EL530-Introduction to Embedded Artificial Intelligence

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

Instructor: Tapas Kumar Maiti (TKM)

Mode of Delivery:

- Lectures and Labs delivery mode: **Physical**
- Course materials will be available in **Google-Classroom**

Lecture time in each week (CEP106):

- **Tuesday : 8:00 – 8:50**
- **Thursday : 10:00 – 10:50**
- **Friday: 10:00 – 10:50**

Lab time in each week (LAB211):

Friday: 16:00 - 18:00

No Lecture and Lab on holidays

Course Contents

Part-I Fundamentals of Embedded AI

- Microcontroller, FPGA, Power, and Energy
- C++ and Python for Embedded-AI

Part-II Embedded AI with Microcontroller

- Arduino, Raspberry Pi Pico,
- Naive Bayes Classifier, Nearest Neighbor Classification
- Neuron Networks, TensorFlow Lite
- Use of C++ and Python for Embedded-AI

Part-III Embedded AI with FPGA

- Intel ModelSim, FPGA, OpenLANE
- Neuron Networks, Activation Function
- Use of Python for Embedded-AI

Part-IV Embedded AI Applications

- Naive Bayes Classifier, Anomaly Detection, Audio Classification, Weather Station, Indoor Scene Classification, Human Gesture Recognition, NN on FPGA
- Embedded AI-Ops

Evaluation Scheme

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

Embedded AI

Intersection between artificial intelligence (AI) and embedded systems for smart applications

Artificial Intelligence	Embedded Devices
• Naive Bayes • Nearest Neighbor • Neural Network • Deep Learning	Microcontrollers • ARM Processor • FPGA • GPU

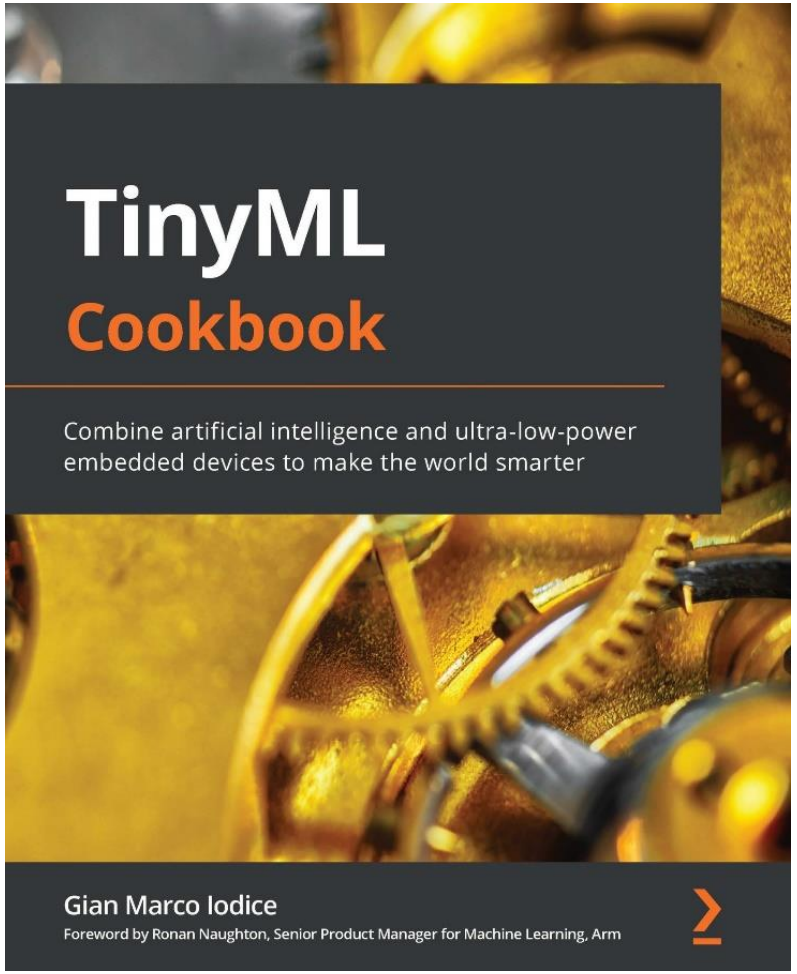
Metrics

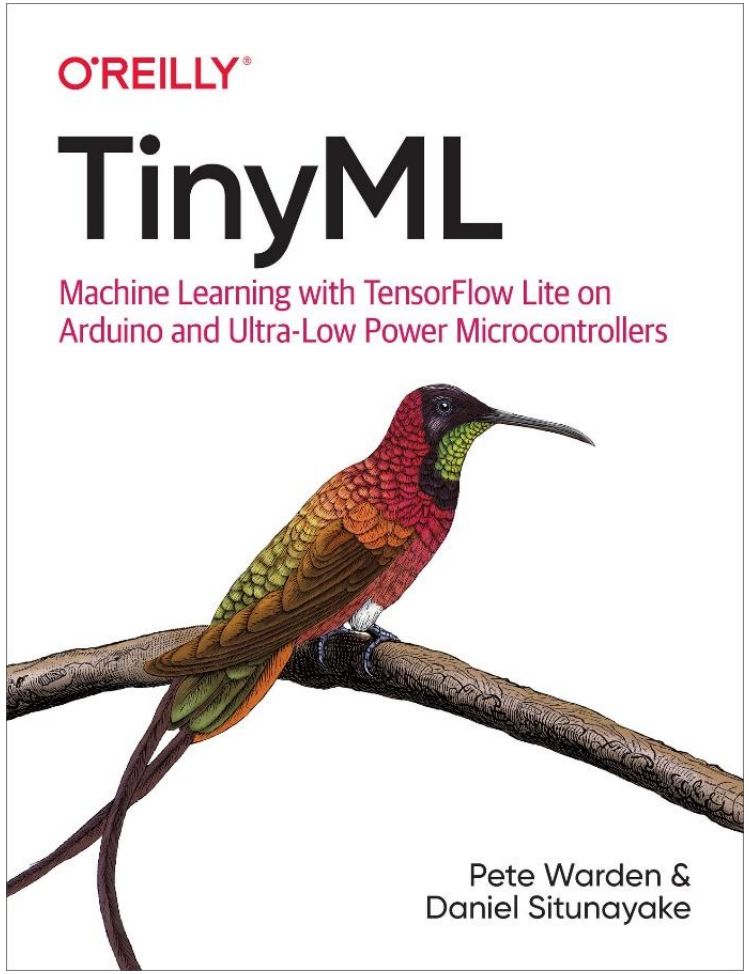
Low-Power	High-Speed	Memory Size
mW or below	mS or less for single task	KB or less

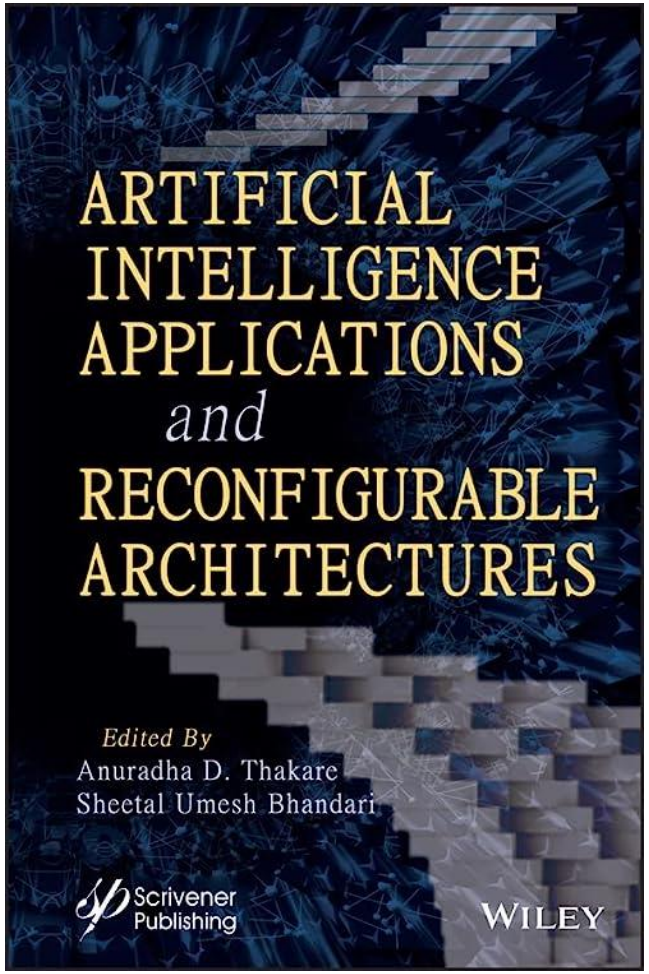
What you will Learn

- Fundamentals of microcontroller programming for embedded AI. Examples: Arduino, Raspberry Pi Pico, Jetson Nano, FPGA
- Use of sensors such as IMU, Camera, and microphone
- Use of actuators such as dc motor and servo-motor
- Run artificial intelligence (AI) on microcontroller
- Prototyping embedded AI for real-world applications

Books

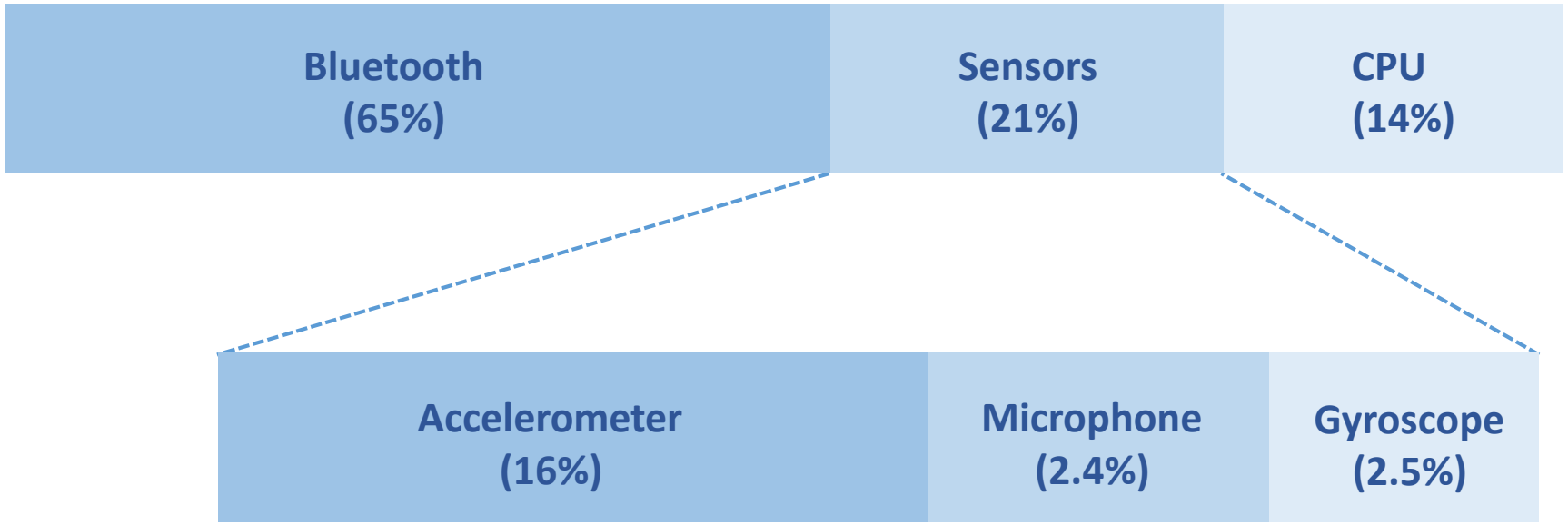

TinyML
Cookbook
Combine artificial intelligence and ultra-low-power embedded devices to make the world smarter
Gian Marco Iodice
Foreword by Ronan Naughton, Senior Product Manager for Machine Learning, Arm


O'REILLY
TinyML
Machine Learning with TensorFlow Lite on Arduino and Ultra-Low Power Microcontrollers
Pete Warden &
Daniel Situnayake

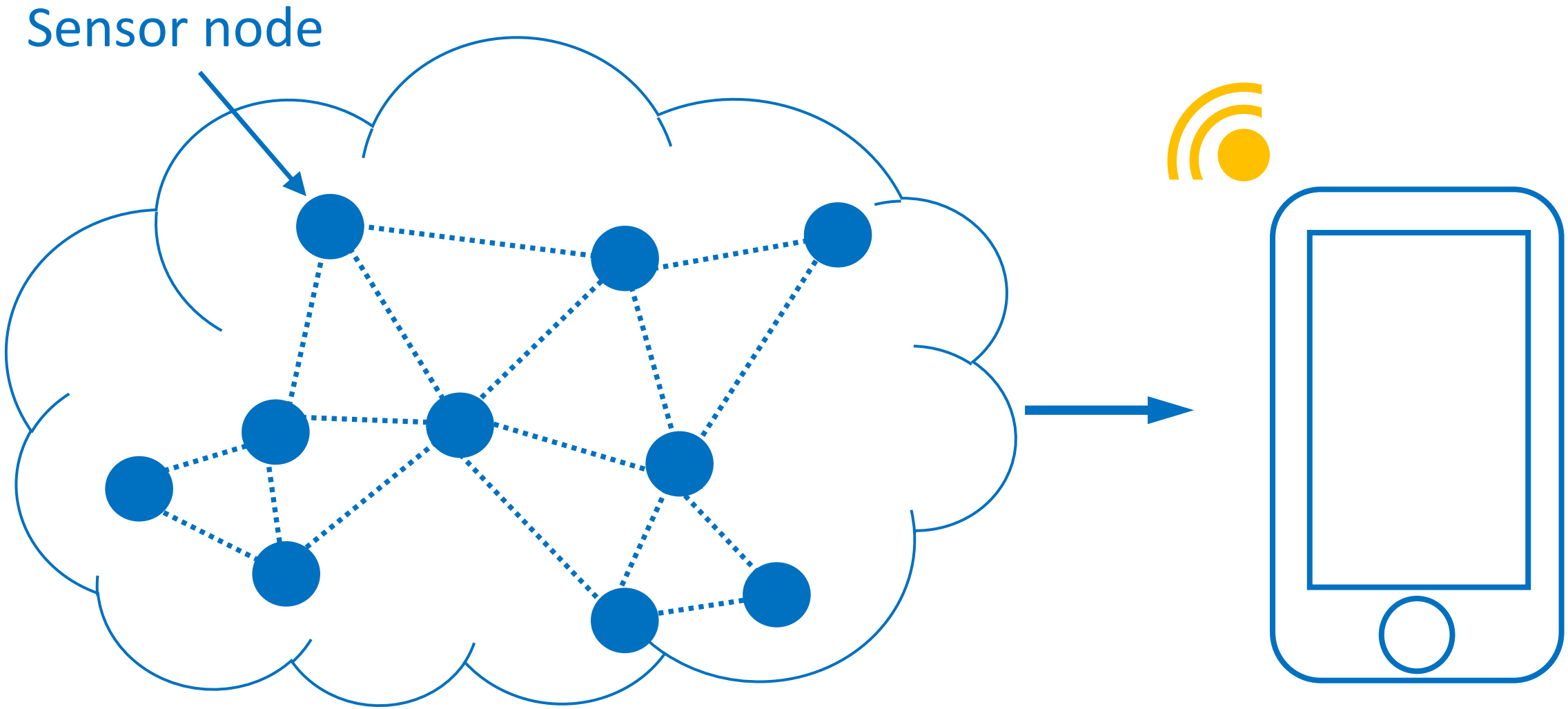

ARTIFICIAL INTELLIGENCE APPLICATIONS and RECONFIGURABLE ARCHITECTURES
Edited By
Anuradha D. Thakare
Sheetal Umesh Bhandari
Scrivener Publishing
WILEY

Power Consumption Breakdown

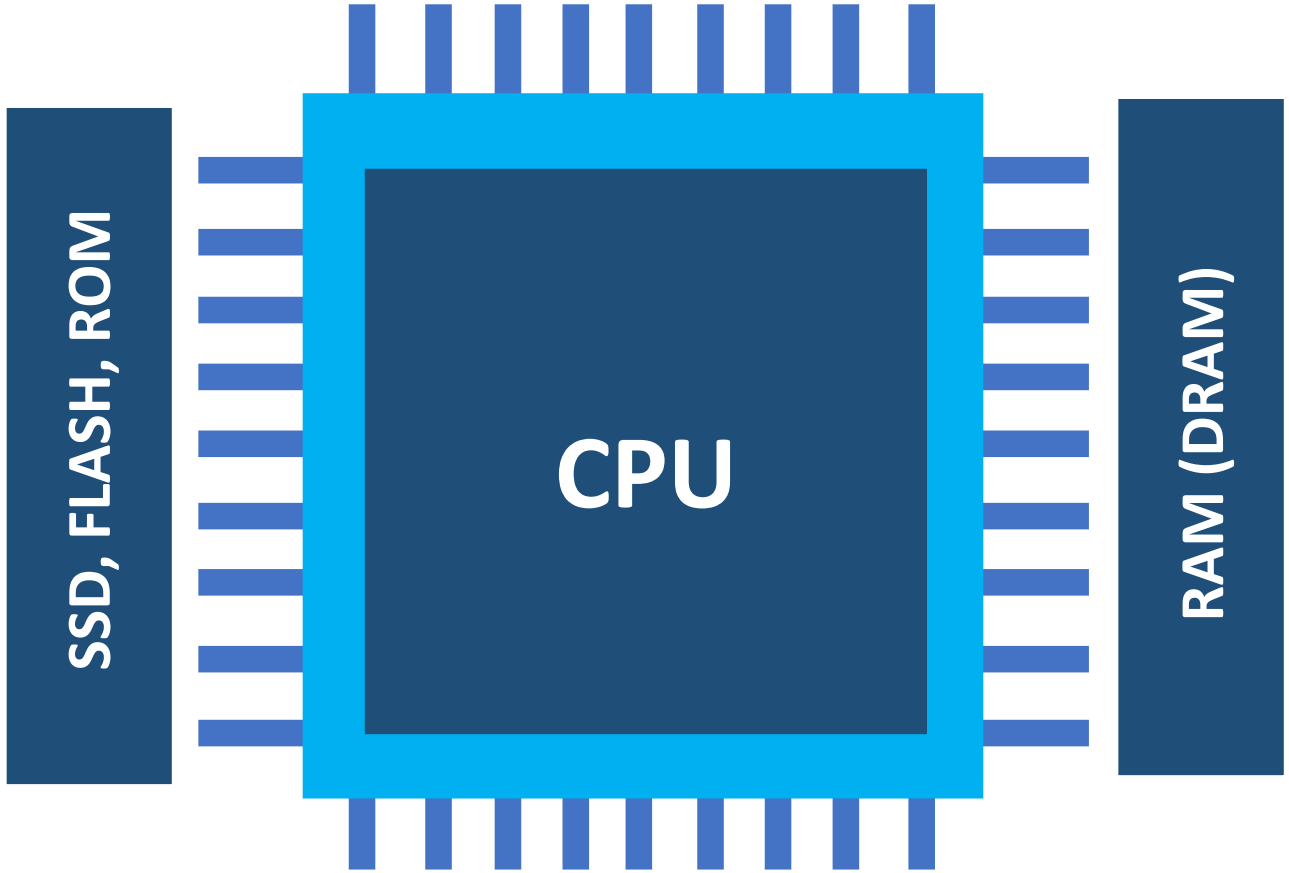
(For the Arduino Nano 33 BLE Sense board)



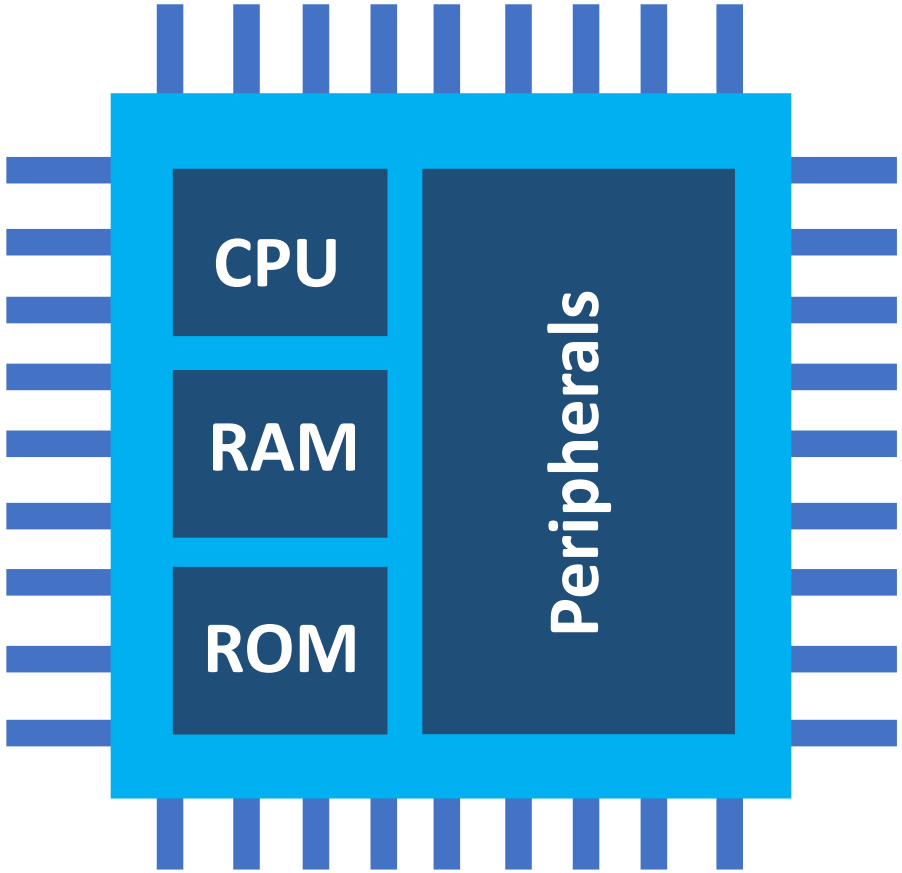
Wireless Sensor Network (WSN)



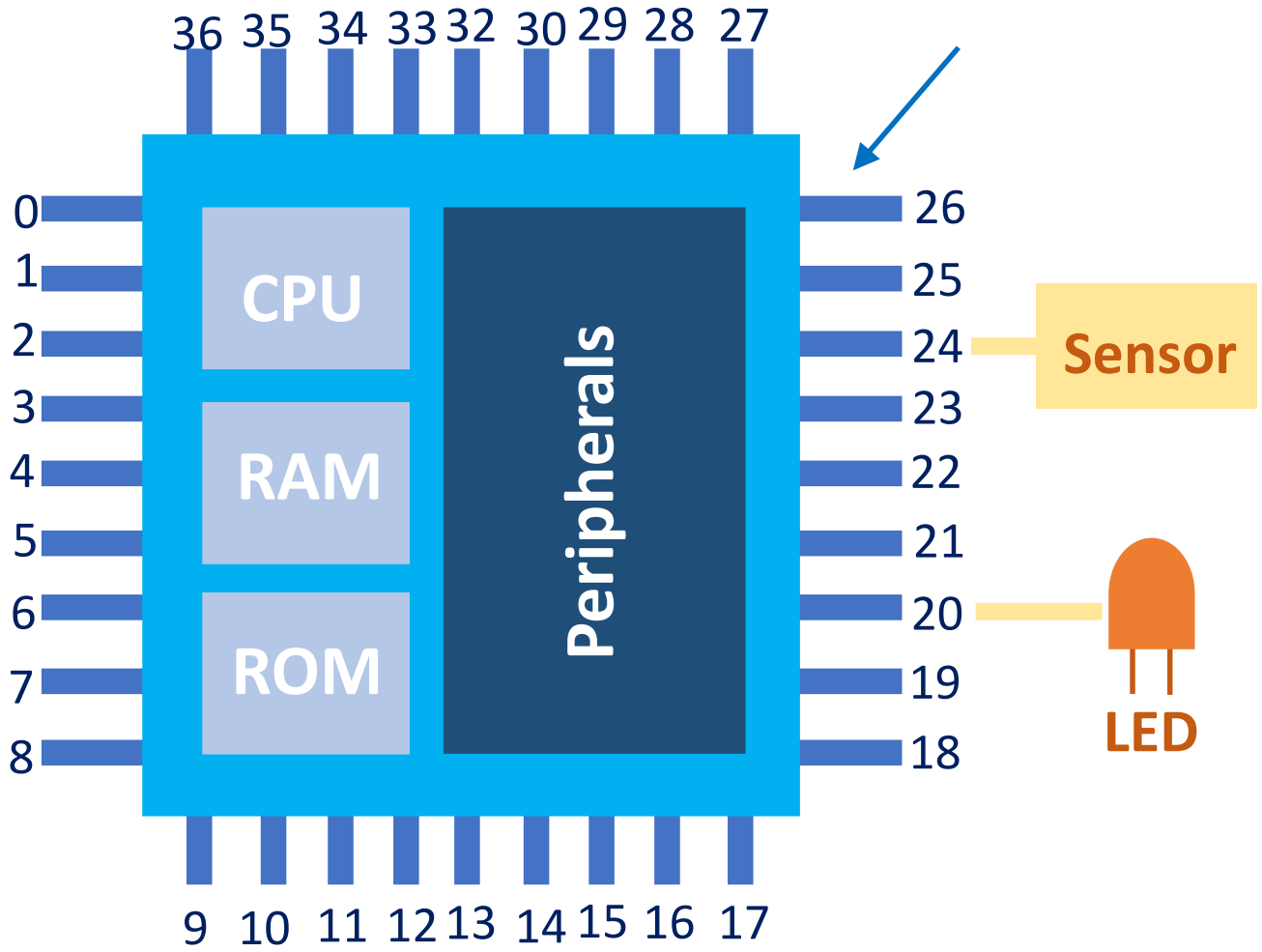
Microprocessor



Microcontroller

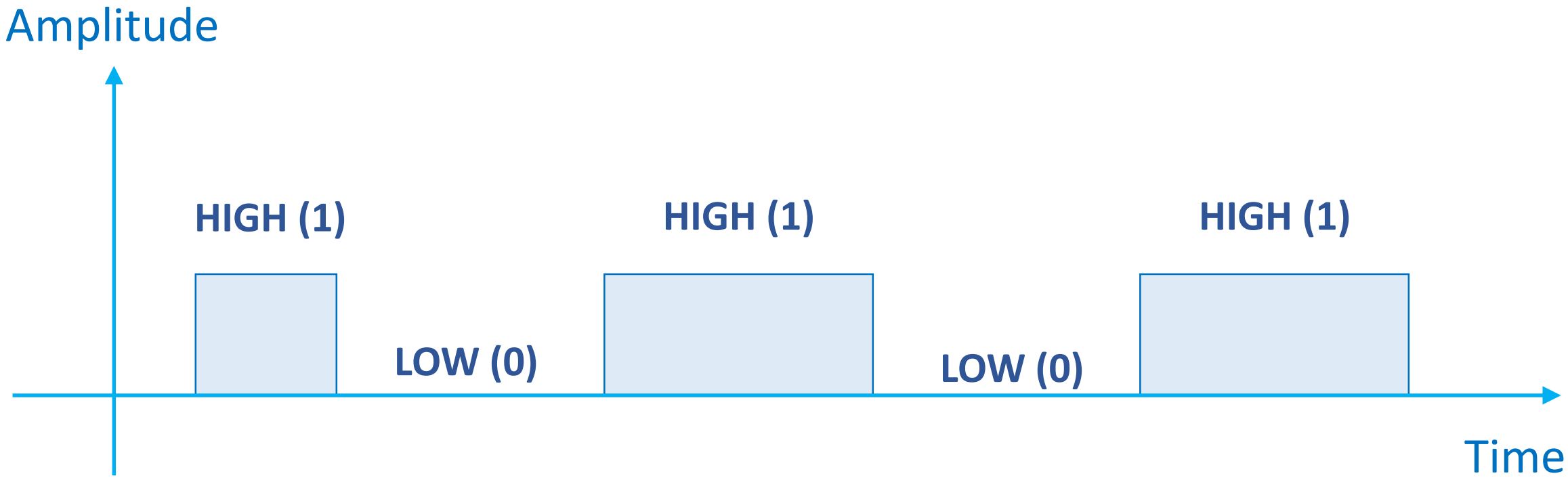


Microcontroller

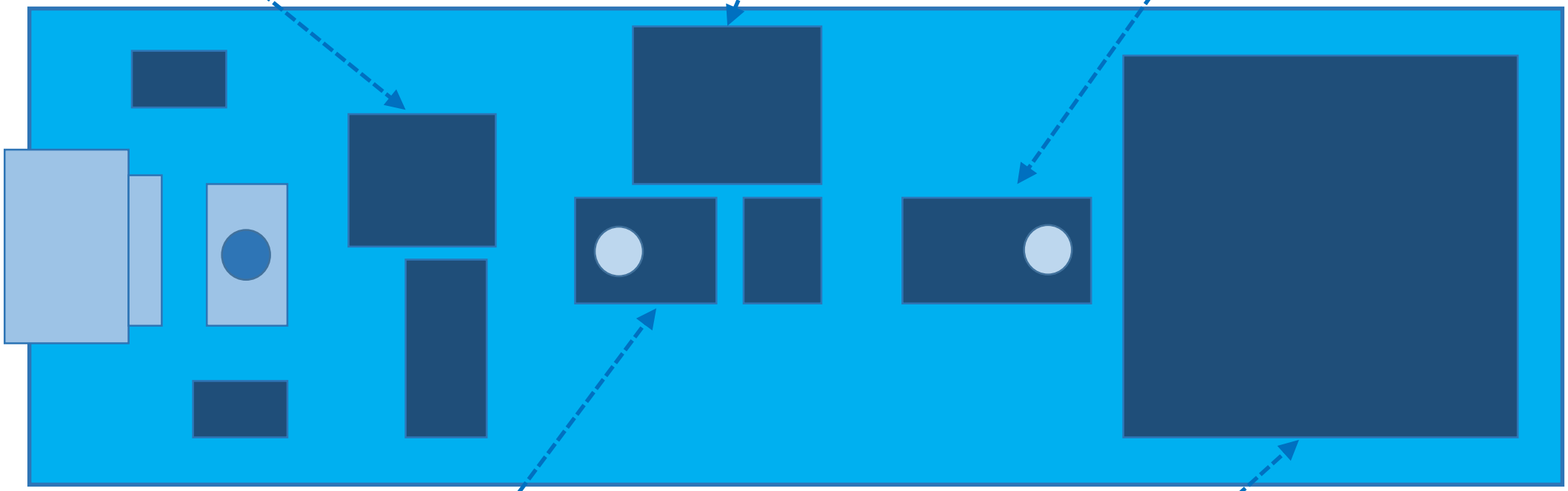


Peripheral Type	PIN Number
-----	----
PINA	24
-----	----
PINB	20
-----	-----
----	-----

General-purpose input/output (GPIO/IO)



Arduino Nano 33 BLE Sense Board



Sensor
Temperature, humidity, and pressure

Sensor
Motion, vibration, and orientation

Sensor
Microphone

Sensor
Colour, brightness, proximity, gesture

Microcontroller (nRF52840)
Arm Cortex-M4 CPU with Bluetooth Low Energy (BLE) Module

Raspberry Pi Pico

Microcontroller (RP2040)

Dual-Core Arm Cortex-M0+ CPU

Embedded AI Development Boards

Board	MCU	CPU	Clock	Memory	IO	Sensor(s)	Radio
Arduino Nano 33 BLE Sense	Nordic nRF52840	32-bit ARM Cortex-M4F	64 MHz	1 MB flash 256 KB RAM	x8 12-bit ADCs x14 DIO UART, I2C, SPI	Mic, IMU, temp, humidity, gesture, pressure, proximity, brightness, color	BLE
Espressif ESP32-DevKitC	ESP32 D0WDQ6	32-bit, 2-core Xtensa LX6	240 MHz	4 MB flash 520 KB RAM	x18 12-bit ADCs x34 DIO** UART, I2C, SPI	Hall effect, capacitive, touch***	WiFi, BLE
Espressif EYE	ESP32 D0WD	32-bit, 2-core Xtensa LX6	240 MHz	4 MB flash* 520 KB RAM	SPI via surface pads	Mic, camera	WiFi, BLE
MAX32630FTHR	Maxim MAX32620	32-bit ARM Cortex-M4F	96 MHz	2 MB flash 512 KB RAM	x4 10-bit ADCs x16 DIO UART, I2C, SPI	Accelerometer, gyroscope	BLE
Teensy 4.0	NXP iMXRT1062	32-bit ARM Cortex-M7	600 MHz	2 MB flash 1 MB RAM	x14 10-bit ADCs x40 DIO** UART, I2C, SPI	Internal temperature, capacitive touch	None

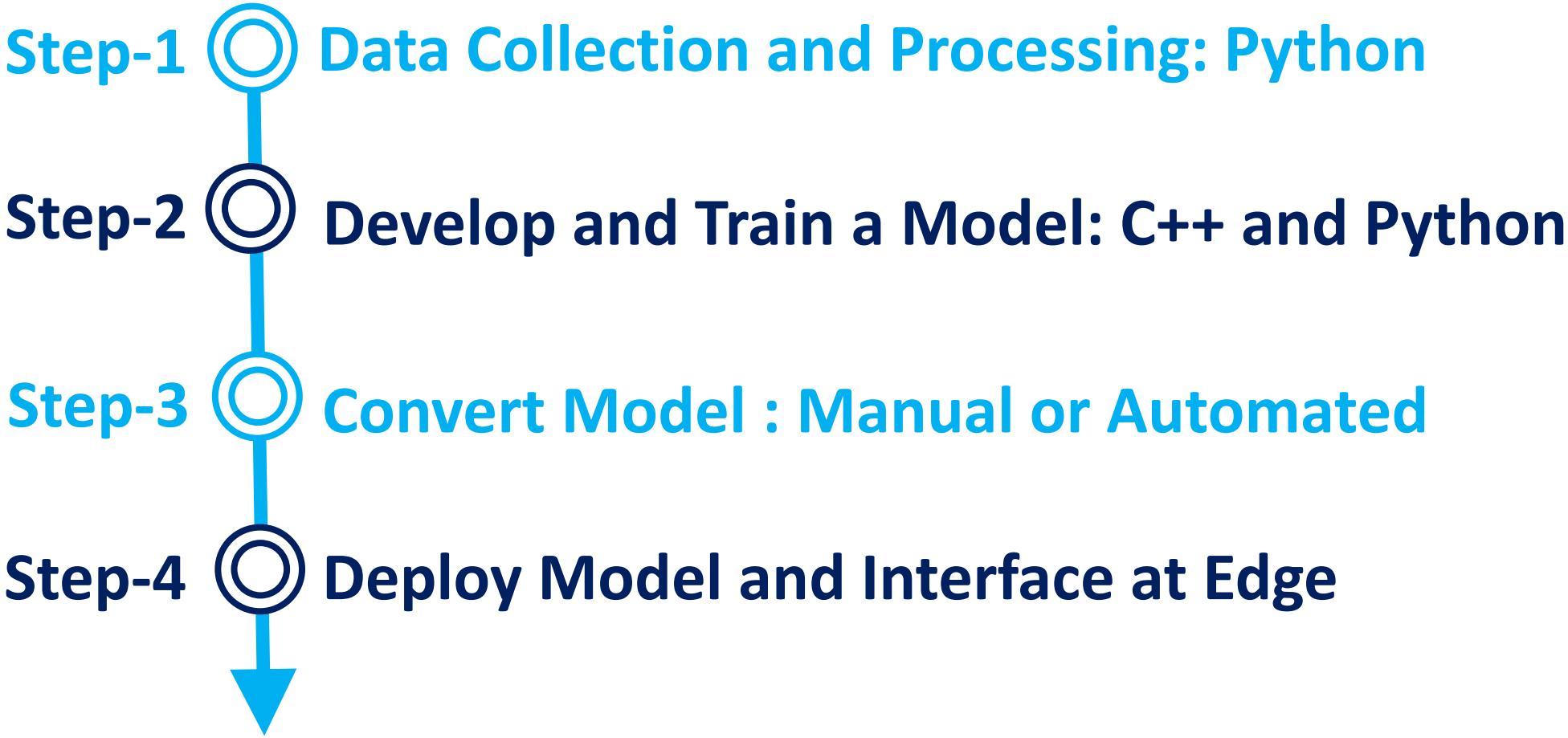
Graphics Processing Unit (GPU)

