# Intelligent Robotic Arm: FPGA-Powered Neural Network for Automated Identification and Precision Unfastening of Nuts, Screws, and Bolts

Thomas van der Sterren

January 13, 2024

**Abstract**

With an ever-increasing value in robotic systems integrating artificial intelligence (AI), it is easy to get overwhelmed in the choices during the development process. This, sadly, leaves a lot of technological advances on the table. One of which being the integration of FPGA processors in robotic systems. Within this project, a robotic arm will be developed that makes use of an Artificial Intelligence Neural Network running on an FPGA. This will show how such system could be developed and build upon.

## 1 Introduction

Currently a lot of systems use a central processing units (CPUs). These CPUs are found in almost any digital system, ranging from micro-controllers to stronger systems such as mobile phones or computers. Although CPUs are often immensely powerful, they are sometimes not best equipped for the situation. Some systems would benefit from stronger, or more precise system architectures, think for example of Graphical Processing Units (GPUs), which are dedicated to the processing of graphical data. Being able to handle much more data in parallel. The one that will be most investigated during this report would be the Field Programmable Logic Array (FPGA). This architecture has some key differences that would make it stand out for specific use-cases.

The use-case within this report will be to implement a neural network that runs on an FPGA chip alongside a CPU to gain the best of both worlds. This will be accomplished by using a Xillinx Ultra64 board. The goal is to create an implementation of a robotic arm that utilizes a camera to identify the difference between a screw, bolt, and nut. Using that information, a pose estimation can be done so that the end-effector of the robotic arm can be placed to the correct location, finally being able to unscrew the entity. Extra documentation, demonstration and code can be found on the Github. [1]

## 2 Background

### 2.1 The different chip architectures

The market is flooded with different chip architectures, however three stand out the most when it comes to embedded systems used within robotics. The different options would be CPU, GPU, and the FPGA. The difference can be quite elaborate however very vital to the different use-cases. Within this chapter, light will be shed on the different chips and their role within the evolution of robotics.

#### 2.1.1 CPU

The Central Processing Unit (CPU) is a critical and principal component to a computer system. It is tasked with executing general-purpose processing. CPUs are architecturally designed for sequential processing. Which makes them perfect for handling a few complex logic tasks. Generally, a CPU chip consist of multiple cores allowing the CPU to run some processes in parallel. Programming languages that run on a CPU are written in sequential languages such as C, C++, Java, or Python.

### 2.1.2 GPU

The Graphics Processing Unit (GPU) was initially developed for rendering graphics in video games. Their architecture makes them excellent for processing parallel data, as would be needed for rendering information to a large array of pixels in a monitor. Since they have such good parallel capabilities, they are vastly used for (scientific) simulations, machine-learning, crypto-mining, and of course the aforementioned game use. [2]

### 2.1.3 FPGA

A field programmable gate array (FPGA) chip is very different from the previously mentioned chips. They offer a unique flexibility that cannot be found in the other chip sets. Unlike a CPU or GPU, FPGA does not rely on a fixed set of cores. Instead, the chip consists of a large amount of logic gates that can be wired in a custom manner. Thus, within the FPGA chip it is possible to create complete circuits that can run parallel and without the need of clock, and only the parts used will be powered, having a lower power consumption. As this chip is closer to hardware, it also requires a specified language for this, a hardware descriptive language, like Verilog or VHDL needs to be used. A larger downside in contrast to other options is that development for this chip have been known to be more cumbersome. During this project, it is obvious that, although it might be true, it does provide new options within embedded systems. [3]

These are the most common and relevant chips, though not the only architectures seen. Looking at this it is already possible to conclude that the difference of chip makes them ideal for different use cases. Thankfully, it is possible to use multiple different chips within one system, one embedded system. Resulting in a powerful and flexible system that can complete more complex tasks than a single chip would be able to do.

## 2.2 Robotic Arms in Research

Robotic arms have proven to be a valuable tool in various fields, showing the effectiveness and flexibility when preforming tasks. The fields in which they are used range from medical, industry or military, which demonstrates the versatility of robotic arms. The advancements made, through research of robotic arms, guide the automation and enhancing precision in a multitude of domains. Robotic arms have emerged as indispensable tools in various fields,
The robotic arms have been a topic of research since late 1940's. The very first robotic arm to become used was by George Devol in his company Unimation. His robotic arm was the first to be used within the automotive industry. After this revolutionary device was brought to life, more companies started forming around the robotic industry. [4] Ever since then, the robots have become more and more adapted in society and industry. The need to control these arms have therefore also increased. At first robotic systems used pneumatic actuators, and logic gates, while the newer ones run on computer logic on embedded systems.
Even though robotic arms have already made ginormous leaps in various research domains, new challenges keep appearing. Within this report, the challenge of implementation of Artificial Intelligence is addressed, proposing possible solution and illustrate exciting opportunities for future development of intelligent embedded systems.

# 3 System Architecture

This section will provide a general overview of the hardware components, that later will be discussed within the different chapters.

## 3.1 Role of FPGA

The FPGA is the vital part of this system, as it is the embedded part that is used to accelerate the neural network. Within this project, the Ultra96-V2 board is used, which has an Arm-based, AMD Zynq UltraScale+ chip. This chip contains both the CPU and FPGA as well as the necessary hardware

to connect the two. This makes a good flow of data between the CPU and the FPGA. Allowing the overall architecture to make use of the fast capabilities of the FPGA. [5]

## 3.2 Integration of CPU

The CPU will provide a connection to both the Robotic Arm, as well as the camera. There are a couple of elements which run on the CPU, to get more understanding of these elements it is vital to first have a basic grasp of the software used, ROS2, before more information is giving on the elements.

