

Report

Hardware Design and Lab: Final Project

第一組

106062333 刁彥斌

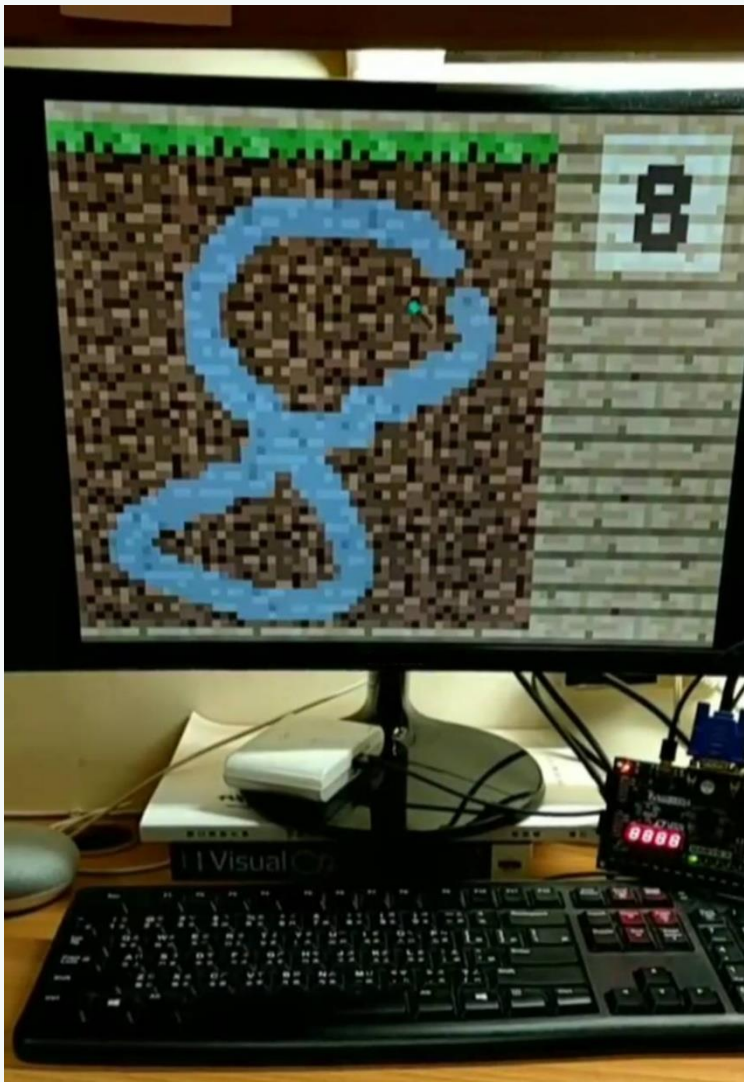
106062203 孫偉芳

Introduction

Project Title

全知全能智多星可愛畫布？

The Omniscient and Omnipotent Smartie, Cutie Canvas?



Summary

The **Omniscient and Omnipotent Smartie, Cutie Canvas?** is a project on Basys3 Artix-7 FPGA, consisting of a **canvas painter**, and a simple on-board **neural network** that recognizes the digit (0-9) drawn on the canvas.