Hardware	GPU	CPU	Framework
AWS DeepLens	Generation 9 (Gen 9) Graphics	Intel Atom	Apache MXNet, TensorFlow, Caffe
AWS Panorama	NVIDIA Jetson Xavier	ARMv8.2 64-bit	Apache MXNet, TensorFlow, Keras, Darknet
Raspberry Pi 3, 4	Broadcom VideoCore VI	ARMv8-A	Apache MXNet, TensorFlow, OpenCV, Google Coral
NVIDIA Jetson TX2	NVIDIA Pascal	ARM Cortex-A57	Apache MXNet, TensorFlow, Caffe, PyTorch, MATLAB

AI Frameworks

Framework	Developer	System Requirement
TensorFlow Lite	Google	Android, iOS, Microcontrollers
PyTorch	Facebook	Android, iOS
CoreML3	Apple	iOS
MXNET	Apache	Linux, Microprocessors
Embedded Learning Library (ELL)	Microsoft	Linux, Microprocessors

AI Deployment



EL530-Introduction to Embedded Artificial Intelligence

Lecture 2

Basics of AI/ML

Choice of Model

- **Classification Model**
- **AI/ML-Based Classifier**

Types of AI/ML Problems

- Classification
- Regression
- Clustering
- Dimensionality Reduction

Traditional AI/ML Models

- **Classification Models:** Naïve Bayes, SVMs, Decision Trees, Random Forest
- **Regression:** Linear, Lasso, Ridge, SVR
- **Clustering:** K-means, DBSCAN, Spectral Clustering
- **Dimensionality Reduction:** Manifold Learning, Factor

“Traditional” AI/ML-based systems rely on experts (you) to decide what features to pay attention to- and how

Representation ML Models

“Traditional” AI/ML-based systems

figure out by themselves what features

to pay attention to- and how

Example: Deep Learning Models

Traditional vs. Deep Learning Models

Traditional AI/ML Model

- Domain experts
- More insight
- **Scikit-learn**

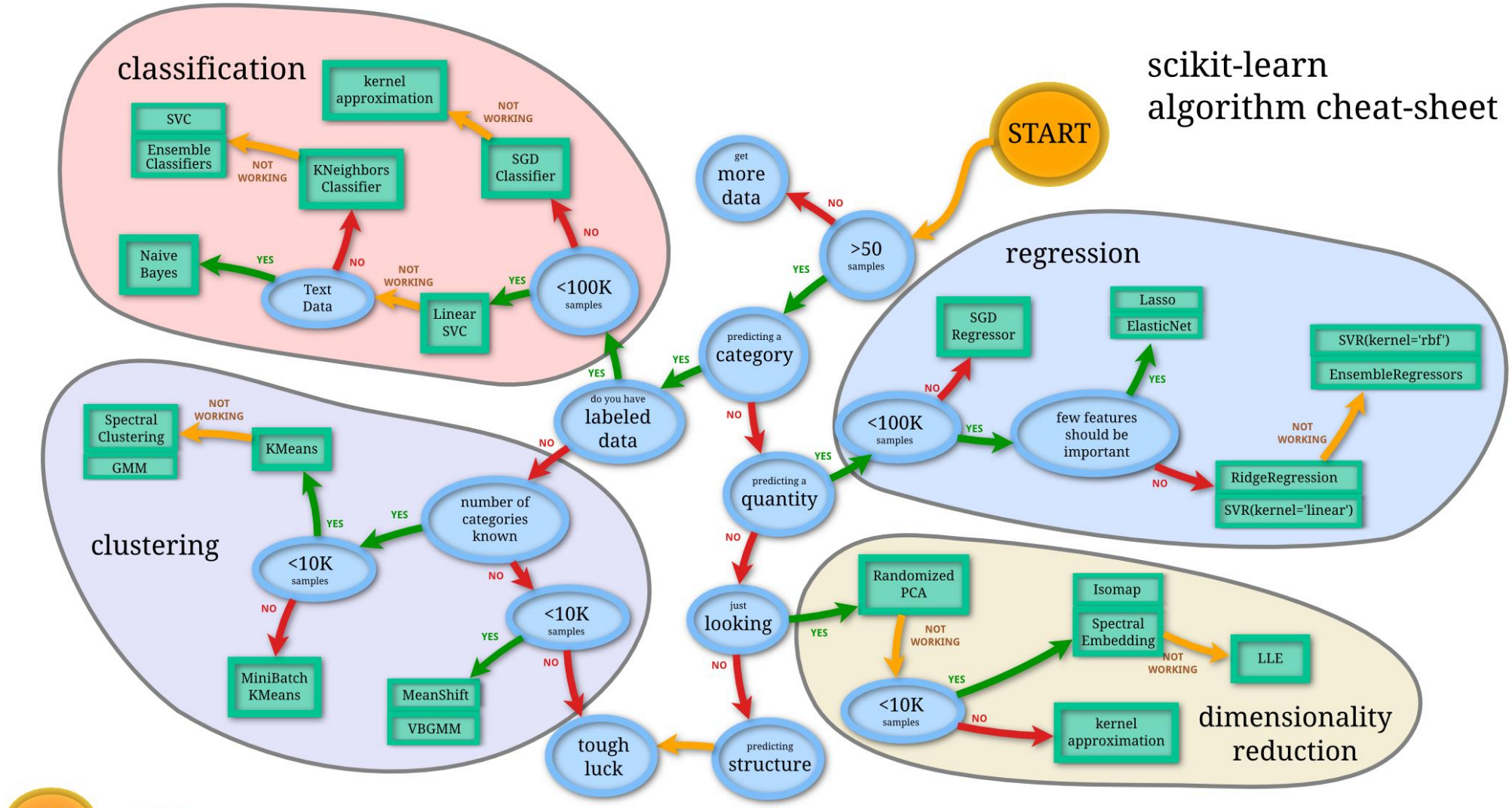
Deep Learning ML Model

- Model itself
- Black-Box
- **Tensorflow, Keras, PyTorch**

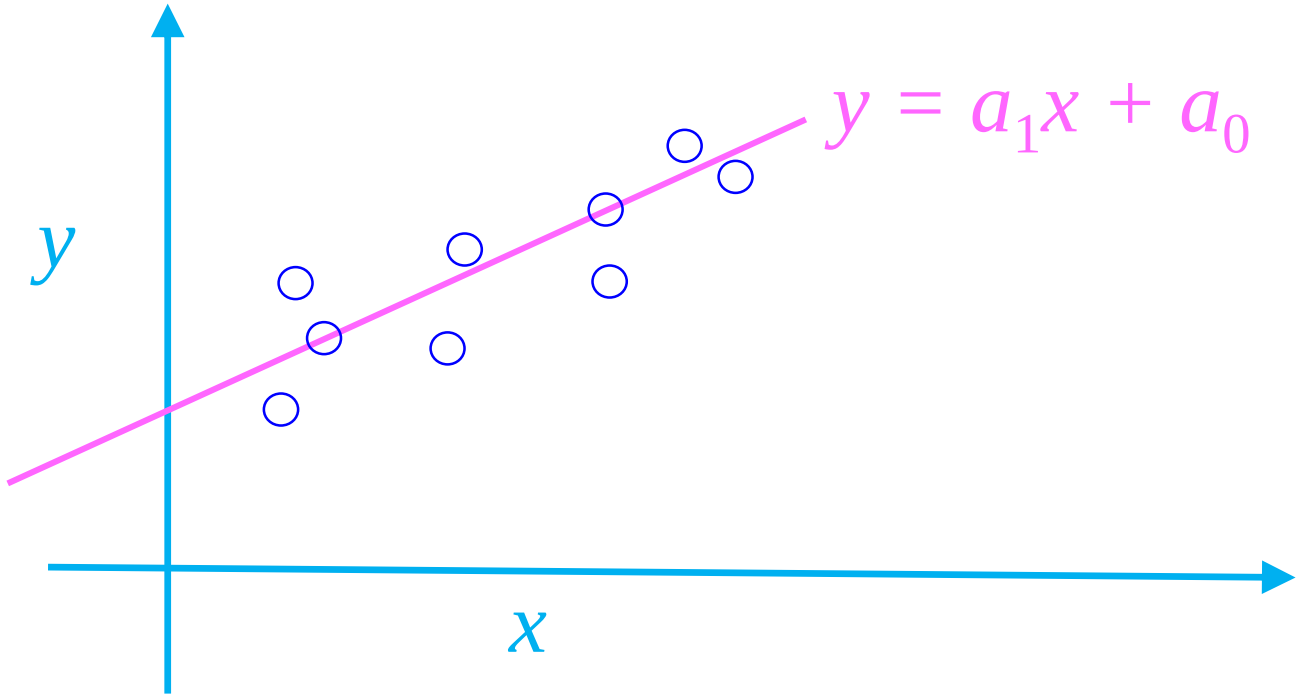
Scikit-learn Based on

- Numpy
- SciPy
- Matplotlib
- Simpy
- Pandas

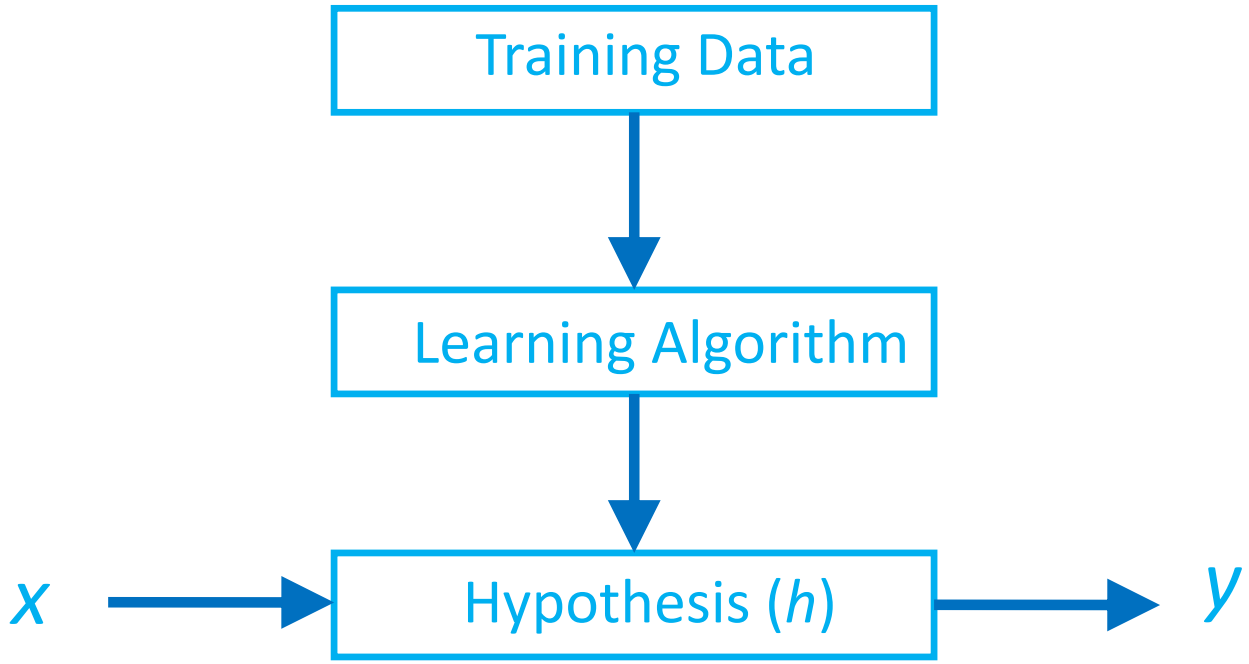
Choosing the Right Model



Linear Regression



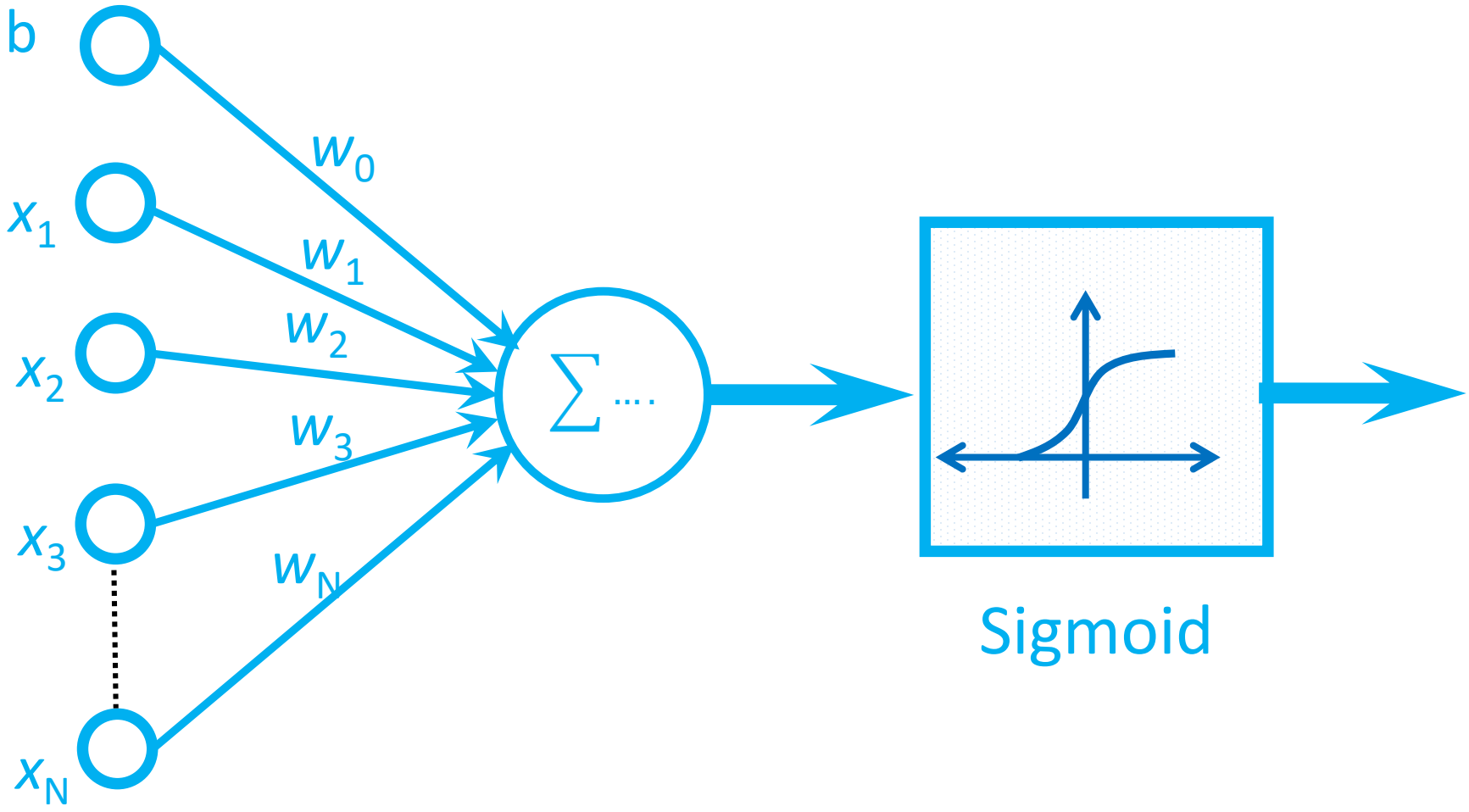
Linear Regression



$$h = a_1 x + a_0$$

$$error = h - y$$

Neural Network



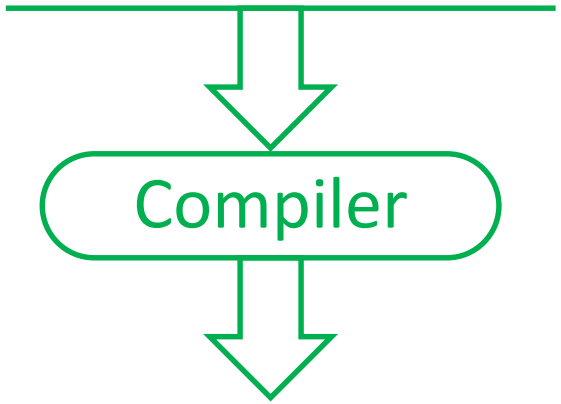
AI/ML Architectures

Programming Languages

AI Model
Python/C++

Model in
High-level
Language

```
m = x + y
n = u * w
If (m == n) then
    result = 2*w
Else
    result = 5*y
```



Model in
Machine
Language

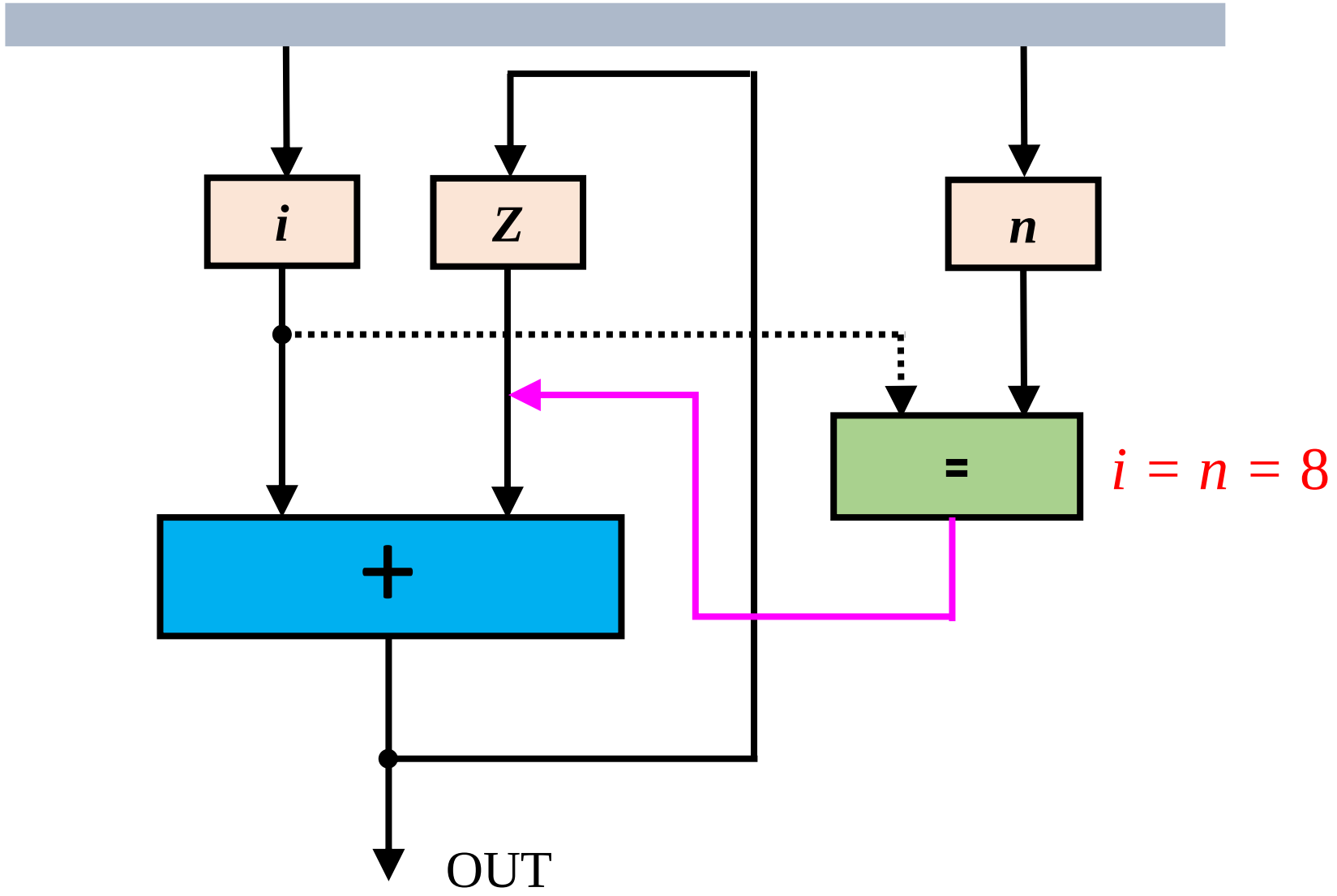
```
0000 0111 0101 1010 0011 1100 0111
0110 0101 0111 1110 0001 1000 0101
1100 0101 1101 1000 0101 1110 1011
0101 1111 1001 1011 1101 0000 0001
.....
```