### 3.2.1 ROS2

The CPU will be running ROS2, a publisher-subscriber communication network, allowing code to run in separate blocks (nodes) and communicate among each other through topics. The nodes can subscribe to a topic to read what other nodes publish to it. Logically, a node can also publish to topics. This form of communication allows a one-to-one, one-to-many, many-to-one, or even a many-to-many relation. Meaning information can be easily spread between separate code blocks, which run in separate threads. [6]

### 3.2.2 Nodes: The different sections

There are three different nodes within the system that run on the CPU. The three nodes each have their own task and responsibility. There is one node responsible for communicating with the FPGAs Neural Network, which is written in C++ code. One node will be responsible for moving the robotic system and reading the camera data. Which is written in Python and connected to the hardware using Serial (via USB). Finally, there is one node responsible for pose estimation of the element that needs unscrewing, which is written in Python and makes use of libraries to identify linear patterns. In the next section, a representation of the connection will be delineated.

## 3.3 Connection of components

As this system uses may different components a global architecture is drawn out in Figure 1. In this figure, one can clearly see the hardware on the left side, consisting of the Robotic Arm and the End-Effector (EE) motors as an output of the system, and the End-Effector Camera as an input to the system. This data is connected to the CPU, using a serial connection via USB. This data flows through to the first node, being the `Robot Driver Node`. The `Robot Driver Node` will send the camera data it collects and prepares, to the `/Capture` topic. This topic is subscribed by both the `FPGA Communication Node` and the `Pose Estimate Node`. The `FPGA Communication Node` will send the camera image to the FPGA through the AXI lite bus. The FPGA will process the image using the Neural Network, which will later be discussed in section 4. The FPGA returns its verdict back to the `FPGA Communication Node` which propagates it to the `/Result` topic. The final node in the system is the `Pose Estimate Node` which uses both the image as well as the result of the identification neural network, to estimate the orientation of the element. The orientation is needed by the `Robot Driver Node` to position the end-effector, it gets this through the `/Pose_EE` topic. Like that the cycle is closed.

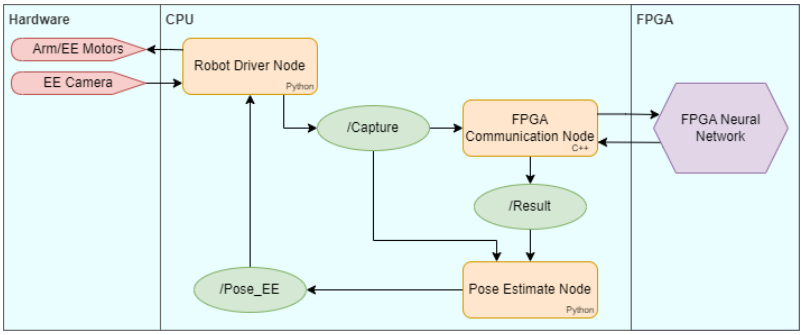


Figure 1: Schematic overview of the general system architecture.

# 4 Neural Network

Neural networks are an ever-growing field, bringing many challenges with it. One of the challenges that research in neural networks still face is efficiency and power consummation. This also brings the possibility of using FPGA to solve this issue. [7] Through many documentations and tools online, the development of a simple neural network is straightforward. Later it will also be discussed how to implement the custom network inside an FPGA.

## 4.1 Development

When developing a neural network, the size is something to always consider, however, it is a topic that is very much discussed. So far there are some theoretical approaches for choosing a sizing, though in practice it often boils down to trying different options. For this basic principle, the neural network configuration that was chosen is schematically shown in figure 2. Which is a network with 768 input, 4 layers, which are finally connecting to the output. The layer construction consists of the first layer being 80 nodes, the second layer being 32 nodes, which moves down to the third layer consisting of 16 nodes, finally ending in three nodes. The final node layer connects to the prediction. All these layers are inter-connected using weight classes, these classes describe how much influence the previous node, will have another node. A higher weight means it is more likely to pass a value to the linked neuron/node.

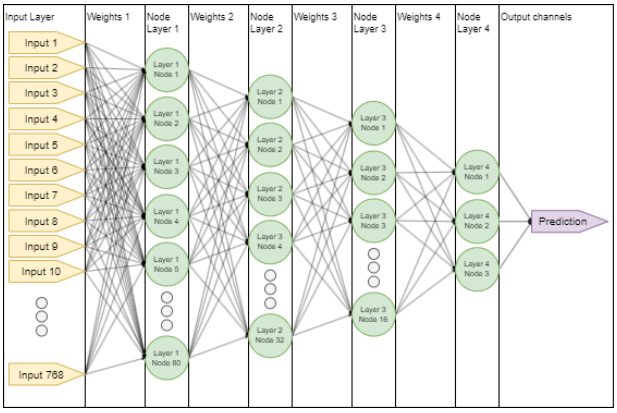


Figure 2: Schematic overview of the Neural network designed.

## 4.2 Data Collection

A exceptionally large and time-consuming job within machine learning, and making these networks is data collection. To train the network which part is which, it requires a lot of examples to work from. The more data that is available, the better the network can train. This is one of the reasons that data is becoming a more import aspect in our daily life. To acquire such large, and different data there are multiple different methods. It can be generated using software, where using simulations, many different angles of the object a rendered. Another method is to generate a large dataset of real images. The preference is always to make images as closely resembled as to what the purpose will be, however, to achieve the large amount of data to optimize the learning, a mix can be made between synthetic data and gathered data.

