Study and Implementation of Artificial Neural Networks in low powered embedded Systems

R.Lokesh Krishna, Vedhansh Yadav

Abstract

The era of machine learning was hugely promoted by the drastic growth of computational capabilities and a high amount of parallelism in such systems. The preexisting CPU and GPU based platforms do a great job in training these neural networks and they fairly satisfy the demands of deployment in real-life cases and mobile applications. Thus, there is a growing demand to deploy compromisable and cost-efficient Machine learning-related solutions in day to day automation and human-free areas of operation. However, the usage of a sophisticated desktop arrangement is perhaps overkill for a certain set of Neural Network-based tasks, as it requires a complete OS, dependency software and unrelated background tasks. Hence, in this paper, we try to study and implement an ANN-based framework on the famous 8bit microcontroller Atmega328 and compare it with preexisting methodologies to solve a certain set of problems. This can be used in relatively low-level ANNbased task like sensor-filtering, small scale autonomous robots, sensor transfusion, etc. We are thereby highlighting the possible areas where we could eliminate the need for a full-fledged desktop arrangement/microprocessor for your ANNbased tasks and prove how a simple microcontroller could be an efficient and viable alternative.

Introduction

In today's world, we could find Machine learning-related solutions everywhere, from self-driving cars, Human face generating GAN's, self-learning robots, game bots, etc. All these rely on huge amounts of computing power and some of them might not even be available to an average man. Attempts have been made to remove this hardware bottleneck by cloud-based services like Colab, Google cloud, AWS at the higher level and single board computer solutions like raspberry pi, Nvidia's Jetson boards for low-level applications. However, there is still an unremovable limitation that the above platforms suffer, which is the need for a full-fledged Operating System running a huge amount of dependencies and software and backend tasks for deploying ANN-based solutions. Though the performance of such systems are unparalleled and are only getting better day by day, we, on the other hand, argue that such amount of additional requirements for these systems might not even be required and is greatly irrelevant for small scale ANN-based tasks. Hence take a different approach towards the problem. We essentially would like to answer the following questions,

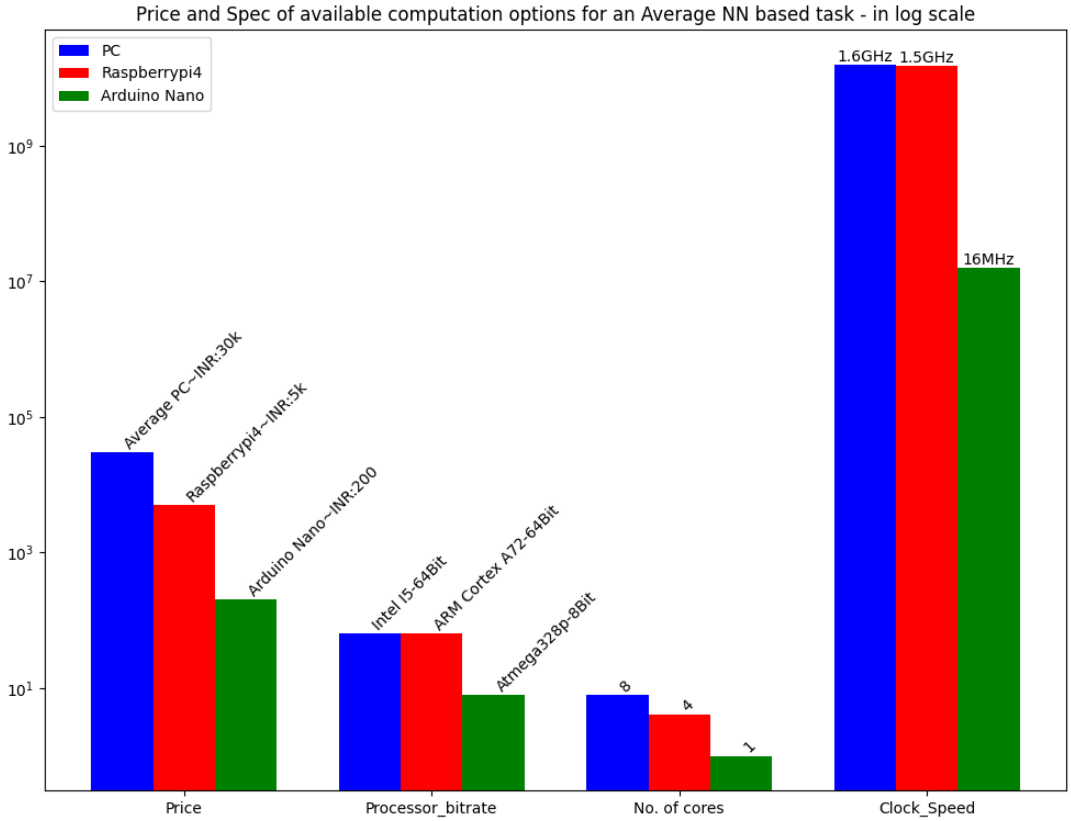
1. Are these sophistications are required for relatively low-level ANN-based tasks?