AI with Microcontroller
and FPGA

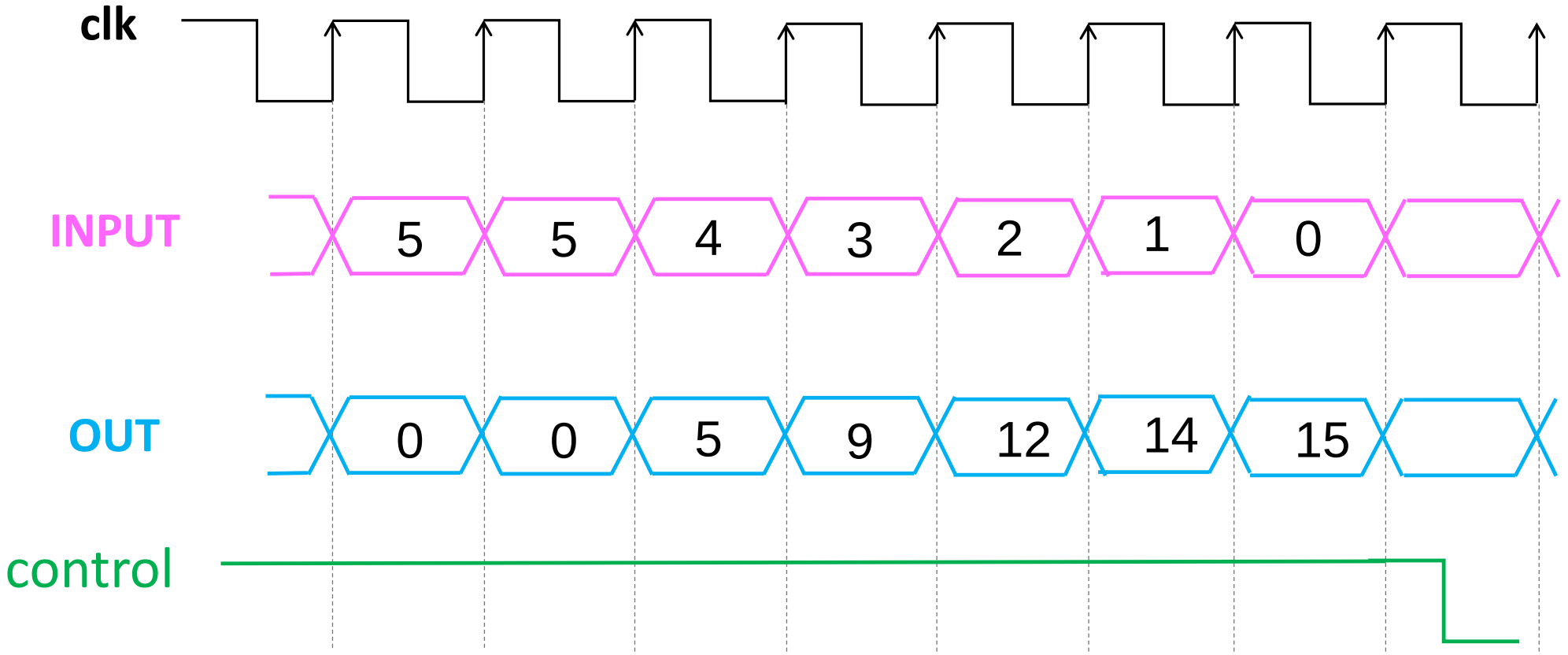
Compute $Z = \sum_{i=1}^{n=8} X(i)$

```
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

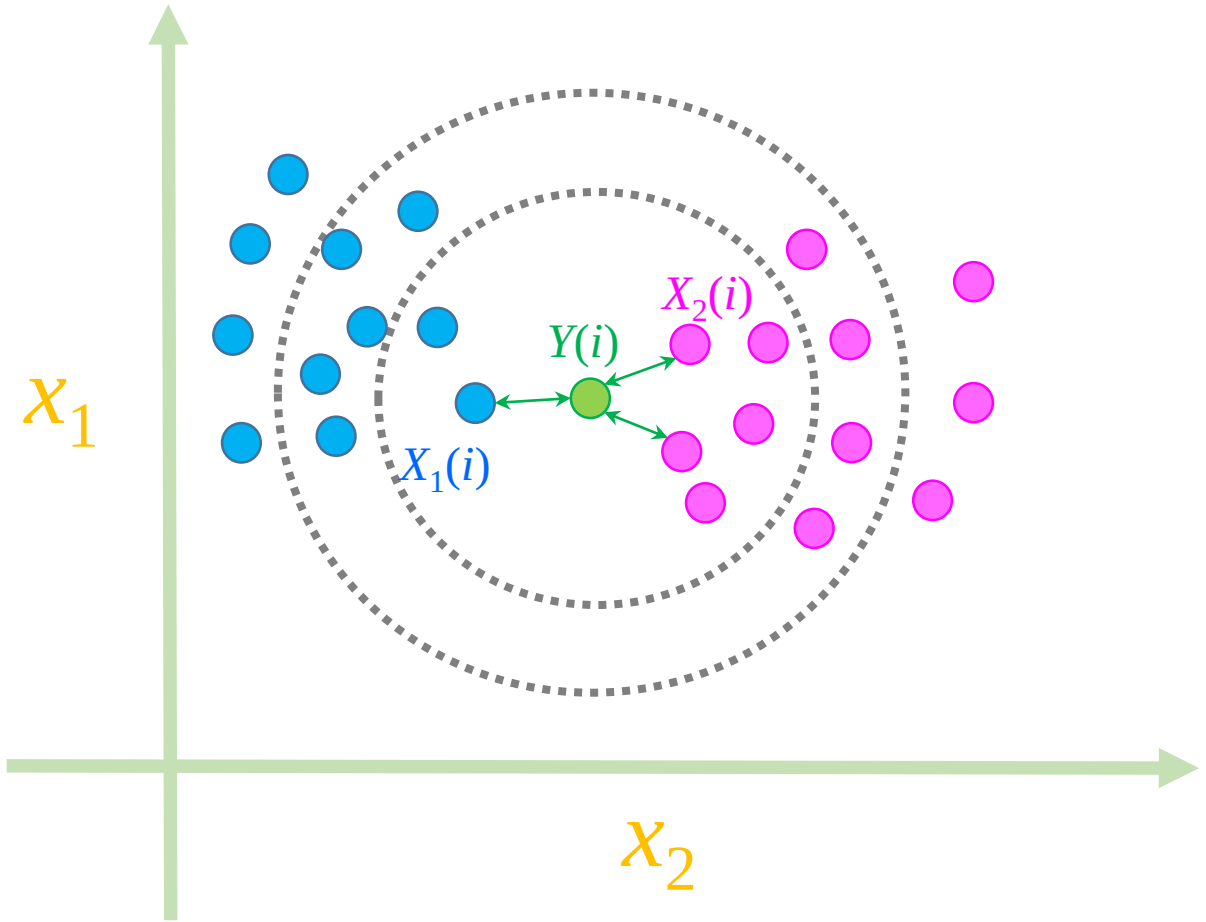
Manhattan Distance: $Z = \sum_{i=1}^{n} \left| X(i) - Y(i) \right|$

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

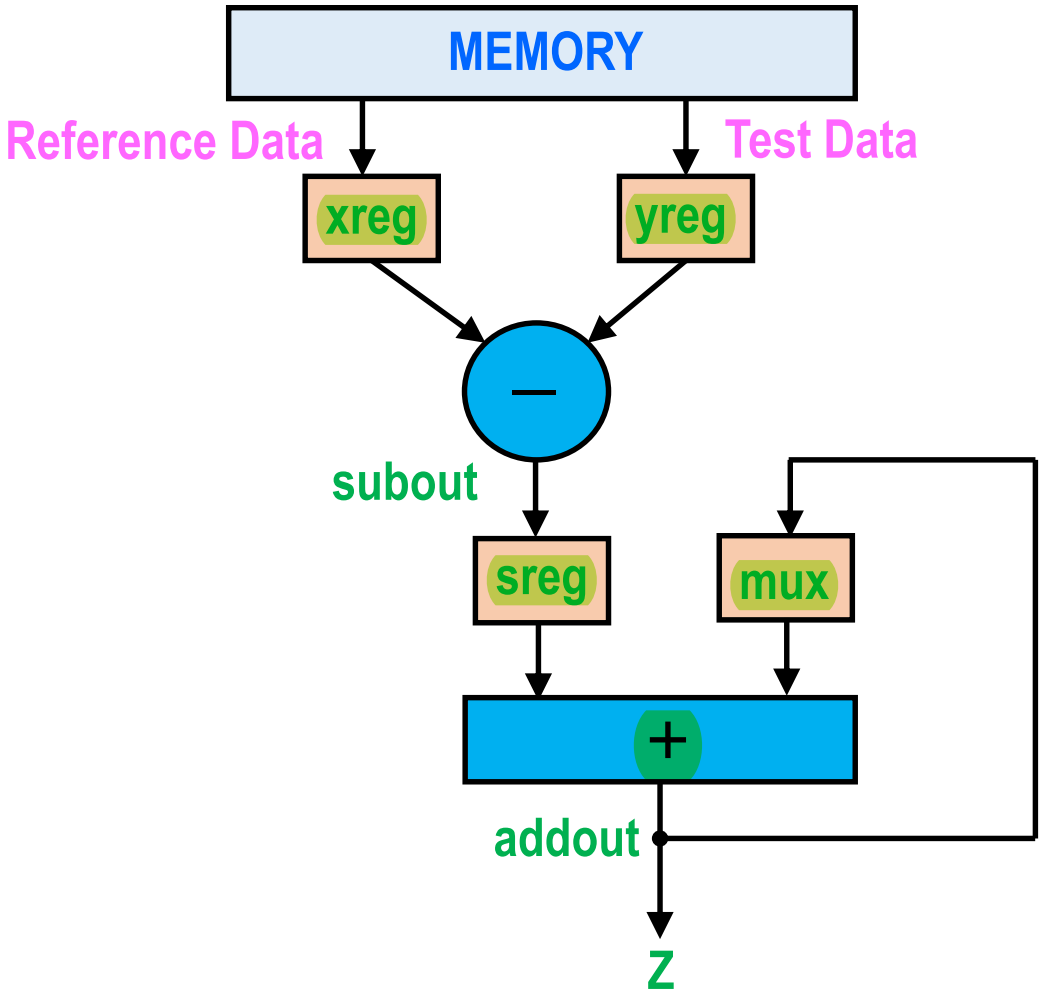
Support Vector Machine (SVM): $Z = \exp\left(-\gamma \left| X(i) - Y(i) \right|^2 \right)$

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

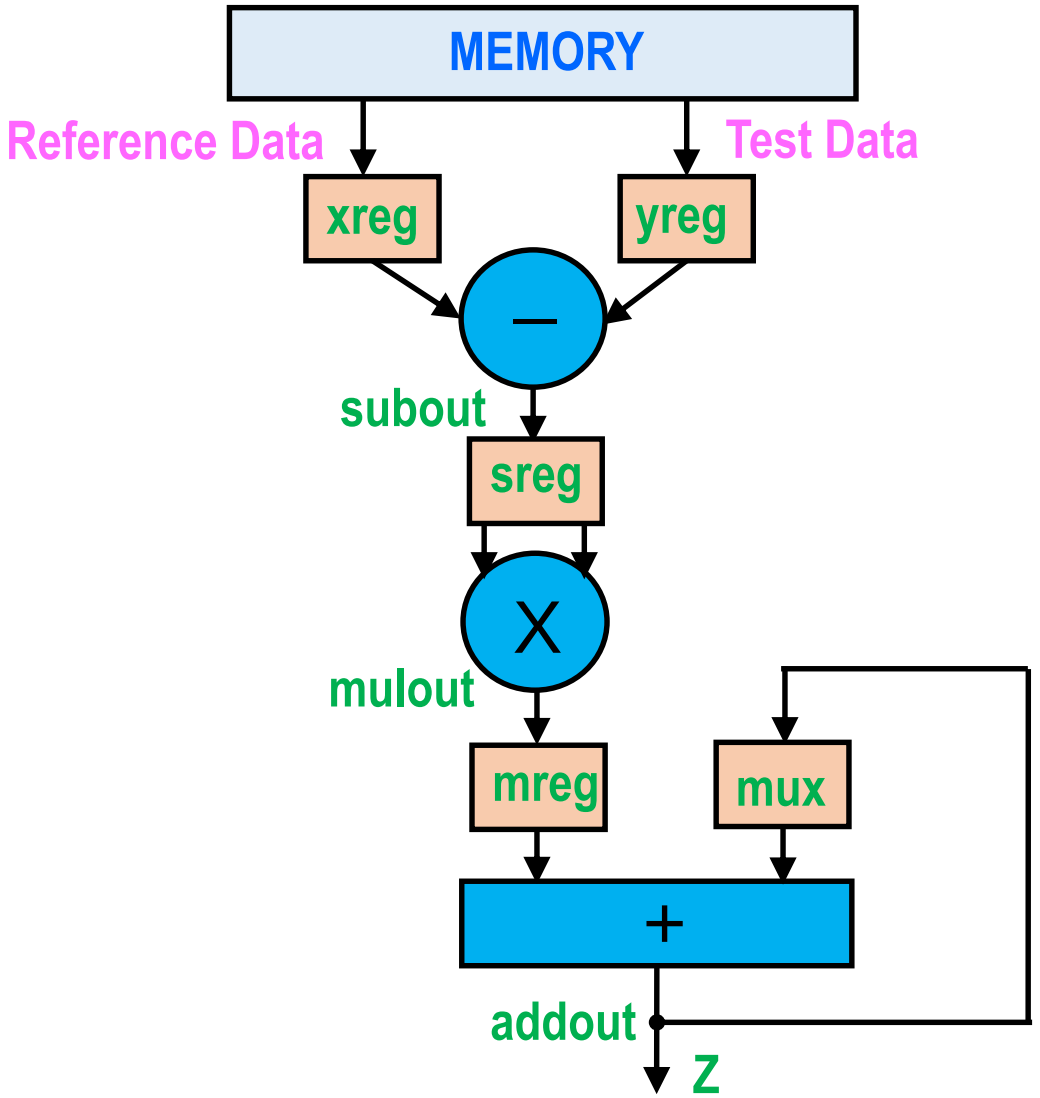
Manhattan Distance: $Z = \sum_{i=1}^{n} |X(i) - Y(i)|$



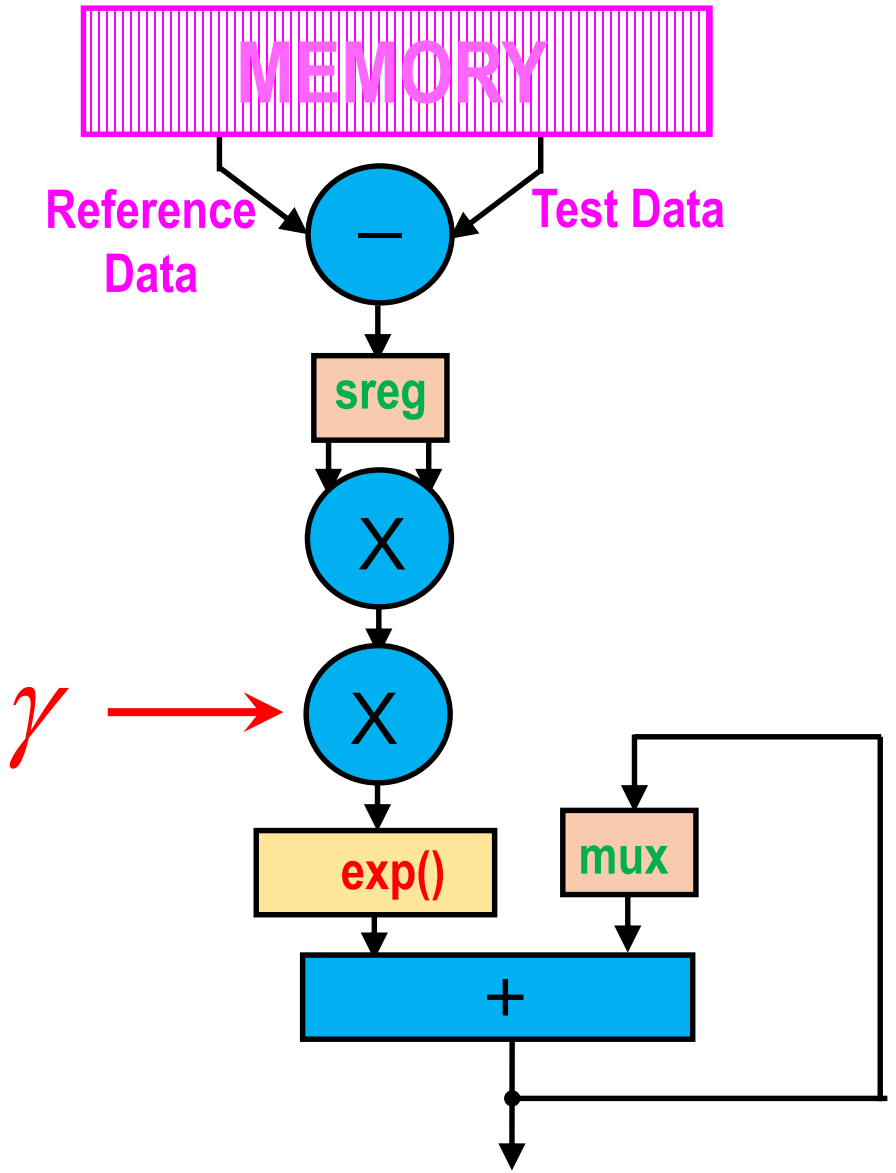
Manhattan Distance: $Z = \sum_{i=1}^{n} |X(i) - Y(i)|$



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

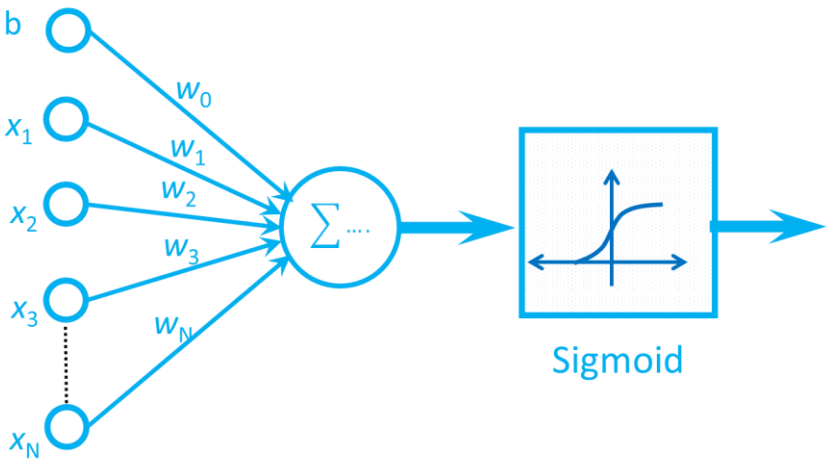


SVM: $Z = \exp\left(-\gamma \left| X(i) - Y(i) \right|^2\right)$



Neural Networks (NN)

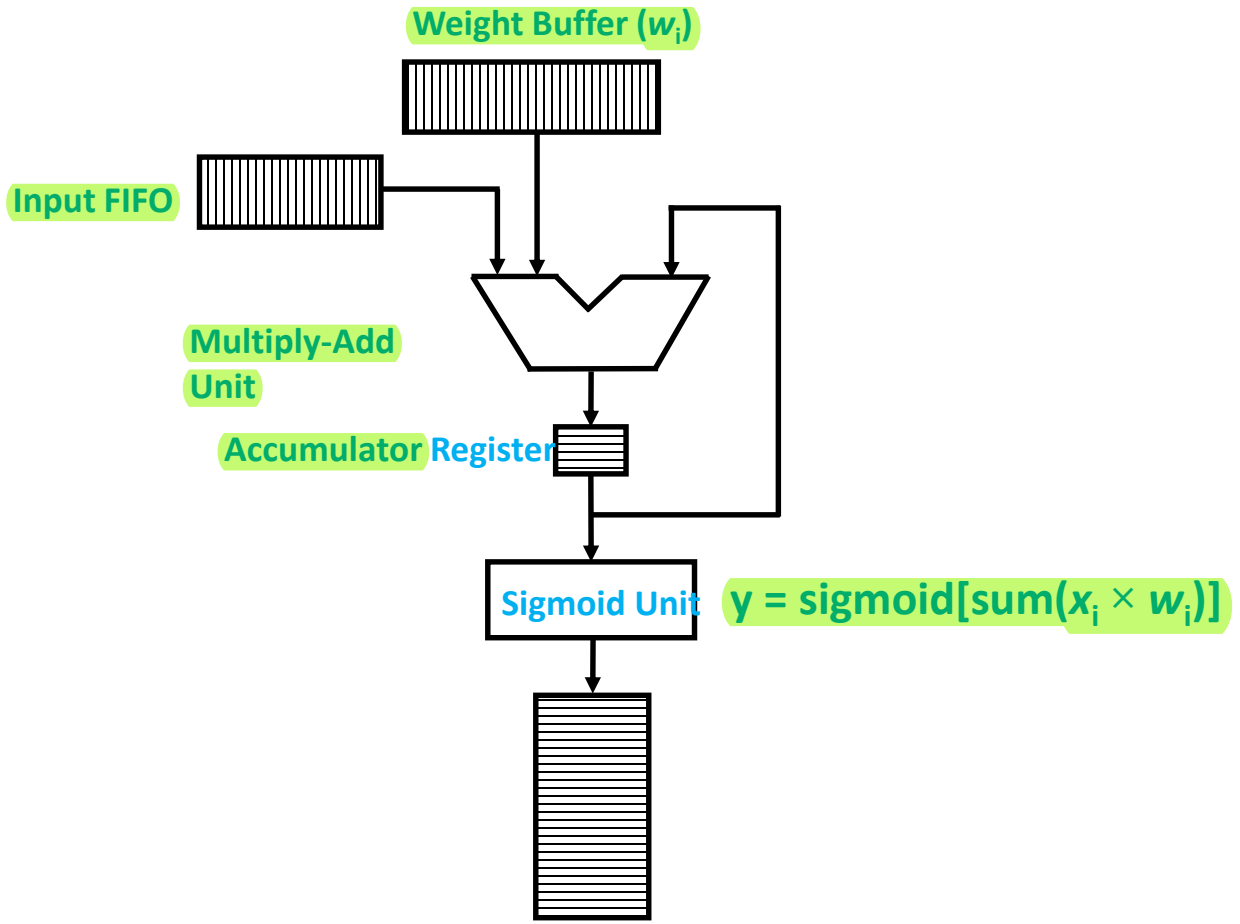
Sigmoid function: $Z = \frac{1}{1+\exp\left[-w(i)\bullet x(i)\right]}$



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)



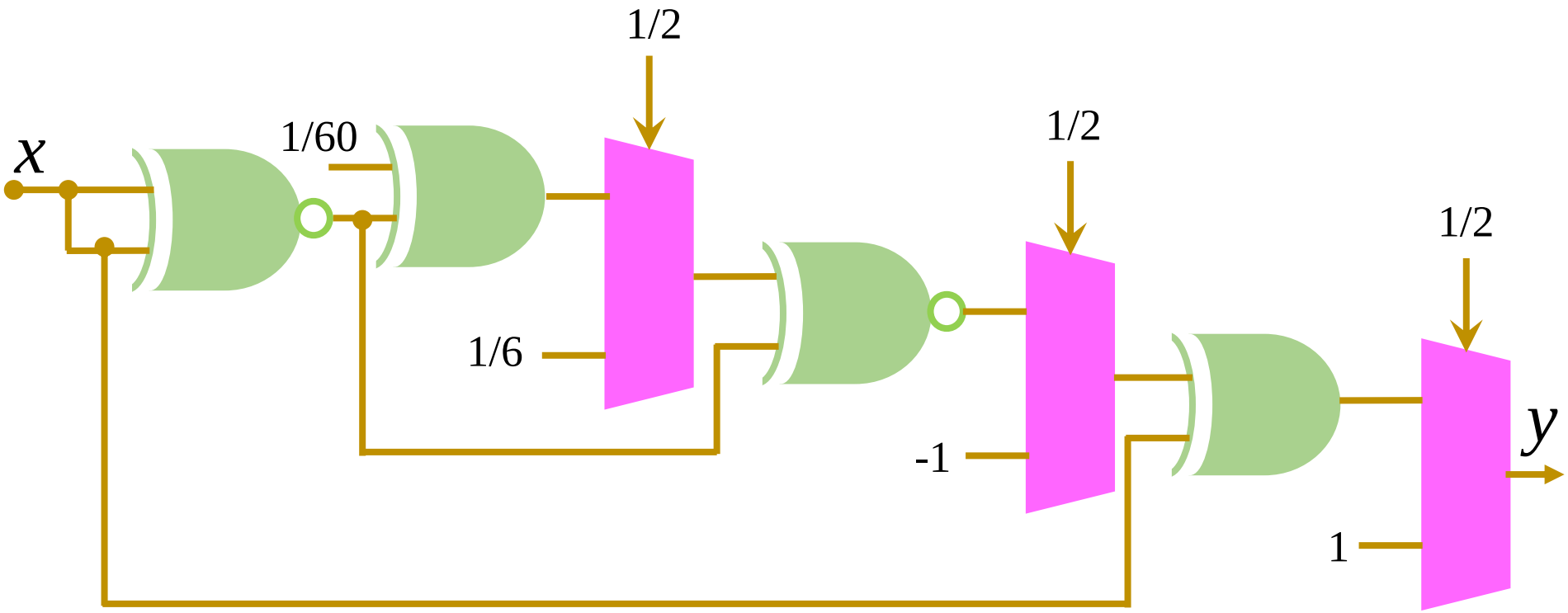
Sigmoid Function

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

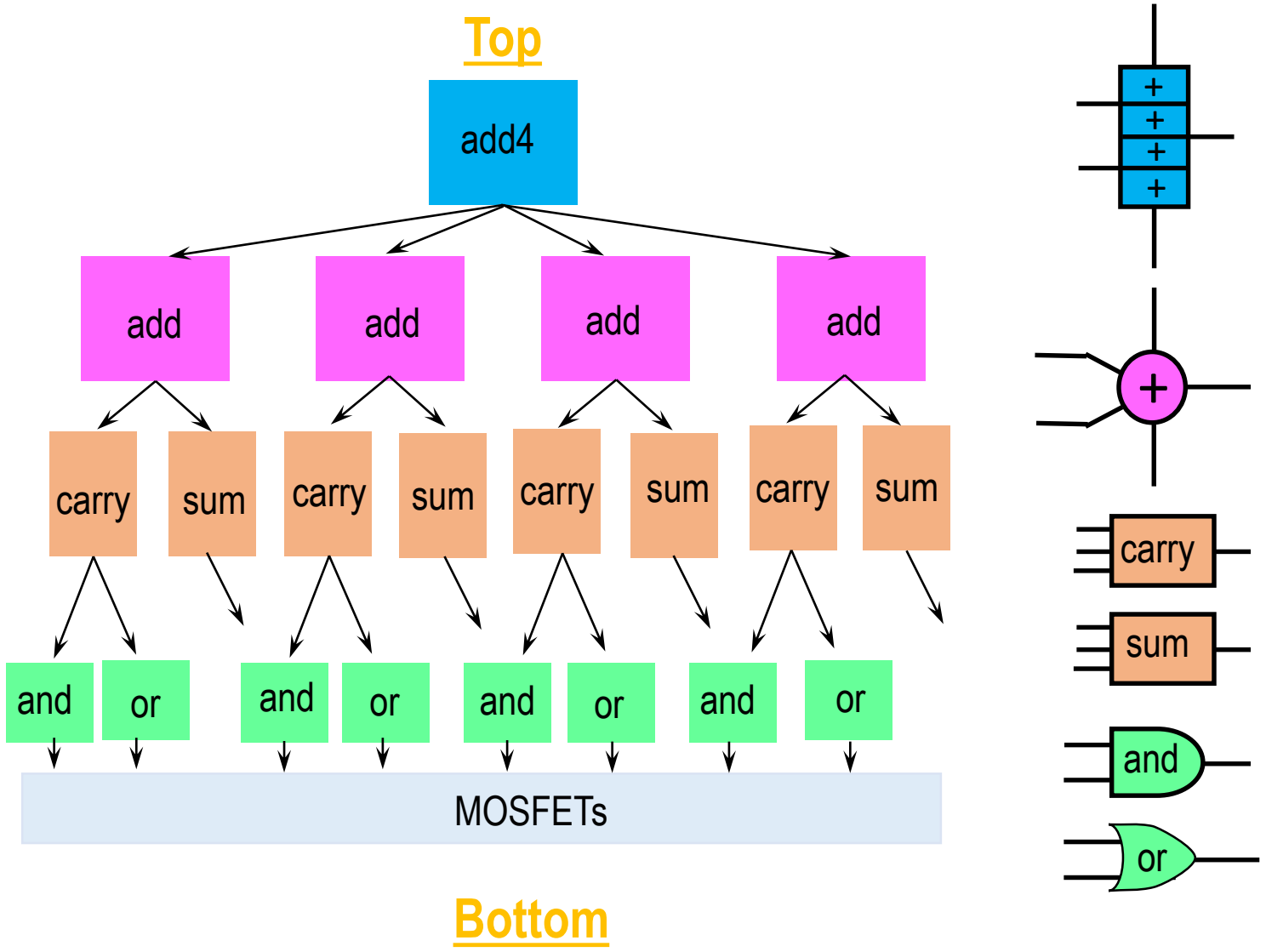
$$\approx \frac{1}{2} + \frac{x}{4} + \frac{x^3}{48} + \frac{x^5}{480}$$

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

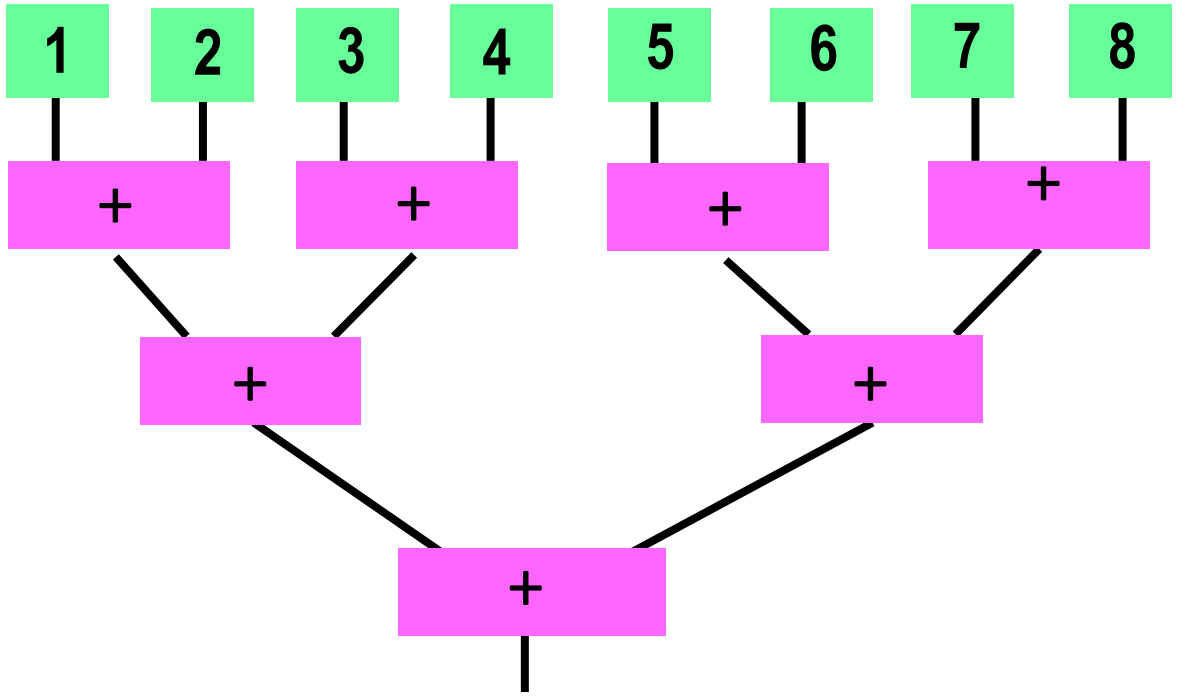
Sigmoid Unit



Architectural Decomposition of a 4-bit Adder

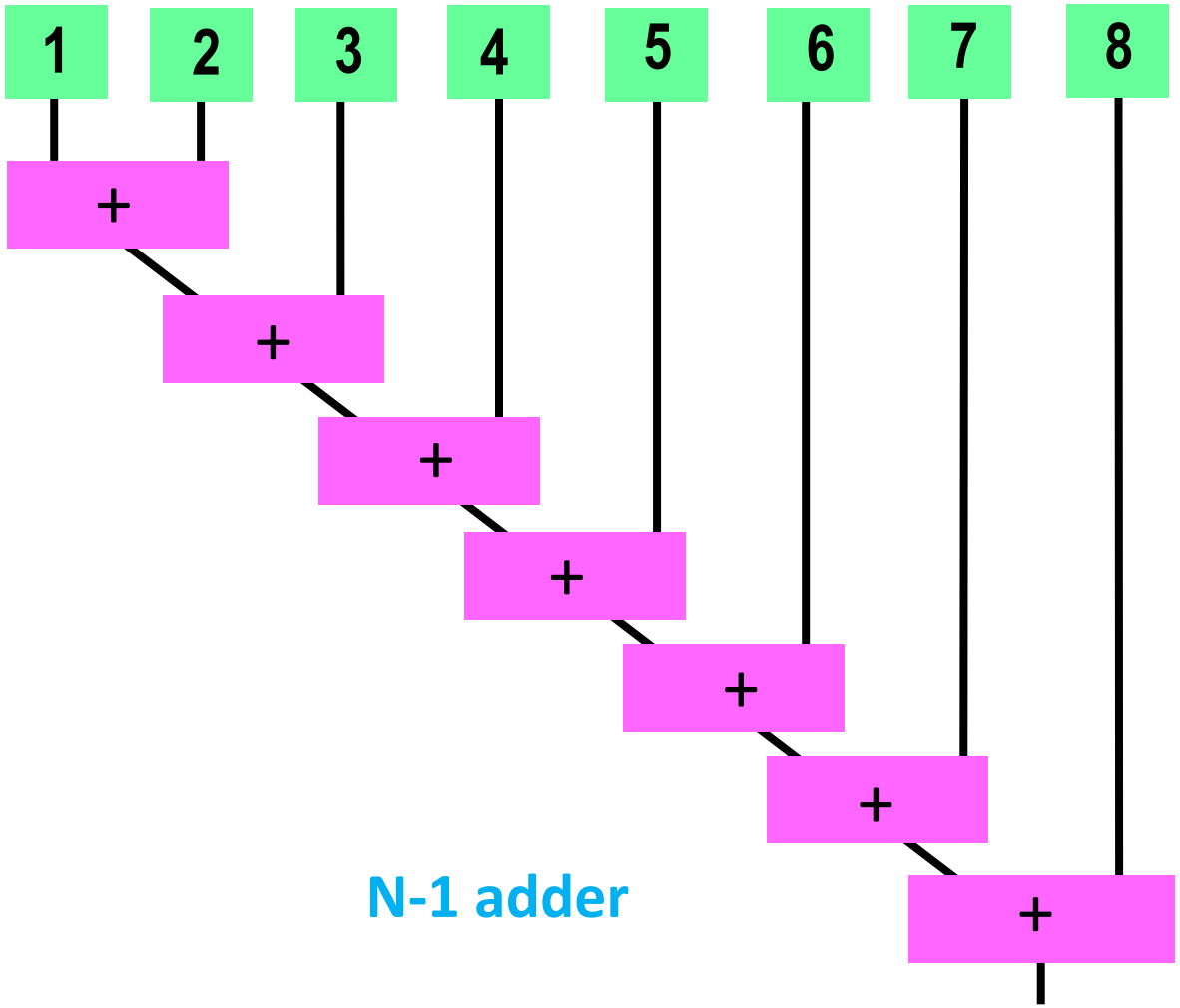


Summation $Z = \sum_{i=1}^{n=8} X(i)$



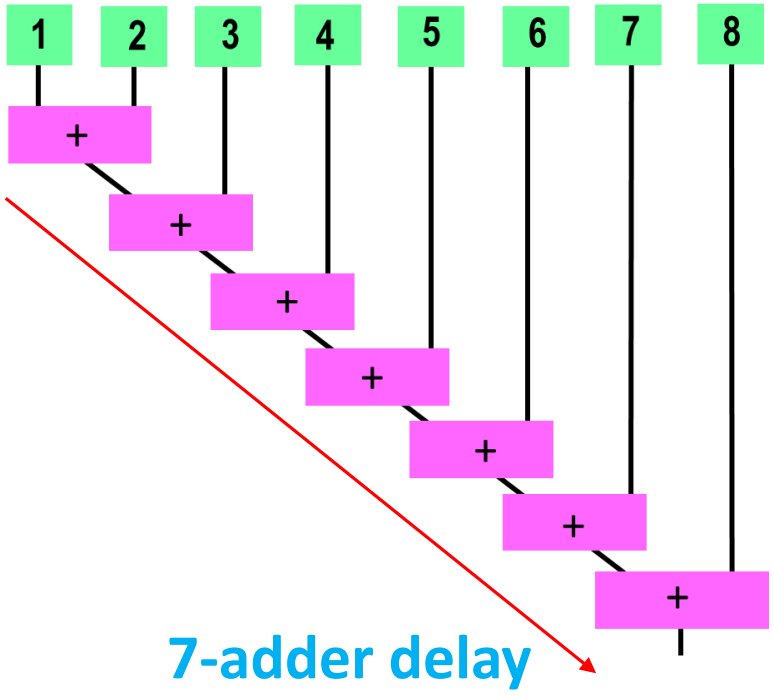
N-1 adder

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

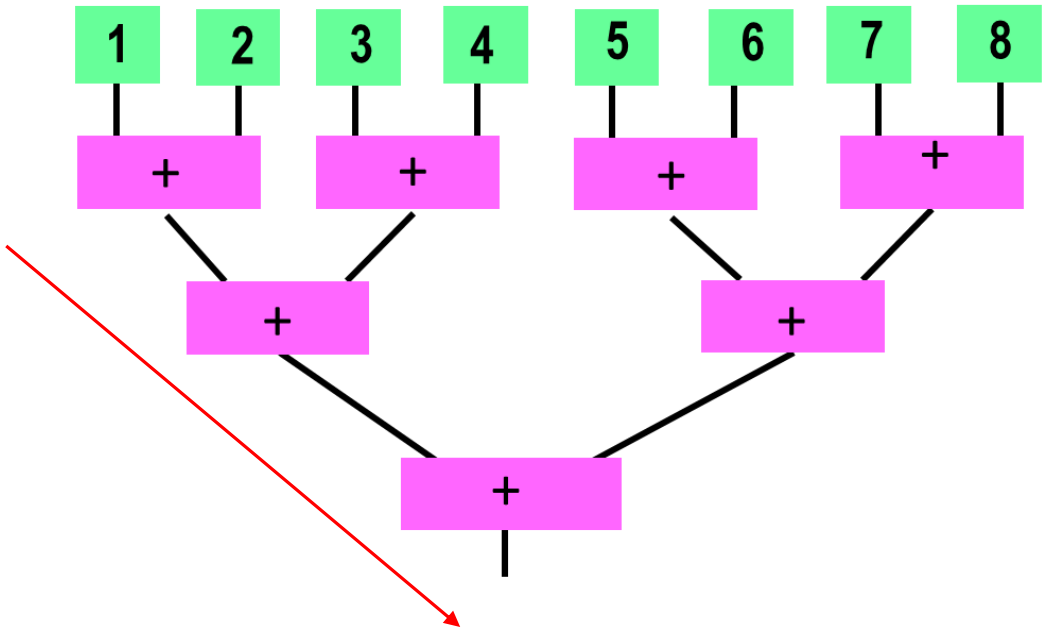


N-1 adder

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

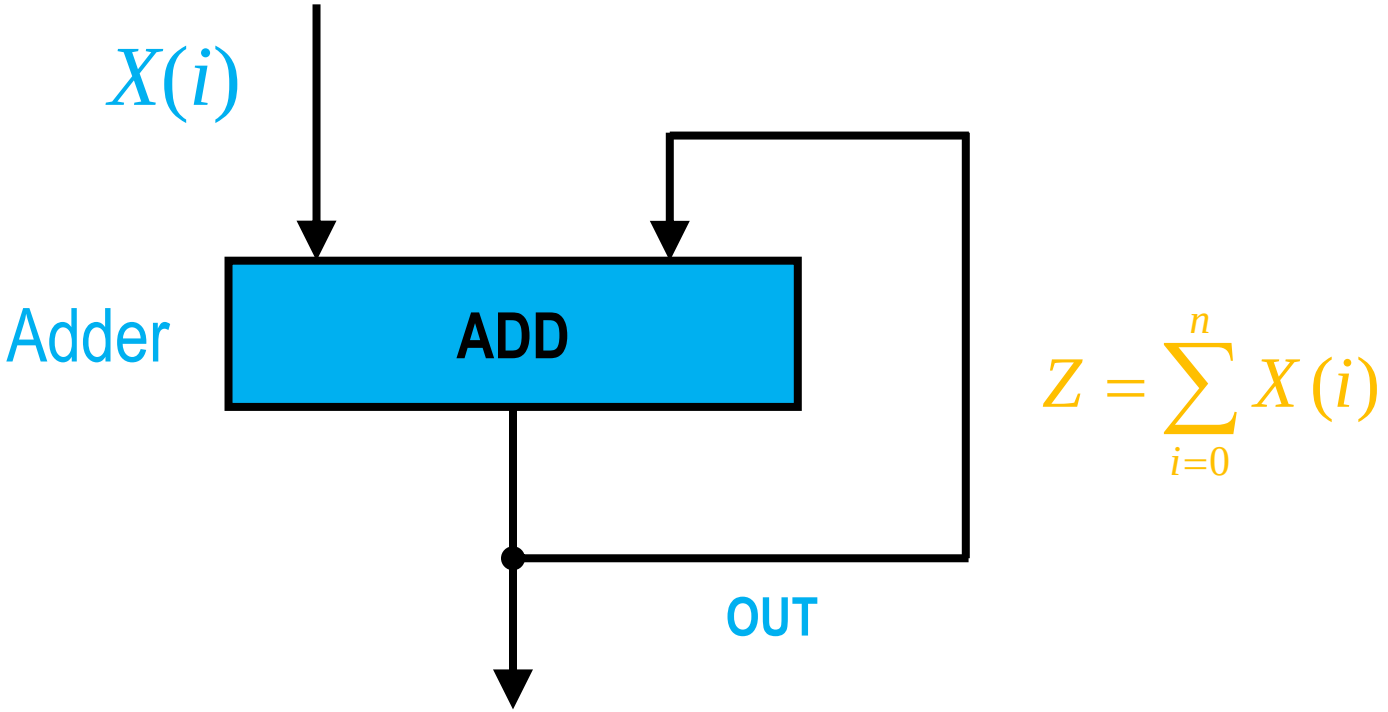


7-adder delay

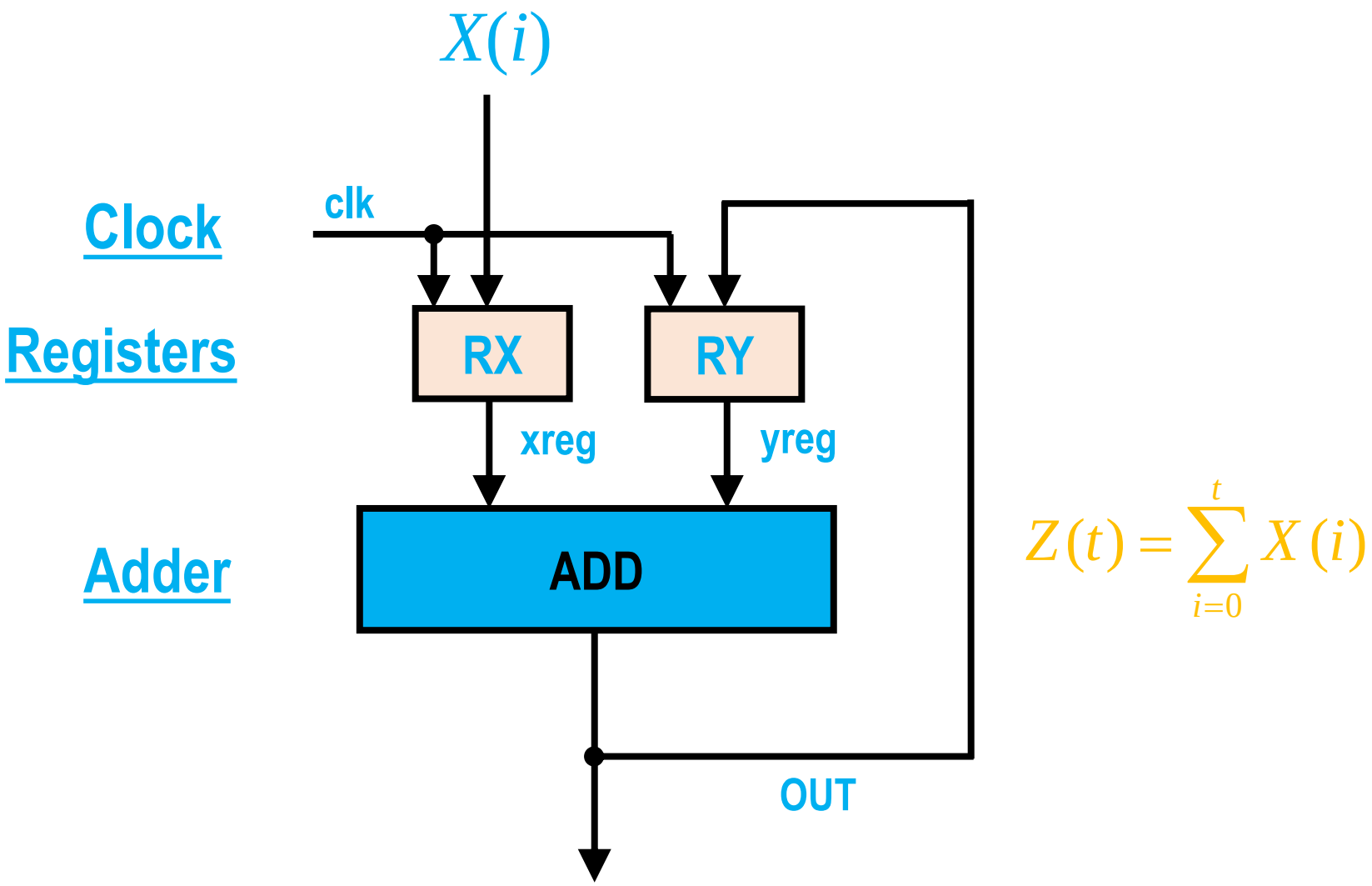


3-adder delay

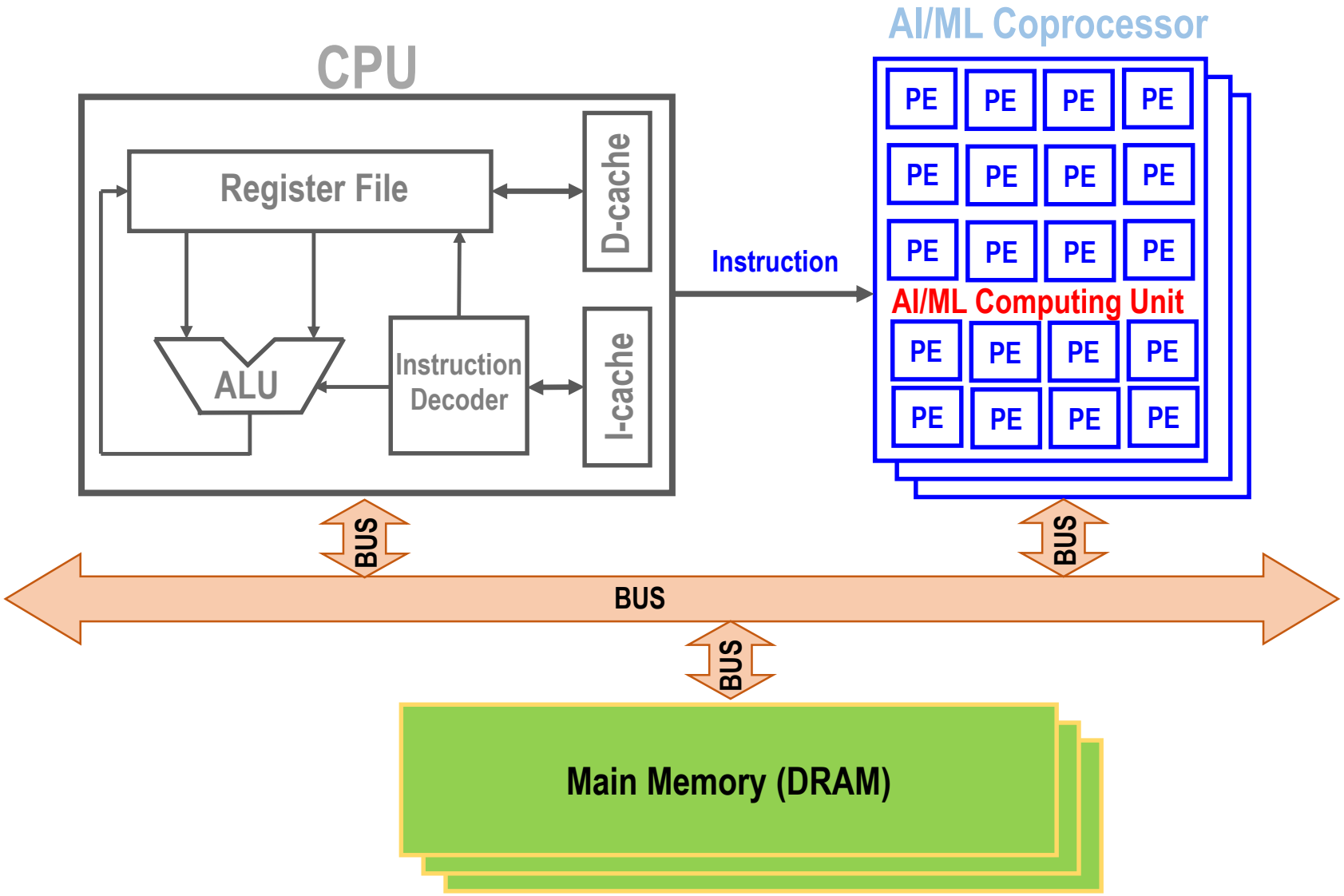
Architecture Computes $Z = \sum_{i=0}^{n} X(i)$



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



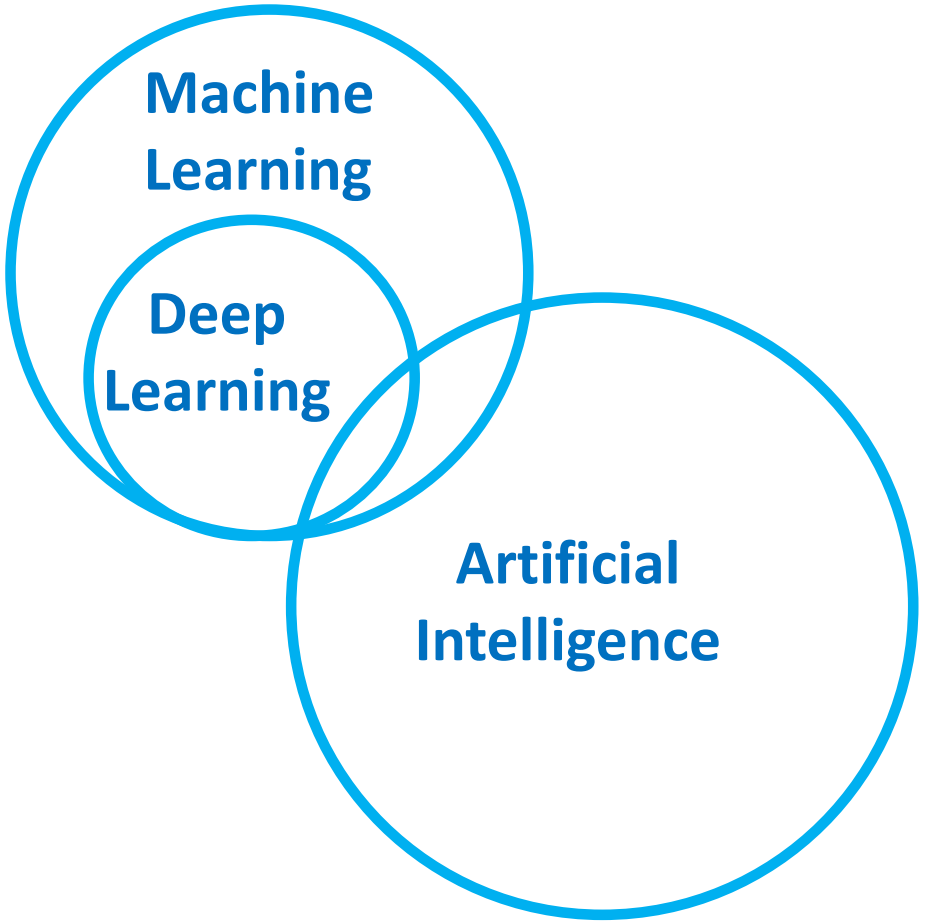
AI/ML Processor



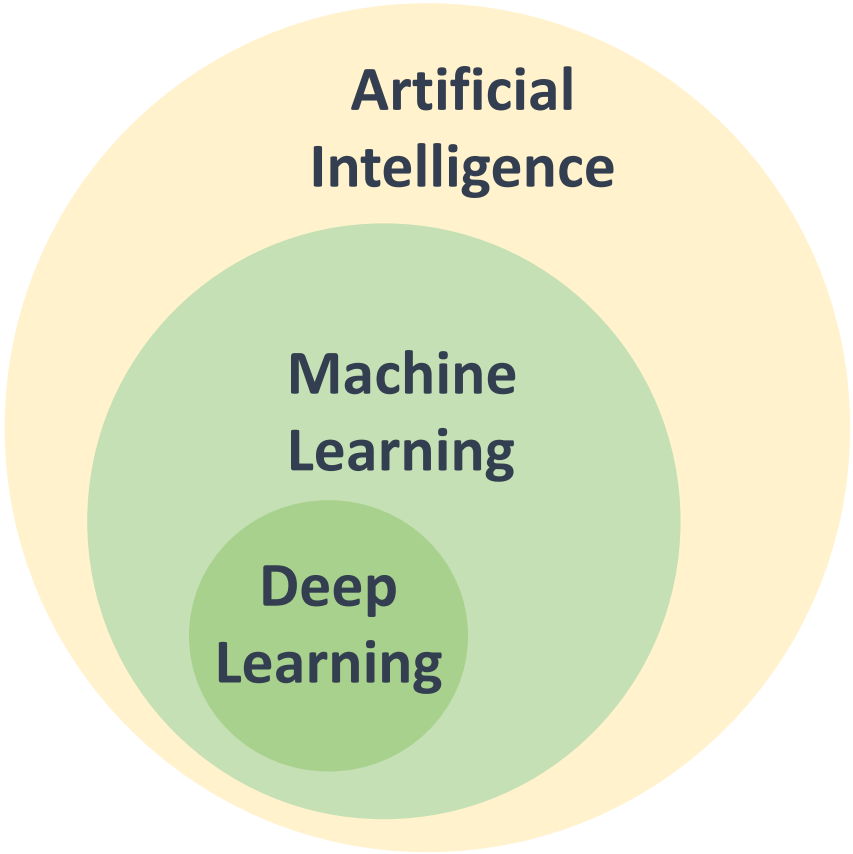
EL530-Introduction to Embedded Artificial Intelligence

Lecture-3

AI/ML/DL



Group 1



Group 2

Supervised Learning

Unsupervised Learning

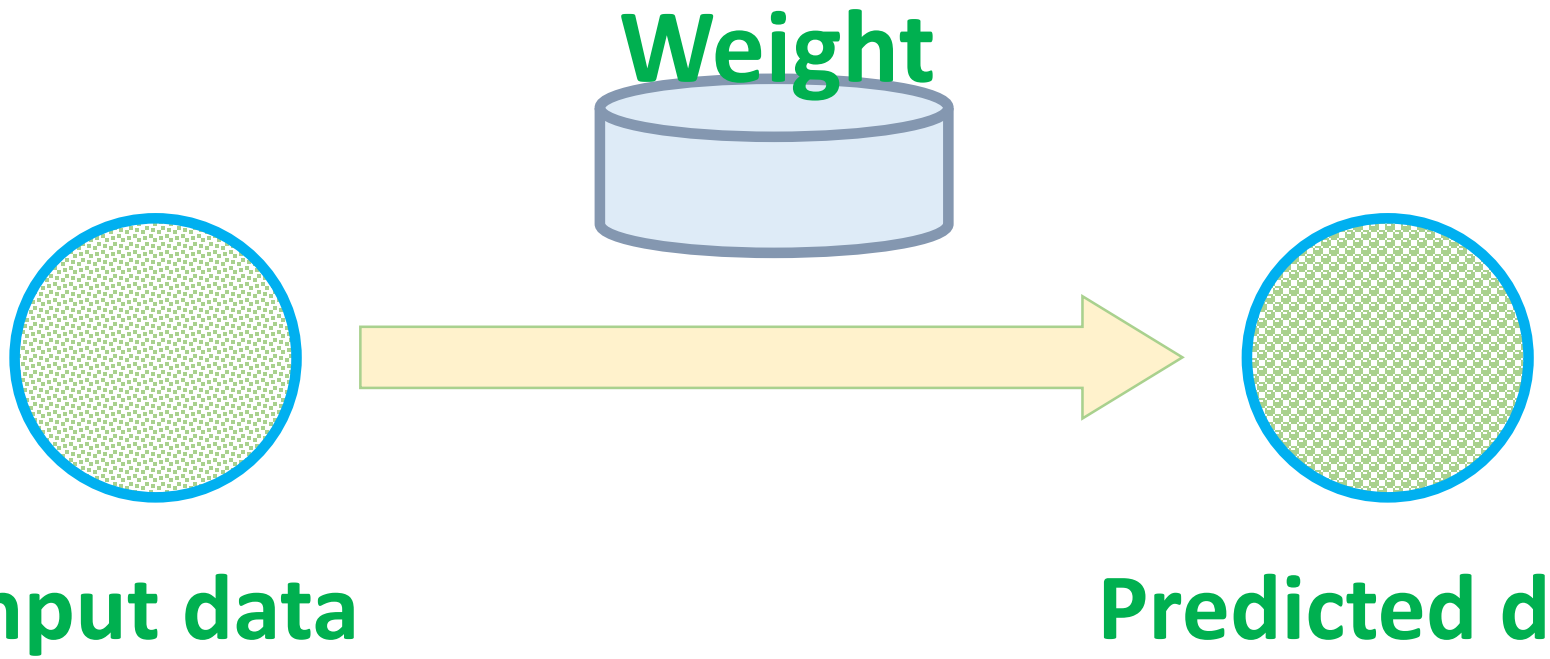
- *Man*
- *Dog*
- *Cat*
- *Tree*
- *Street*

2
1
3
2
1

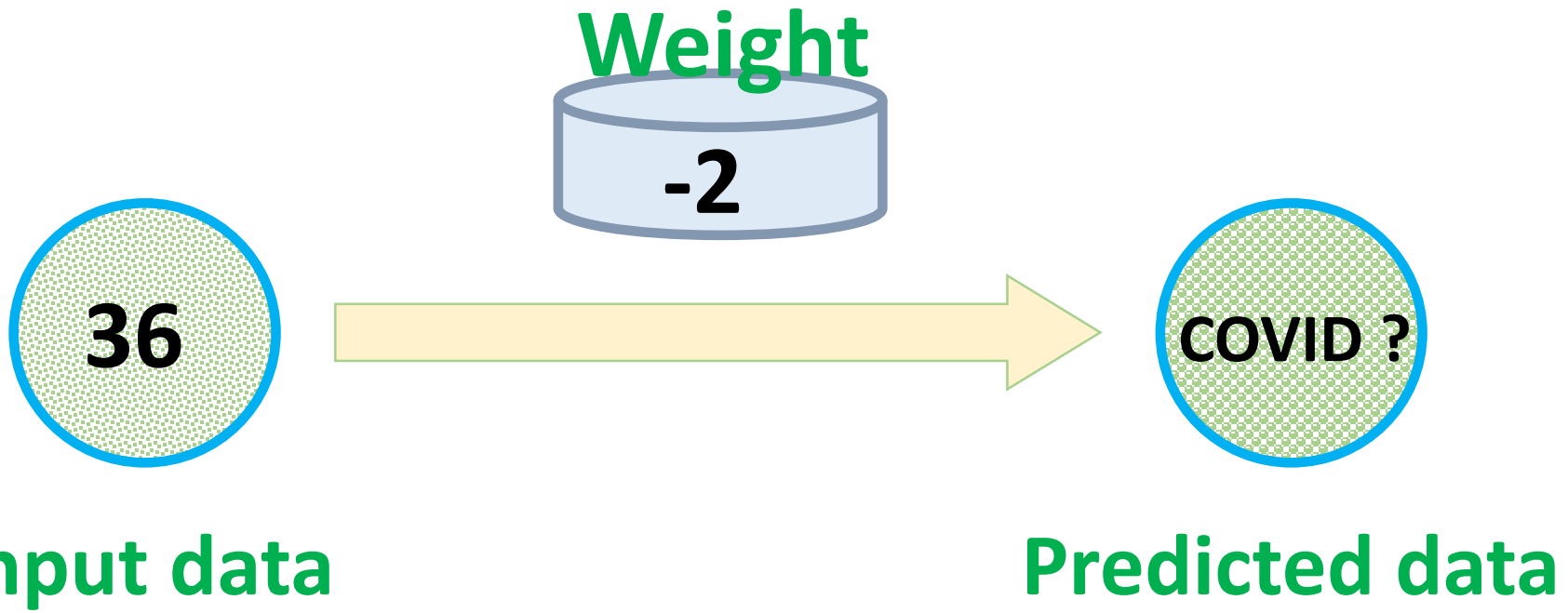
Example of ML on Edge Devices

- Smart home security
- Visitor tracking
- Drone avionics
- Plant disease detection

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



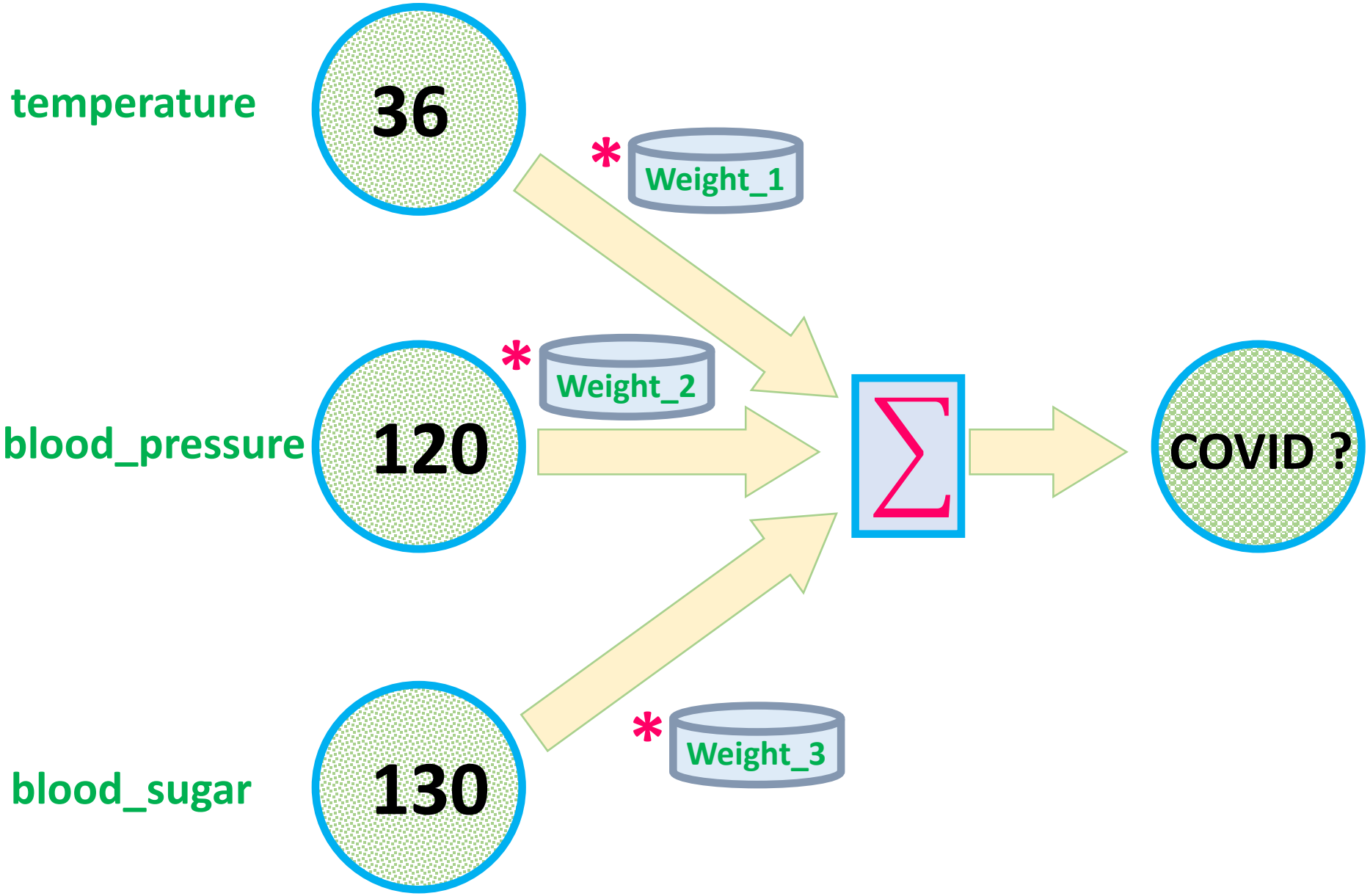
Simple Neural Network model