### 4.2.1 Real Images

In figure 3, an image shows the setup that was used for gathering the real data. This setup consisted of a bracket that is attached to a motor. At the top of the bracket is the camera pointing down. One important aspect to note is that prior to collecting the data, the focal length was adjusted for its purpose. The reason why this is clearly stated is that once again, the aim is to get the best data that would match the real goal. Since the robotic arm is going to identify the entities at a specific height, it is possible to fix the focal length in place. This resulted in significantly better images. The

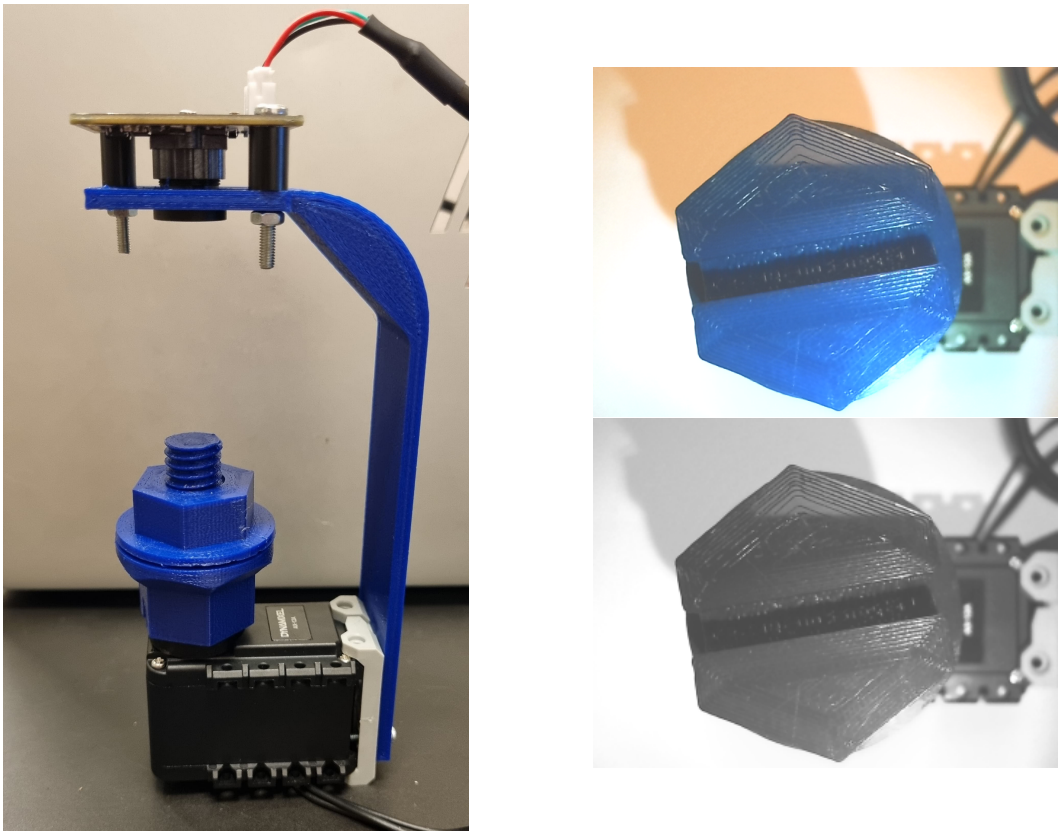

Figure 3: A) Data collection through hardware setup. B) Example images collected. C) Image after compressed and made to Black-White for the network.

reason the motor is connected to the setup, is so that it is possible to gain a lot of diverse data, by successfully integrating a script. The script would rotate the motor an increment at a time, hold it still for an image, and then rotate to the next position. This was done for the different items, with different lighting conditions to create a large dataset.

#### 4.2.2 Synthetic Data

Another solution to increase the dataset was synthetic data. Like described earlier, synthetic data is data that is created virtually. Allowing for an easier, larger amount to be generated. For this project, synthetic data was created using Blender and Unity. Blender is a software used to create or edit models. The entities STL files were loaded into Blender to change color and scaling, after which the model could be loaded into Unity. Unity, often used for developing games, is also a great tool for work such as this. The camera, lighting and model position can be changed using the C# scripts, creating different renders of the entity. This can be seen in figure 4. Due to the substantial number of diversities of data, it would make it possible to train the data on a larger variety of data, giving a higher likelihood of succeeding.

### 4.3 Training

When a dataset is collected, and a neural network design is thought out, it is possible to develop the network. Using existing libraries within Python, generating a network as intended is quite trivial. Within this project, the TensorFlow library was used, allowed a simple script to be developed that would, train, test and output the neural weight classes. The results of the training will later be discussed in section 8. The essence of the script is quite simple. The script would first import all the images and classify them with their outcome value. So, as an example, screws would be value 0, bolts 1 and nuts 2. After this classification is done, the images would be split, part for training data, another part for testing data. The reason this is split, is that it would be unfair to test the network

