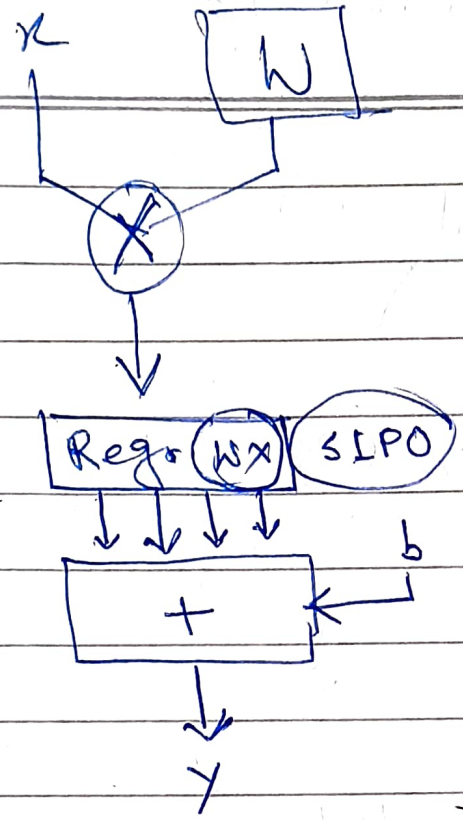
NN for FPGA Implementation

1. Feed forward Network.
2. Pre-trained
3. Number Representation
   - fixed point representation.
   - IEEE floating point representation (32 bits / 64 bits)
4. Activation function.
   - Sigmoid
   - Hyperbolic tangent

   → Look-up-table

Date: 19/10/2023

Date: 20/10/2023



Date: 26/10/2023

Sigmoid function

$$y = \frac{1}{e^{-x} + 1}$$

$$= \frac{1}{e^{\left(\sum w_i x_i + b\right)} + 1}$$

* Activation function
  - i) Sigmoid function → (exp) look-up table.
  - ii) ReLU
  - iii) Soft max

As sigmoid fn. is very complex to implement in FPGA we use lookup table to store the sigmoid fn data. Data store in binary format in the .mif file.

Date: 27/10/2023.

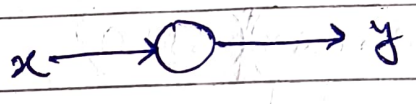
Date: 02/11/2023

NN from Scratch

Single-in Single Out NN

(SISO NN)

$y = wx + b$.



-- XYZ --

Add bias.

Date: 09/11/2023.

Architecture level implementation —

Timing Analysis

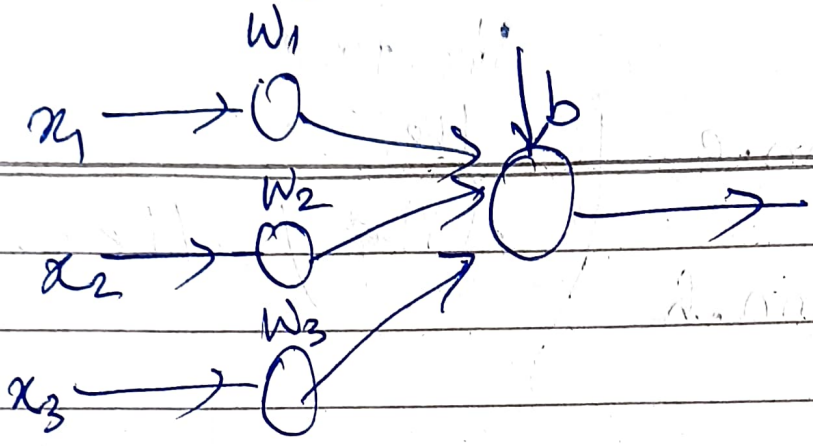
Simple machine learning

FPGA implementation process:

Use verilog code to write the model in FPGA. Get the weight and bias from Edge impulse/py and write the model manually to implement in the FPGA. Take data from .mif file and make the model. It will improve the operation speed and reduce the energy use.

MISO

Date.: 10/ 11/ 2023

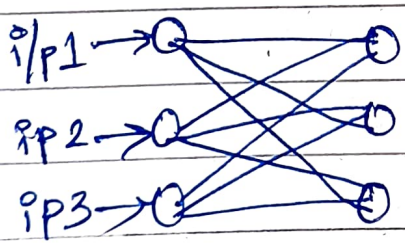


Code in C. → .C file, .h file. main.c file

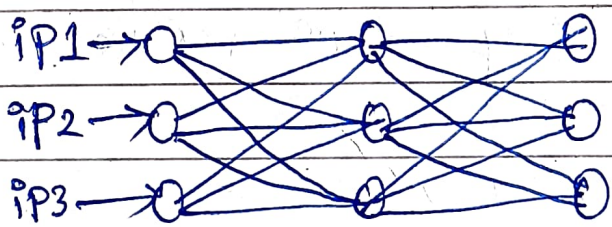
Date: 21/ 11/ 2023.