```
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

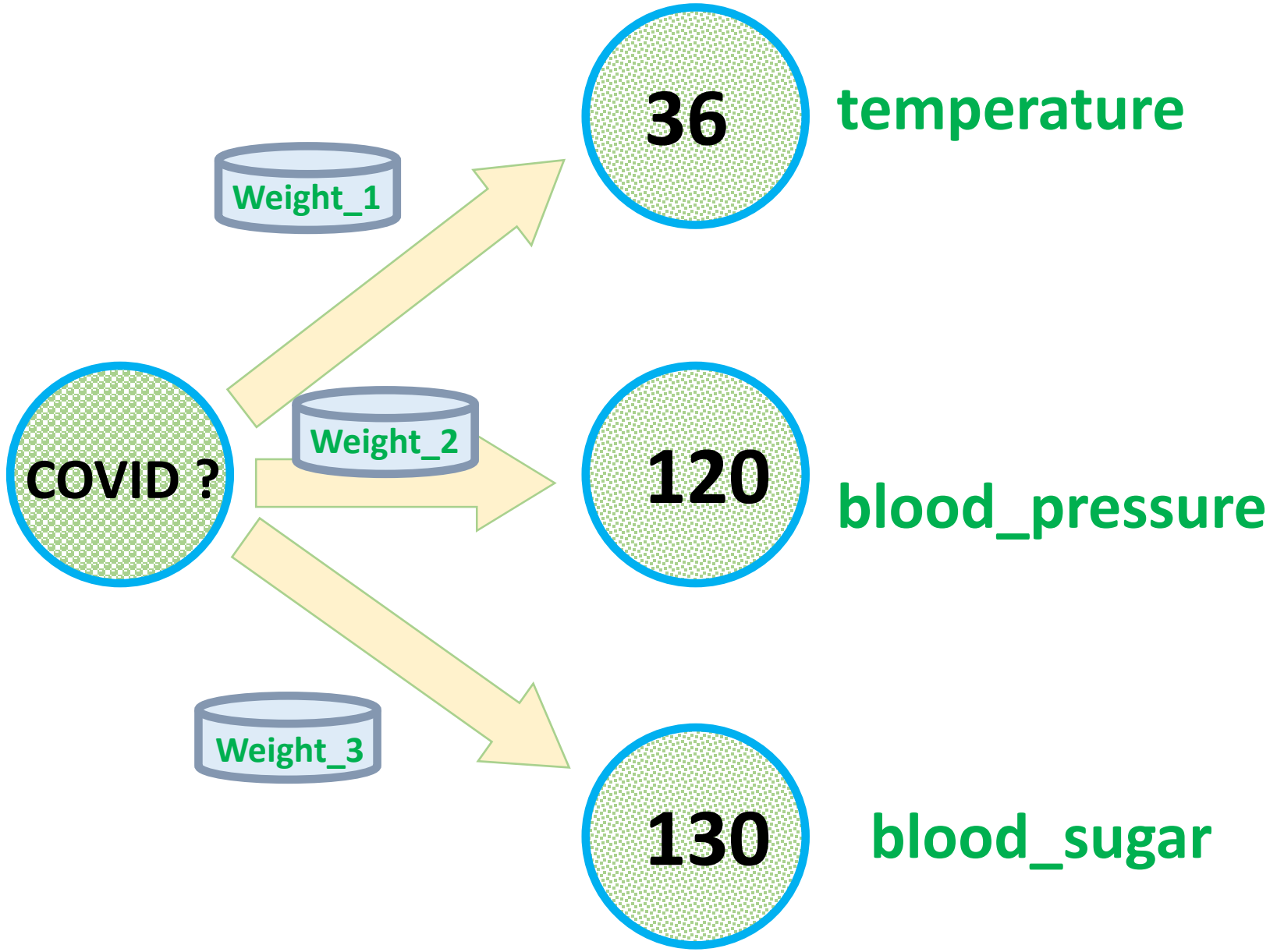
```
int weighted_sum(input, weight){
    for(int i=0;i<INPUT_LEN; i++)
            output += input[i]*weight[i];
             return output;  }
```

```
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};
blood_pressure []    =   {110, 120, 130, 140, 150, 160};
blood_sugar []       =   {130, 140, 150, 160, 170, 180};
```

Single input multiple outputs Neural Network



Neural Network model

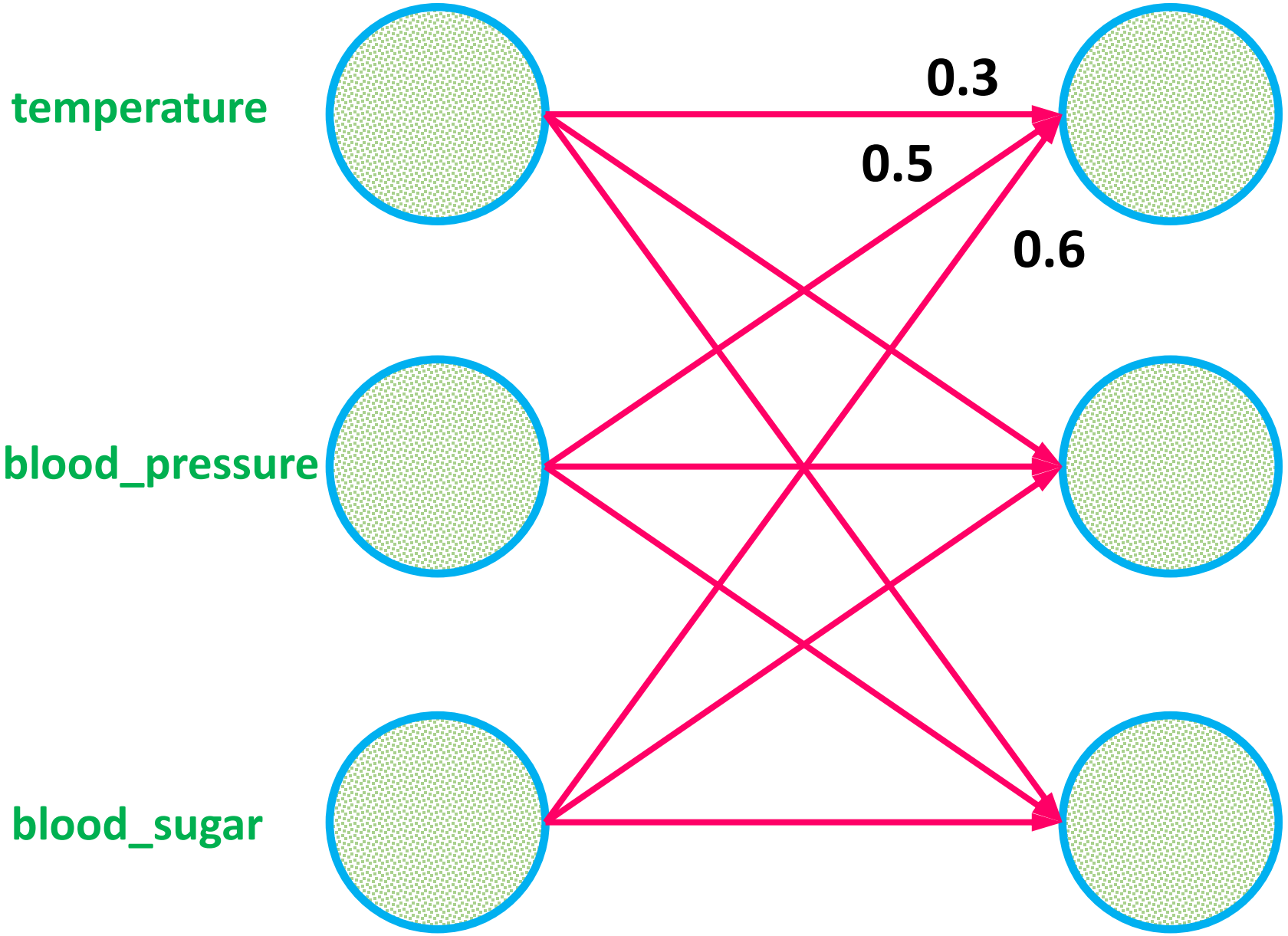
```
void elementwise_multiple( input, weight, output, LEN){
        for(int i =0; i<LEN; i++)
          output[i] =  input_scalar *weight_vector[i]
}
```

```
void single_input_multiple_output_nn(input, weight, output, LEN){
 elementwise_multiple(input, weight, output,  LEN);
}
```

Inputs

```
temperature prediction      =   covid * weight_1 ;
blood_pressure prediction   =   covid * weight_2 ;
blood_sugar prediction      =   covid * weight_3 ;
```

Multiple inputs Multiple outputs Neural Network



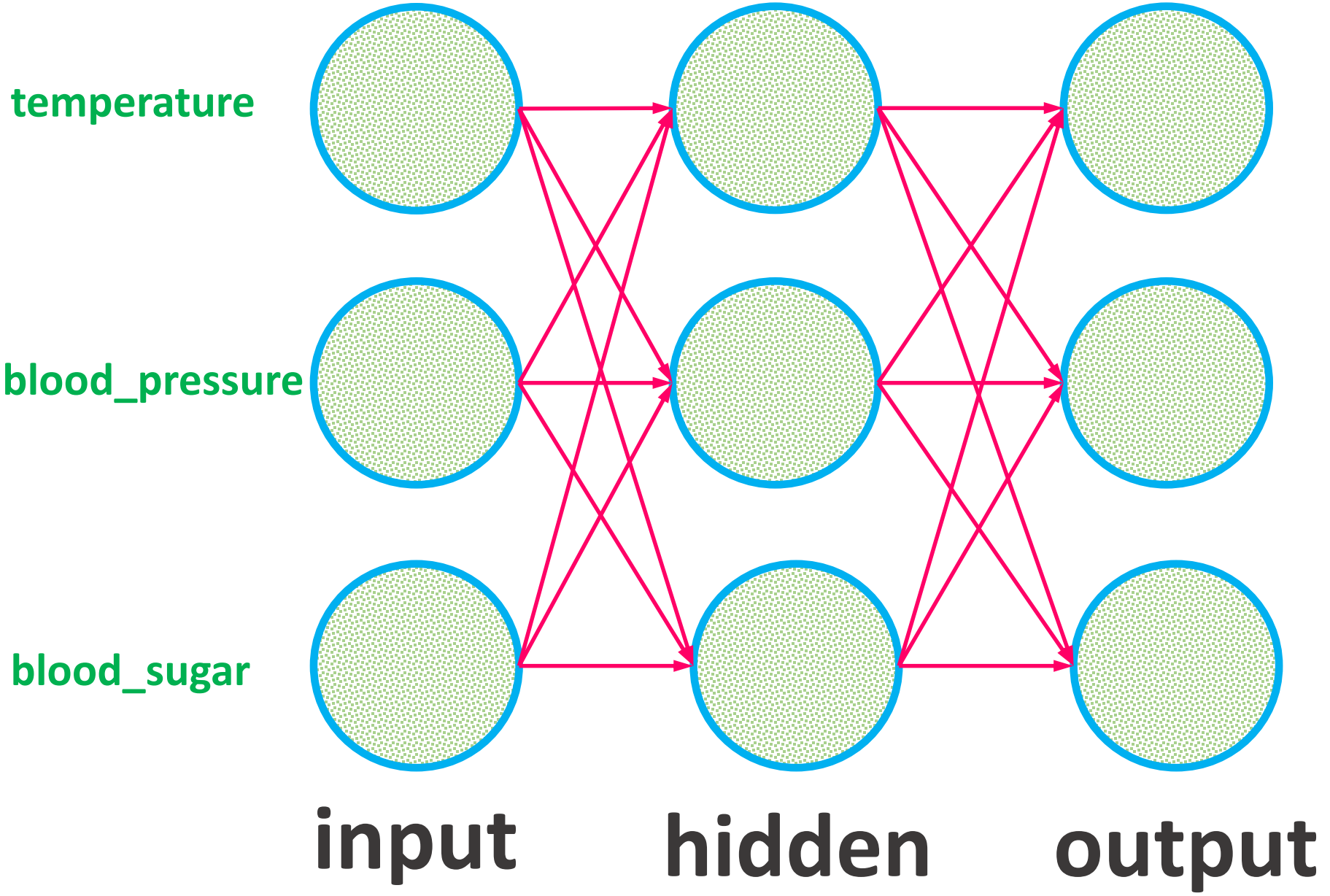
Neural Network model

```
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 suger
Weights [] [] = { {-2.0, 9.5, 2.01},      //covid?
                  {-0.8, 7.2, 6.3 },      //sick?
                  {-0.5, 0.45, 0.9}       // non-covid ?
```

Hidden Layer Neural Network



Metrics of Embedded AI

1

Low-Power

mW or below

2

High-Speed

mS or less
for single task

3

Memory Size

KB or less

1

Low-Power

mW or below

2

High-Speed

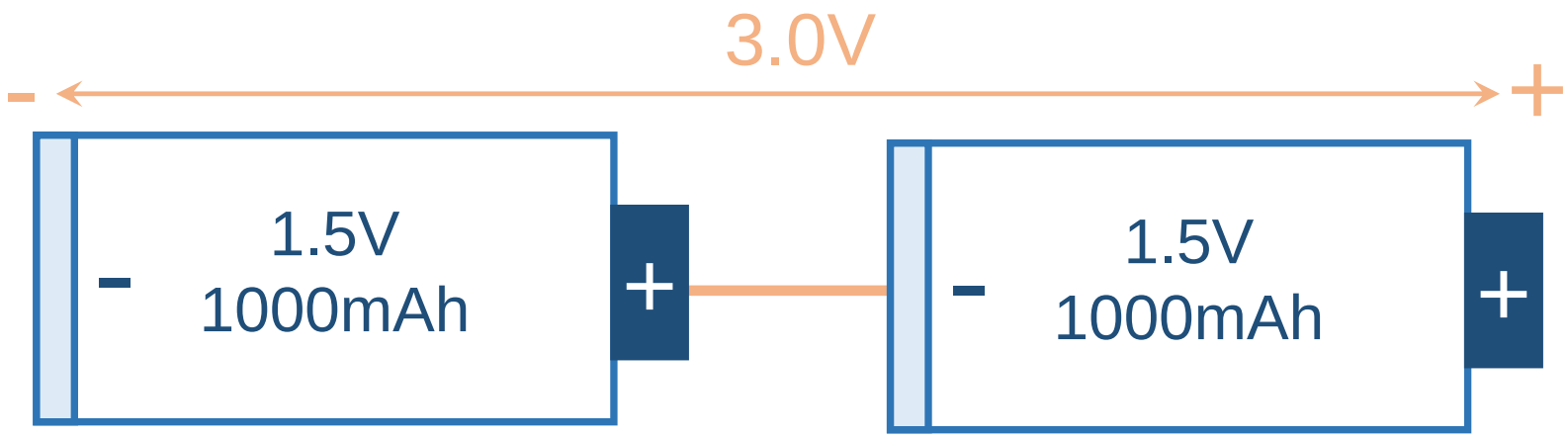
mS or less
for single task

3

Memory Size

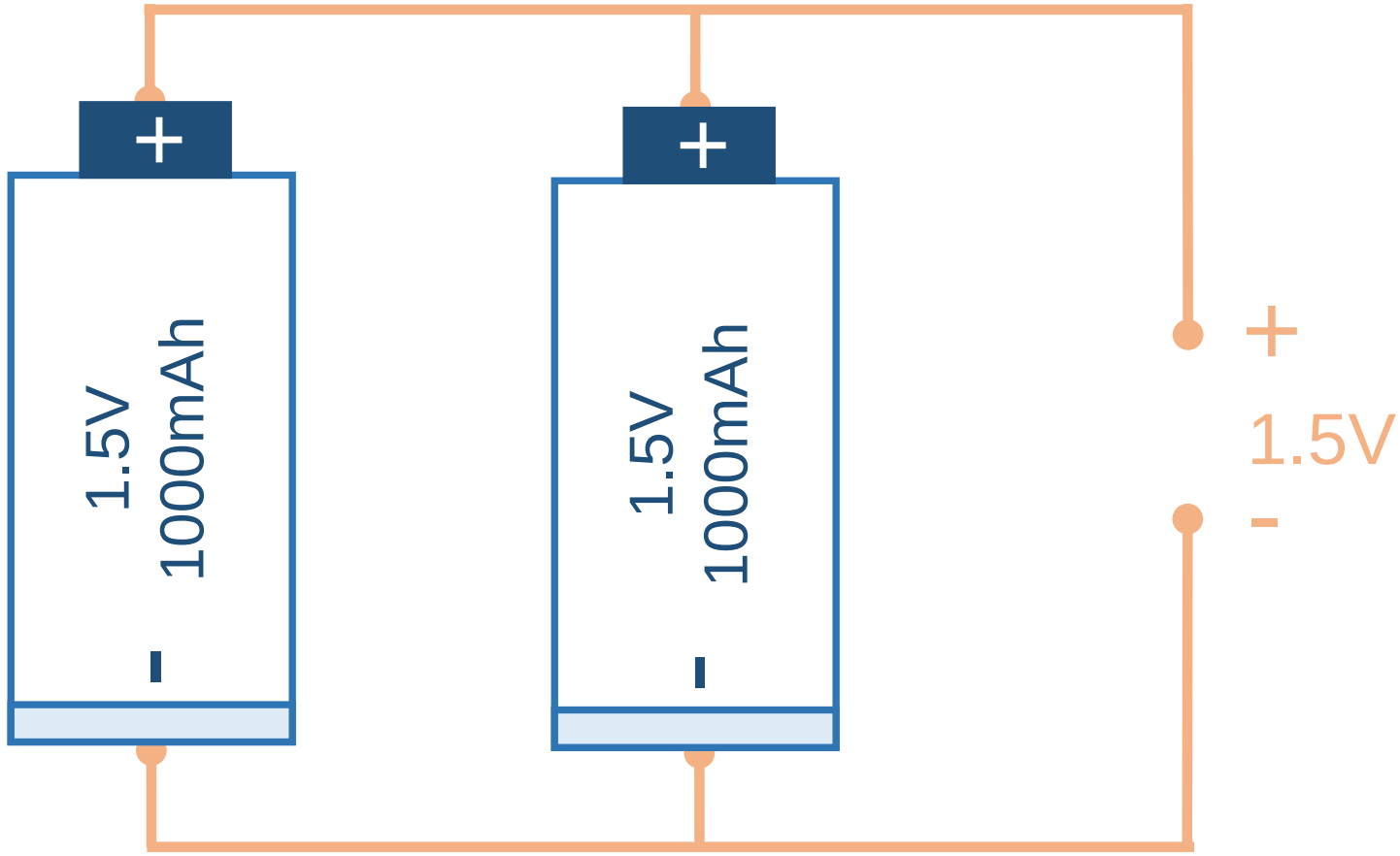
KB or less

Battery



$$V_{new} = V_{battery} \,.\, N$$

Increasing the output voltage by connecting batteries in series. Since one battery supplies 1.5*V* for 2400 mAh, connection of two batteries in series to produce 3.0*V* for the same energy capacity.



$$BC_{new} = BC_{battery} \cdot N$$

This approach will not increase the output voltage but just the battery capacity (BC). since one battery has a battery capacity of 2400 mAh, we could connect two batteries in parallel to increase the battery capacity by two times.

Battery Lifetime

$$B_L = \frac{B_C}{I_L}$$

Quantity	Unit	Meaning
B_L	Hours (h)	Battery life
B_C	mAh	Battery capacity
I_L	mA	Load current consumption in microcontroller

Power and Energy

- Energy is the capacity for doing work (for example, using force to move an object)

- Power is the rate of consuming energy. In practical terms, **power tells us how fast we drain the battery**, so high power implies a faster battery discharge.

$$P = V \cdot I$$

$$E = P \cdot T$$

Physical Quantity	Unit	Meaning
P	Watt (W)	Power
E	Joule (J)	Energy
V	Volt (V)	Voltage supply
I	Ampere (A)	Current consumption
T	Second (s)	Operating time

Physical quantities in the power and energy formulas

- On microcontrollers, the voltage supply is in the order of a few volts **(for example, 3.3 V)**
- Current consumption is in the range of **micro-ampere (μA)** or **milli-ampere (mA)**
- For this reason, we commonly adopt
  - **microwatt (μW) or milliwatt (mW) for power**
  - **microjoule (μJ) or millijoule (mJ) for energy**

Example

Suppose you have a processing task and you have the option to execute it on two different processors **PU1** and **PU2**. These processors have the power consumptions **12mW** and **3mW**, respectively. What processor would you use to execute the task?

- Although PU1 has **higher (4x)** power consumption than PU2, this does not imply that PU1 is less energy-efficient.
- On the contrary, PU1 could be more computationally efficient than PU2 (**for example, 8x**).
- Making it the best choice from an energy perspective, as shown in the following formulas:

$$E_{PU1} = 12 \cdot T_1$$

$$E_{PU2} = 3 \cdot T_2 = 3 \cdot 8 \cdot T_1 = 24 \cdot T_1$$

- Here, we can say that PU1 is our better choice because it requires less energy from the battery under the same workload.
- Commonly, we adopt **OPS per Watt (arithmetic operations performed per Watt)** to bind the power consumption to the computational resources of our processors.