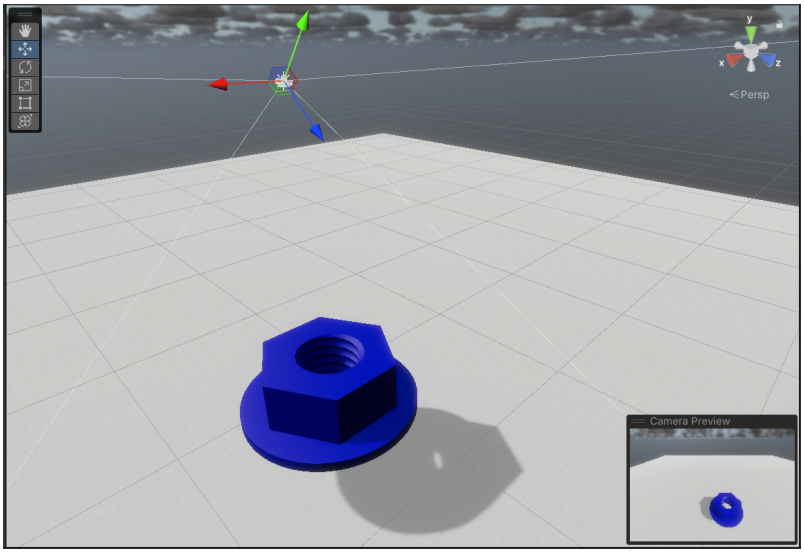


Figure 4: Unity interface. In the bottom right is a capture of the output.

on the exact same image used for training. The training and testing data is both randomized and the training is started. After the training has successfully been done, the network would be tested. This testing shows how accurate it is when giving data. Due note that when the actual situation differs from the testing and training data, the accuracy cannot be exact. Thus, this accuracy is not an exact representation of the real-world scenario. After the data was trained the weight classes can be exported and will be used within the next step. The generation of the matrix calculations within Vitis HLS to be ran from the FPGA.

# 5 Vitis HLS, Vivado and a Linux Environment

This section will describe the development of the FPGA part. This part will be responsible for running the neural network trained in the previous section, by running a series of matrix multiplications.

## 5.1 Generation of an IP

The development of the IP (intellectual property) used within the FPGA development environment, Vivado, is developed using Vitis HLS. This software allows the user to write in a C code architecture which Vitis HLS will convert into a FPGA usable language, such as Verilog, or in the case of this project, VHDL.
The development of the script is based upon the script provided in the repository mentioned earlier in section 4 [8]. The script from the repository was altered to allow for an extra layer and different layer sizes. Once the layout of the script is generated to house the size of the network intended in this project, the learned data is added. To do this, the resulting weights from the training will be implemented inside the script. Throughout the script are also multiple lines starting with `#pragma`'s short for pragma directives. These lines give extra commands to the compiler. In this case, the pragma directives are used to initialize ports within the HLS block, as well as give commands on where the script can work more parallel.

### 5.1.1 Development for PYNQ

In the beginning of the project, the neural network was developed for the PYNQ board and a bare-metal test was done, before continuing to the Ultra96 board. The network was initially developed using primarily the BRam. Later it was found this was not preferred for the Ultra96. Since the aim was to run this system on a custom Ubuntu version with an integration to the FPGA, the Vitis HLS script was altered. In section 8.2.1, results, a brief discussion will be held about this topic.

### 5.1.2 Development for Ultra96

As mentioned earlier, there are multiple pragma directives that could be used to instruct the compiler to build the IP differently. As the BRam was not the solution used when working with the Ultra96 and the ROS2 environment, a different pragma was used to create a direct AXI_lite bus as seen in figure 5. The AXI lite bus is connected to a ROS node which handles the data to send to the FPGA logic block. As schematically displayed in section 3.

```
int nn_inference(float input_img[n_inputs]){
//#pragma HLS INTERFACE mode=s_axilite port=return
#pragma HLS INTERFACE s_axilite port=return bundle=AXI_CPU
#pragma HLS INTERFACE s_axilite port=input_img bundle=AXI_CPU

//#pragma HLS INTERFACE mode=m_axi port=input_img

//#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=input_img

    ap_fixed<32,24> fp_input_img[n_inputs] = {1.0};
    float_to_fixed(input_img, fp_input_img);

    ap_fixed<32,24> temp_output[1][n_layer1] = {1};
    ap_fixed<32,24> temp_output2[1][n_layer2] = {1};
    ap_fixed<32,24> temp_output3[1][n_layer3] = {1};
    ap_fixed<32,24> temp_output4[1][n_layer4] = {1};
    ap_fixed<32,24> prediction = -1;
    int prediction_int = -1;

    hwmm_layer1(fp_input_img, weights::layer1_weights, temp_output);
    hw_act_layer1(temp_output, temp_output);
    hwmm_layer2(temp_output, weights::layer2_weights, temp_output2);
    hw_act_layer2(temp_output2, temp_output2);
    hwmm_layer3(temp_output2, weights::layer3_weights, temp_output3);
    hw_act_layer3(temp_output3, temp_output3);
    hwmm_layer4(temp_output3, weights::layer4_weights, temp_output4);
    hw_act_layer4(temp_output4, prediction);

    prediction_int = prediction;

    return prediction_int;

}
```

Figure 5: Implementation of the AXI lite bus into the Neural Network on the FPGA.

Figure 5 also clearly shows the different layers that the information is passed through with the different weights. Resulting in the outputted prediction_int.

The code developed in the Vitis HLS environment was tested using testbench before continuing to the exporting and preparation for the Vivado development. Later in section 8, the results of these tests will be discussed.

## 5.2 Using the PL

After synthesizing the Vitis HLS, and testing it, the next step will be implementing the newly created HLS IP block into Vivado. In figure 6 shows the block diagram of the Vivado implementation. This shows how the AXI bus of the Ultra96 is connected to the developed HLS IP block using the port developed earlier. After the block diagram was successfully built, a synthesis could be ran, followed by the bitsteam generation. This bitstream would then be used in the next parts of the process.