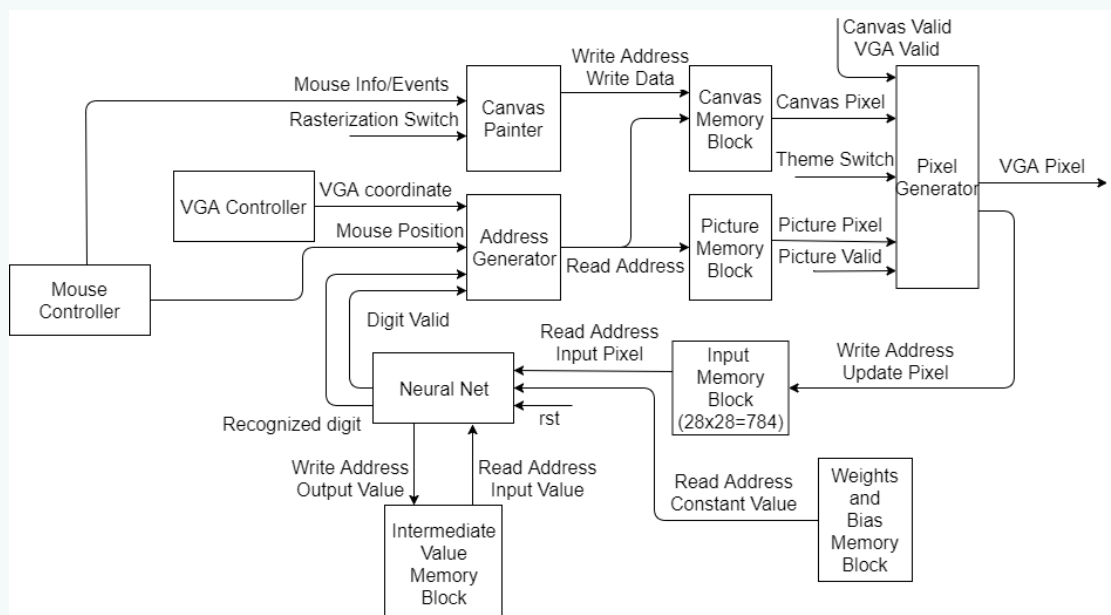
Control and I/Os

Button Bottom:	Reset
Mouse Left:	Draw
Mouse Middle:	Erase
Mouse Right:	Clean Canvas
LED:	Indicates recognized digit
7-Segment:	Indicates recognized digit
SW 15 & 14:	Theme Settings (4 possible themes)
SW0:	Rasterization On/Off

VGA and USB interface (for mouse) are also used.

Block Diagram

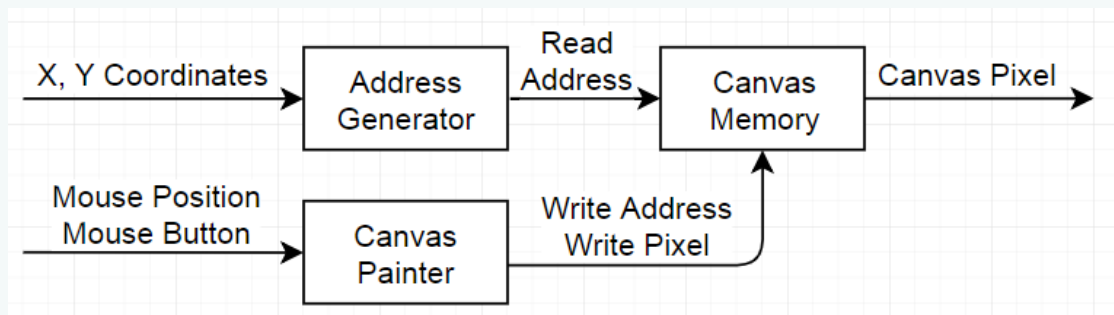
The left part is the user input, and the right part is the VGA output. The upper part is the UI controller, and the lower part is the Neural Net.



Implementation

Canvas

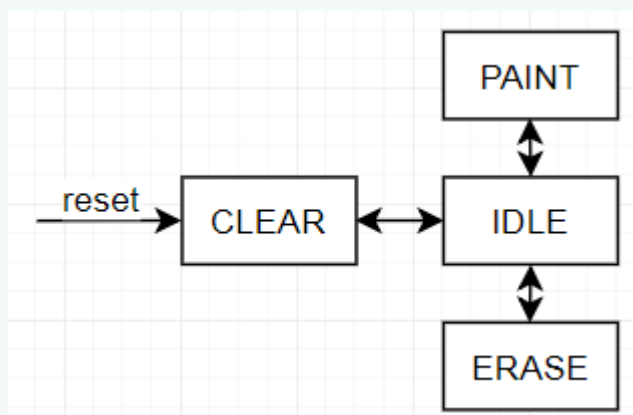
- **Canvas** is a 56x56 **memory block** (dual port).
- **canvas_mem_addr_gen** outputs the current **address** for the memory, given the VGA x-y signal.
- **CanvasPainter** also interacts with the memory. It determines when, where, and what to draw on the canvas with an FSM.



Painter

Canvas Painter is an FSM that determines when, where, and what to draw on the canvas, based on the mouse info.

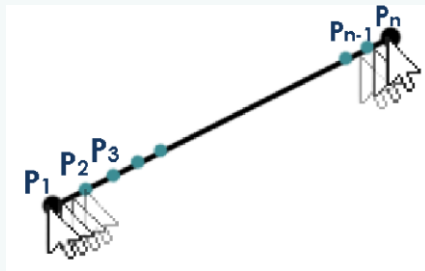
The initial state is **CLEAR** so that the canvas is cleaned by pressing reset button.



- **CLEAR:** Clean the whole canvas. It takes 56x56 clocks before completion (set every pixel to 0) and returns to **IDLE** state.
- **IDLE:** Wait for next operation. Left click for **PAINT**, right click for **ERASE**.

CLEAR, and middle click for **ERASE**.

In **PAINT** and **ERASE** state,



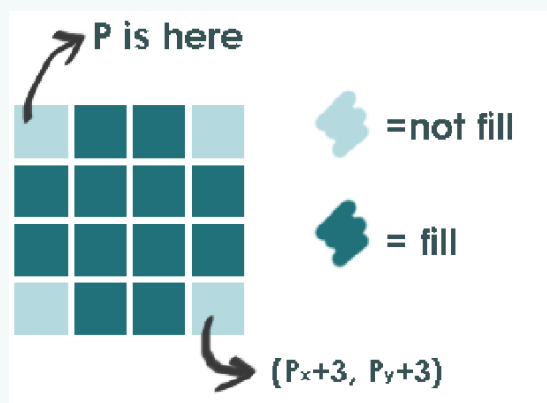
P_1 and P_n are the **most recent 2 positions of mouse**.