1

Low-Power

mW or below

2

High-Speed

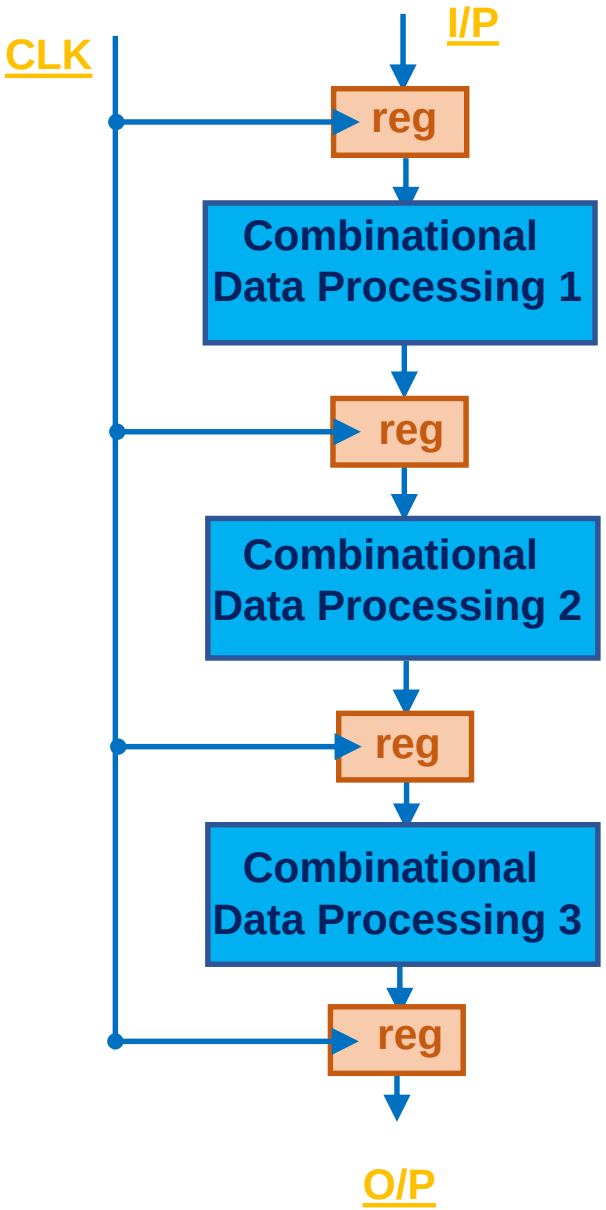
mS or less
for single task

3

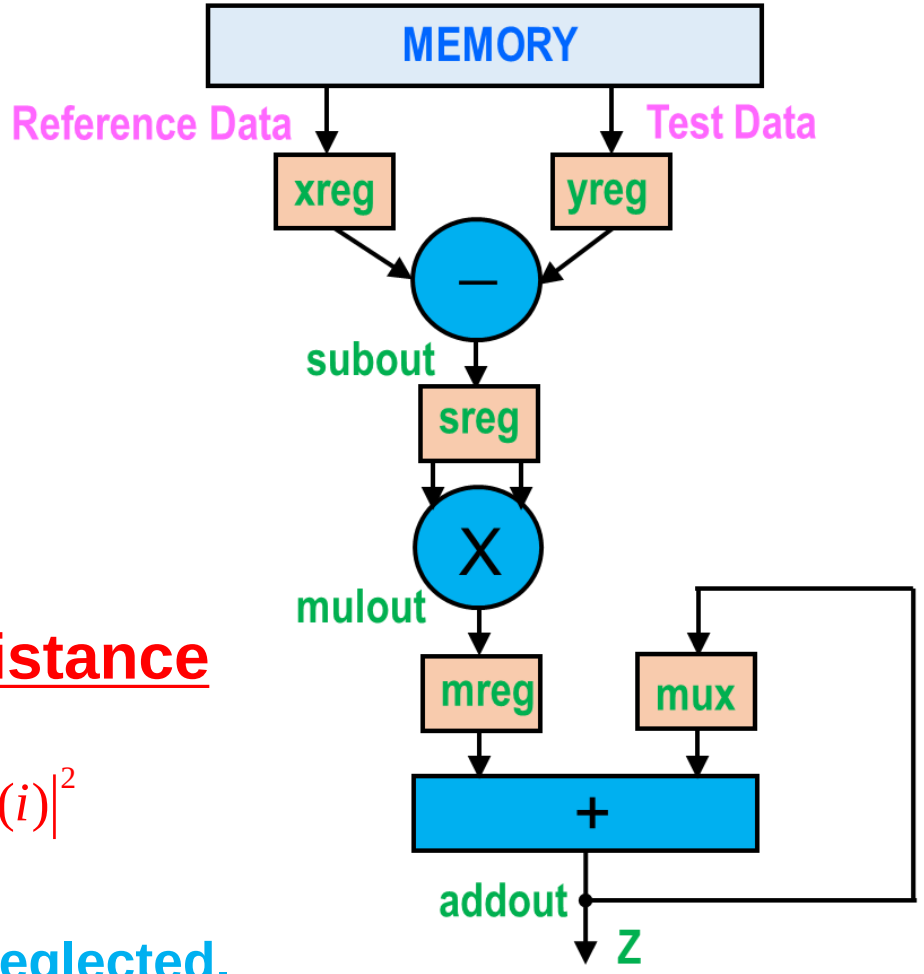
Memory Size

KB or less

Sequential Data Processing



Euclidean Distance Search

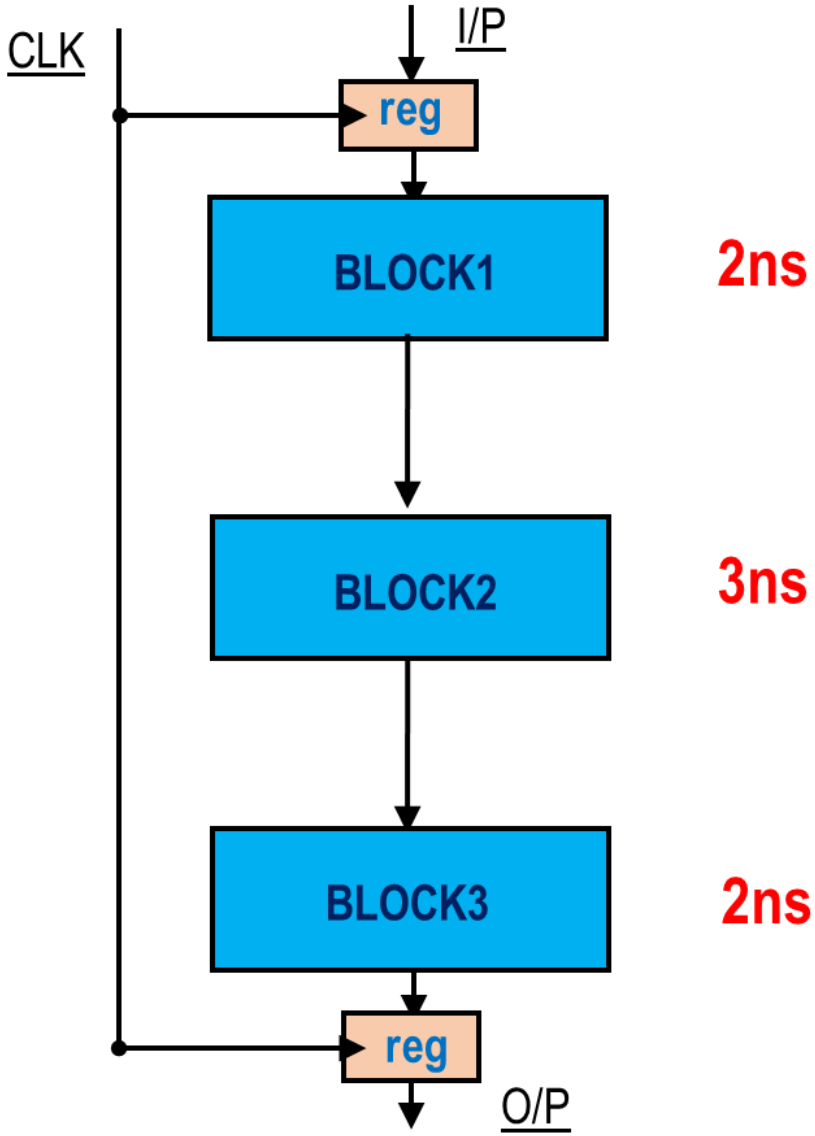


Euclidean Distance

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

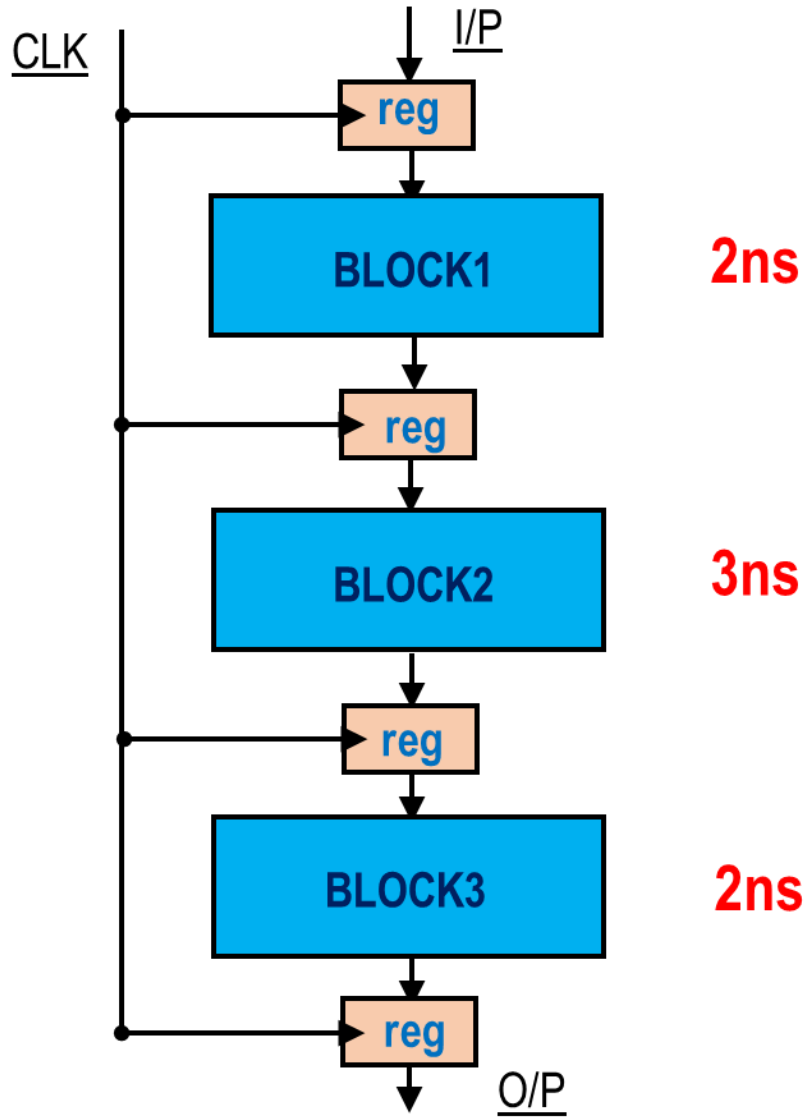
Square root is neglected.

Block Level Representation



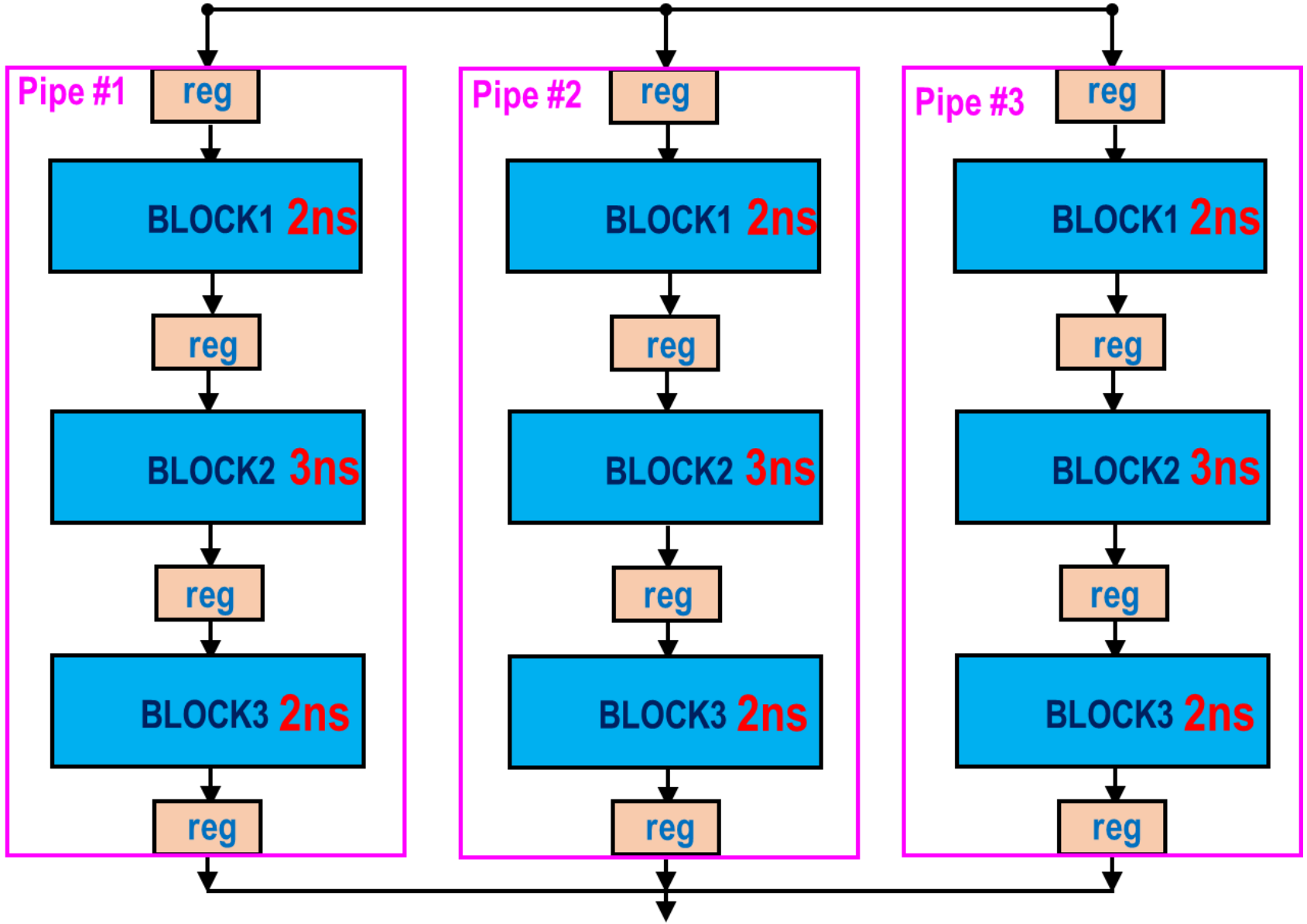
- Total: 7 ns
- Throughput: 1 output / 7ns
- Latency: 7ns

Pipelining

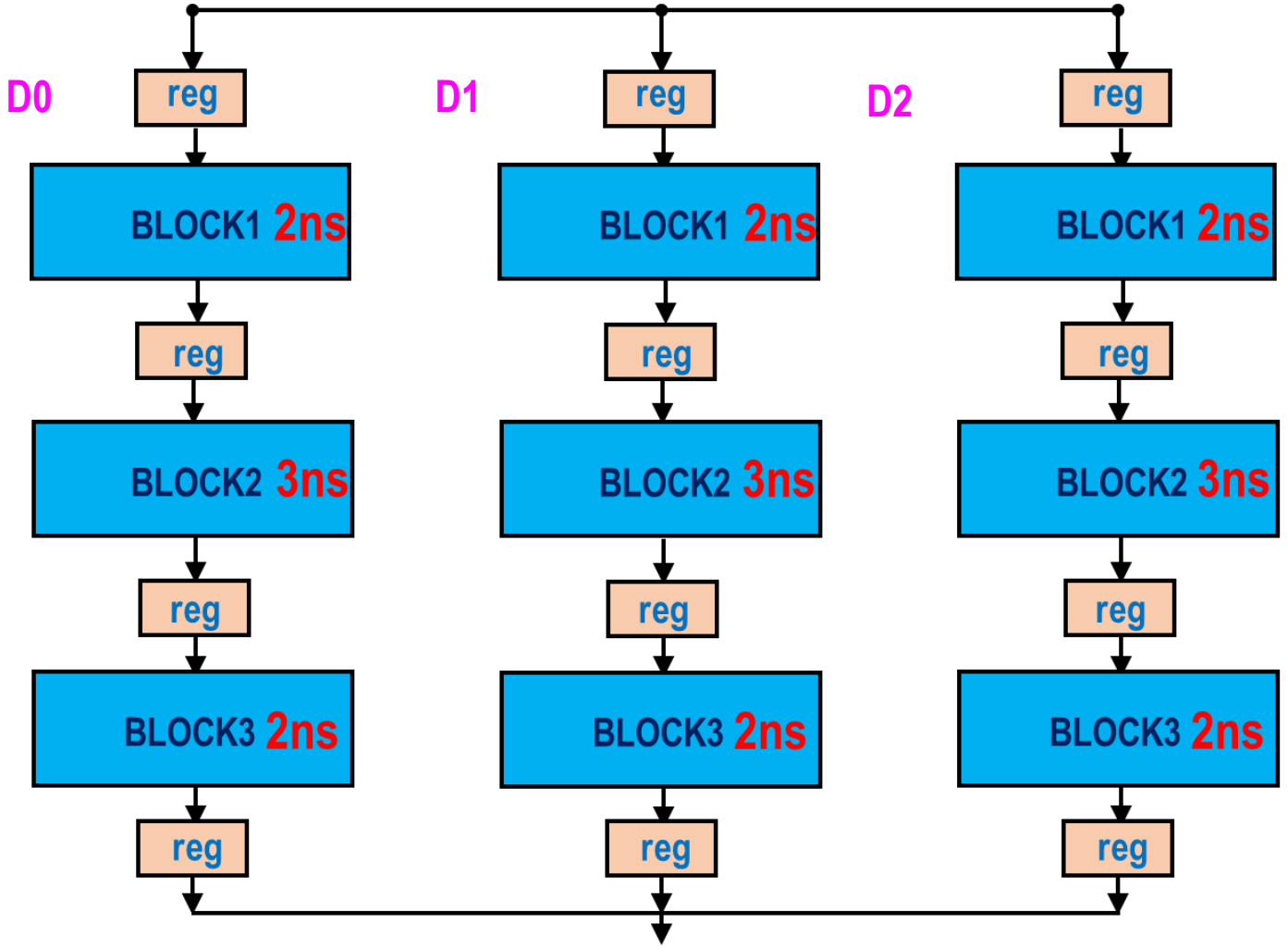


- Throughput: 1 output / 3ns
- Latency: 9ns

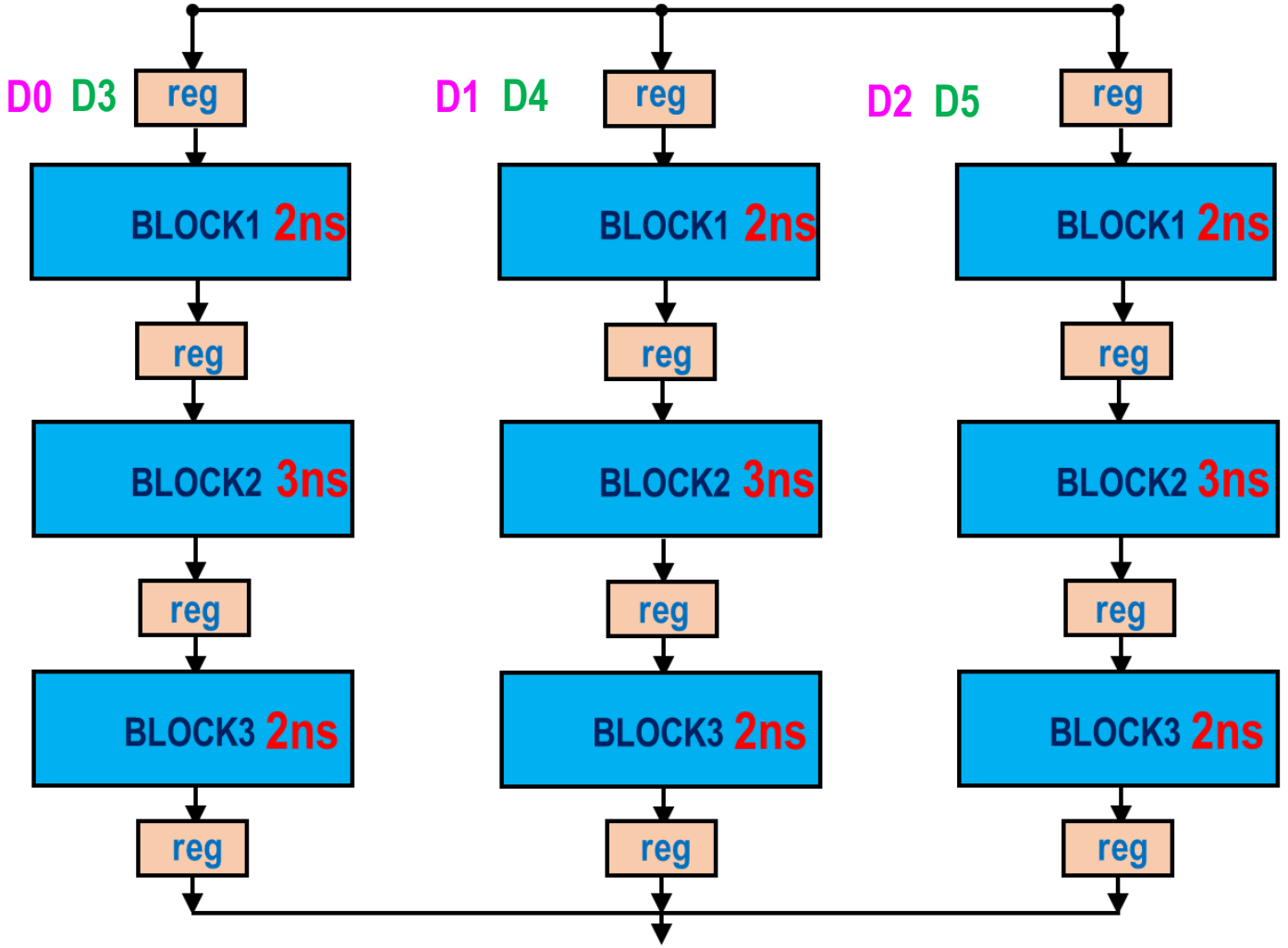
Parallel and Pipelining



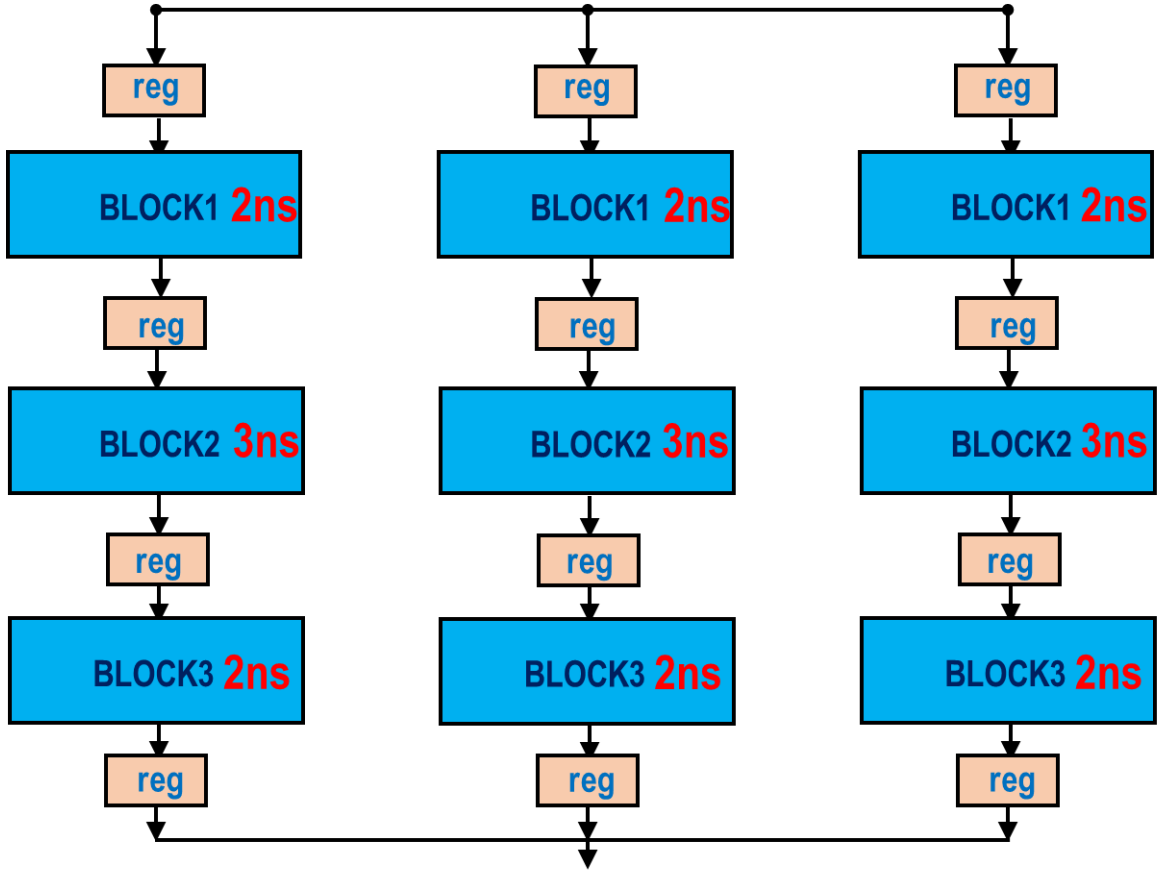
Parallel and Pipelining



Parallel and Pipelining



Parallel and Pipelining



Utilization

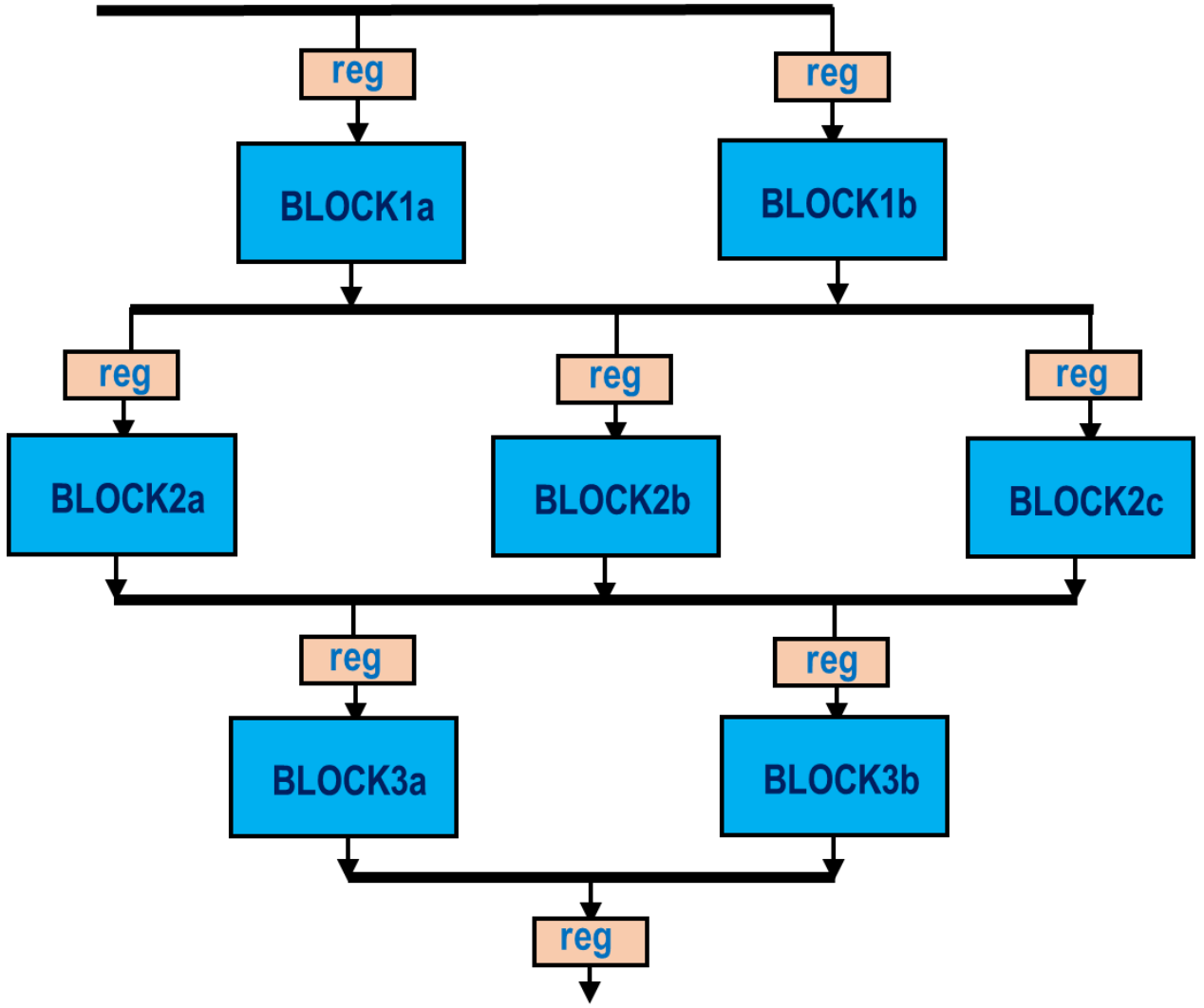
66.67%

100%

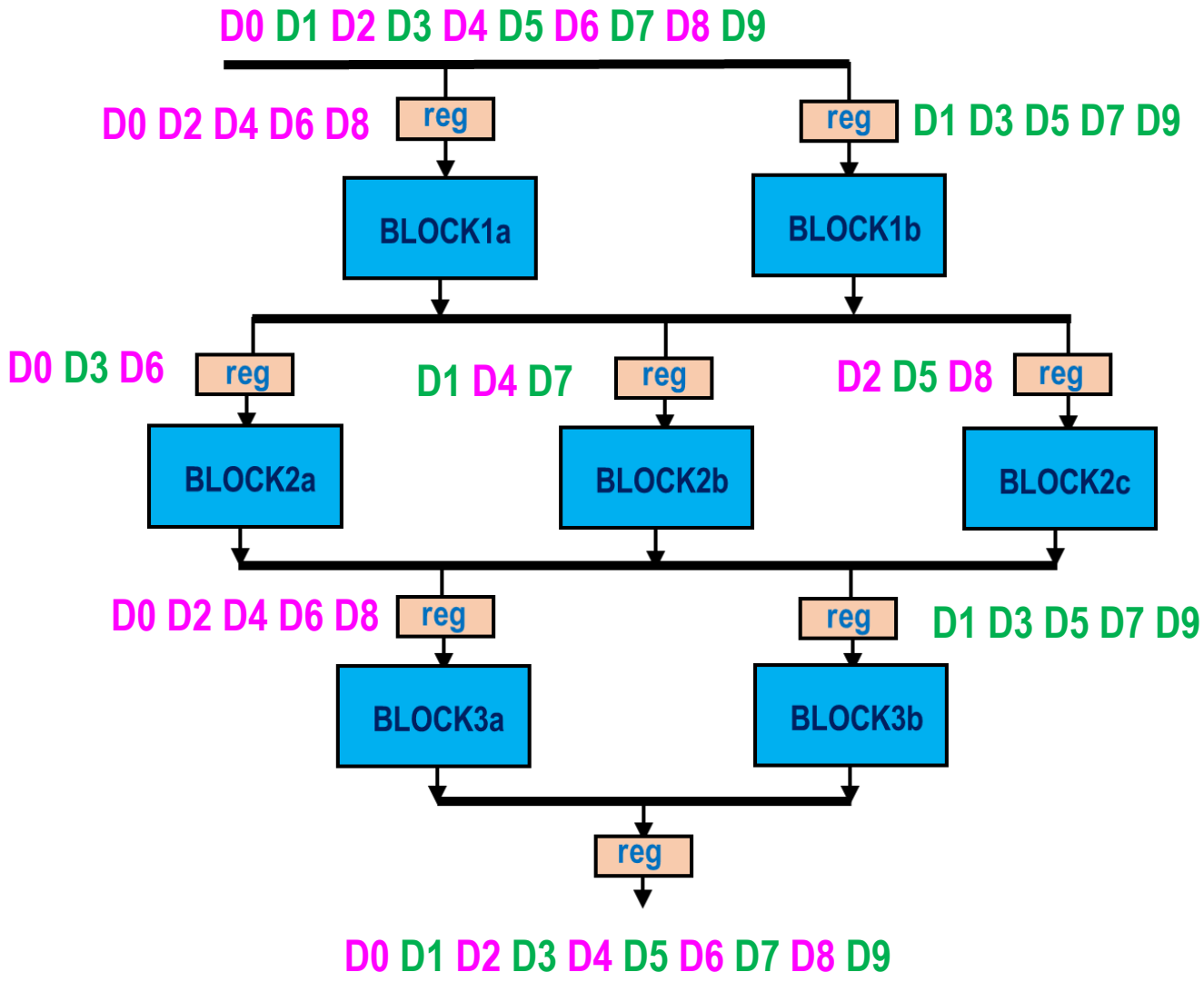
66.67%

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

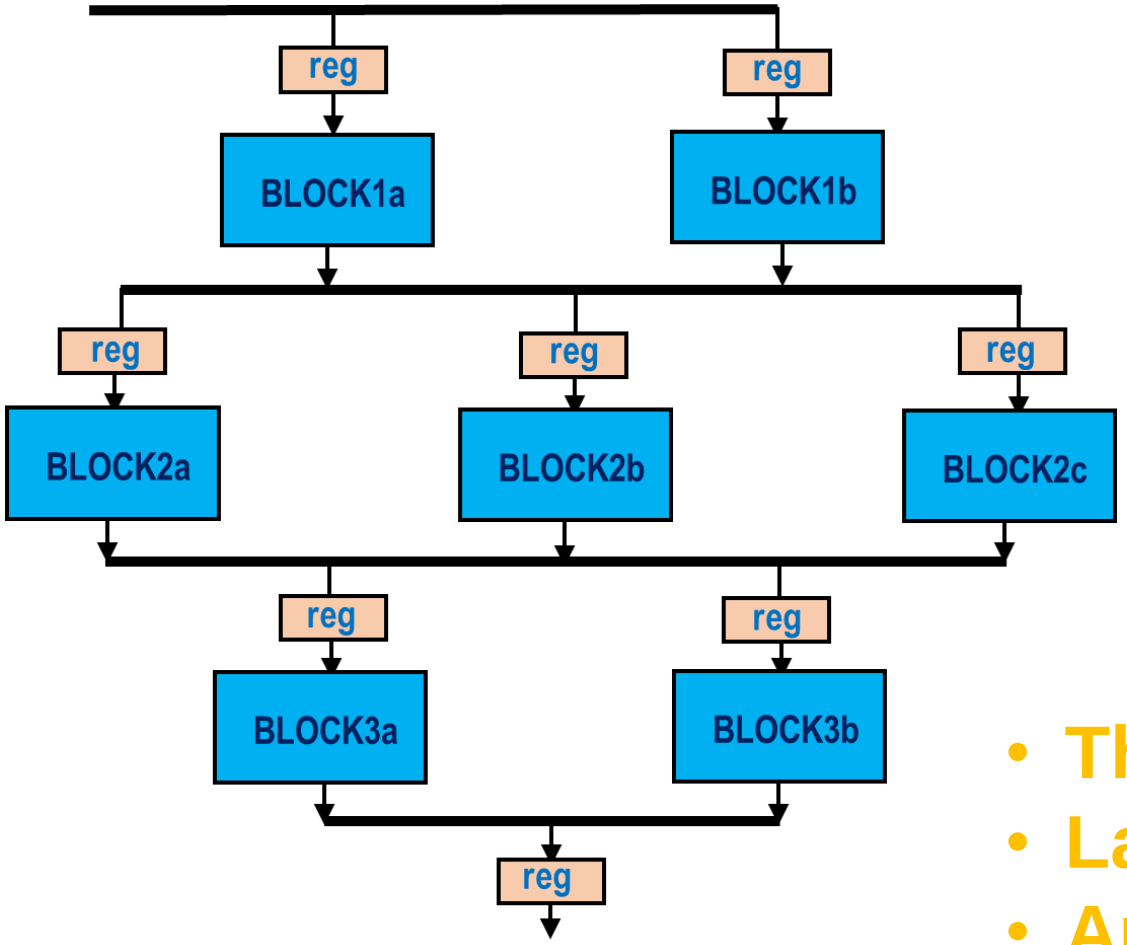
Combining Parallel and Pipelined Structures



Combining Parallel and Pipelined Structures



Combining Parallel and Pipelined Structures



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

Naive Bayes, Part 1

What is Naive Bayes?