2. What is the minimum hardware required to deploy an ANN-based solution that has performance comparable to the modern-day desktop?

3. What are the possible trade off's and limitations in realising such solutions?

The answer to our first question proves to be a No. Early stages of AI research happened in the early fifties were the available software was nowhere comparable to out modern-day computers. So, essentially a single program that does the required tasks of an ANN is sufficient

The answer to the next questions needs a detailed analysis of the available nominal computational facilities today.



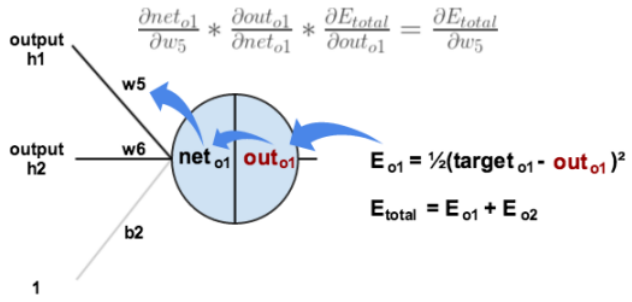
This plot clearly shows the conventional CPU based solutions available to an average user. We thus select the Arduino Nano, a microcontroller powered by the Atmel Atmega 328p microcontroller with the following specifications,

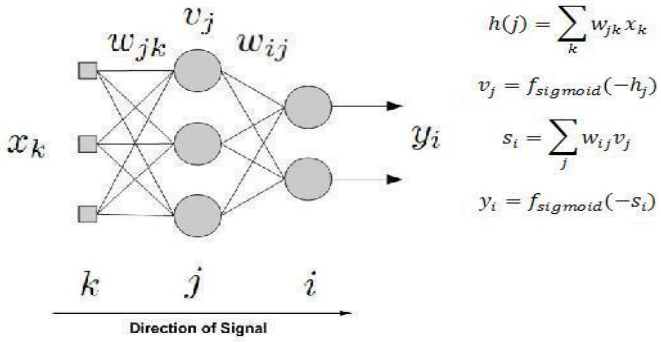
S.No	Feature	Available hardware
1.	CPU	Atmega 328p
2.	RAM	2KB SRAM
3.	ROM	1kb EEPROM
4.	Flash Memory	32 KB of which 2 KB used by bootloader
5.	Clock Speed	16 MHz
6.	Power consumtion	~100mW

The third question is answered entirely by our experiments below were we made a library for Neural Network training and deployment in Arduino platforms and tested it on an Arduino Nano and the results are as follows.

Section – 1
Definitions and Formulations .

I A Neural Network

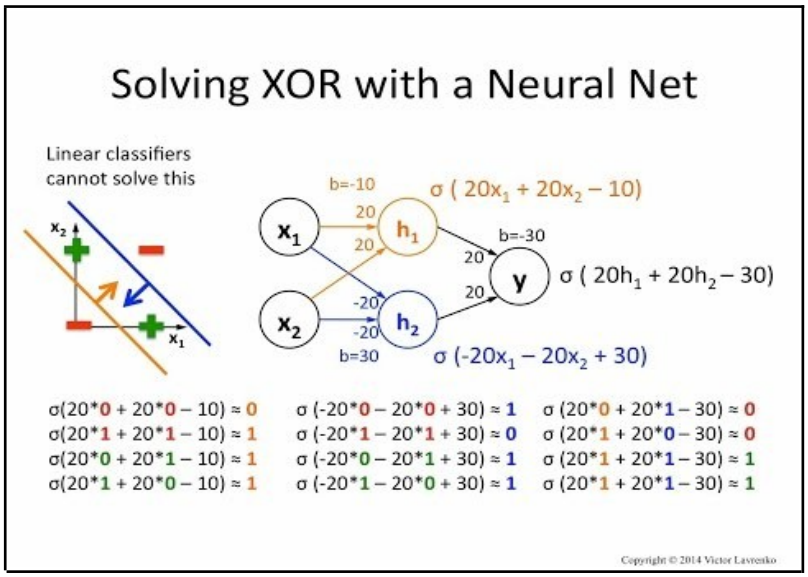




Neural Networks are a universal function approximators that could fit a given data set. There are many architectures of Neural Networks available like MLP, CNN, RNN, LSTM, GAN's etc. As for our needs, a Multilayered Perceptron (MLP) or otherwise known as a Fully connected Neural Networks are enough for low-level regression and classification tasks. We thus implemented a NeuralNetwork class in C++ for the Feedforward (inference) and backpropagation (training). The Backpropagation algorithm is a simple gradient descent performed over all data points.