## 5.3 Connecting to the PL

Using Vitis, not to be confused with Vitis HLS, a script can be written that connects the CPU environment with the FPGA environment that was developed in the previous step. The bitstream that was generated in the previous step would be exported to a Xillinx Support Achieve (.xsa), holding the hardware specified information, which, once imported in Vitis, would allow a bare-metal test. The point of this test is to see if it is possible to connect the FPGA to the CPU the way that was intended. During this process failures occurred. The exact issue will be discussed in the section 8. Though multiple attempts to debug the issue by uncommenting was undertaken. Although it became clear the issue lied in the UART port, it did not become clear whether the error was within the Vitis libraries, or the Vivado implementation. Thus, eventually, the building of a bare-metal test was skipped.

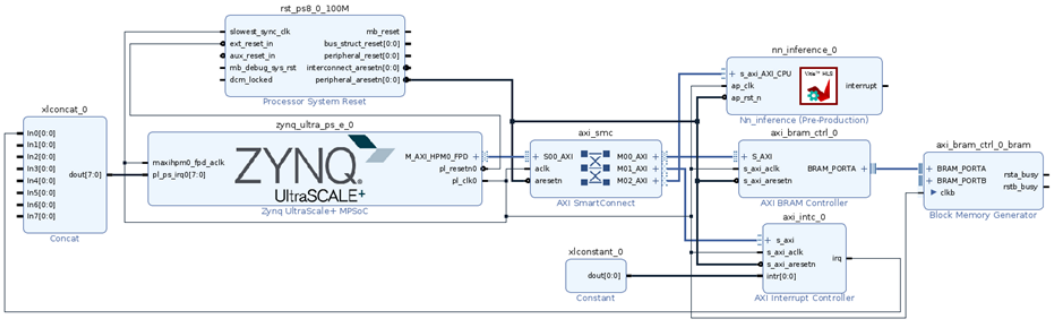


Figure 6: Block diagram of the HLS IP block and the Ultra96 system in Vivado.

### 5.4 Ubuntu Linux Build

Using the files generated with the Vitis HLS and Vivado software, the process to build a Linux distribution is still possible. This, being done by PetaLinux. During the process of the development of the Linux distribution, another error occurred. Halting the process for the FPGA.

## 6 Pose Estimation

Pose Estimation in itself is an entire study. In this project, only a fraction of the capabilities is being utilized. Thankfully, just like with the neural network, there are libraries that will help support the development of this work. The primary library used within this section is OpenCV [9].

The goal of pose estimation is to find the orientation of the entity that is in the image being investigated. In the case of this report, which can either be a screw, bolt, or a nut. Since there is only a 2d image, it is important to extrapolate details from that information. Although the rough steps are all the same for each entity, the first and last step will vary per entity. It is for that reason that this part of the architecture also needs the entity identification, next to the raw image.

### 6.1 General steps

Although the screw requires some steps prior, to detect the center, and the last step of choosing the primary line is per part individual, the rest of the steps are identical. In figure 7 the steps of the screw are displayed and will be used as a reference to explain how the steps are implemented.

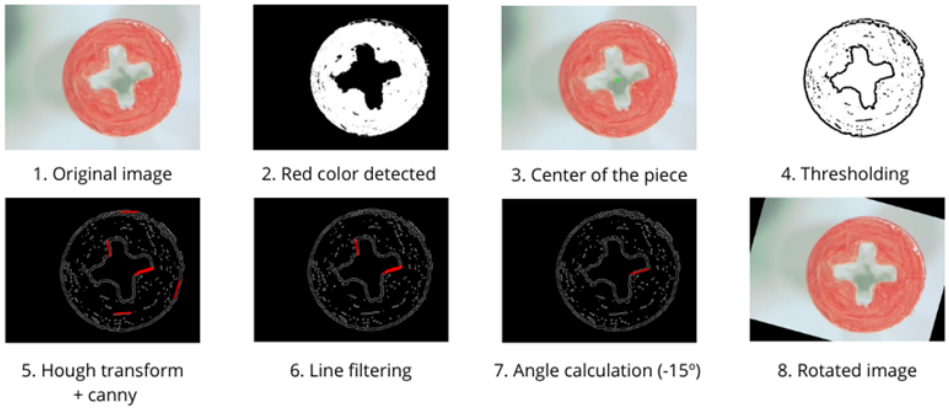


Figure 7: The steps for the pose estimation for the Screw entity.

### 6.1.1 Center finding

Only for the screw, the center is needed. The center of the entity is needed to filter in a later step. To find the center of the entity, first a threshold is overlaid to create a black and white image. Using the black and white image, it is possible to find the middle of the 'white' pixel cluster. This inevitably is always the center of entity.

### 6.1.2 Thresholding

The next thing (or first for the other entities) that is done, is to use thresholding to distinguish the parts top surface from the rest of the image. Thresholding is done by identifying the color and/or brightness of a pixel and comparing it to a reference. Using this the image can create an outline of where the pixels changes from above the reference to below the reference. Leaving an outline or contour of the entity.

### 6.1.3 Canny and Hough transform

Using the Canny detection, the edges are detected. This results in a black image, with white edges. The result of this will be fed into the Hough transform. The Hough Transform is used to identify circles and straight lines in the image. The idea of this step is to parameterize the different lines within the entity image. Using these lines, with known properties it is possible to continue to the next step.