Naive Bayes is a probabilistic machine learning algorithm. It is based on Bayes' theorem. This algorithm makes an assumption that all features are independent of each other. In other words, changing the value of one feature, does not directly change the value of any of the other features. This assumption is naive because it is (almost) never true.

For example, if you have temperature and humidity as input features, Naive Bayes assumes that temperature and humidity are independent of each other. So, changing the value of temperature, does not directly change the value of humidity. However, in reality, temperature and humidity are related to each other.

Conditional Probability

Before you go into Naive Bayes, you need to understand what Conditional Probability is. Let's start with an example.

When you toss a fair coin, it has a probability of 1/2 of getting heads or tails. Mathematically, it is written as $P(Head) = \frac{1}{2}$ and $P(Tail) = \frac{1}{2}$.

Another example, suppose you pick a card from the deck and you already know that your card is an ace. What is the probability of getting a diamond given the card is an ace? Well, we have already a condition that the card is an ace. So, the population (denominator) is 4 not 52. There is only one diamond in aces. So, the probability of getting a diamond given the card is an ace is 1/4. Mathematically, it is written as $P(Diamond|Ace) = \frac{1}{4}$. This is called conditional probability.

Naive Bayes by Example 1

Let's start with a simple example of Naive Bayes algorithm. Suppose you have 100 fruits which could be either apple or orange. Your training data of these fruits is shown in the following table. In this training data, we have one feature which is color that has three possible values: red, green, and orange. Then, we have one label that has two possible values: apple and orange. We denote color as X and fruit as y.

No.	Color	Fruit
1	Red	Apple
2	Red	Apple
3	Green	Apple

No.	Color	Fruit
4	Green	Orange
5	Orange	Orange
...	...	...
100	Green	Orange

The objective of Naive Bayes classifier is to predict if a given fruit (by knowing its color) is an apple or orange. Let's say the color of a fruit is green ($X = green$), can you predict what fruit (y) is it? In other words, you can predict y when only the X variables in training data are known.

The idea is to compute the two probabilities, that is the probability of the fruit being an apple or orange. Whichever fruit type gets the highest probability wins. Mathematically, you need to compute both $P(y = apple|X = green)$ and $P(y = orange|X = green)$, then compare the results. If $P(y = apple|X = green) > P(y = orange|X = green)$ then the prediction is an apple. If $P(y = apple|X = green) < P(y = orange|X = green)$ then the prediction is an orange.

This is the Naive Bayes formula for computing these probabilities:

$$P(y = apple|X = green) = \frac{P(X = green|y = apple)P(y = apple)}{P(X = green)}$$

$$P(y = orange|X = green) = \frac{P(X = green|y = orange)P(y = orange)}{P(X = green)}$$

The step-by-step of Naive Bayes algorithm:

Step 0: create a frequency table for each feature of the training data

First, you need to build a table called frequency table for each feature of the training data. Since we have only one feature, which is color, so we only need to build one frequency table.

Given the training data, you need to count how many red apples, green apples and orange apples are there. The same for the oranges, you need to count how many red oranges, green oranges and orange oranges are there.

No.	Color	Fruit
1	Red	Apple
2	Red	Apple
3	Green	Apple
4	Green	Orange

No.	Color	Fruit
5	Orange	Orange
...	...	...
100	Green	Orange

I have omitted the other rows for the sake of simplicity. The following table shows the frequency table. We have 33 red apples, 0 red orange, 20 green apples, 7 green oranges, 1 orange apple, and 39 orange oranges.

Color	Apple	Orange	Total
Red	33	0	33
Green	20	7	27
Orange	1	39	40
Total	54	46	100

Finally, you need to add them together to get the total number of apples, oranges, and the total number of fruits are there. The same for every color, you need to add them together to get the total number red fruits, green fruits, orange fruits, and the total number of fruits are there.

Step 1: compute the probabilities for each of the fruits (label)

In this step, you need to compute the $P(y = apple)$ and $P(y = orange)$. Out of 100 fruits, you have 54 apples and 46 oranges. So the respective probabilities are:

$$P(y = apple) = \frac{54}{100}$$

$$P(y = orange) = \frac{46}{100}$$

Step 2: compute the probability of the color (feature)

In this step, you need to compute $P(X = green)$. Out of 100 fruits, you have 27 greens. So the probability is:

$$P(X = green) = \frac{27}{100}$$

Step 3: compute the conditional probabilities

In this step, you need to compute $P(X = green|y = apple)$ and $P(X = green|y = orange)$. Out of 54 apples, you have 20 greens. So the probability is:

$$P(X = green|y = apple) = \frac{20}{54}$$

Out of 46 oranges, you have 7 greens. So the probability is:

$$P(X = green|y = orange) = \frac{7}{46}$$

Step 4: substitute all the three probabilities into the Naive Bayes formula

In this step, you need to subtitute all the three probabilities into the Naive Bayes formula.

$$P(y = apple|X = green) = \frac{\frac{20}{54}\frac{54}{100}}{\frac{27}{100}} = \frac{20}{27}$$

$$P(y = orange|X = green) = \frac{\frac{7}{46}\frac{46}{100}}{\frac{27}{100}} = \frac{7}{27}$$

$P(y = apple|X = green) > P(y = orange|X = green)$, which means apple get higher probability than orange, therefore apple will be our predicted fruit.

The Naive Bayes Formula

From the previous example, you can rewrite the Naive Bayes formula in more general form:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

where X is input feature and y is output label that you want to predict. $P(y|X)$ is called posterior probability. $P(X|y)$ is called likelihood. $P(y)$ is called label/class prior probability. $P(X)$ is called feature/predictor prior probability.

Now, if you notice in step 4, in the previous example, the value of denominators of both formulas remain constant ($\frac{27}{100}$) for given input. Since we compare these two probabilities, you can remove that term. So, for Naive Bayes **classifier** you can simplify the formula to:

$$P(y|X) \propto P(X|y)P(y)$$

Laplace Smoothing

In the previous example, the $P(X = red|y = orange)$ is zero. It makes sense because out of 46 oranges you have 0 red, but if you have many input features, the entire probability will become zero because one of the feature's value is zero. It will wipe out all the information in the other probabilities. This case is called zero frequency, and it needs to be avoided. You can use Laplace Smoothing to solve the problem of zero probability.

Laplace Smoothing can best be explained by an example. Let's say we want to compute $P(X = red|y = orange)$.

Without using Laplace Smoothing, the probability is

$$P(X = red|y = orange) = \frac{0}{46} = 0$$

It gives us a probability of zero.

Laplace Smoothing is usullay applied by **adding one count to the numerator and adding number of possible values of feature to the denominator**. In this case, number of possible values of feature is three (red, green, and orange). So, the probability after Laplace Smoothing is

$$P(X = red|y = orange) = \frac{0 + 1}{46 + 3} = \frac{1}{49}$$

So, by using Laplace Smoothing, it gives us a non-zero probability.

Even though Naive Bayes is naive, it perform very well in such applications, even when the features are not independent of each other. Furthermore, compared to other algorithms, Naive Bayes is fast. So, it could be used for making predictions in real time.

Naive Bayes, Part 2

Naive Bayes by Example 2

In part 1, you have seen a simple example of Naive Bayes classifier for fruit calssification. The model only has one input feature which is color. Now let's start with a slightly complex example that has more than one input feature.

Consider a fictional training data that describes the weather conditions as shown in the following table. Given the weather conditions, each tuple classifies the conditions as will it rain ('Yes') or not ('No').

No.	Outlook	Temperature	Humidity	Rain
1	Cloudy	Cool	High	Yes
2	Sunny	Mild	High	No
3	Cloudy	Hot	Normal	No
4	Sunny	Cool	Normal	No
5	Sunny	Hot	Low	No
6	Cloudy	Mild	Normal	Yes
7	Cloudy	Hot	High	Yes
8	Cloudy	Cool	Low	No
9	Sunny	Cool	High	Yes
10	Sunny	Hot	High	No
11	Cloudy	Hot	Low	No
12	Cloudy	Cool	Normal	Yes
13	Sunny	Mild	Low	No
14	Cloudy	Cool	Low	Yes
15	Sunny	Mild	Low	No

In this training data, we have three features which are outlook, temperature, and humidity. Outlook has two possible values: cloudy and sunny. Temperature has three possible values: cool, mild, and hot. Humidity has three possible values: low, normal, and high. Then, we have one label that has two possible values: yes and no.

We denote features as X_i where $i = \{1, 2, 3\}$ that corresponds to outlook, temperature, and humidity, repectively. We denote label as y. Naive Bayes classifier assumes each of the

features is independence. In fact, the independence assumption is never true, but often works well in practice.

In part 1, you have Naive Bayes classifier as follow:

$$P(y|X) \propto P(X|y)P(y)$$

where X is input feature and y is output label that you want to predict. If you have more than one feature, then the Naive Bayes classifier formula becomes:

$$P(y|X_1, \dots, X_n) \propto P(X_1|y) \dots P(X_n|y)P(y)$$

where X_1 to X_n are input features. You can caclulate the likelihood probability by multiplying all conditional probability of the features X_i given label y which can be expressed as:

$$P(y|X_1, \dots, X_n) \propto P(y) \prod_{i=1}^{n} P(X_i|y)$$

Back to our example, let's say today is cloudy, the temperature is cool, and the humidity is normal, can you predict will it rain today? Well, let's apply the same steps as in the example in part 1. Here, we want to calculate the following:

$$P(y = yes|X_1 = cloudy, X_2 = cool, X_3 = normal) \propto P(y = yes)P(X_1 = cloudy|y = yes$$

$$P(y = no|X_1 = cloudy, X_2 = cool, X_3 = normal) \propto P(y = no)P(X_1 = cloudy|y = no)P$$

First, create a frequency table for each feature of the training data as follows:

Outlook	Yes	No	Total
Cloudy	5	3	8
Sunny	1	6	7
Total	6	9	15

Temperature	Yes	No	Total
Cool	4	2	6
Mild	1	3	4
Hot	1	4	5
Total	6	9	15

Humidity	Yes	No	Total
Low	1	5	6
Normal	2	2	4

Humidity	Yes	No	Total
High	3	2	5
Total	6	9	15

Step 1: compute the probabilities for each value of the label

Out of 15 observations, you have 6 yes and 9 no. So the respective probabilities are:

$$P(y = yes) = \frac{6}{15}$$

$$P(y = no) = \frac{9}{15}$$

Step 2: compute the conditional probability

Out of 6 yes, you have 5 cloudy. So the probability is:

$$P(X_1 = cloudy|y = yes) = \frac{5}{6}$$

Out of 6 yes, you have 4 cool. So the probability is:

$$P(X_2 = cool|y = yes) = \frac{4}{6}$$

Out of 6 yes, you have 2 normal. So the probability is:

$$P(X_3 = normal|y = yes) = \frac{2}{6}$$

Out of 9 no, you have 3 cloudy. So the probability is:

$$P(X_1 = cloudy|y = no) = \frac{3}{9}$$

Out of 9 no, you have 2 cool. So the probability is:

$$P(X_2 = cool|y = no) = \frac{2}{9}$$

Out of 9 no, you have 2 normal. So the probability is:

$$P(X_3 = normal|y = no) = \frac{2}{9}$$

Step 3: subtitute all the three probabilities into the Naive Bayes formula

$$P(y = yes|X_1 = cloudy, X_2 = cool, X_3 = normal) \propto \frac{6}{15}\frac{5}{6}\frac{4}{6}\frac{2}{6} = \frac{240}{3240} = 0.074074074$$

$$P(y = no|X_1 = cloudy, X_2 = cool, X_3 = normal) \propto \frac{9}{15}\frac{3}{9}\frac{2}{9}\frac{2}{9} = \frac{108}{10935} = 0.009876543$$

Since
$P(y = yes|X_1 = cloudy, X_2 = cool, X_3 = normal) > P(y = no|X_1 = cloudy, X_2 = cool,$
, which means 'yes' get higher probability than 'no', then 'yes' will be our predicted output.

The Naive Bayes Classifier Formula (Updated)

To be more mathematically precise, you can rewrite the Naive Bayes classifier formula in this form:

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(X_i|y)$$

where $\hat{y}$ is the prediction. In statistics, the hat is used to denote an estimator or an estimated/predicted value. The $\arg\max$ of a function is the value of the input, i.e. the 'argument' at the maximum. In other words, it is the label y that has maximum probability. In the above example, the label y that has maximum probability is 'yes'. So, $\hat{y} = yes$.

Naive Bayes, Part 3

Naive Bayes by Example 3

In part 2, you have seen an example of Naive Bayes classifier for rain prediction. It has three input features. Now let's start with a new example. In this example, we want to build an intelligent lighting. The light has two states either 'On' or 'Off' depending on user behaviour based on time and light intensity in the environment.

Consider a training data that describes the light conditions as shown in the following table. You have two input features (light sensor and time) and one label (light state). The value of light sensor is normalized to 0-1. The value of time is from 00:00 AM to 23:00 PM. Given the light sensor and time, each tuple classifies the light conditions as 'On' or 'Off'.

In [ ]:
```
import pandas as pd
df = pd.read_csv("intelligent-lighting.csv")
df
```

Out[ ]:

	Light Sensor	Time	Light State
0	0.0	00:00	Off
1	0.0	01:00	Off
2	0.0	02:00	Off
3	0.0	03:00	Off
4	0.0	04:00	Off
5	0.1	04:00	Off
6	0.1	05:00	On
7	0.2	06:00	On
8	0.3	07:00	On
9	0.4	08:00	Off
10	0.2	08:00	On
11	0.5	09:00	Off
12	0.3	09:00	On
13	0.6	10:00	Off
14	0.7	11:00	Off
15	0.9	12:00	Off
16	1.0	12:00	Off
17	0.4	12:00	On
18	0.9	13:00	Off
19	1.0	13:00	Off
20	0.8	14:00	Off
21	0.3	14:00	On
22	0.7	15:00	Off
23	0.2	15:00	On
24	0.6	16:00	Off
25	0.6	17:00	Off
26	0.1	17:00	On
27	0.5	18:00	Off
28	0.3	18:00	On
29	0.5	19:00	Off

	Light Sensor	Time	Light State
30	0.2	19:00	On
31	0.1	20:00	On
32	0.0	20:00	On
33	0.1	21:00	On
34	0.0	21:00	On
35	0.0	22:00	On
36	0.0	23:00	On

Before we build our Naive Bayes model, I want to group light sensor values into four levels as follows:

$$\text{Light Sensor} \leftarrow \begin{cases} 0 & \text{if } 0.00 \leq \text{Light Sensor} \leq 0.25 \\ 1 & \text{if } 0.26 \leq \text{Light Sensor} \leq 0.50 \\ 2 & \text{if } 0.51 \leq \text{Light Sensor} \leq 0.75 \\ 3 & \text{if } 0.76 \leq \text{Light Sensor} \leq 1.00 \end{cases}$$

In [ ]:
```
# group light sensor value into 4 levels
for i in range(len(df)):
    if (df.loc[i,'Light Sensor'] >= 0.0 and df.loc[i,'Light Sensor'] <= 0.25):
        df.loc[i,'Light Sensor'] = 0
    elif (df.loc[i,'Light Sensor'] >= 0.26 and df.loc[i,'Light Sensor'] <= 0.50):
        df.loc[i,'Light Sensor'] = 1
    elif (df.loc[i,'Light Sensor'] >= 0.51 and df.loc[i,'Light Sensor'] <= 0.75):
        df.loc[i,'Light Sensor'] = 2
    elif (df.loc[i,'Light Sensor'] >= 0.76 and df.loc[i,'Light Sensor'] <= 1.0):
        df.loc[i,'Light Sensor'] = 3
df['Light Sensor'] = df['Light Sensor'].astype('int64')    # convert float64 to int
df
```

Out[ ]:

	Light Sensor	Time	Light State
0	0	00:00	Off
1	0	01:00	Off
2	0	02:00	Off
3	0	03:00	Off
4	0	04:00	Off
5	0	04:00	Off
6	0	05:00	On
7	0	06:00	On
8	1	07:00	On
9	1	08:00	Off
10	0	08:00	On
11	1	09:00	Off
12	1	09:00	On
13	2	10:00	Off
14	2	11:00	Off
15	3	12:00	Off
16	3	12:00	Off
17	1	12:00	On
18	3	13:00	Off
19	3	13:00	Off
20	3	14:00	Off
21	1	14:00	On
22	2	15:00	Off
23	0	15:00	On
24	2	16:00	Off
25	2	17:00	Off
26	0	17:00	On
27	1	18:00	Off
28	1	18:00	On
29	1	19:00	Off

	Light Sensor	Time	Light State
30	0	19:00	On
31	0	20:00	On
32	0	20:00	On
33	0	21:00	On
34	0	21:00	On
35	0	22:00	On
36	0	23:00	On

Now, let's build our Naive Bayes model for predicting light state. Let X_i where $i = \{1, 2\}$ be the features that corresponds to light sensor and time, repectively. Let y be the label (light state). You can make a prediction of light state using the following Naive Bayes equation:

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(X_i|y)$$

where $\hat{y}$ is the prediction and $n = 2$.

Let's say light sensor level is 2 and time is 12:00, can you predict the light state? Well, let's apply the same steps as in the previous examples. Here, we want to calculate the following:

$$\hat{y} = \arg\max_y(P(y = off)P(X_1 = 2|y = off)P(X_2 = 12|y = off), P(y = on)P(X_1 = 2|$$

Step 0: create a frequency table for each feature of the training data

Light Sensor	Off	On	Total
Level 0	6	12	18
Level 1	4	5	9
Level 2	5	0	5
Level 3	5	0	5
Total	20	17	37

Time	Off	On	Total
00:00	1	0	1
01:00	1	0	1
02:00	1	0	1
03:00	1	0	1
04:00	2	0	2

Time	Off	On	Total
05:00	0	1	1
06:00	0	1	1
07:00	0	1	1
08:00	1	1	2
09:00	1	1	2
10:00	1	0	1
11:00	1	0	1
12:00	2	1	3
13:00	2	0	2
14:00	1	1	2
15:00	1	1	2
16:00	1	0	1
17:00	1	1	2
18:00	1	1	2
19:00	1	1	2
20:00	0	2	2
21:00	0	2	2
22:00	0	1	1
23:00	0	1	1
Total	20	17	37

Step 1: compute the probabilities for each value of the label

Out of 15 observations, you have 20 'Off' and 17 'On'. So the respective probabilities are:

$$P(y = off) = \frac{20}{37}$$

$$P(y = on) = \frac{17}{37}$$

Because there are zero probabilities, you need to add Laplace Smoothing:

$$P(y = off) = \frac{20 + 1}{37 + 2.1} = \frac{21}{39}$$

$$P(y = on) = \frac{17 + 1}{37 + 2.1} = \frac{18}{39}$$

Step 2: compute the conditional probability

Out of 20 'Off', you have 5 'Level 2'. So the probability is:

$$P(X_1 = 2|y = off) = \frac{5}{20}$$

Out of 20 'Off', you have 2 '12:00'. So the probability is:

$$P(X_2 = 12|y = off) = \frac{2}{20}$$

Out of 17 'On', you have 0 'Level 2'. So the probability is:

$$P(X_1 = 2|y = on) = \frac{0}{17}$$

Out of 17 'On', you have 1 '12:00'. So the probability is:

$$P(X_2 = 12|y = on) = \frac{1}{17}$$

Because there are zero probabilities, you need to add Laplace Smoothing:

$$P(X_1 = 2|y = off) = \frac{5+1}{20+4.1} = \frac{6}{24}$$

$$P(X_2 = 12|y = off) = \frac{2+1}{20+24.1} = \frac{3}{44}$$

$$P(X_1 = 2|y = on) = \frac{0+1}{17+4.1} = \frac{1}{21}$$

$$P(X_2 = 12|y = on) = \frac{1+1}{17+24.1} = \frac{2}{41}$$

Step 3: subtitute all the three probabilities into the Naive Bayes classifier

$\hat{y} = \arg\max_y(P(y = off)P(X_1 = 2|y = off)P(X_2 = 12|y = off), P(y = on)P(X_1 = 2|y$

$$\hat{y} = \arg\max_y(\frac{21}{39}.\frac{6}{24}.\frac{3}{44}, \frac{18}{39}.\frac{1}{21}.\frac{2}{41})$$

$$\hat{y} = \arg\max_y(0.00918, 0.00107)$$

$$\hat{y} = off$$

So, our predicted light state when light sensor level is 2 and time is 12:00 is 'Off'.

Machine Learning Part 1

Linear Regression:

You learned about how traditional programming is where you explicitly figure out the rules that act on some data to give an answer like this:

And then you saw that Machine Learning changes this, for scenarios where you may not be able to figure out the rules feasibly, and instead have a computer figure out what they are. That made the diagram look like this:

Then you read about the steps that a computer takes -- where it makes a guess, then looks at the data to figure out how accurate the guess was, and then makes another guess and so on.

So, consider if I give you a set of numbers like this:

X: -1, 0, 1, 2, 3, 4

And then I give you another set of numbers like this:

Y: -3, -1, 1, 3, 5, 7

Can you figure out the relationship between the two sets? There's a function that converts -1 to -3, 0 to -1, 1 to 1, 2 to 3, 3 to 5 and 4 to 7. Can you figure out the relationship. Think about it for a moment.

…

Often when I ask people about it, they see that the 0 is matched to -1, so Y is (something) times X -1. Maybe they'll take a guess at the something, and come up with 3.
Then fill in the gaps, if Y=3X-1, then

X: -1, 0, 1, 2, 3, 4

Becomes

Y: -4, -1, 2, 5, 8, 11

Other than working for 0, it fails for everything else. In ML terms, you can define this as your loss is ‘high’. With what you learned from that, you might think, what if it’s Y=2X-1?

Then, when you fill in the results for Y=2X-1, you’ll get:

X: -1, 0, 1, 2, 3, 4

Becomes

Y: -3, -1, 1, 3, 5, 7

...which matches your original data perfectly. Your loss is zero.

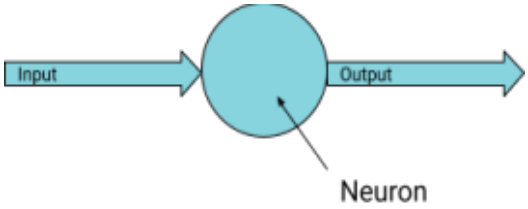
You’ve just gone through this process:

In the next you’ll explore this in a little more depth.

Simple Neural Networks

Up to now you've been looking at matching X values to Y values when there's a linear relationship between them. So, for example, you matched the X values in [-1, 0 , 1, 2, 3, 4] to the Y values in [-3, -1, 1, 3, 5, 7] by figuring out the equation Y=2X-1.

You then saw how a very simple neural network with a single neuron could be used for this.



This worked very well, because, in reality, what is referred to as a 'neuron' here is simply a function that has two learnable parameters, called a 'weight' and a 'bias', where, the output of the neuron will be:

Output = (Weight * Input) + Bias

So, for learning the linear relationship between our Xs and Ys, this maps perfectly, where we want the weight to be learned as '2', and the bias as '-1'. In the code you saw this happening.

When multiple neurons work together in layers, the learned weights and biases across these layers can then have the effect of letting the neural network learn more complex patterns. You'll learn more about how this works later in the course.

In your first Neural Network you saw neurons that were densely connected to each other, so you saw the **Dense** layer type. As well as neurons like this, there are also additional layer types in TensorFlow that you'll encounter. Here's just a few of them:

- **Convolutional** layers contain filters that can be used to transform data. The values of these filters will be learned in the same way as the parameters in the Dense neuron you saw here. Thus, a network containing them can learn how to transform data effectively. This is especially useful in Computer Vision, which you'll see later in this course. We'll even use these convolutional layers that are typically used for vision models to do speech detection! Are you wondering how or why? Stay tuned!
- **Recurrent** layers learn about the relationships between pieces of data in a sequence. There are many types of recurrent layer, with a popular one called LSTM (Long, Short Term Memory), being particularly effective. Recurrent layers are useful for predicting sequence data (like the weather), or understanding text.

You'll also encounter layer types that *don't* learn parameters themselves, but which can affect the other layers. These include layers like **dropouts**, which are used to reduce the density of connection between dense layers to make them more efficient, **pooling** which can be used to reduce the amount of data flowing through the network to remove unnecessary information, and **lambda** layers that allow you to execute arbitrary code.

Your journey over the next few videos will primarily deal with the **Dense** network type, and you'll start to explore how multiple layers can work together to infer the rules that match your data to your labels.

Initialization in Machine Learning

Recall that Machine Learning, the core process powering embedded AI, is the process of having Data and Answers, and from them attempting to infer the rules that link them:

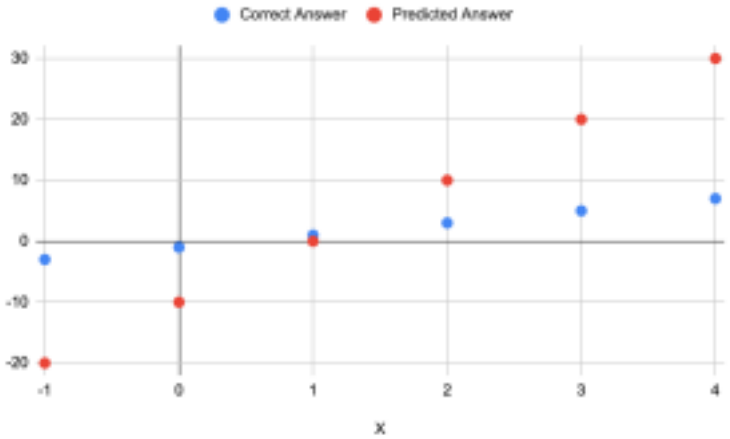
So, if your **data** is the numbers in this set: [-1, 0 , 1, 2, 3, 4] , and your corresponding **answers** are the numbers in this set: [-3, -1, 1, 3, 5, 7], one way to begin to infer the relationship between them (assuming it's linear) is to have a function, where we say Y = wX+b, and we have to try to figure out the values of w and b.

A process to do this is shown below:

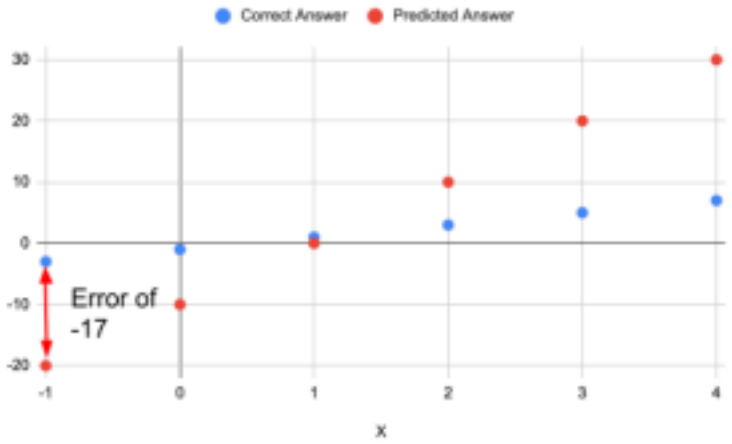
We could make a guess as to the values of w and b, and measure how accurate that guess is.

So, for example, if we guess that w=10, and b = -10, we would get the set of **answers** for our data as [-20, -10, 0, 10, 20, 30], where our *correct* answers are [-3, -1, 1, 3, 5, 7].

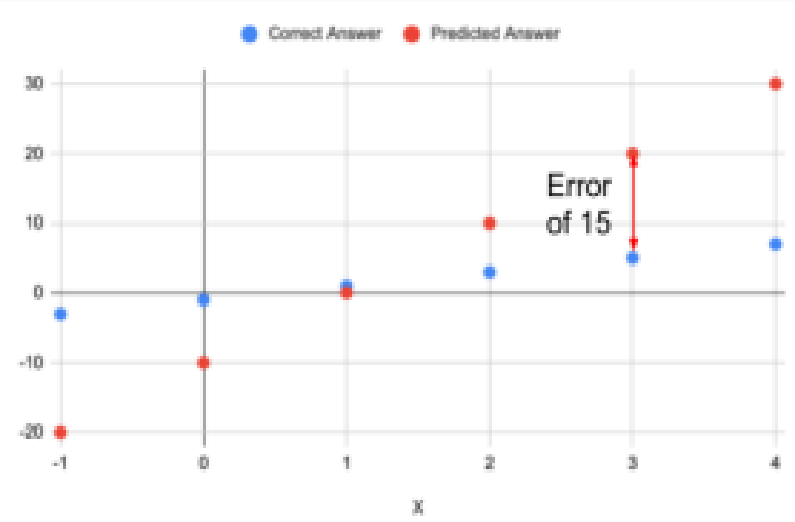
From this we can measure our *loss*, by plotting our predicted answer against our actual answer.



So, for example, for x=-1, we predicted -20, when the answer is actually -3, so we are off by -17



And when x=3, we predicted the answer to be 20, when in fact it is supposed to be 5, so we have an error of 15.

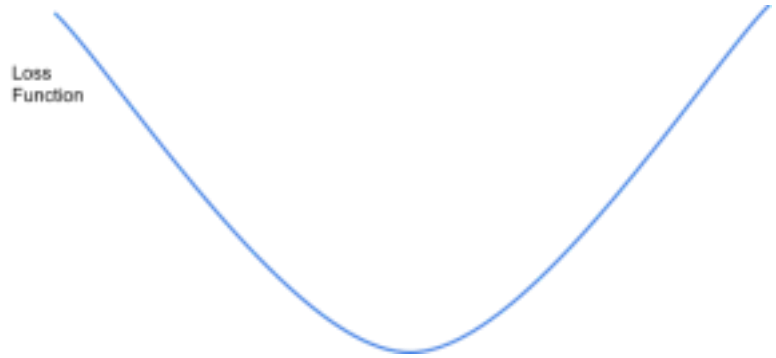


If we want to find out our error, we could average each of these.

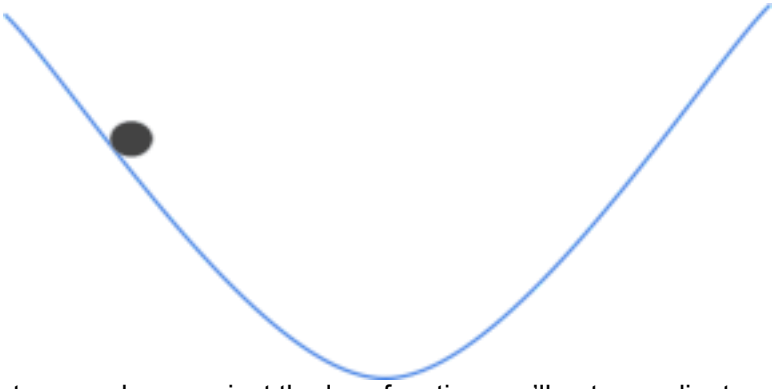
However, by doing this our negative (-17) and positive (15) errors would mostly cancel out, so the smart thing to do to ensure that we don't have them do this is to *square* the error amount, average out the squares of the errors, and then get the square root of that.

This gives us a loss *function* of root mean squared error.

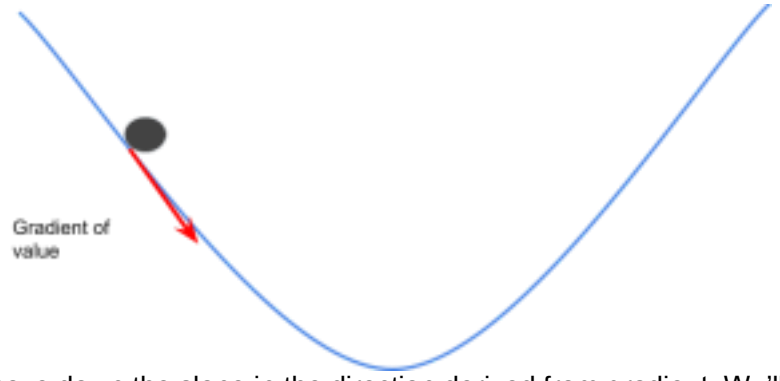
When we now want to optimize our guess, and do better than these results, we can look at the loss function, and figure out a way to minimize our loss. As the loss function uses a *square*, the curve of the function is parabolic.



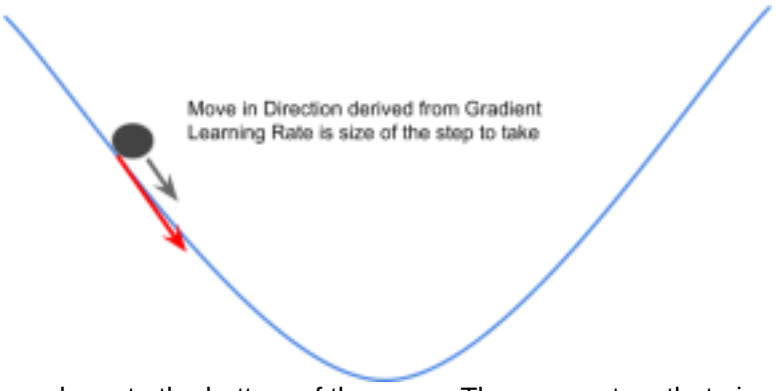
And the value of our loss with our current parameters can be plotted on this. Our loss was high, so we can say we're pretty far up the curve.



When we differentiate our values against the loss function, we'll get a gradient, and we can use the gradient to figure out which direction we have to go in to move down the slope!



And we can then move down the slope in the direction derived from gradient. We'll 'jump' by a certain amount, and we can call that amount the 'Learning Rate'



After that, we are now closer to the bottom of the curve. The parameters that give us our location on the curve can then be the next ‘guess’, and by definition, these will have a lower loss, and we can repeat the loop. This is called *back propagation*.

The process can be repeated, and over time the value will get closer and closer to the bottom of the parabola, which is the minimum value of the loss function.

This process of using the gradient of the value to figure out the direction of the minimum, and then jumping down the curve towards the minimum is called *gradient descent.*

Understanding of Neurons in Action

In the previous video we created a function that had two parameters -- w and b, and returned a value f(x) = wx+b. You then saw how to use the machine learning training loop to adjust these parameters so that the correct values could be 'learned' over time.

You also, earlier, saw how this works in TensorFlow with machine learning.

```
my_layer = keras.layers.Dense(units= 1 ,
input_shape=[ 1 ]) model =
tf.keras.Sequential([my_layer])
model. compile (optimizer= 'sgd' , loss= 'mean_squared_error' )

xs = np.array([- 1.0 , 0.0 , 1.0 , 2.0 , 3.0 , 4.0 ],
dtype= float ) ys = np.array([- 3.0 , - 1.0 , 1.0 , 3.0 ,
5.0 , 7.0 ], dtype= float )

model.fit(xs, ys, epochs= 500 )
```

I've changed the code slightly so that instead of having the layer code within the tf.keras.Sequential(), I declared it as my_layer. Now, if you train this network, and try to predict a value for a given X like this:

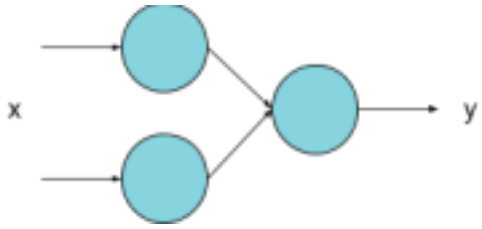
```
print (model.predict([ 10.0 ]))
```

You'll see a value that's close to 19. It did this by learning the internal parameters of the neuron, and you can inspect the neuron by looking at my_layer using it's get_weights() parameter:

```
print (my_layer.get_weights())
[array([[ 1.9980532 ]], dtype=float32), array([- 0.9939642 ],
dtype=float32)]
```

You'll see that you get back two arrays. The first contains the w value -- which after running for 500 epochs gives you a value that's very close to 2! Similarly the second contains the b value, which got learned to be very close to -1.