Section – 2
Experiments conducted:

Experiment -1 , Learning Bit wise XOR operation



Input		Output
A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

A classical problem in ANN research was the XOR problem as learning it using a single perceptron was proved to be impossible by Minksy and Papert in 1969(Minsky, M. Papert, S. (1969). Perceptron: an introduction to computational geometry. A limitation of the single-layer perceptron is that it is only capable of separating data points with a single line. This is unfortunate because the XOR inputs are not linearly separable. This can be observed below as the hyperplanes separating the classes 0 and 1 are non-linear. This required a logistic regression-based approach to classify. We thus validate our neural network implementation with the following XOR test.

Experiment and results:

<photo of arduino and a led>

We wanted to model a 3 input XOR gate with the implementation of our neural network. The network was made of an input layer with 3 nodes,2 hidden layers with 9 neurons each and the output layer with a single neuron. The input for each node in the input layer is either a 0 or a 1 and the output is the predicted output which is either close to 0 or 1 based on the inputs We use a sigmoid activation function and learning rates as 0.3 and 0.06 for the weights and biases respectively.

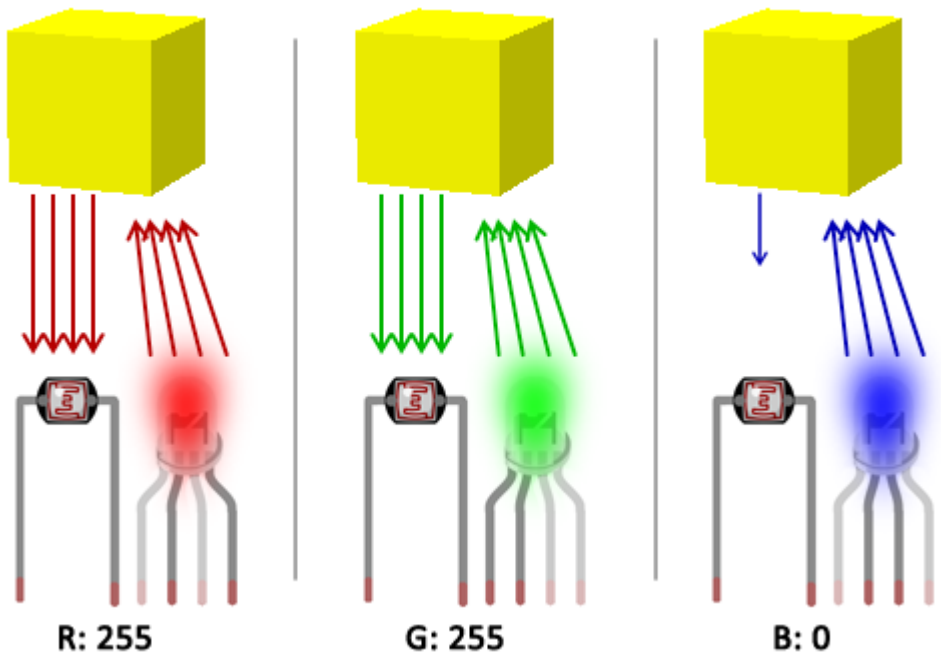
Hyperparameter	Value
Learning rates of weights	0.3
Learning rate of Biases	0.06
Activation function	Sigmoid function
No of epochs	500
Network weight initialization	Random initialization (values for -1 to 1)

We were able to get an accuracy of 100% after training for 500 epochs.This is justified as the network was trained over all possible input values and is aswell tested on the same.