We are to paint (or erase) every point on the path (i.e. P_1, P_2, \dots, P_n)

Let a point P traverse through this path. We call this a **TRAIL**.

When a **TRAIL** ends, mouse click is detected to determine whether to stay in **PAINT** (or **ERASE**) or get back to **IDLE**. If mouse stays clicked, the current position of mouse is again sampled to form a new **TRAIL**.

Since the brush is not a single pixel but a **4x4 region**, we need to draw some pixels around P . This is done for every $P=P_i$, and is called a **STEP**.



A **STEP** is done in around 12 clock ticks (actually a little more), and then P traverses to the next point, thus beginning a new **STEP**.

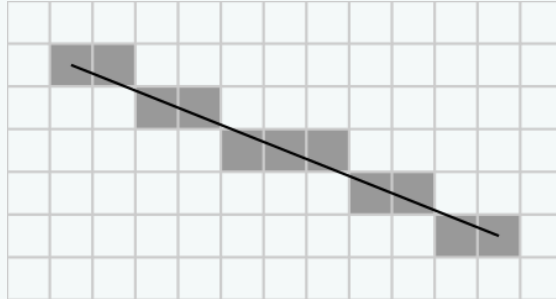
In every **STEP**, the pixels to be filled is defined by **offsets**. Therefore, we can design several modules with several offset settings, creating **brushes of different shapes**, for flexibility and scalability.

Rasterization

Rasterization is a process that **maps shapes onto screen** pixels.

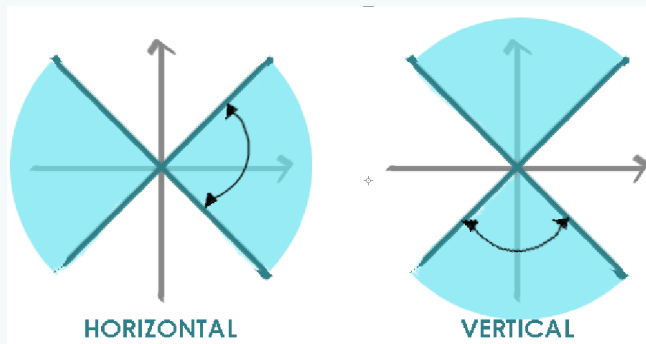
In this case, we have to map a **TRAIL line** onto the screen, so we can determine points P_i along the path.

There are several algorithms capable of achieving this; however, the trivial (yet pretty useful) approach is adopted: **Slope**.

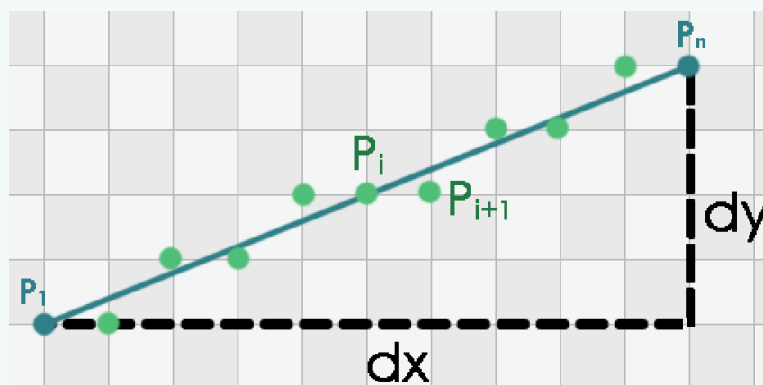


We can consider two types of lines with their chief direction: **horizontal** and **vertical**.

For horizontal lines, since there are **more X's between P_1 and P_n than Y**, we let the x coordinate of P be that of P_1 to P_n , and **calculate the corresponding y coordinate**, so that we won't create gaps by going too fast. (similar for vertical)

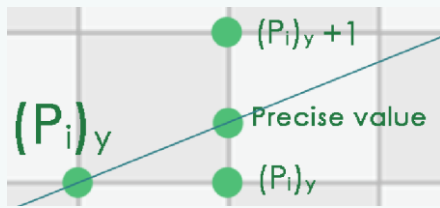


Take the calculation of horizontal lines for example.



$$(P_{i+1})_x = (P_i)_x + 1.$$

The **precise value** of y of P_{i+1} is $[(P_1)_y + \frac{dy}{dx}(i + 1)]$



Compare the precise value with $(P_i)_y + 1$.