### 6.1.4 Filtering and selection of lines

As can be seen in figure 7 (5), there is noise, which is predominately seen within the images of the screws. Thus, to select a usable line, different filters need to be used. For the screw, only lines within a distance from the center will be used. This isolates the cross of the entity. For the bolt, the longest line is used, since this would always result in the slit of the middle. For the nut, no filter is used, as all the lines would always correspond to the hexagon edges. If there are more than one line found that are usable, the bottom line will be used.

### 6.1.5 Pose

The angle of the final line will then be measured in comparison to the horizon of the image. This angle is outputted by the system, and for visualization during debugging, the image is rotated and displayed. Showing that the orientation of the entity is now horizontal. The final angle can then be used by the end effector to orientate the head, ready to unscrew the part.

# 7 Robotic Arm

The robotic arm is an essential part of the system. Without the robotic arm, there would be no interaction with the surroundings. Developing a good arm to prove the systems value, is thus another crucial part. It is, however, important to stay realistic of the expectations of the work that is developed. The robotic arm discussed within this section will purely have the function to demonstrate a possible embedded system.

## 7.1 Design

The development of the arm starts with research and sketching. Since it was familiar that the orientation of the entities would remain the same, it was possible to keep the end-effector horizontal. This would allow the investigation of the different arm designs. The design that was chosen was a parallelogram arm. The great benefit of such an arm is that the motors for the arm movement would all be located at the base, allowing for a lightweight construction. The construction was drawn in a 3D CAD software, SolidWorks in this case. The final design that was drawn out is displayed in figure 8. The design consists of two parallelograms linked together with a joint, At the base plate, the two brackets are designed to keep the motors in the correct location. The motors were preset into a position before screwing them in place. This insured that the during power up, the motors would not start from an unknown location, and allowed planning of the range of motion. As an example, if this

would not have been done, it could be possible that the motor is at its end limit when screwing it in place. Resulting in a possible unreachable position of the arm, since the motor needed to turn both ways (moving the arm forward or back). At the end of the arm is the end-effector. The end-effector holds both the motor that will unscrew the entity, as well as the camera responsible for gathering sensory data that will be digested by the system. As stated earlier, the use of the parallelogram arm allows this end effector to always be horizontal.

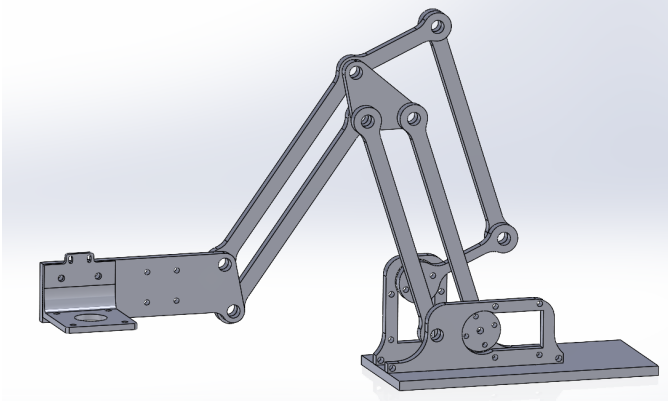

Figure 8: SolidWorks CAD drawing of Robotic Arm.

## 7.2 Inverse Kinematics

An inverse kinematics of a system is a mathematical function that returns the position of the joints based upon an input of the coordinate system. As an example, say the arm must move to location X,Y. It would be possible to figure out the position of the joints. This can either be done using matrices, modeling or in this case since the arm is simple, geometry.

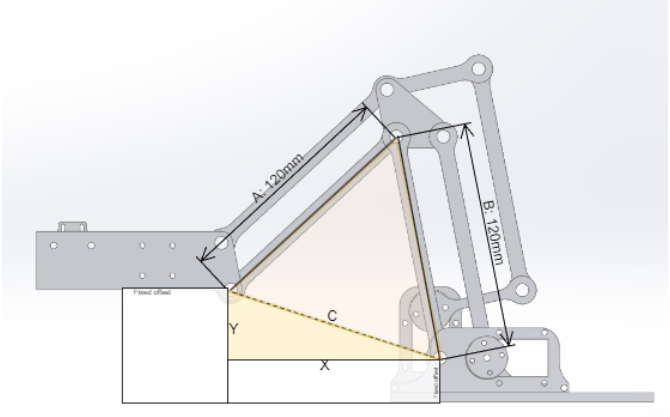


Figure 9: Schematic overview for kinematics calculations.

Calculating the inverse kinematics of the robot arm using geometry is done by finding multiple triangles. Showing in figure 9, are the two triangles. One of the triangles is a right-angle triangle from the first joint, to the X,Y position. The other triangle is a scalene triangle running along the joints and the X,Y point. (However, since in this robot design both arm lengths are the same, it is technically an isosceles triangle. For the calculations this does not make a difference.) By using the cosine rule, shown in equation 1, the unknown angles can be calculated, which in turn give the joint positions.

$$c^2 = a^2 + b^2 - 2ab\cos(C) \tag{1}$$

First, the hypotenuse of the right-angle triangle is calculated using Pythagoras theorem. After which all the unknowns for the cosine rule are known. Since the arm lengths are constant. The formula for the upper joint would be restructured to equation 2.

$$Angle_1 = \cos^{-1}\left(\frac{Arm1^2 + Arm2^2 - Hypotenuse^2}{2 * Arm1 * Arm2}\right) \tag{2}$$