Experiment 2. Multi Colour detection in Arduino using a photo resisitor

To verify the validity of our approach in classification based tasks we conducted a preliminary experiment to check the feasibility. Colour detection is a rather famous problem in Computer Vision. The traditional approach being masking and thresholding in various colour spaces such as RGB, HSV, etc. We, on the other hand, take an alternate approach by detecting almost 10 colours by using a simple photoresistor and a few LEDs. Based on the classical colour theory we train a neural network to effectively separate the colours based on the 3-dimensional values from the light-dependent resistor

Colour Theory:

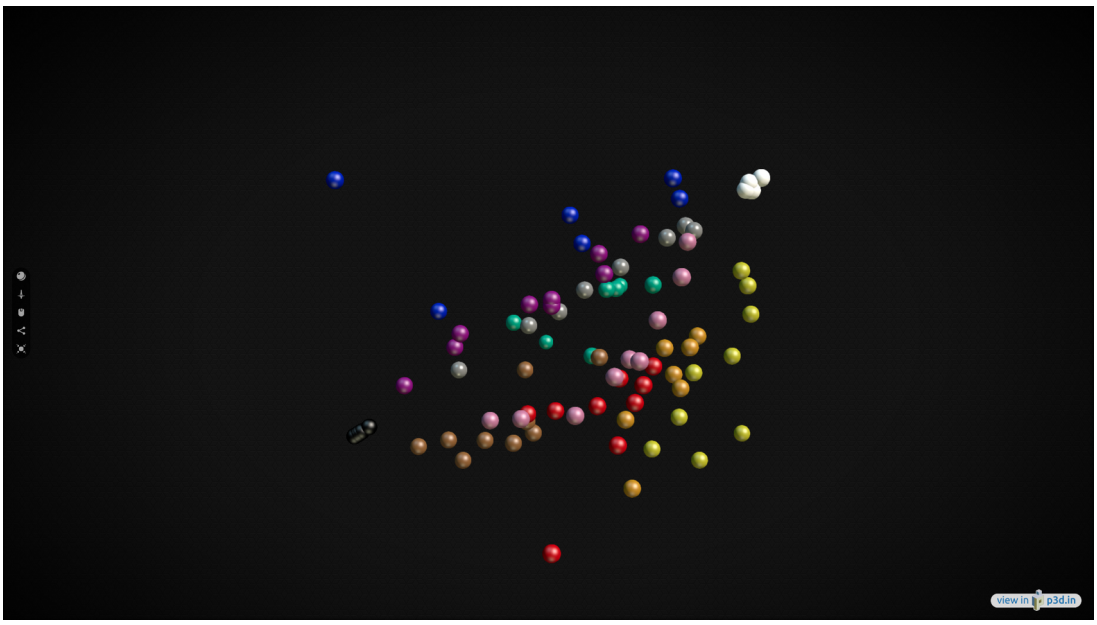


Colour theory states that, when light (of a certain colour) reaches an object, it is reflected according to properties of that object's colour. For example, a red light reaching a yellow object will be reflected according to how much red exists in the composition of that yellow colour, so it is expected to have a lot of red light being reflected, what makes sense when we think of the RGB composition of yellow (essentially red and green). However, when a blue light reaches the yellow object, no strong reflection is expected due to the low presence of blue in the colour composition. It is, however, worth noting that this holds good only when we isolate the sensor setup from ambient light. Hence, we build a covering to prevent the photoresistor to receive light from any other external unwanted sources.

Experiment and results:

<photo of arduino with the photoresistor setup>

Based on the above theory we created a data set with 75 sample points. The 3D visualization of our dataset is illustrated below.



The Neural network has input as the RGB values and output as the probability of the 10 respective colours. For this experiment, we train the Neural Network in the normal desktop and just transfer the trained weights and biased to the Arduino. The sensor values range from 0 to 1024 due to the 10 bit ADC of Atmega 328p. We normalize these values to 0 to 1 and feed into the neural network. The Network contains an input layer with 3 neurons, a hidden layer with 6 neurons and finally an output layer with 10 neurons, one for each colour.

We trained for multiple random seeds and the best accuracy was reached with the following hyperparameter values

Hyperparameter	Value
Learning rates of weights and bias	0.1
Momentum	0.8
Activation function	Sigmoid function
No of epochs	1000
Network weight initialization	Random initialization (values for -1 to 1)
Learning Saturation error	0.001

After training for about 1000 epochs we get an accuracy of 95 % when tested on the actual Arduino hardware.

Experiment 3, Sensor Filtering and Accurate value prediction

Sensor noise filtering is a very common problem that is been faced in almost every real-life application of electronic devices as there are prone to environmental interference. To be specific the problem is more accurately termed as Noise reduction and is the process of simply removing from a signal. In this work, however, we take an alternate learning-based approach to solve this problem. We propose to train a neural network mapping a history of noisy sensor output values to the correct ie actual reading.