But what would it look like if you used more than just a single neuron? So, for example, if you used a multiple neural network that looks like this?



To implement this in code, you'd use 2 layers, the first with 2 neurons, and the second with 1 neuron. It would look like this:

```
my_layer_1 = keras.layers.Dense(units= 2 ,
input_shape=[ 1 ]) my_layer_2 =
keras.layers.Dense(units= 1 )
model = tf.keras.Sequential([my_layer_1, my_layer_2])
model. compile (optimizer= 'sgd' , loss= 'mean_squared_error' )

xs = np.array([- 1.0 , 0.0 , 1.0 , 2.0 , 3.0 , 4.0 ],
dtype= float ) ys = np.array([- 3.0 , - 1.0 , 1.0 , 3.0 ,
5.0 , 7.0 ], dtype= float )

model.fit(xs, ys, epochs= 500 )
```

So, if you look at the *parameters* in this case, there'll be a little bit of a difference in the second layer. Typically we have said that our neuron has an input, x, and an output, y, and y will equal wx+b, where w and b are learned parameters. However, when we have 2 inputs, as you can see above, what will happen is that the formula will change where there is a separate w for *each* input.

The neuron in the second layer has 2 inputs, so it will, instead of having y = w*x+b, it will have y = w1*x1+w2*x2+b, where x1 is the output of the first neuron in the previous layer, and x2 is the output of the second neuron in the previous layer. Naturally, if there are more than 2 neurons in the previous layer, then that number of weights will be learned.

If you run the above code to learn the parameters, then inspect the parameters:

```
print (my_layer_1.get_weights())
```

You'll see something like this:

```
[array([[ 1.4040651 , - 0.7996106 ]], dtype=float32), array([-
0.50982034 , 0.20248567 ], dtype=float32)]
```

This will give you the weights and biases for the neurons in the layer. Note that it isn't a list of weight and bias for the first neuron followed by weight and bias for the second. In the above example 1.4040651 is the learned weight for the first neuron and - 0.7996106 is the learned weight

for the second. Similarly - 0.50982034 , 0.20248567 are the learned biases for the first and second neurons respectively.

Similarly: print (my_layer_2.get_weights()) . Will give you something like:

[array([[1.2653419],[- 0.27935725]], dtype=float32), array([- 0.29833543], dtype=float32)]
As mentioned earlier -- you can see that there are 2 weight values in this array, and a single bias. These weights are applied to the output of the previous neuron, and then they are summed and added to the bias.

You can inspect them manually, and apply the sum yourself like this:

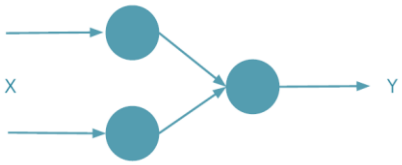
```
value_to_predict = 10.0

layer1_w1 = (my_layer_1.get_weights()[ 0 ][ 0 ][ 0 ])
layer1_w2 = (my_layer_1.get_weights()[ 0 ][ 0 ][ 1 ])
layer1_b1 = (my_layer_1.get_weights()[ 1 ][ 0 ])
layer1_b2 = (my_layer_1.get_weights()[ 1 ][ 1 ])


layer2_w1 = (my_layer_2.get_weights()[ 0 ][ 0 ])
layer2_w2 = (my_layer_2.get_weights()[ 0 ][ 1 ])
layer2_b = (my_layer_2.get_weights()[ 1 ][ 0 ])

neuron1_output = (layer1_w1 * value_to_predict) +
layer1_b1   neuron2_output = (layer1_w2 *
value_to_predict) + layer1_b2

neuron3_output = (layer2_w1 * neuron1_output) +
(layer2_w2 *  neuron2_output) + layer2_b

print (neuron3_output)

Will output: [ 18.999996 ]
```

This same process will apply for bigger and more dense neural networks, and will allow you to build models that learn more sophisticated patterns. You'll see this shortly with an introduction to Computer Vision, where the model can learn the patterns in pixels in an image and classify them against labels. Then, later in the course, we'll explore how these models need to be adapted in the context of embedded AI.

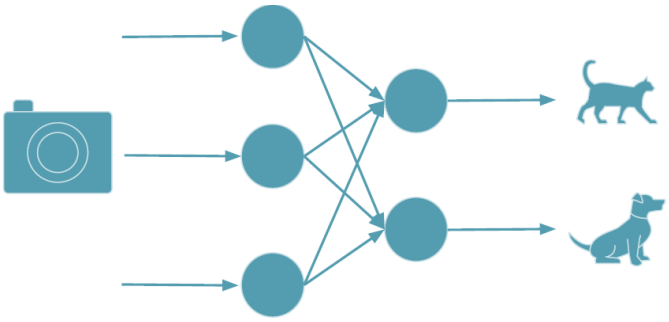
Machine Learning Part 2
Neural Network from Regression to Classification

- Single-In Single-Out (SISO) NN

- Multiple-In Single-Out (MISO) NN



- Classification



- Data and Label

Data Label

[1, 0]

[0, 1]

Example: Digit Recognition

0 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

1 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

2 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

3 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

4 [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

5 [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]

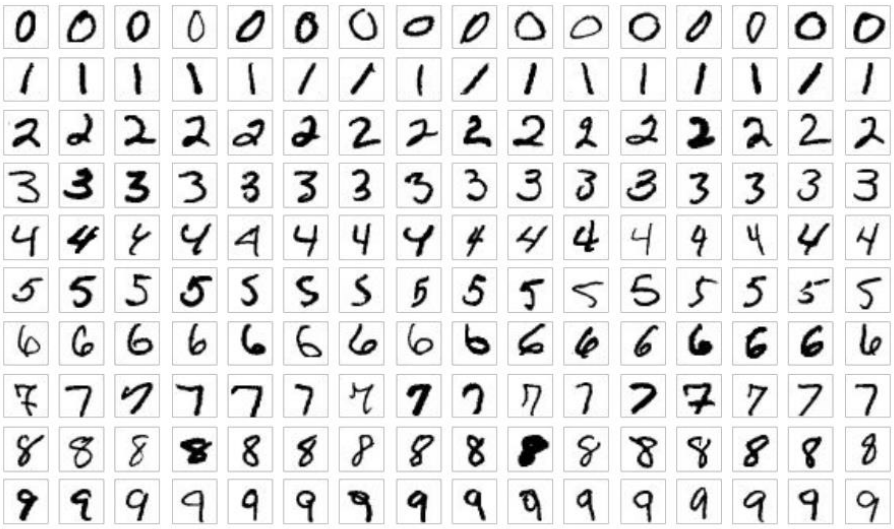
6 [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]

7 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]

8 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]

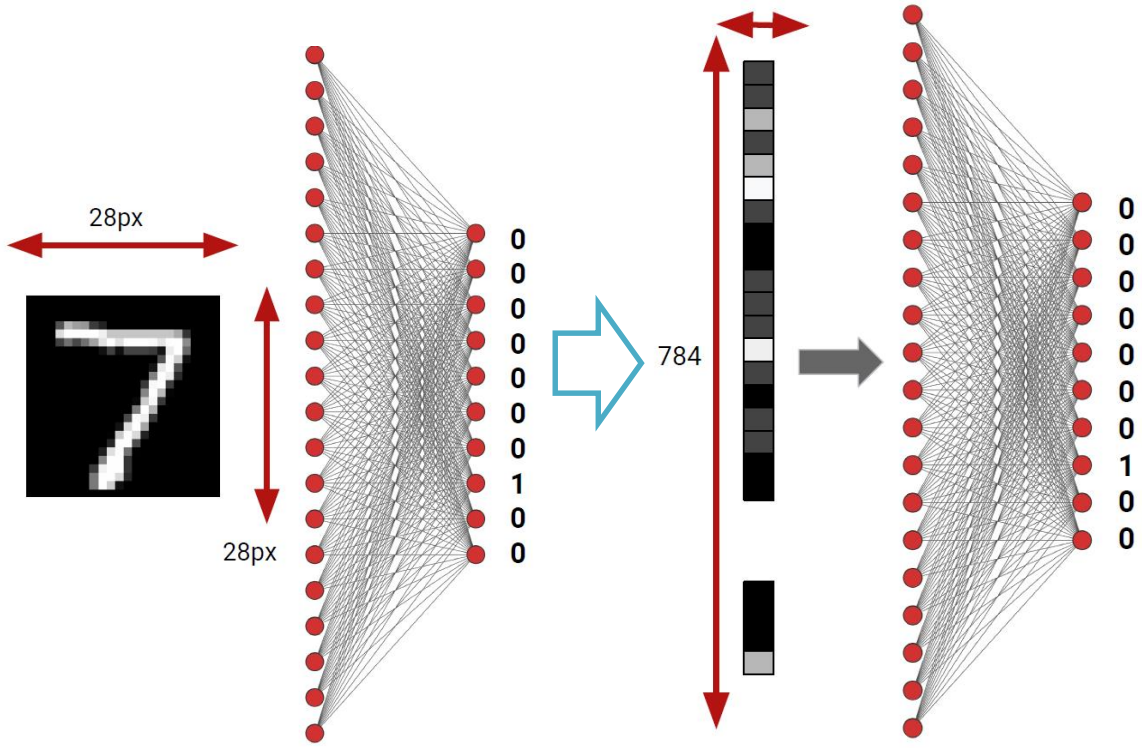
9 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

MNIST Data:



60,000 Labelled Training Examples
10.000 Labelled Validation Examples

Understanding of Data:



When writing code for neural networks, often the bulk of your attention will be on the model architecture -- because you want to find the optimum architecture for accuracy on your training set, and that accuracy also carries over to validation and testing.

This is good practice, but do keep in mind that often you will have many more lines of code to handle everything else in your system (e.g., data collection and preparation).

With MNIST, your code looked like this:

```
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))
```

Note that it only took 3 lines of code to load and prepare your data -- this is because the MNIST dataset was already available to you, so you could get it with 1 line, and then it was just another 2 lines to normalize the images. As you build real systems, getting the data may not always be this easy -- and it might require significant coding to get it into a place where you can train a neural network with it. Going into the specifics of this process for every possible data source is beyond the scope of this course, but as you continue to learn neural networks, it's good to practice with different types of data, so you can prepare them for ingestion into your neural network architecture.

```
classifications = model.predict(val_images)
print(classifications[0])
print(test_labels[0])
```

```
[2.4921512e-09 1.3765138e-10 8.8281205e-08 1.0477231e-
03 2.8455029e-12 4.0820678e-06 2.0070659e-16
9.9894780e-01 1.0296049e-07 2.9972372e-07]
```

In this course we will explore three main types of data in the context of embedded AI: audio, visual, and motion. As you will explore later, we often need to include pre and post processing steps to this data to get good results from our machine learning models.

Indeed, often you'll find about 80% of your code might be in the preparation, and only 10% (or less) in your neural network architecture, leaving the other 10% for testing and other tidy up.

Machine Learning Part-2 Extra

Start with a simple neural network for MNIST

Note that there are 2 layers, one with 20 neurons, and one with 10.

The 10-neuron layer is our final layer because we have 10 classes we want to classify.

Train this, and you should see it get about 98% accuracy

In [ ]:
```
# Load libraries
import sys

import tensorflow as tf
```

In [ ]:
```
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.soft

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

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

Examine the test data

Using model.evaluate, you can get metrics for a test set. In this case we only have a training set and a validation set, so we can try it out with the validation set. The accuracy will be slightly lower, at maybe 96%. This is because the model hasn't previously seen this data and may not be fully generalized for all data. Still it's a pretty good score.

You can also predict images, and compare against their actual label. The [0] image in the set is a number 7, and here you can see that neuron 7 has a 9.9e-1 (99%+) probability, so it got it right!

In [ ]:
```
model.evaluate(val_images, val_labels)
```

```
classifications = model.predict(val_images)
print(classifications[0])
print(val_labels[0])
```

Modify to inspect learned values

This code is identical, except that the layers are named prior to adding to the sequential. This allows us to inspect their learned parameters later.

In [ ]:
```
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
layer_1 = tf.keras.layers.Dense(20, activation=tf.nn.relu)
layer_2 = tf.keras.layers.Dense(10, activation=tf.nn.softmax)
model = tf.keras.models.Sequential([tf.keras.layers.Flatten(input_shape=(28,28)),
                                    layer_1,
                                    layer_2])

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

model.fit(training_images, training_labels, epochs=20)

model.evaluate(val_images, val_labels)

classifications = model.predict(val_images)
print(classifications[0])
print(val_labels[0])
```

Inspect weights

If you print layer_1.get_weights(), you'll see a lot of data. Let's unpack it. First, there are 2 arrays in the result, so let's look at the first one. In particular let's look at its size.

In [ ]:
```
print(layer_1.get_weights()[0].size)
```

The above code should print 15680. Why?

Recall that there are 20 neurons in the first layer.

Recall also that the images are 28x28, which is 784.

If you multiply 784 x 20 you get 15680.

So...this layer has 20 neurons, and each neuron learns a W parameter for each pixel. So instead of y=Mx+c, we have y=M1X1+M2X2+M3X3+....+M784X784+C in every neuron!

Every pixel has a weight in every neuron. Those weights are multiplied by the pixel value, summed up, and given a bias.

In [ ]:
```
print(layer_1.get_weights()[1].size)
```

The above code will give you 20 -- the get_weights()[1] contains the biases for each of the 20 neurons in this layer.

Inspecting layer 2

Now let's look at layer 2. Printing the get_weights will give us 2 lists, the first a list of weights for the 10 neurons, and the second a list of biases for the 10 neurons

Let's look first at the weights:

In [ ]:
```
print(layer_2.get_weights()[0].size)
```

This should return 200. Again, consider why?

There are 10 neurons in this layer, but there are 20 neurons in the previous layer. So, each neuron in this layer will learn a weight for the incoming value from the previous layer. So, for example, the if the first neuron in this layer is N21, and the neurons output from the previous layers are N11-N120, then this neuron will have 20 weights (W1-W20) and it will calculate its output to be:

W1N11+W2N12+W3N13+...+W20N120+Bias

So each of these weights will be learned as will the bias, for every neuron.

Note that N11 refers to Layer 1 Neuron 1.

In [ ]:
```
print(layer_2.get_weights()[1].size)
```

...and as expected there are 10 elements in this array, representing the 10 biases for the 10 neurons.

Hopefully this helps you see how the element of a simple neuron containing y=mx+c can be expanded greatly into a deep neural network, and that DNN can learn the parameters that match the 784 pixels of an image to their output!

Exploring Convolutions

What are Convolutions?

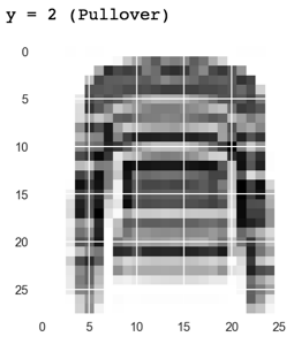
What are convolutions? In this lab you'll explore what they are and how they work. In later lessons, you'll see how to use them in your neural network.

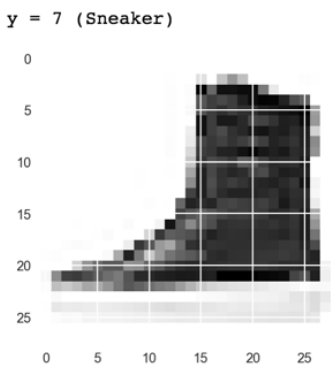
Together with convolutions, you'll use something called 'Pooling', which compresses your image, further emphasising the features. You'll also see how pooling works in this lab.

Limitations of the previous DNN

In an earlier exercise you saw how to train an image classifier for fashion items using the Fashion MNIST dataset. This gave you a pretty accuract classifier, but there was an obvious constraint: the images were 28x28, grey scale and the item was centered in the image.

For example here are a couple of the images in Fashion MNIST





The DNN that you created simply learned from the raw pixels what made up a sweater, and what made up a boot in this context. But consider how it might classify this image?

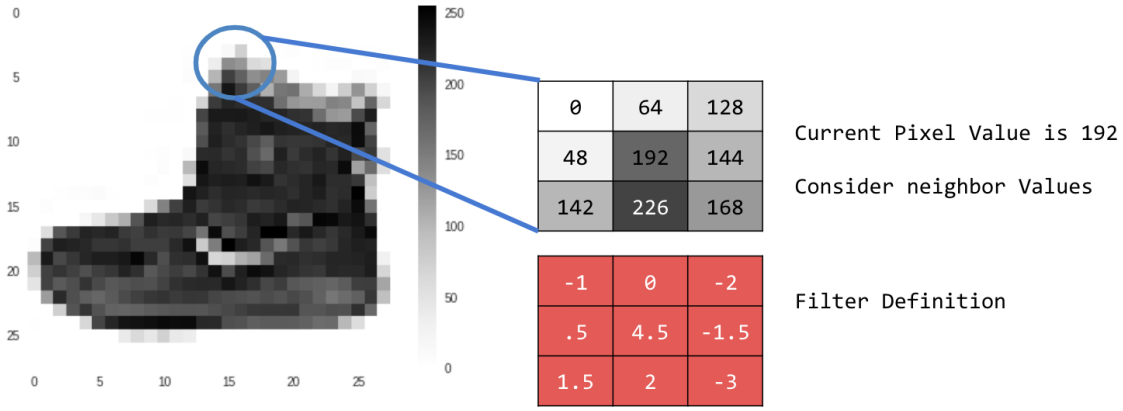
(Image is Public domain CC0 from Pixabay: https://pixabay.com/photos/boots-travel-railroad-tracks-181744/)

While it's clear that there are boots in this image, the classifier would fail for a number of reasons. First, of course, it's not 28x28 greyscale, but more importantly, the classifier was trained on the raw pixels of a left-facing boot, and not the features that make up what a boot is.

That's where Convolutions are very powerful. A convolution is a filter that passes over an image, processing it, and extracting features that show a commonolatity in the image. In this lab you'll see how they work, but processing an image to see if you can extract features from it!

Generating convolutions is very simple -- you simply scan every pixel in the image and then look at it's neighboring pixels. You multiply out the values of these pixels by the equivalent weights in a filter.

So, for example, consider this:




```
CURRENT_PIXEL_VALUE = 192
NEW_PIXEL_VALUE = (-1 * 0) + (0 * 64) + (-2 * 128) +
                  (.5 * 48) + (4.5 * 192) + (-1.5 * 144) +
                  (1.5 * 42) + (2 * 226) + (-3 * 168)
```

In this case a 3x3 Convolution is specified.

The current pixel value is 192, but you can calculate the new one by looking at the neighbor values, and multiplying them out by the values specified in the filter, and making the new pixel value the final amount.

Let's explore how convolutions work by creating a basic convolution on a 2D Grey Scale image. First we can load the image by taking the 'ascent' image from scipy. It's a nice, built-in picture with lots of angles and lines.

Let's start by importing some python libraries.

In [ ]:
```
import cv2
import numpy as np
from scipy import datasets
i = datasets.ascent()
```

Next, we can use the pyplot library to draw the image so we know what it looks like.

In [ ]:
```
import matplotlib.pyplot as plt
plt.grid(False)
plt.gray()
plt.axis('off')
plt.imshow(i)
plt.show()
```

We can see that this is an image of a stairwell. There are lots of features in here that we can play with seeing if we can isolate them -- for example there are strong vertical lines.

The image is stored as a numpy array, so we can create the transformed image by just copying that array. Let's also get the dimensions of the image so we can loop over it later.

In [ ]:
```
i_transformed = np.copy(i)
size_x = i_transformed.shape[0]
size_y = i_transformed.shape[1]
```

Now we can create a filter as a 3x3 array.

In [ ]:
```
# This filter detects edges nicely
# It creates a convolution that only passes through sharp edges and straight
# lines.

#Experiment with different values for fun effects.
#filter = [ [0, 1, 0], [1, -4, 1], [0, 1, 0]]

# A couple more filters to try for fun!
filter = [ [-1, -2, -1], [0, 0, 0], [1, 2, 1]]
#filter = [ [-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]

# If all the digits in the filter don't add up to 0 or 1, you
# should probably do a weight to get it to do so
# so, for example, if your weights are 1,1,1 1,2,1 1,1,1
# They add up to 10, so you would set a weight of .1 if you want to normalize them
weight  = 1
```

Now let's create a convolution. We will iterate over the image, leaving a 1 pixel margin, and multiply out each of the neighbors of the current pixel by the value defined in the filter.

i.e. the current pixel's neighbor above it and to the left will be multiplied by the top left item in the filter etc. etc. We'll then multiply the result by the weight, and then ensure the result is in the range 0-255

Finally we'll load the new value into the transformed image.

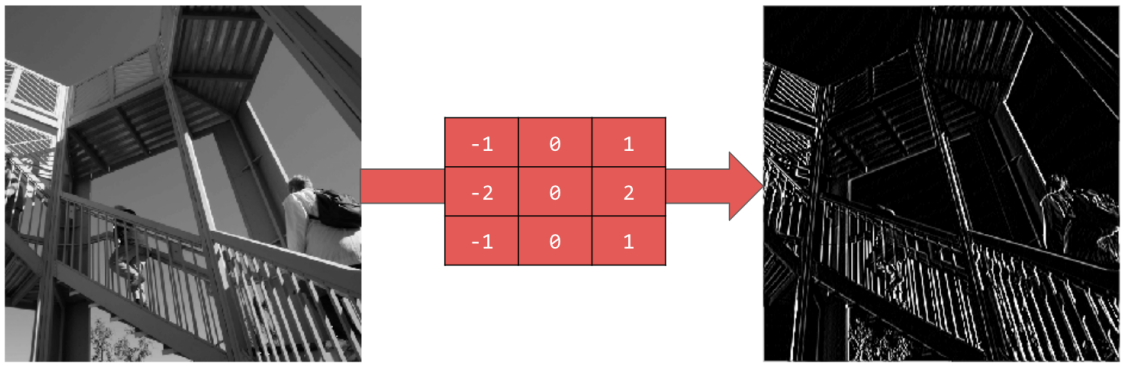
In [ ]:
```
for x in range(1,size_x-1):
  for y in range(1,size_y-1):
      convolution = 0.0
      convolution = convolution + (i[x - 1, y-1] * filter[0][0])
      convolution = convolution + (i[x, y-1] * filter[1][0])
      convolution = convolution + (i[x + 1, y-1] * filter[2][0])
      convolution = convolution + (i[x-1, y] * filter[0][1])
      convolution = convolution + (i[x, y] * filter[1][1])
      convolution = convolution + (i[x+1, y] * filter[2][1])
      convolution = convolution + (i[x-1, y+1] * filter[0][2])
      convolution = convolution + (i[x, y+1] * filter[1][2])
      convolution = convolution + (i[x+1, y+1] * filter[2][2])
      convolution = convolution * weight
      if(convolution<0):
        convolution=0
      if(convolution>255):
        convolution=255
      i_transformed[x, y] = convolution
```

Now we can plot the image to see the effect of the convolution!

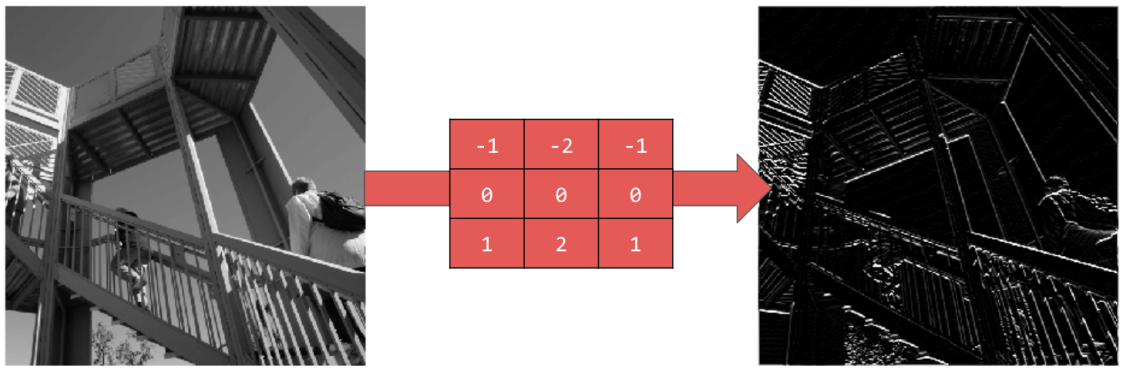
In [ ]:
```
# Plot the image. Note the size of the axes -- they are 512 by 512
plt.gray()
plt.grid(False)
plt.imshow(i_transformed)
#plt.axis('off')
plt.show()
```

So, consider the following filter values, and their impact on the image.

Using -1,0,1,-2,0,2,-1,0,1 gives us a very strong set of vertical lines:



Using -1, -2, -1, 0, 0, 0, 1, 2, 1 gives us horizontal lines:



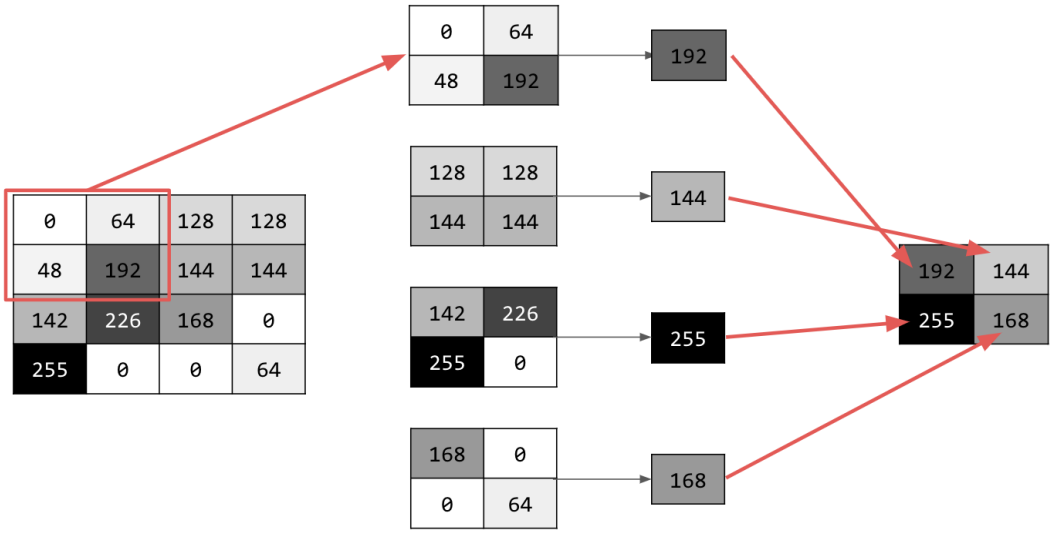
Explore different values for yourself!

Pooling

As well as using convolutions, pooling helps us greatly in detecting features. The goal is to reduce the overall amount of information in an image, while maintaining the features that are detected as present.

There are a number of different types of pooling, but for this lab we'll use one called MAX pooling.

The idea here is to iterate over the image, and look at the pixel and it's immediate neighbors to the right, beneath, and right-beneath. Take the largest (hence the name MAX pooling) of them and load it into the new image. Thus the new image will be 1/4 the size of the old -- with the dimensions on X and Y being halved by this process. You'll see that the features get maintained despite this compression!



This code will show (4, 4) pooling. Run it to see the output, and you'll see that while the image is 1/4 the size of the original in both length and width, the extracted features are maintained!

In [ ]:
```
new_x = int(size_x/4)
new_y = int(size_y/4)
newImage = np.zeros((new_x, new_y))
for x in range(0, size_x, 4):
  for y in range(0, size_y, 4):
    pixels = []
    pixels.append(i_transformed[x, y])
    pixels.append(i_transformed[x+1, y])
    pixels.append(i_transformed[x+2, y])
    pixels.append(i_transformed[x+3, y])
    pixels.append(i_transformed[x, y+1])
    pixels.append(i_transformed[x+1, y+1])
    pixels.append(i_transformed[x+2, y+1])
    pixels.append(i_transformed[x+3, y+1])
    pixels.append(i_transformed[x, y+2])
    pixels.append(i_transformed[x+1, y+2])
    pixels.append(i_transformed[x+2, y+2])
    pixels.append(i_transformed[x+3, y+2])
    pixels.append(i_transformed[x, y+3])
    pixels.append(i_transformed[x+1, y+3])
    pixels.append(i_transformed[x+2, y+3])
    pixels.append(i_transformed[x+3, y+3])
    pixels.sort(reverse=True)
    newImage[int(x/4),int(y/4)] = pixels[0]
```

```
# Plot the image. Note the size of the axes -- now 128 pixels instead of 512
plt.gray()
plt.grid(False)
plt.imshow(newImage)
#plt.axis('off')
plt.show()
```

In the next lab you'll see how to add convolutions to your Fashion MNIST neural network to make it more efficient -- because it will classify based on features, and not on raw pixels.

NNs: From DNN to CNN

In previous lessons you saw how to do image recognition using a Deep Neural Network (DNN) containing three layers -- the input layer (in the shape of the data), the output layer (in the shape of the desired output) and a hidden layer. For convenience, here's the entire code again. Run it and take a note of the accuracy that is printed out at the end.

In [ ]:
```
import tensorflow as tf
mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (val_images, val_labels) = mnist.load_data()
training_images=training_images / 255.0
val_images=val_images / 255.0
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  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=['a
model.fit(training_images, training_labels, validation_data=(val_images, val_labels
```

Your accuracy is probably about 89% on training and 87% on validation...not bad...But how do you make that even better? One way is to use something called Convolutions. I'm not going to get into the details of Convolutions here, but the ultimate concept is that they narrow down the content of the image to focus on specific, distinct, features.

If you've ever done image processing using a filter (like this: https://en.wikipedia.org/wiki/Kernel_(image_processing)) then convolutions will look very familiar.

In short, you take an array (usually 3x3 or 5x5) and pass it over the image. By changing the underlying pixels based on the formula within that matrix, you can do things like edge detection. So, for example, if you look at the above link, you'll see a 3x3 that is defined for edge detection where the middle cell is 8, and all of its neighbors are -1. In this case, for each pixel, you would multiply its value by 8, then subtract the value of each neighbor. Do this for every pixel, and you'll end up with a new image that has the edges enhanced.

This is perfect for computer vision, because often it's features like edges that distinguish one item for another. And once we move from raw image data to feature data, the amount of information needed is then much less...because you'll just train on the highlighted features.

That's the concept of Convolutional Neural Networks. Add some layers to do convolution before you have the dense layers, and then the information going to the dense layers is more focussed, and possibly more accurate.

Run the below code -- this is the same neural network as earlier, but this time with Convolutional layers added first. It will take longer, but look at the impact on the accuracy:

In [ ]:
```
import tensorflow as tf
mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (val_images, val_labels) = mnist.load_data()
training_images=training_images.reshape(60000, 28, 28, 1)
training_images=training_images / 255.0
val_images=val_images.reshape(10000, 28, 28, 1)
val_images=val_images/255.0
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(28, 28, 1)),
  tf.keras.layers.MaxPooling2D(2, 2),
  tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
  tf.keras.layers.MaxPooling2D(2,2),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(20, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['a
model.summary()
model.fit(training_images, training_labels, validation_data=(val_images, val_labels
```

It's likely gone up to about 97% on the training data and 91% on the validation data.

That's significant, and a step in the right direction!

Try running it for more epochs -- say about 100, and explore the results! But while the results might seem really good, the validation results may actually go down, due to something called 'overfitting' which will be discussed later.

(In a nutshell, 'overfitting' occurs when the network learns the data from the training set really well, but it's too specialised to only that data, and as a result is less effective at seeing *other* data. For example, if all your life you only saw red shoes, then when you see a red shoe you would be very good at identifying it, but blue suade shoes might confuse you...and you know you should never mess with my blue suede shoes.)

Then, look at the code again, and see, step by step how the Convolutions were built:

Step 1 is to gather the data. You'll notice that there's a bit of a change here in that the training data needed to be reshaped. That's because the first convolution expects a single tensor containing everything, so instead of 60,000 28x28x1 items in a list, we have a single 4D list that is 60,000x28x28x1, and the same for the validation images. If you don't do this, you'll get an error when training as the Convolutions do not recognize the shape.