To get the angle of the bottom joint, the result of its cosine rule will be added to the angle of the right-angle triangle, simply calculated using the inverse tangent. The final equation would thus be an altered version of the cosine rule plus the inverse tangent, presented in equation 3.

$$Angle_2 = \cos^{-1}\left(\frac{Hypotenuse^2 + Arm1^2 - Arm1^2}{2 * Arm2 * Hypotenuse}\right) + tan^{-1}\left(\frac{X}{Y}\right) \tag{3}$$

The aforementioned two equations can be written for this robot arm in scripting, including the offset of the motor (starting point), the length of the arms, and the correction for the parallelogram to give code below. These can be used within the python script, which runs the motors.

```
H = math.sqrt(x**2 + y**2)
lower = 60+ math.degrees(math.acos( (H**2)/(H*240) ) + math.atan2(y,x) )
upper = 300- (math.degrees( math.acos(((120**2) + (120**2) - (H**2))/
    (2*120*120)) ) +10-(90-lower))
```

Due note, that the X,Y location is initially calculated from the origin of the first motor to the pivot-point on the end effector, thus if the baseplate is raised, or the end effector is longer these distances must be removed from values used in the calculations.

# 8 Results

In this section the results of the process will be discussed. This chapter is set up in such a way that it follows the flow of the process.

## 8.1 Neural Network Results

The neural network came with different results under different conditions. When it was first tested, after development within the python scripts, it resulted slightly different to the implementation in the HLS. This is further discussed within the upcoming sub-sections.

### 8.1.1 TensorFlow Training Results

The neural network was trained on a similar dataset as that it was tested to. In figure 10 the results showed that the accuracy of the network was 1.0, this correlates to a 100% accuracy. This can then later also be presented by the five singular tests that have been ran. Each outputting the correct prediction as the inputted value would suggest. Although these results seem highly promising, a 100% accuracy is rarely possible when exposing the system to more environmental variable, such as a higher variance in lighting conditions, color of entities or color of background. To make this system robuster, the dataset would have to grow with the use-case, meaning that when the system needs to react the same within a different environment, the dataset would require to do the same.

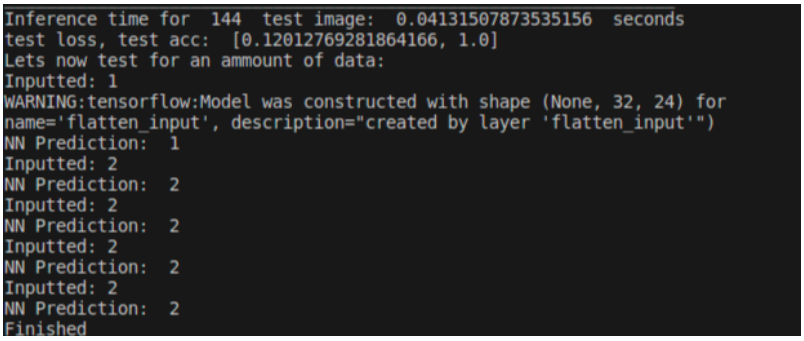


Figure 10: Test results within Python.

### 8.1.2 Vitis HLS Testbench Results

After the Vitis HLS block was developed, an identical test as the TensorFlow training was done. Five unique images would be inserted through the testbench, where the output of the system would be outputted in console. In figure 11 the results are displayed. The input images were of value 2, 0, 1, 1, 1. Meaning that the last image was not correctly estimated. Since the training results suggested a 100% accuracy, it already seems to deviate. The most logical explanation for this irregularity could be that the data size within the FPGA is smaller than that of the Python runtime. Hence, while still processing the image, the FPGA might round creating small abnormalities in the expected behavior.

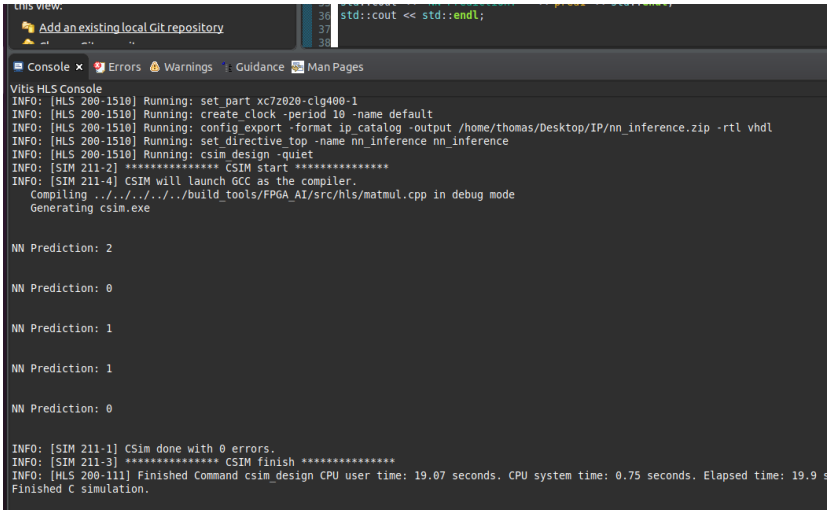


Figure 11: Test results within Vitis.

## 8.2 Building for Ultra96

While building for the Ultra96, some issues arouse. Although the building of the Vitis HLS IP block and the synthesis of the Vivado integration seems to work, the step to further to the bare-metal testing and the building of a Linux distribution seemed to fail. The individual issues will be addressed below.