The process is twofold and is broken down as follows

1. Train a Neural Network over previous labelled sensor values, ie on a dataset with the input labels as n previous sensor values and output labels as the correct sensor values.

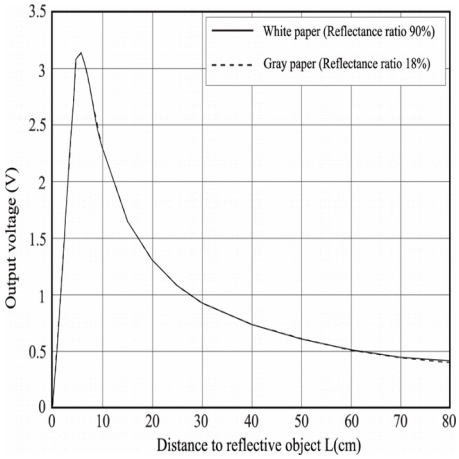
2. Deploy it in on an Arduino with a sensor, feed n past sensor values into the Neural Network to predict the actual value.

We will elaborately explain the Experimental setup below.

Experimental setup

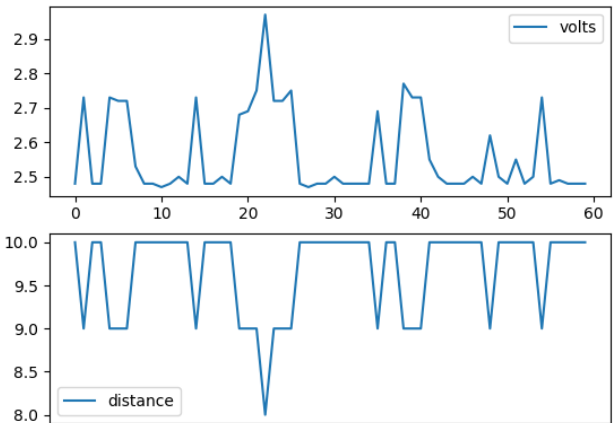
Unlike our previous approaches, in this part, we train the Neural network in the Arduino itself and deploy the Neural Network, in the Arduino as well.

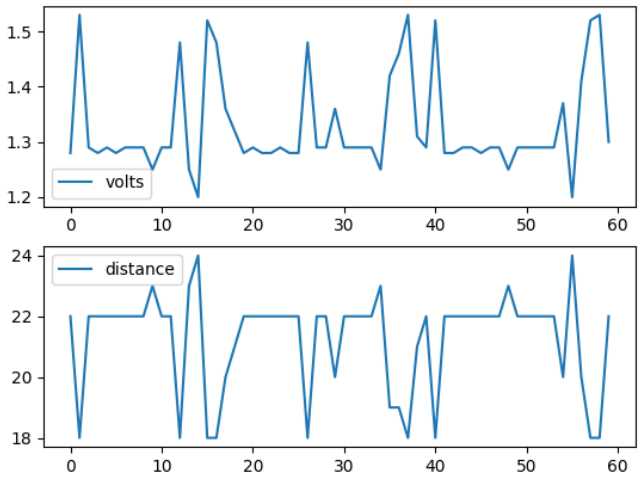
Hardware:

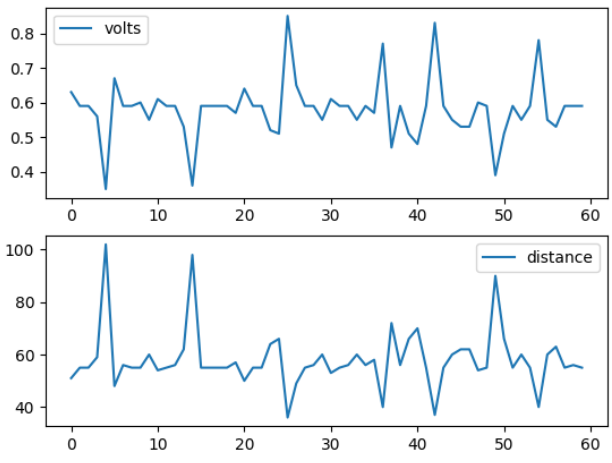


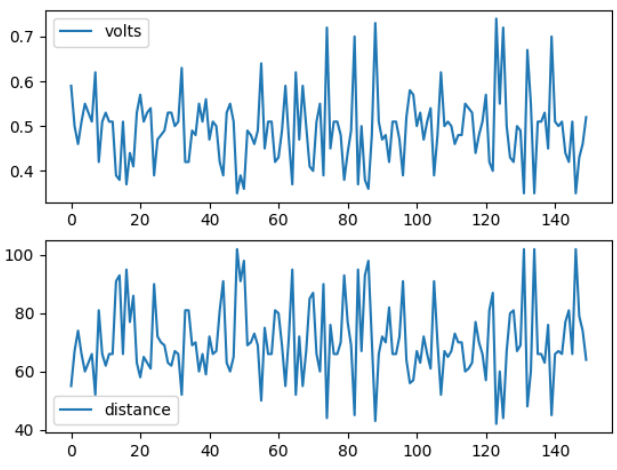
We are using the SHARP GP2Y0A21YK0F, Distance Measuring Sensor Unit as the primary sensor whose signal needs to be filtered. The sensor can measure an object placed in front of it at a range of 10 cm to 80 cm. It is composed of a PSD (position sensitive detector), IRED (INFRARED EMITTING DIODE) and a Signal Processing Unit. The Output voltage Vs distance plot of the sensor from its datasheet is depicted above.The distance is thus measured from the voltage signal using the formula below

Measured distance = 29.988*(output signal voltage)^-1.173









The plots showing the raw measured values for 60 samples of different distances are depicted.It can be clearly observed that the sensor reading are quite noisy and it gets even the more worse as the distance increses.

The Neural Network Used:

As mentioned earlier we are training the neural network in the Arduino board itself and continuously use the trained neural network to predict accurate sensor values. We model the Neural Network as a causal system as the outputs of the network will depend on the present and past values from the sensor. Thus the input to the neural network is the current sensor value and 4 previous values, normalized to a range between 0 to 1. The output of the neural network is a single predicted value which is discretized to belong to a single class. The Neural network has an input layer with 5 neurons and 2 hidden layers with 10 nodes each and finally a single output node. First, we discretize the sensor output as 4 equidistant points in the output space of the senor as 10cm,30m,50cm,70cm and give each of them a class label and a prediction value as follows

Distance Value	Class No.	Prediction value
10 cm	0	0.125
30 cm	1	0.375
50 cm	2	0.625
70 cm	3	0.875

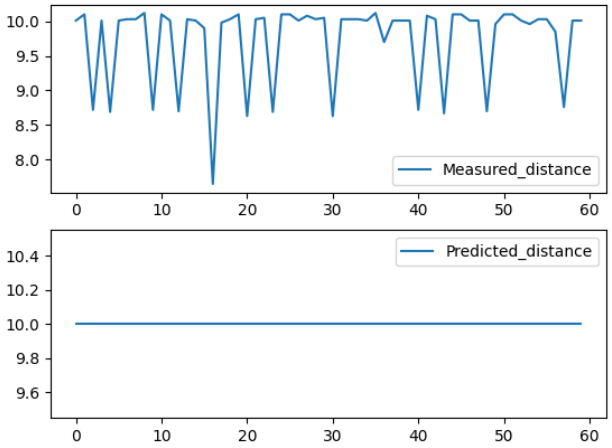
So, if the neural network predicts a value of 50cm as the real values based on the previous 5 measured sensor values, it will be outputting a value of 0.625.

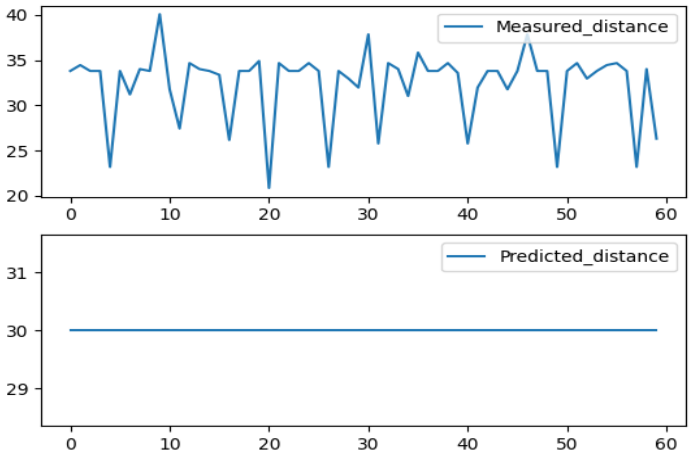
The data set:

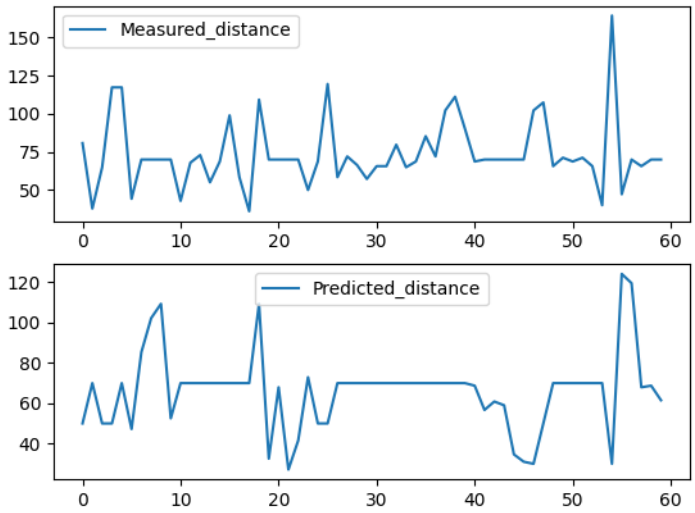
We prepared a data set with 80 data points, with 20 data pairs for each class. This data set was good enough to generalize over for 4 classes but we believe a larger dataset will be required for more classes and accurate continuous space prediction. We require a minimum of 350 epochs to learn this dataset and the training requires a time of about 80 seconds (1min 20 seconds) on the Arduino nano. The hyperparameters are as follows,

Hyperparameter	Value
Learning rates of weights	0.5
Learning rate of Biases	0.1
Activation function	Sigmoid function
No of epochs	>350
Network weight initialization	Random initialization (values for -1 to 1)

Experiment's Results:



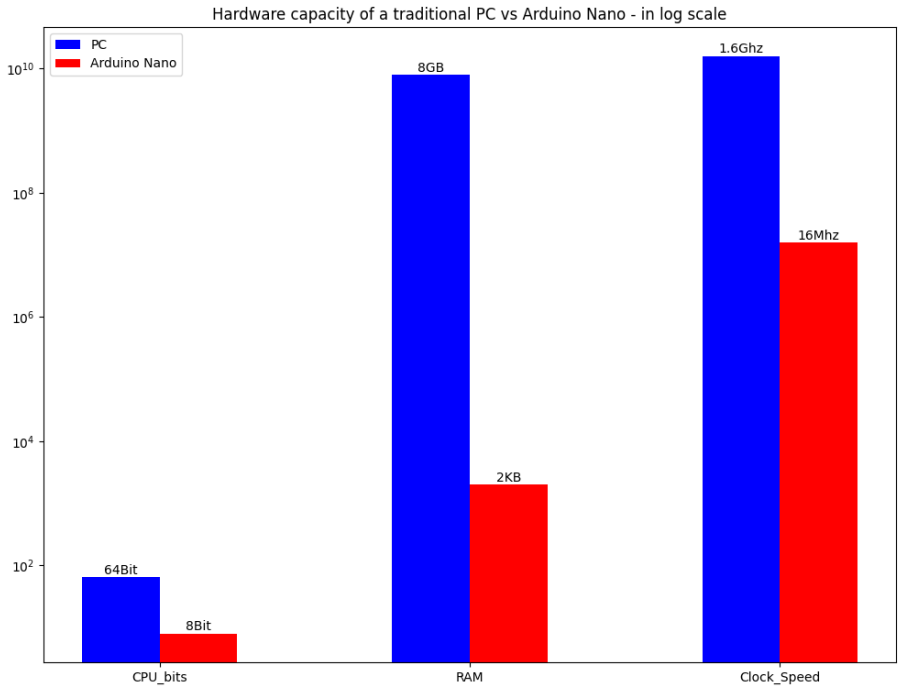




We find the neural network to be greatly accurate in predicting shorter distances (10cm and 30 cm) and wen measured a larger distance it significantly keeps the measurement consistent but is still not robust enough for the greatly oscillating sensor feed. We reason this as a limitation of the sensor and can pretty much be removed by fine-tuning the neural network parameters and more training.

Overall results:

From experiments 1,2 and 3 we compare the performance of our approach with that of a similar neural network implemented in the state of the art framework Pytorch that is run in an intel i5 processor with 8 cores and 8GB RAM.To bring things to perspective, the specifications of the two computer are plotted and tabulated below,



Specificaton	Traditional PC	Arduino Nano
CPU bit rate	64 Bit	8 Bit
RAM	8 GB	2 KB
Clock Speed	1.6 GHz	16 MHz

We conducted each of the above experiment 15 times for statistical consistency and averaged them out as results in the table below. We evaluate the efficiency of our approach based on the following aspects,

1. Power consumed
   The amount of power required to run the arrangement for a second.
2. Training time for 350 epochs
   The amount of time takes to perform the backpropagation for the given Neural Netowork for 350 iterations of the complete dataset.