If the precise value is larger, then $(P_{i+1})_y = (P_i)_y + 1$

Otherwise, $(P_{i+1})_y = (P_i)_y$

Since I do not want to deal with negatives and fractions, the inequality can be written as:

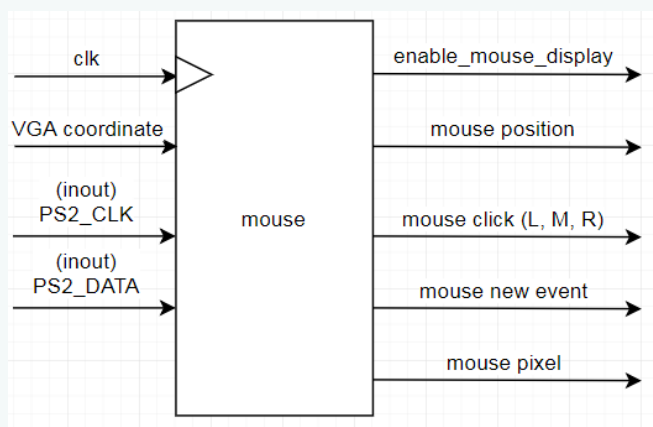
$$(i + 1) \cdot dy + dx \cdot y_1 \geq dx \cdot (y_i + 1)$$

Thus, we can determine the next P after every **STEP**.

Surely there are improvements to this method, but since it does not wander away from the line at each sampling, this method does a decent job. An obvious boost was to also calculate the difference between 3 points so as to make the best decision. Though the difference is few since we map a **448x448** region onto a **56x56** canvas, and then zoom into **28x28** for neural net.

Mouse

The mouse module is a self-contained module with some customized parameters.

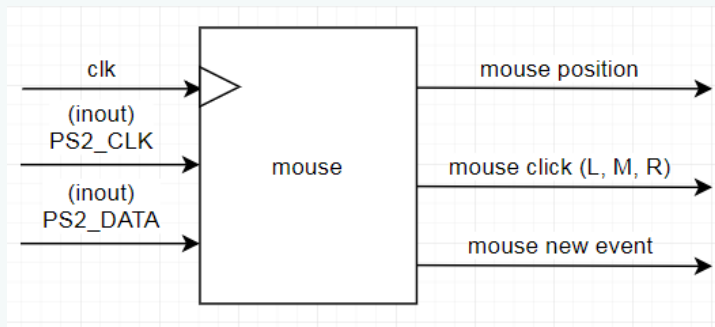


Given the **PS2 info**, it is able to provide mouse position and **click**

events. Plus, it also take the **VGA coordinate** and compute display information such as **enable_mouse_display** and **mouse pixel**. This gives a default mouse cursor icon.

(The enable mouse display indicates whether this specific VGA pixel is occupied by mouse)

I wanted to use my own icons, so I did a little modification and it became:



I let another module calculate the two signals, so it can change with different themes.

UI & Themes

The pixel generator module takes all possible objects on the screen and decide which to display at the end (outputs color pixel for VGA). Either the object can provide its info, or pixel gen can compute it (fixed position like the position of canvas or background).

To change themes, use the **leftmost 2 switches**:

00	Minecraft Simulator	Minecraft 模擬器
01	Pink Lady	紅粉佳人
10	Carefree	寧夏午後
11	High on Weed	魔幻大麻

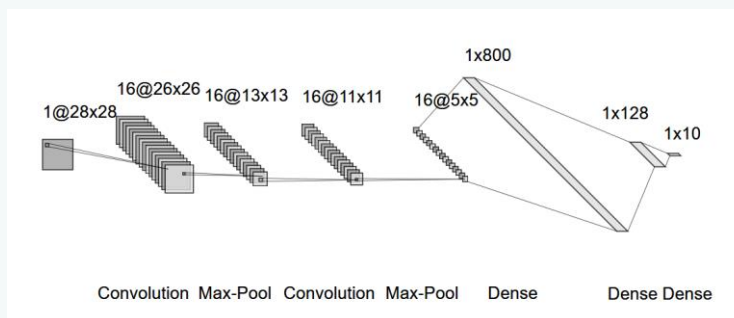
The Minecraft theme uses several pictures to stack up, but others are results of certain color assignments or computation. The brush and eraser

in Minecraft Simulator are a shovel and pickaxe, made by **precious diamond** found deeply down the mine (This is not true). For the other themes, they do not deserve the luxury so they used a simple brush and eraser. They differ in color, though.

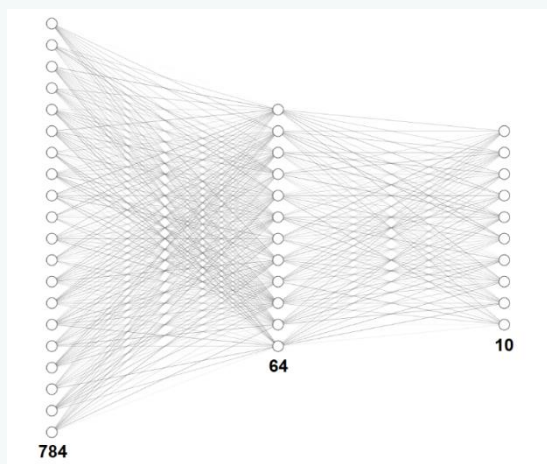
Neural Network

Initially, we want to use **CNN** to recognize digits by implementing the network in [keras/examples/mnist_cnn.py](#). However, we can easily calculate that for the flattened-CNN to be densified (fully-connected) to 128 neurons isn't practical. (**1179776 parameters** solely in this layer)

We've compressed the CNN network by tweaking the hyper-parameters such as kernel size, kernel counts, add more pooling, decrease the neurons, etc. We've finally discovered a network that has less parameters that can fit in the FPGA 1.8Mb block memory.