NN model Implementation in C

(1) Single-input-Single-Output
(11) Multiple-input- Single - Output
(111) Multiple-input- Multiple Output
(1V) Multiple- input- Multiple-Output with hidden layer.



MIMO



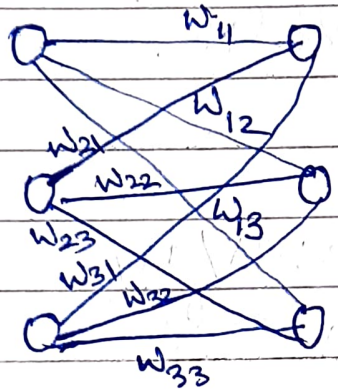
MIMO with hidden layer.

Weight value ⊗ we will get from Edge impulse

Simple_nn
- main.c
- simple_nn.c

header
- simple_nn.h

} Structure of the C code file / folder.



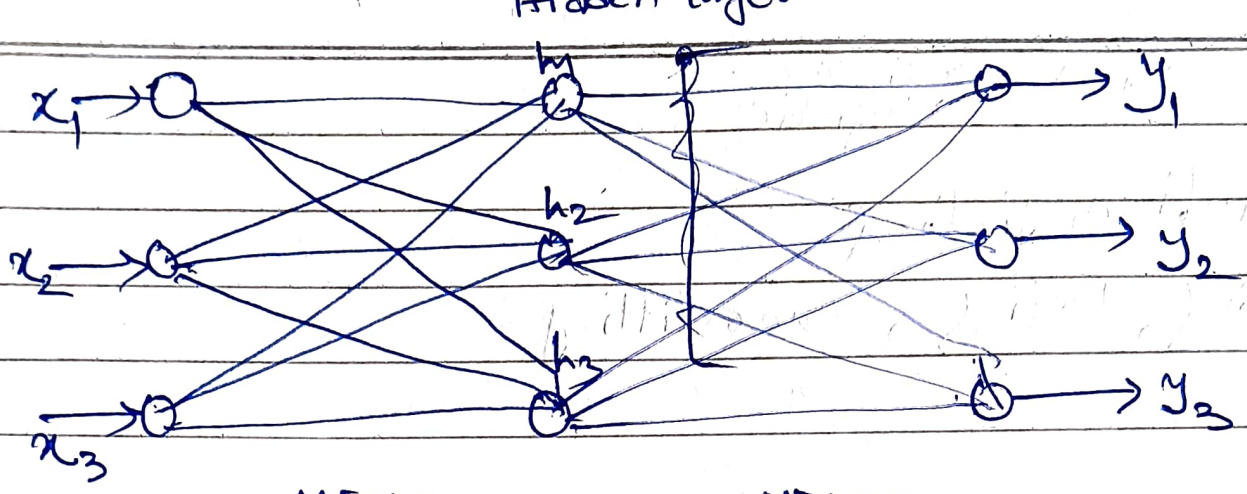
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$y[i] = \sum_j w_{ij} x_j$$

Write the C code in the files. (Header and simple_nn.c)

* Write C program as per the requirement.

Simple NN with hidden layer



$$y_i = \sum W_K y_j$$
$$y_j = \sum W_L x_i$$

~~for two layers~~ two MIMO layer is one hidden layer.

Microcontroller implementation :

→ Get the bias and weight value from py/Edge impulse. Training / testing is performed using python / Edge impulse

→ write the header file to store the structure, weight and bias value and their relation. The header file we can call from the main.c file. This is written in C langue.

→ Then we write the c++ code for Arduino which call the header file to calculate NN outputs providing inputs from sensor.

Date : 23/11/2023

Regression → linear
logistic

Convolution, Pooling

Training, testing, Over-fitting

Sine wave model prediction using python

Nearest neighbor

Draw architecture for $y = mx + c$
Adder, Down counter,
Multiplayer, ...

Use sigmoid function.

Linear Regression model → ~~may be~~ 1 Question

Convolution + Pooling → Question
(CNN / DNN)

Power Optimization → Question