```
import tensorflow as tf
mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (val_images, val_labels) =
mnist.load_data()
training_images=training_images.reshape(60000, 28, 28, 1)
```

```
training_images=training_images / 255.0
val_images = val_images.reshape(10000, 28, 28, 1)
val_images=val_images/255.0
```

Next is to define your model. Now instead of the input layer at the top, you're going to add a Convolution. The parameters are:

1. The number of convolutions you want to generate. Purely arbitrary, but good to start with something in the order of 64
2. The size of the Convolution, in this case a 3x3 grid
3. The activation function to use -- in this case we'll use relu, which you might recall is the equivalent of returning x when x>0, else returning 0
4. In the first layer, the shape of the input data.

You'll follow the Convolution with a MaxPooling layer which is then designed to compress the image, while maintaining the content of the features that were highlighted by the convolution. By specifying (2,2) for the MaxPooling, the effect is to quarter the size of the image. Without going into too much detail here, the idea is that it creates a 2x2 array of pixels, and picks the biggest one, thus turning 4 pixels into 1. It repeats this across the image, and in so doing halves the number of horizontal, and halves the number of vertical pixels, effectively reducing the image to 25% of its original size.

You can call model.summary() to see the size and shape of the network, and you'll notice that after every MaxPooling layer, the image size is reduced in this way.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=
(28, 28, 1)),
  tf.keras.layers.MaxPooling2D(2, 2),
```

Add another convolution

```
  tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
  tf.keras.layers.MaxPooling2D(2,2)
```

Now flatten the output. After this you'll just have the same DNN structure as the non convolutional version

```
  tf.keras.layers.Flatten(),
```

The same 20 dense layers, and 10 output layers as in the pre-convolution example:

```
  tf.keras.layers.Dense(20, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Now compile the model, call the fit method to do the training, and evaluate the loss and accuracy from the validation set.

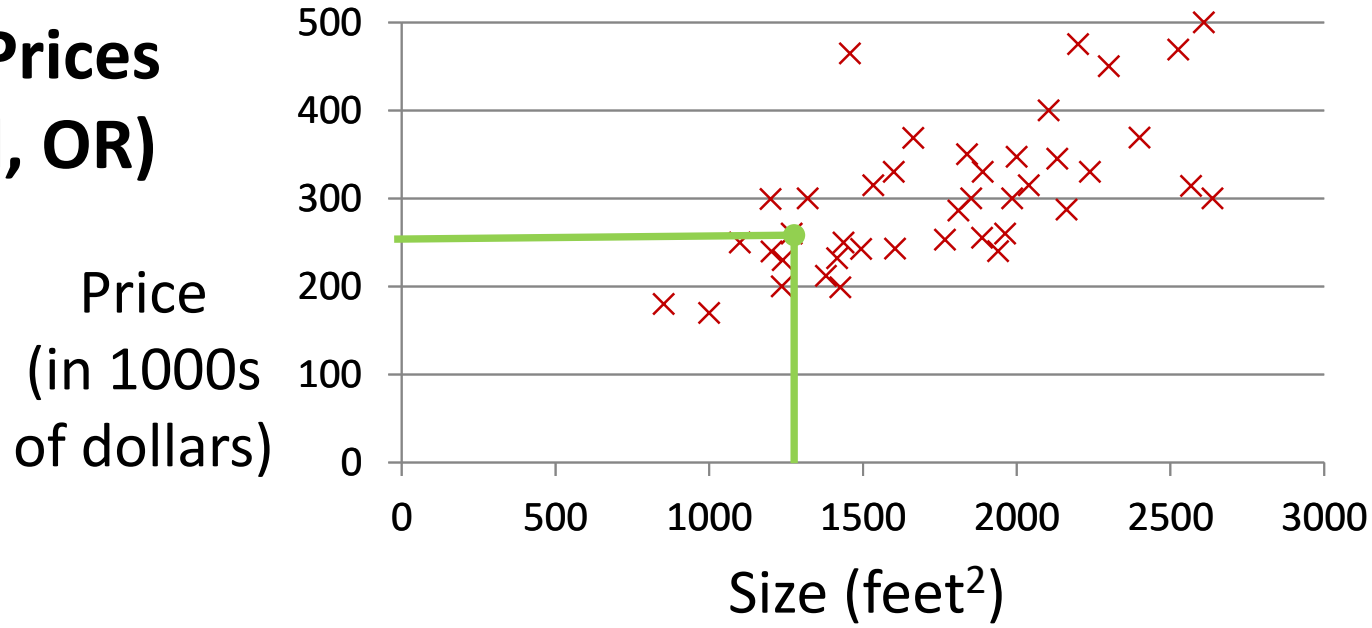
```
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(training_images, training_labels, validation_data=
(val_images, val_labels), epochs=20)
```

Optional Additonal Exercises

1. Try editing the convolutions. Change the 32s to either 16 or 64. Explore what impact this will have on accuracy and/or training time.
2. Remove the final Convolution. What impact will this have on accuracy or training time?
3. How about adding more Convolutions? What impact do you think this will have? Experiment with it.
4. Remove all Convolutions but the first. What impact do you think this will have? Experiment with it.

Linear regression with one variable

Housing Prices (Portland, OR)

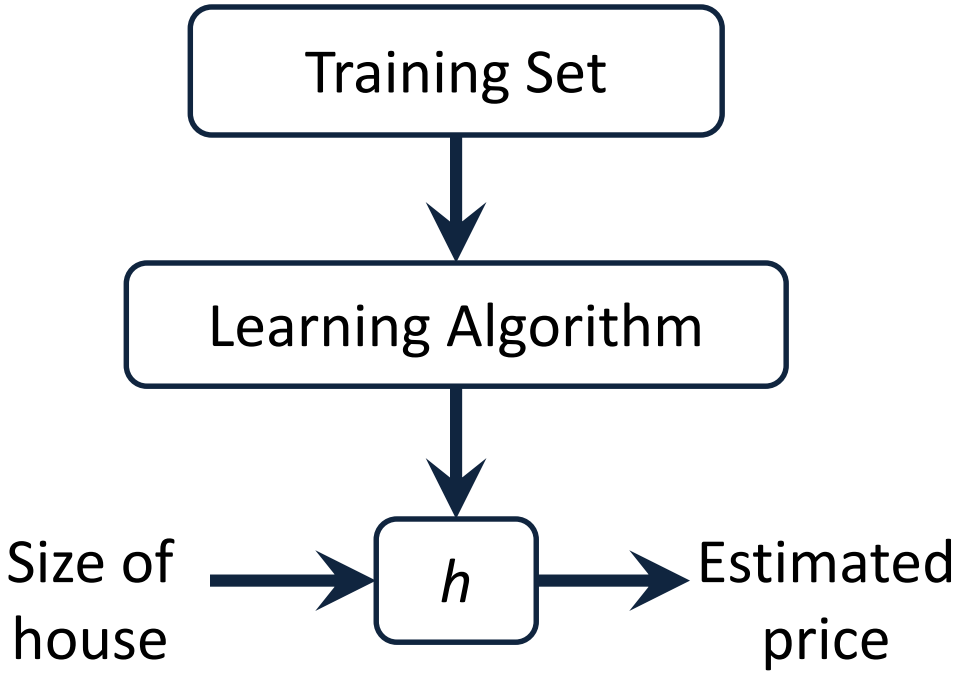


Supervised Learning

Given the "right answer" for each example in the data.

Regression Problem

Predict real-valued output



How do we represent *h* ?

Linear regression with one variable.
Univariate linear regression.

Cost function

Training Set

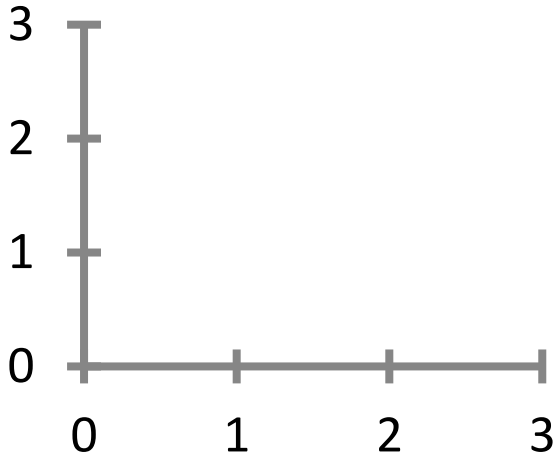
Size in feet² (x)	Price ($) in 1000's (y)
2104	460
1416	232
1534	315
852	178
…	…

Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

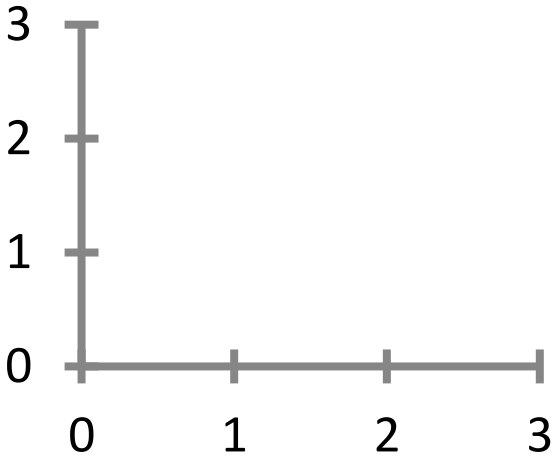
θ_i's: Parameters

How to choose θ_i's ?

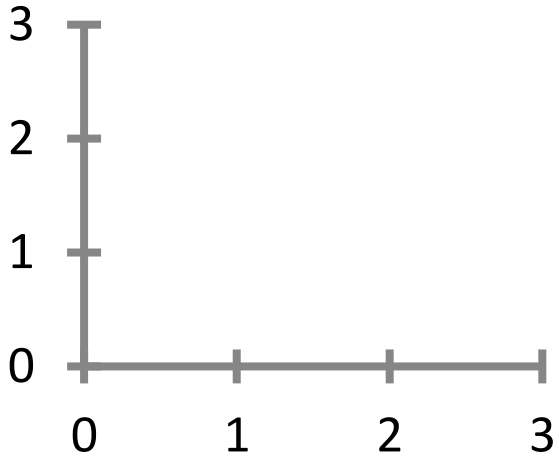
$h_\theta(x) = \theta_0 + \theta_1 x$



$\theta_0 = 1.5$
$\theta_1 = 0$



$\theta_0 = 0$
$\theta_1 = 0.5$



$\theta_0 = 1$
$\theta_1 = 0.5$

Simplified

Hypothesis:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

$$h_\theta(x) = \theta_1 x$$

Parameters:

$$\theta_0, \theta_1$$

$$\theta_1$$

Cost Function:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

$$J(\theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

$\underset{\theta_1}{\text{minimize}}\ J(\theta_1)$

Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

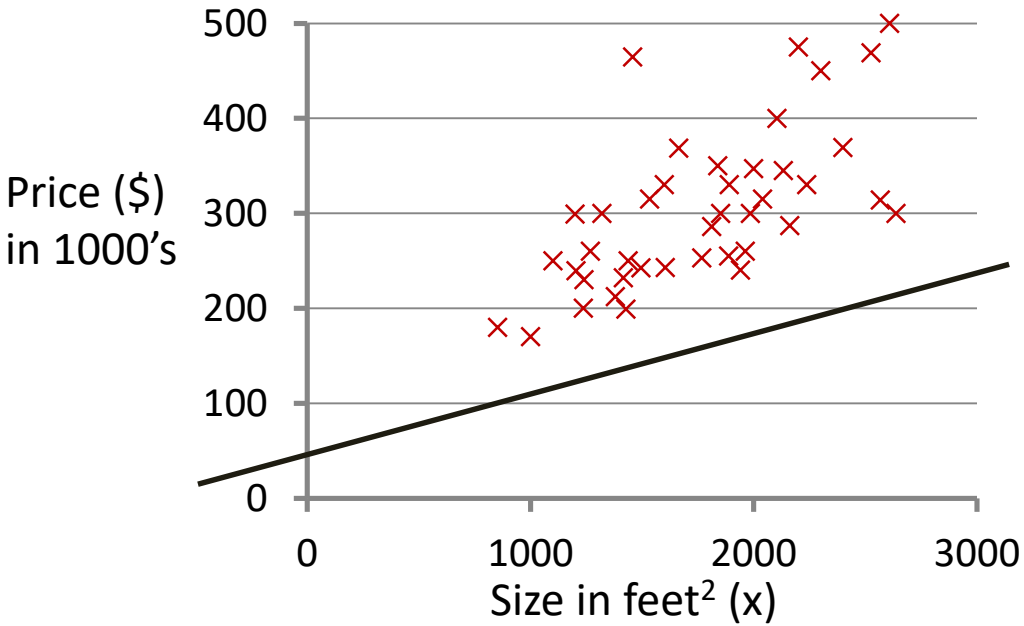
Parameters: θ_0, θ_1

Cost Function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$

Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

$h_\theta(x)$

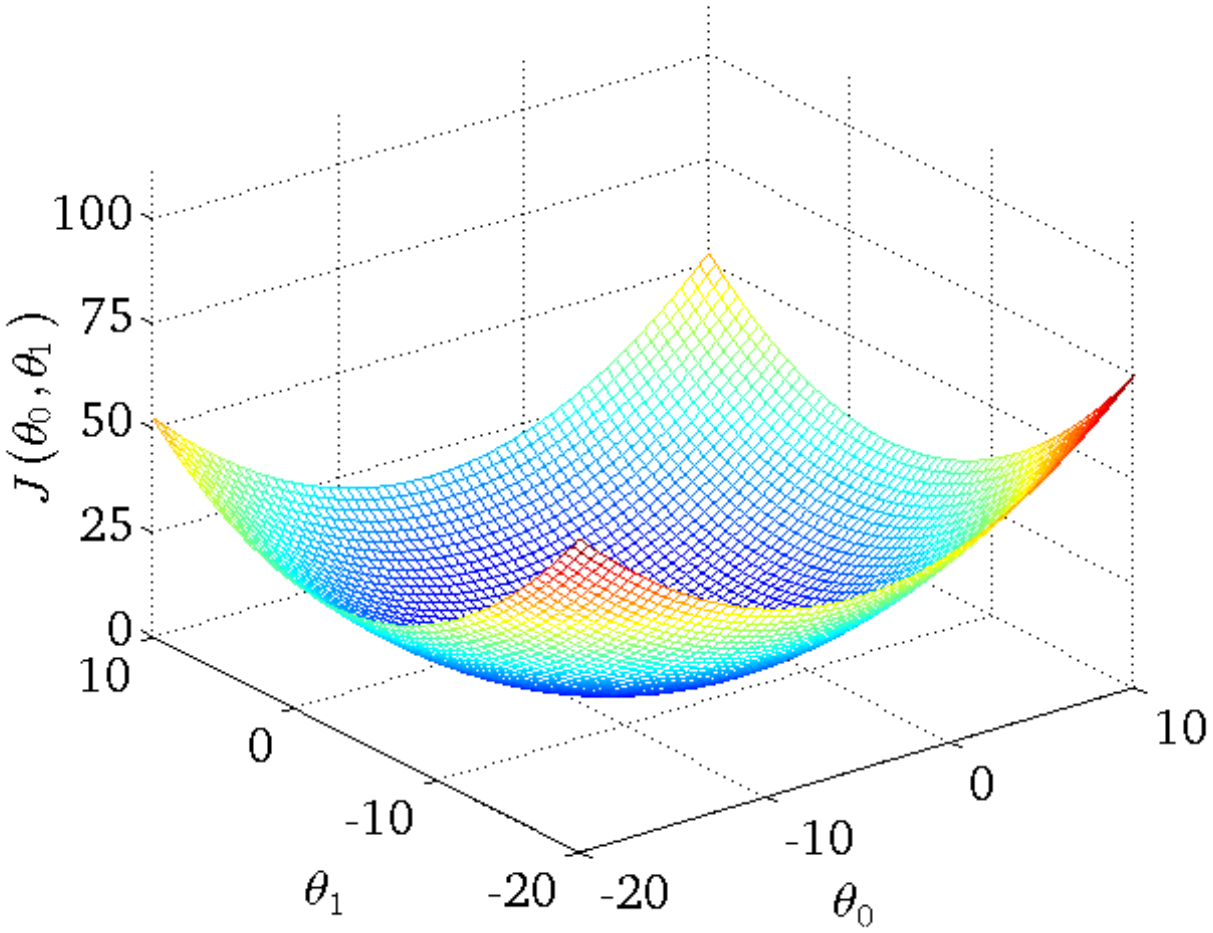
(for fixed θ_0, θ_1, this is a function of x)



$$h_\theta(x) = 50 + 0.06x$$

$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)


100
75
50
25
0
$J(\theta_0, \theta_1)$
10
0
-10
-20
θ_1
-20
-10
0
10
θ_0

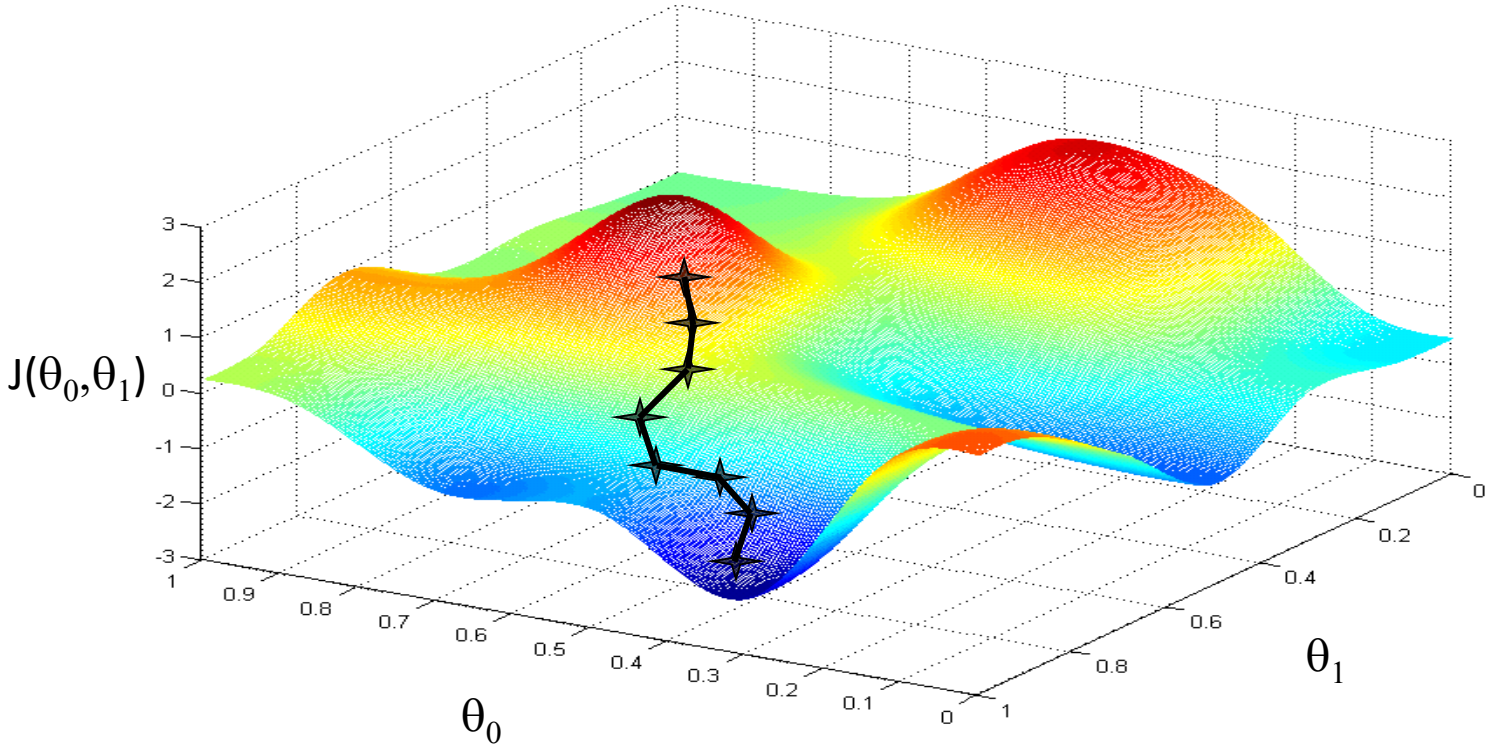
Gradient descent

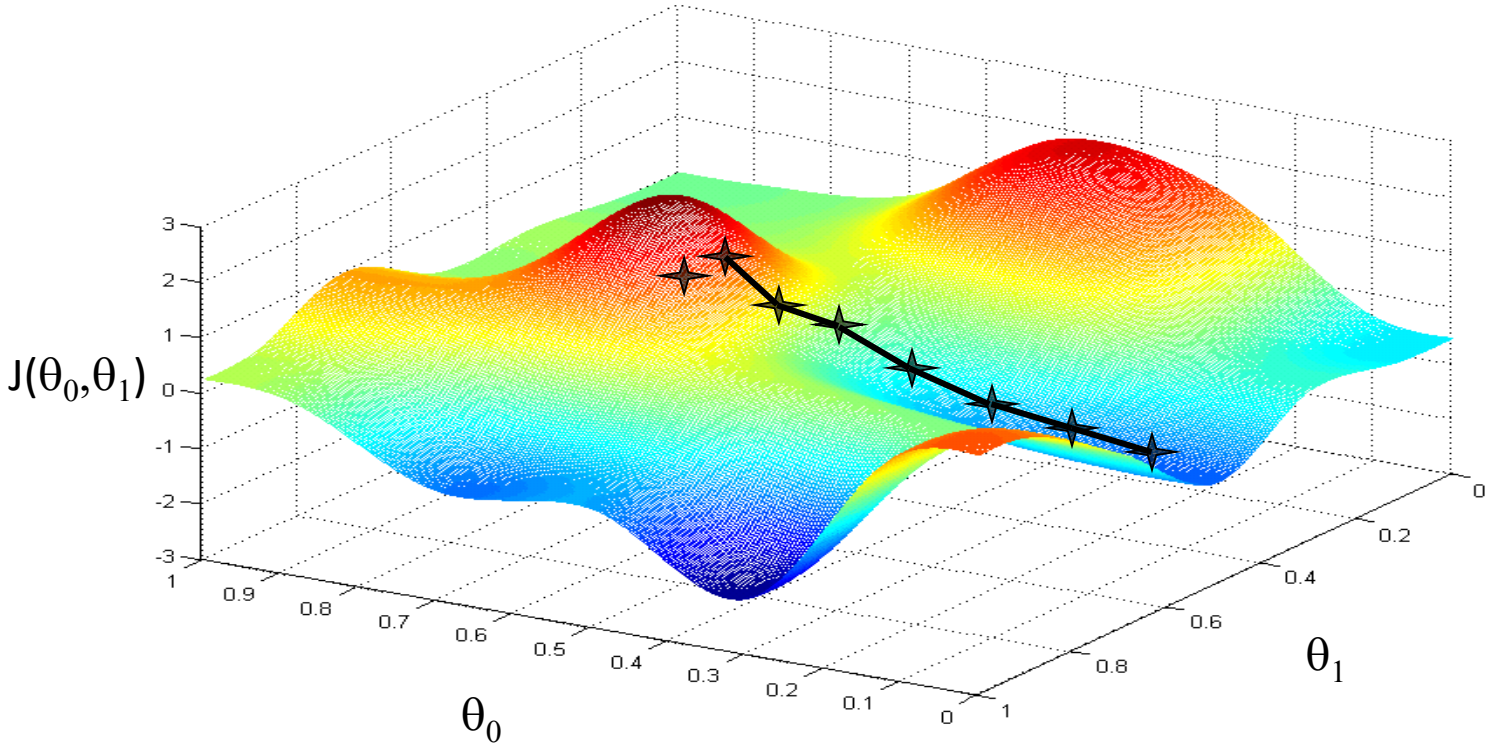
Have some function $J(\theta_0, \theta_1)$

Want $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

Outline:

- Start with some θ_0, θ_1
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we hopefully end up at a minimum


$J(\theta_0,\theta_1)$
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ_0
0
0.2
0.4
0.6
0.8
1
θ_1


$J(\theta_0,\theta_1)$
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ_0
0
0.2
0.4
0.6
0.8
1
θ_1

Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \qquad (\text{for } j = 0 \text{ and } j = 1)$$

}

Correct: Simultaneous update

$$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$
$$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0}$$
$$\theta_1 := \text{temp1}$$

Incorrect:

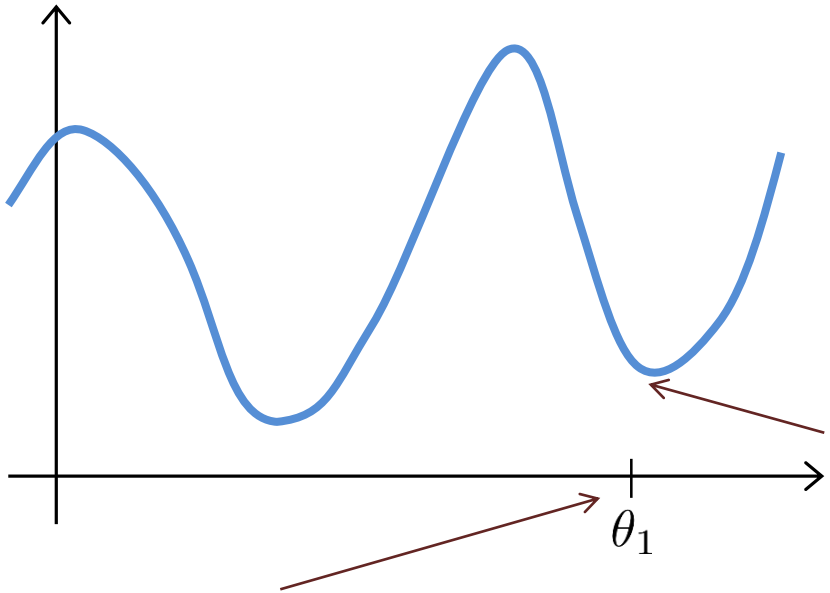
$$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0}$$
$$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$
$$\theta_1 := \text{temp1}$$

Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \qquad \text{(simultaneously update } j = 0 \text{ and } j = 1\text{)}$$

}



θ_1 at local optima

Current value of θ_1

$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

Gradient descent for linear regression

Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

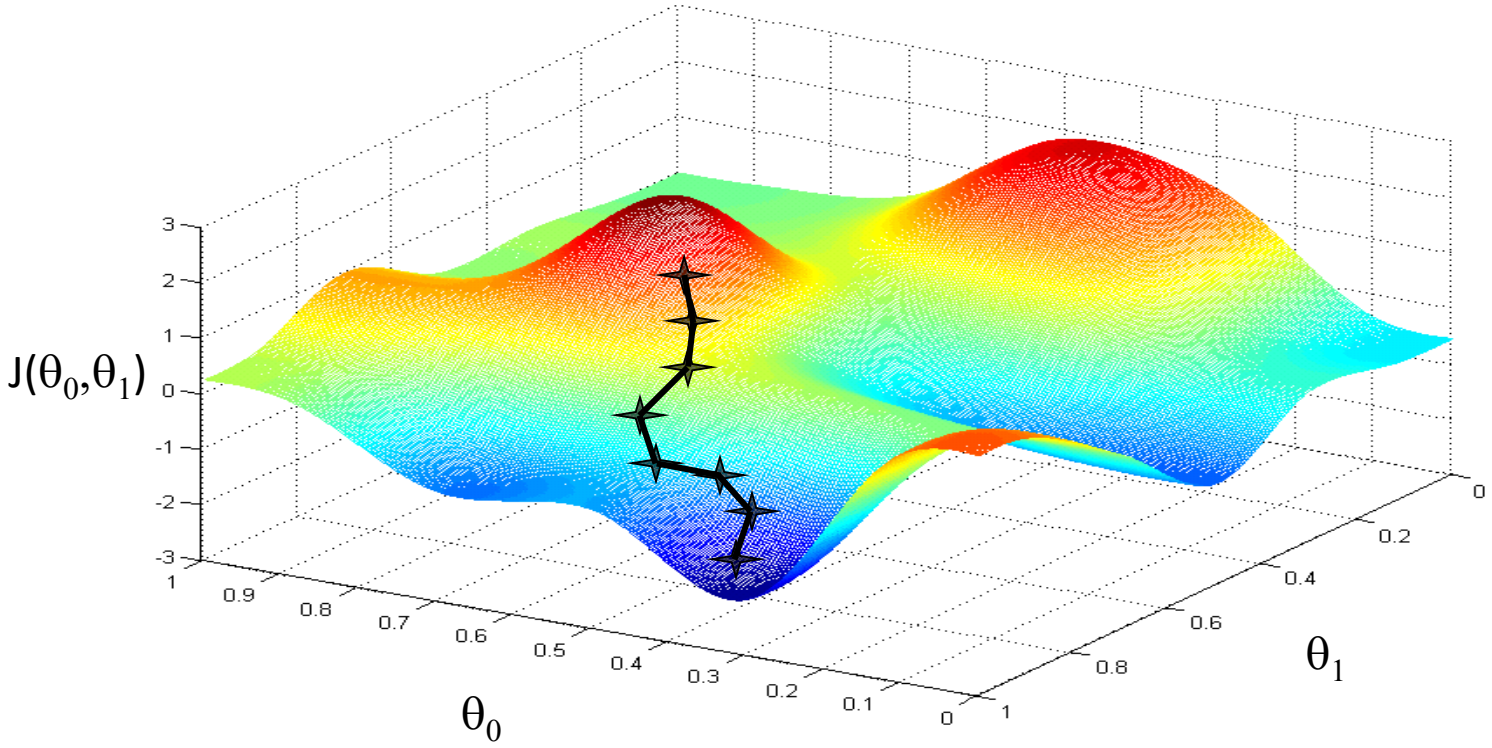
(for $j = 1$ and $j = 0$)

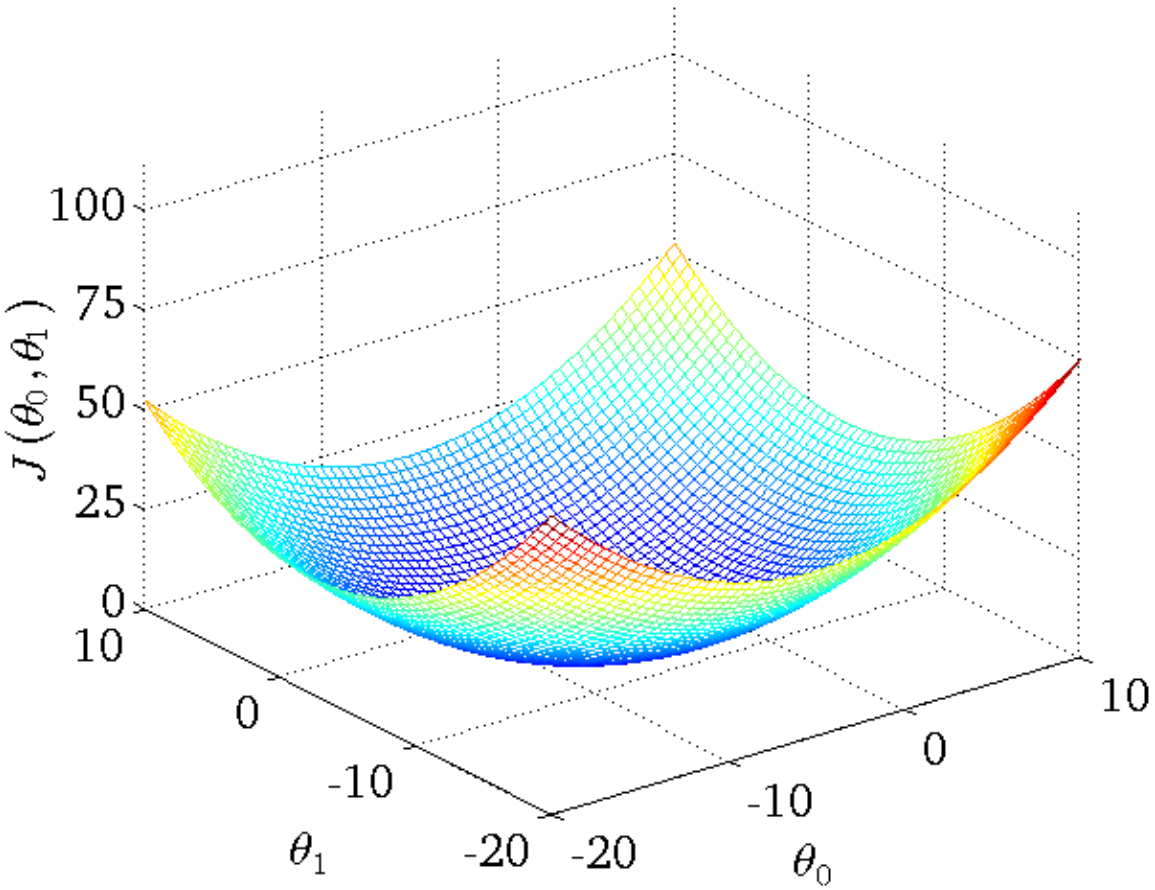
}

Linear Regression Model

$$h_\theta(x) = \theta_0 + \theta_1 x$$

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$


$J(\theta_0,\theta_1)$
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ_0
0
0.2
0.4
0.6
0.8
1
θ_1


100
75
50
25
0
$J(\theta_0, \theta_1)$
10
0
-10
-20
θ_1
-20
-10
0
10
θ_0

Logistic Regression

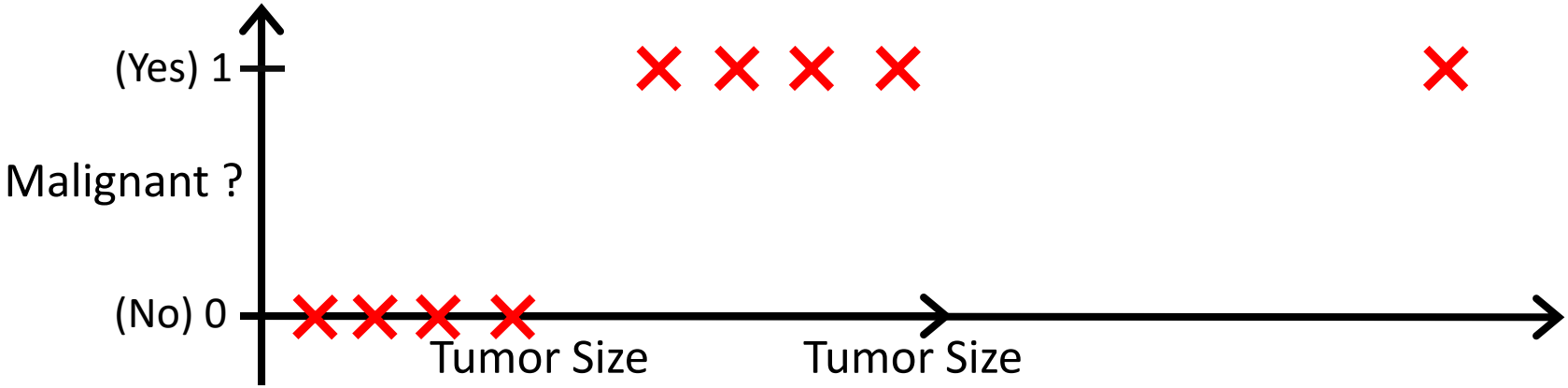
Classification

Classification

Email: Spam / Not Spam?
Online Transactions: Fraudulent (Yes / No)?
Tumor: Malignant / Benign ?

$y \in \{0, 1\}$

0: “Negative Class” (e.g., benign tumor)
1: “Positive Class” (e.g., malignant tumor)



Threshold classifier output $h_\theta(x)$ at 0.5:

If $h_\theta(x) \geq 0.5$, predict “y = 1”

If $h_\theta(x) < 0.5$, predict “y = 0”

Classification: y = 0 or 1

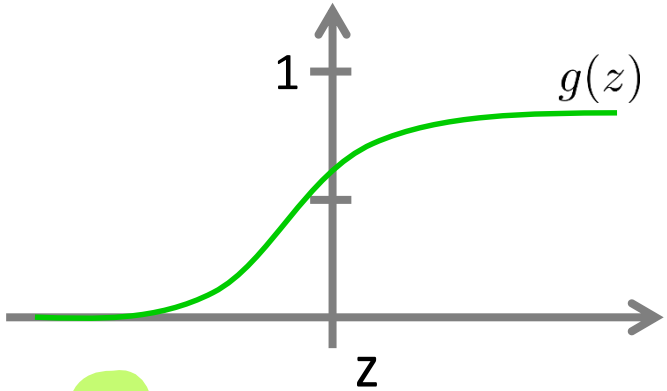
$h_\theta(x)$ can be > 1 or < 0

Logistic Regression: $0 \leq h_\theta(x) \leq 1$

Hypothesis Representation

Logistic regression

$$h_\theta(x) = g(\theta^T x)$$



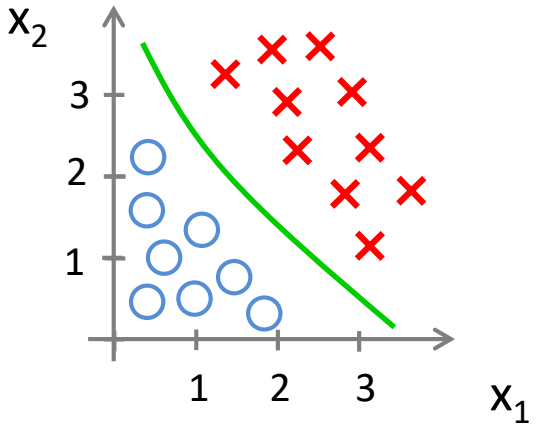
Sigmoid function
Logistic function

$$g(z) = \frac{1}{1+e^{-z}}$$

Suppose predict "$y = 1$" if $h_\theta(x) \geq 0.5$

predict "$y = 0$" if $h_\theta(x) < 0.5$

Decision Boundary



$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

Predict "$y = 1$" if $-3 + x_1 + x_2 \geq 0$

Logistic regression cost function

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

Note: $y = 0$ or 1 always

Logistic regression cost function

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$= -\frac{1}{m} [\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

To fit parameters θ:

$$\min_\theta J(\theta)$$

To make a prediction given new x:

Output $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$

Gradient Descent

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

Want $\min_\theta J(\theta)$:

Repeat $\{$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$\}$ (simultaneously update all θ_j)

Gradient Descent

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

Want $\min_\theta J(\theta)$:

Repeat $\{$

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(simultaneously update all θ_j)

$\}$

Algorithm looks identical to linear regression!

Advanced optimization

Optimization algorithm

Cost function $J(\theta)$. Want $\min_\theta J(\theta)$.

Given θ, we have code that can compute

- $J(\theta)$
- $\frac{\partial}{\partial \theta_j} J(\theta)$ (for $j = 0, 1, \ldots, n$)

Gradient descent:

Repeat $\{$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$\}$

Optimization algorithm

Given θ, we have code that can compute

- $J(\theta)$
- $\frac{\partial}{\partial \theta_j} J(\theta)$ (for $j = 0, 1, \ldots, n$)

Optimization algorithms:

- Gradient descent
- Conjugate gradient
- BFGS
- L-BFGS

Advantages:

- No need to manually pick α
- Often faster than gradient descent.

Disadvantages:

- More complex