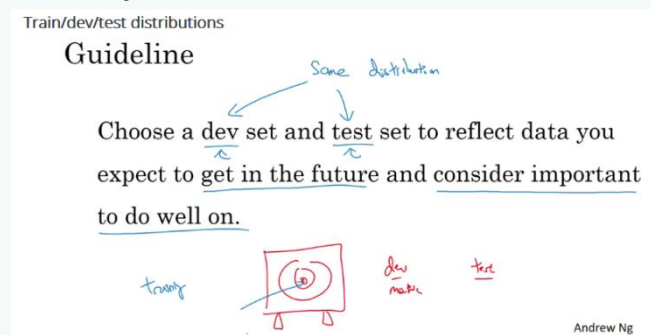
However, CNN's accuracy is about **80%** when testing on our custom data, which isn't much better than **2 fully-connected layers**. We decided to use the 2 fully-connected layers first, and then decide whether to implement CNN depending on the results. (Spoiler: We didn't implement the CNN in the end.)



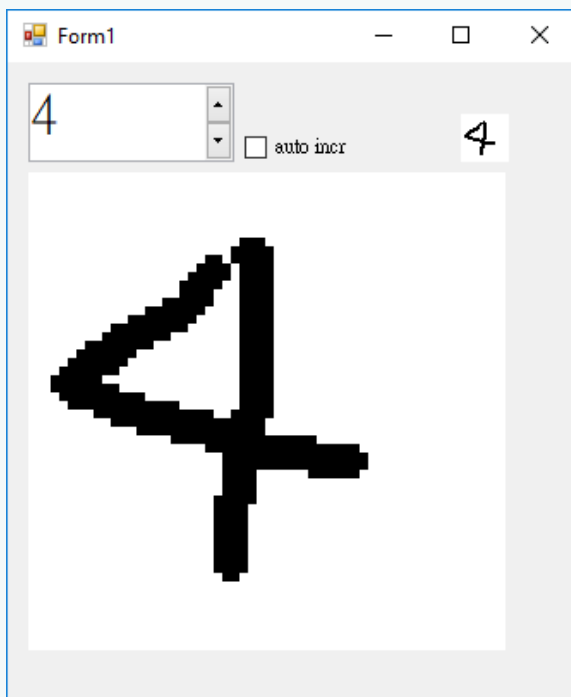
When trained solely on the MNIST data using Keras, 2 fully-connected neural network has about 98% accuracy, but the accuracy

drops dramatically to about 70% when testing on our own custom data. It improved a bit when we convert all MNIST data to black-and-white only images before training, but it's still about 70%.

This problem occurs since our training and testing in different data distributions, the MNIST data set is grayscale, hand-written by Americans; while our data set is black-and-white, mouse-drawn by Taiwanese. So we must put some **custom data** into the training data to achieve better accuracy.



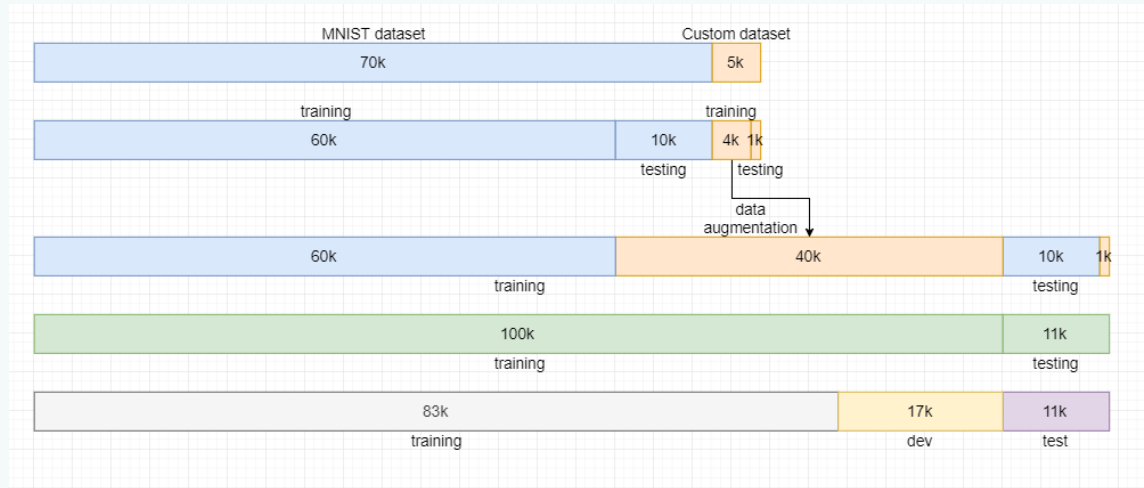
In order to get enough custom data which we expected to test in the future, we must write it by ourselves in the exact same environment. We make **a Windows Forms application to mimic the exact same user interface (canvas) that we'll be using in our FPGA**. The program will auto-save your drawn digit to the correct folder after your mouse have drawn the numbers of strokes of the digit. It's really easy to use, and we get about **5k custom images** in just a few hours.