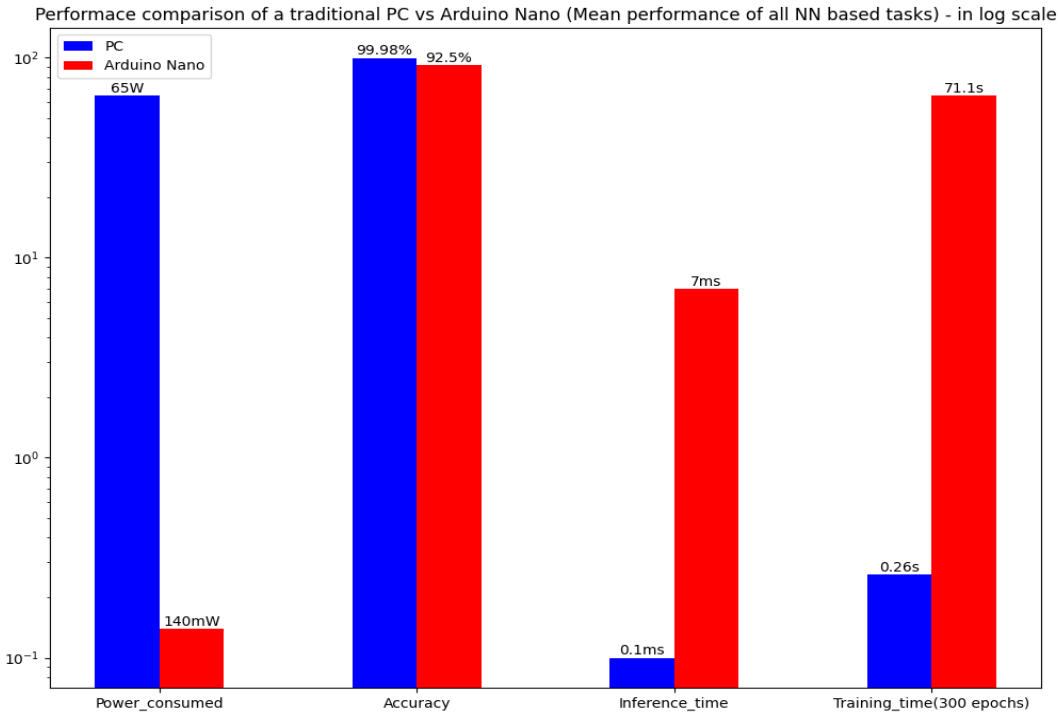
3. Inference time of the neural network

The amount of time taken for one feed-forward step ie the time take for an input vector to flow through the network and come out as the output vector.

4. Accuracy of prediction

It is the ratio of the number of correct predictions/total number of test cases *100

$$\text{Accuracy} = \frac{\text{number of correct predictions} * 100}{\text{total number of test cases}}$$



Performace Index	Traditional PC	Arduino Nano
Power consumed	45 – 65 w	95~140 mW
Training time for n epocs	0.260s	71.1s
Inference time of neural Network	0.0001s	0.007s
Accuracy of prediction	99.98 %	92.5%

Thus from the above results its quite clear that the performance as compared to the hardware capabilities of the individual computers, Arduino Nano is a clear winner when it comes to such low-level problems.

The results are quite intuitive as the microcontroller need not deal with the high-level unwanted background software checks and dependencies. Also, it is worth noting that the implementations in Arduino were in C++ and that in Pytorch was in python but this difference in speed is negligible as a PC has hardware capabilities that are magnitudes higher than its competitor. Though a traditional PC has more accuracy and decrease time values, it is still not compelling enough when compared to the power consumed and the amount of hardware getting wasted for a seemingly simple task.

Conclusion:

In this work, we address a bracket of low-level problems that can be effectively and efficiently solved by using Artificial Neural Networks in low power embedded system like the Arduino Nano, powered by an Atmega328p microcontroller. We are able to reach comparable results to the state of the art ML frameworks in spite of the drastic difference in nature and capability if the hardware resource available. Our work is now limited to simple Vector related tasks and we believe that this can be extended to low-resolution image related tasks by using sensors like the OV7670 CMOS camera unit.