### 8.2.1 Bare-Metal Test

The bare metal testing had earlier worked on the PYNQ, giving results that were expected from the Vitis HLS Testbench. However once rebuilding the Vitis HLS block specifically for the Ultra96 and the Vivado synthesis, it stopped functioning. Sadly, neither the Vitis HLS nor the Vivado software prompted with a possible issue, making debugging of the error thus much harder. With the assumption that the issue might be within the bare-metal test through Vitis, the building of the Linux distribution was tried.

### 8.2.2 Building Linux Distribution

The building within PetaLinux also failed as can be seen in figure 12 below. From the error portrayed in the image, it is still not possible to make up what exactly failed. Returning the problem back to a difficult to solve issue.

Since both errors occurred after a specific point in the process, it can be assumed that the error is indeed within the Vitis HLS code, or the Vivado development. This would require further investigation seeing the direct issue is not being displayed by either program.

Figure 12: Error in the development of the Linux distribution using PetaLinux.

## 8.3 Resulting Entity Pose

The module responsible for estimating the pose of the entity worked partially very well. Where both the Bolt and Nut presented an accuracy of 50 correct estimations out of 50 tries, however the screw did present more difficulty only coming to 23 correct estimations out of 50 tries. This could be due to the reference points needed for the Canny and Hough transform, or lack thereof. In future works, the system could be tried with a higher resolution imaging or perhaps a different filtering technique to create a better Canny edge detection or Hough transform. This could potentially mitigate or decrease the issue of the screw's pose estimation.

## 8.4 Robotic Movement

The robotic arm was developed to present how the solution of the embedded system could become viable. This system worked as intended. Although it was not developed with strength and high precision in mind, it succeeded in the task of orientating itself above the entities using a ROS2 node with controller. This design could serve as a good starting point for the development of a better research driven arm. The final realization of the arm is presented below in figure 13.

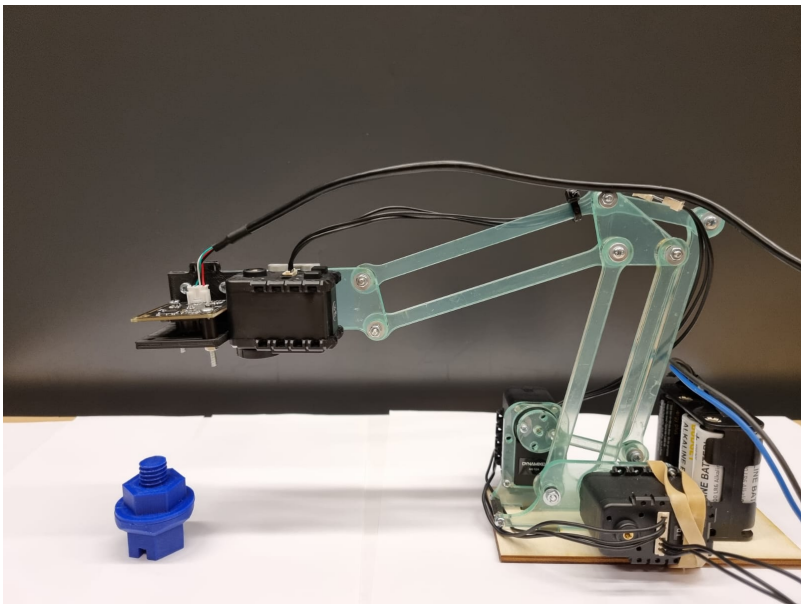

Figure 13: The final arm developed for the Embedded System.

# 9 Conclusion

The robotic scene will always be a further developing paradigm. Since the 1940's robotic arms were being developed to serve humans. With the latest revolution of AI, this brings a large number of possibilities. Looking deeper into the embedded system behind the moving components lies a large field of possibilities. Within this paper the implementation of an FPGA incorporated system was discussed. The point being to bring attention to the possibility of integration of a smart and efficient neural network for industry and research. Although this project showed to still require further research, the concept clearly shows that the potential for a FPGA inclusive embedded system is tremendous. The result of this systems, although not completed, did show how the architecture of such an embedded system is constructed. Additionally it showed the development process required. Displaying how such a embedded system could develop into a robust system, capable of handling tasks as a singular unit.

# References


[1] T. A. J. van der Sterren, D. O. Eslava, A. M. Quesada, and J. G. Pastor, "Using an fpga neural network to make robotic choices." `https://github.com/blatv/FPGA_AI_RobotArm/tree/main`, 2023.

[2] M. K. Alkaeed, Z. Alamro, M. S. Al-Ali, H. A. Al-Mohammed, and K. M. Khan, "Highlight on cryptocurrencies mining with cpus and gpus and their benefits based on their characteristics," in *2020 IEEE 10th International Conference on System Engineering and Technology (ICSET)*, pp. 67–72, 2020.

[3] R. Joost and R. Salomon, "Advantages of fpga-based multiprocessor systems in industrial applications," in *31st Annual Conference of IEEE Industrial Electronics Society, 2005. IECON 2005.*, pp. 6 pp.–, 2005.

[4] A. Gasparetto and L. Scalera, "A brief history of industrial robotics in the 20th century," *Advances in Historical Studies*, vol. 8, no. 1, 2019.

[5] Xillinx, "Avnet ultra96-v2." `https://www.xilinx.com/products/boards-and-kits/1-vad4rl.html`.

[6] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[7] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," 2015.

[8] N. Malle and E. Ebeid, "Open-source educational platform for fpga accelerated ai in robotics." `https://github.com/nhma20/FPGA_AI/tree/main`, 2022.

[9] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.