However, 5k images are not enough, so we've performed some

random rotation, skew and transformations on the data, **augmenting** them to a total of 40k images.

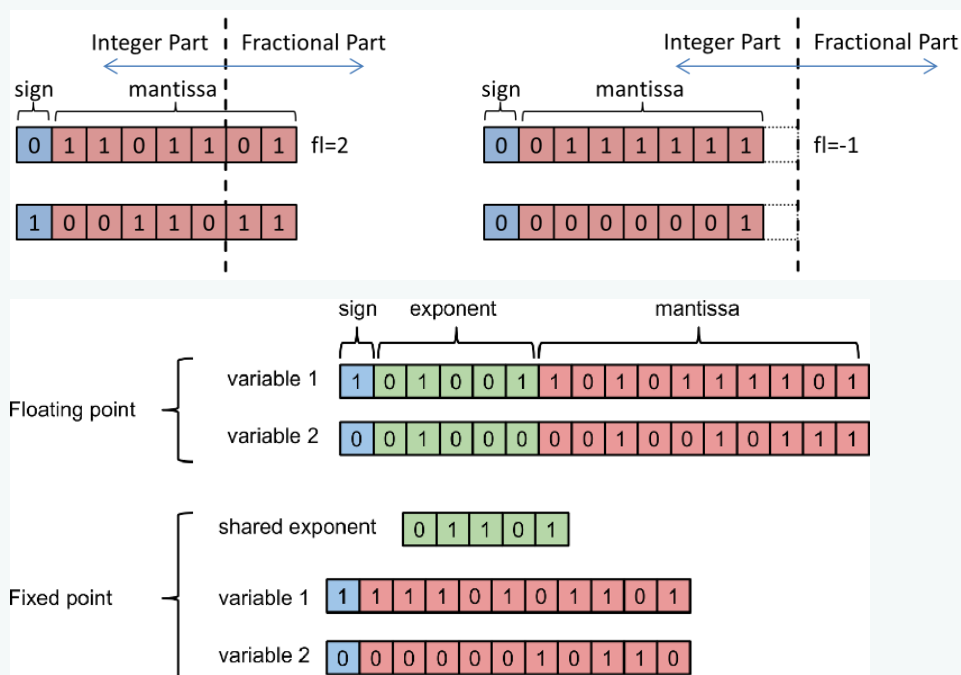
The result is pretty good after a few epochs, we get about **97%** accuracy in the mixed test set and the custom data set.



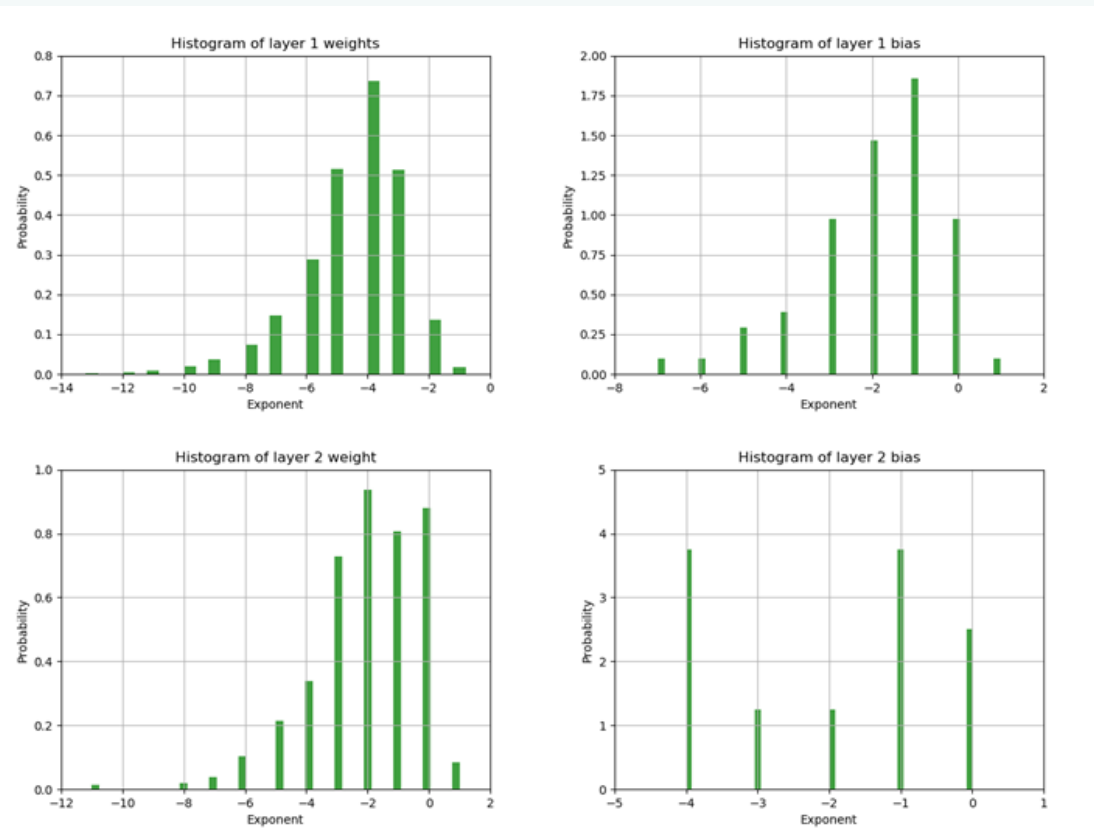
Weights / Bias Quantization

Although the neural network is much smaller than the original CNN, if we use 32-bit float to store the weights, the **memory won't be enough**. We've found [dawsonion/fpu](https://github.com/dawsonion/fpu) on GitHub, and it works well, however it still takes much more cycle to calculate than integer.

Fixed point isn't quite optimal in this case, so we implemented the **Dynamic Fixed Point**. It's like fixed point but stores the shared exponent of each neural net layer. So, all weights can be reduced **from 32-bit float to 8-bit integer** without a huge impact on the accuracy. However, for easier implementation, we only reduced it into **16-bit integer** for better precision.



We store the shared exponent for each layer's bias and weight by plotting their exponent.



By doing this, we only used about **1Mb (50%)** of the block memory that's in the Basys3 FPGA to implement the neural net.

Verilog Result Validation

We implement the neural net in **Python** using Keras to train the weights. We then output its **weights**, **bias** and **intermediate values** of a given input to file. We use **C#** to process those strings and output a space

	rst	clk																																				
state	0		~																																			
n_img_addr	0	1	2	~ 782 783 0 1 2 ~ 782 783 62 63 bias 0 1 2 ~																																		
n_const_addr	0	1	2	~ 782 783 bias 0 1 2 ~ 782 783 bias 0 1 2 ~ 62 63 bias 0 1 2 ~																																		
n_inter_addr				~ 782 783 bias 0 1 2 ~ 782 783 bias 0 1 2 ~ 62 63 bias 0 1 2 ~																																		
img_addr	0	1	2	~ 782 783 0 1 2 ~ 782 783 0 1 2 ~ 62 63 bias 0 1 2 ~																																		
const_addr	0	1	2	~ 782 783 bias 0 1 2 ~ 782 783 bias 0 1 2 ~ 62 63 bias 0 1 2 ~																																		
inter_addr				~ 0 1 2 ~ 782 783 bias 0 1 2 ~ 782 783 bias 0 1 2 ~ 62 63 bias 0 1 2 ~																																		
img	0 1 2		~ 782 783 0 1 2 ~ 782 783 0 1 2 ~ 62 63 bias 0 1 2 ~																																			
const	0 1 2		~ 782 783 bias 0 1 2 ~ 782 783 bias 0 1 2 ~ 62 63 bias 0 1 2 ~																																			
inter				~ 0 1 2 ~ 782 783 bias 0 1 2 ~ 782 783 bias 0 1 2 ~ 62 63 bias 0 1 2 ~																																		
i	0	1	~ 0 1 ~ 63 0 ~																																			
j	0		1	2	~ 781 782 783 784 0 1 2 ~ 781 782 783 784 0 1 2 ~ 61 62 63 64 0 1 2 ~																																	
n_out	0		1	2	~ 0 1 2 ~ 782 783 bias 0 1 2 ~ 782 783 bias 0 1 2 ~ 62 63 bias 0 1 2 ~																																	
out				0	1	2	~ 0 1 2 ~ 782 783 bias 0 1 2 ~ 782 783 bias 0 1 2 ~ 62 63 bias 0 1 2 ~																															
action to n_xx	set		~ mac add set ~ mac add set ~ mac add set ~ mac																																			

We need to prepare the values and constants we need to use beforehand. And since **our block memory is so large, it needs 2 clock cycles to read the output**. We didn't know it before working on this project, so it causes us a lot of trouble.

Information	
Memory Type: Simple Dual Port RAM	
Block RAM resource(s) (18K BRAMs): 1	
Block RAM resource(s) (36K BRAMs): 0	
Total Port B Read Latency (From Rising Edge of Read Clock): 2 Clock Cycle(s)	
Address Width A: 12	
Address Width B: 12	

Testbenches came in handy. We can output the waveform and see what's wrong. We also output all the intermediate values and compare it bit by bit to the C++ output, making sure that it's working as expected. The total calculation only needs 509.02 μ s, so we can **keep calculate the output parallelly in real-time** without the user noticing. (We've doubled the cycle (total process becomes 1ms) in order to meet the timing constraints.)



Mobility

We didn't use the FPGA only as an accelerator, we implemented **everything we needed** in it. To make it more mobile, we use a **Bluetooth wireless mouse** as the controller and store the bitstream in the **SPI flash**. So, the device is like **plug-and-play**, users just need a power source, a mouse and a display to try out the project.

The Bluetooth mouse we use is Logitech M331 SILENT PLUS. It

seems that the dongle **does not require any drivers**, so it can work like a normal mouse when connected to the FPGA. (Seems like the right-upper Programming Mode Jumper needs to be either in USB mode or QSPI mode to make the wireless mouse to work.)

Since we didn't connect it to any external resources, **all calculations are done inside the FPGA, which does not have a CPU, making the whole process even harder.**



Remarks

Completeness

Everything works smoothly and great. However, for the neural network part, we use **2 fully-connected layers** instead of CNN. After using the custom augmented data to train the network, the outcome accuracy is much higher than using CNN only trained using MNIST data. So, implementing **CNN won't have too much improvements** and will **consume a lot more memory**, also **spending much more time recognizing** the drawn number. We only write the code in C++ but didn't implement it in Verilog.

Limits

In order to make the digit recognition work well, the user must paint the digit in the **center** and the **size should be as large as the canvas**.

In the Minecraft theme, we have a diamond shovel to dig out the dirt, and a diamond pickaxe to mine the stone. However, they're just the brush and eraser, no real Minecraft mechanism implemented.

Work Assignments

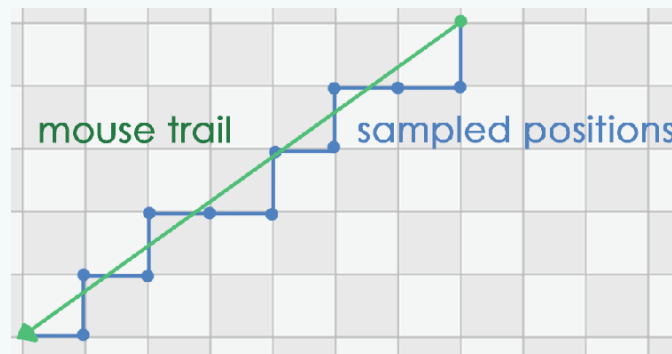
Painter:	Mainly done by 刁彦斌
Neural Network:	Mainly done by 孫偉芳

Testing

See the demo video!

Problems Encountered & solutions

- The mouse is sampled in a bizarre way, as the picture shown below. It **either updates X or Y at a time** (at least so we observed). But we did not have enough time to do some pre-processing for the mouse position change to become smoother.




- Unlike Tensorflow or Keras, we need to be familiar with all **math calculations**, and when using **Dynamic Fixed Points**, we must be familiar of the **IEEE floating-point binary representation**.
- Circuits must have **delays**, and we forced our neural network to be running without the user noticing. If a delay occurs, the result will be entirely different. We use testbench and make sure it meets the **timing constraints** to make sure it works correctly.
- **Real-time recognition** forces us to make the neural net run as fast as it can, while maintainable. We implemented It sequentially using FSMs without any empty cycle, resulting a **1ms delay time**. The user can see the result immediately while drawing.
- The **block memory** in Basys3 Artix-7 FPGA is only 1.8Mb, so in order to fit in the UI and Neural Net, we need to do a lot of data reducing. We only use 2 fully-connected layers (no CNN) and use Dynamic Fixed Point do reduce the size of weights and biases. We only store the **user input as binary image** to instead of grayscale to further reduce the memory usage.
- The distribution of MNIST dataset is different to our actual testing data, causing the **low actual accuracy**. We made a **dedicated app** to create some testing data and use data augmentation to increase its accuracy.

Thoughts

Working on this project is **frustrating and fun**. By spending only a few hours, we can easily implement this in python, and achieve the same outcome. However, implementing this in Basys3 Artix-7 FPGA is really a challenge. It lacks memory and resources and have no built-in CPU. We encountered many problems and solved them respectively. After solving the problems and tidy up the UI, the outcome is very impressive and works better than we thought it would be.

FPGA is surely the trend for purpose-specific computation. Many



companies have started offering such services. The flexibility, scalability and power are growing. A demand of high computation speed of computation core has been spoken for machine learning. In this project, we try to combine several fields of knowledge, creating a simple demonstration of the portable AI engine. Infinite applications still lie uncovered in the world of FPGAs.