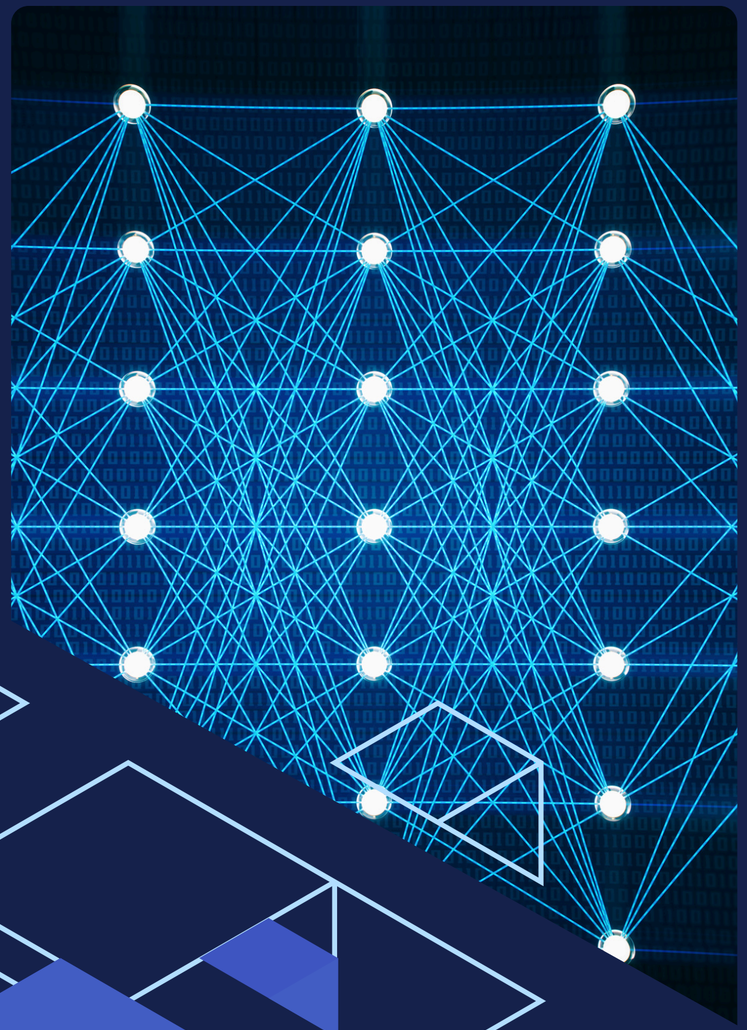
NVIDIA JETSON OCR SYSTEM

ANTONIO GONZALEZ

A2E TECHNOLOGIES SUMMER INTERNSHIP PROGRAM

Table Of Contents



Project Vision

Original Objective

- Develop and implement an Optical Character Recognition (OCR) system on the NVIDIA Jetson Development Platform
- Leverage the power of advanced neural networks and embedded GPU technology to deliver a robust, real-time OCR solution
- Original objective: A functional OCR application capable of accurately recognizing and identifying single characters from a live video feed

Project Vision

Expanded Scope

- Expanded the program’s capabilities to perform inference on multiple individual characters in parallel
- Further developed to allow the program to handle full words and and sentences
- Expected to learn the fundamentals of low level neural network operations, parallel programming, and GPU capabilities

PROJECT TIMELINE

HARDWARE SETUP AND TESTING

Setup and performance evaluation of ArduCam and Jetson

TRAINING DATA COLLECTION

Collected a diverse dataset of character images for training

RESEARCH NEURAL NETWORK ARCHITECTURE

Compared classic N-Layer Neural Network and Convolutional Neural Network (CNN)

CUSTOM CAMERA STAND CREATION

Designed and built for stable image capture with Autodesk Inventor

INITIAL C++ PROGRAM ON JETSON

Basic feed-forward and inference functionality with single characters & still images

PROJECT TIMELINE

INTEGRATION WITH GSTREAMER PIPELINE

Utilized GStreamer to handle character inference with live video

PERFORMANCE OPTIMIZATION WITH CUDA

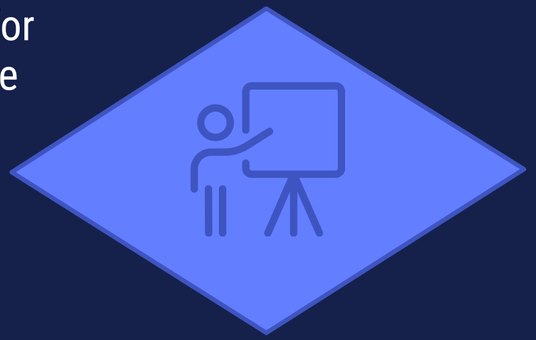
Leveraged parallel processing capabilities with GPU acceleration

REAL-TIME RESULTS EXPANSION

Enhanced program to handle multiple characters and words

MODEL AND PROGRAM REFINEMENT

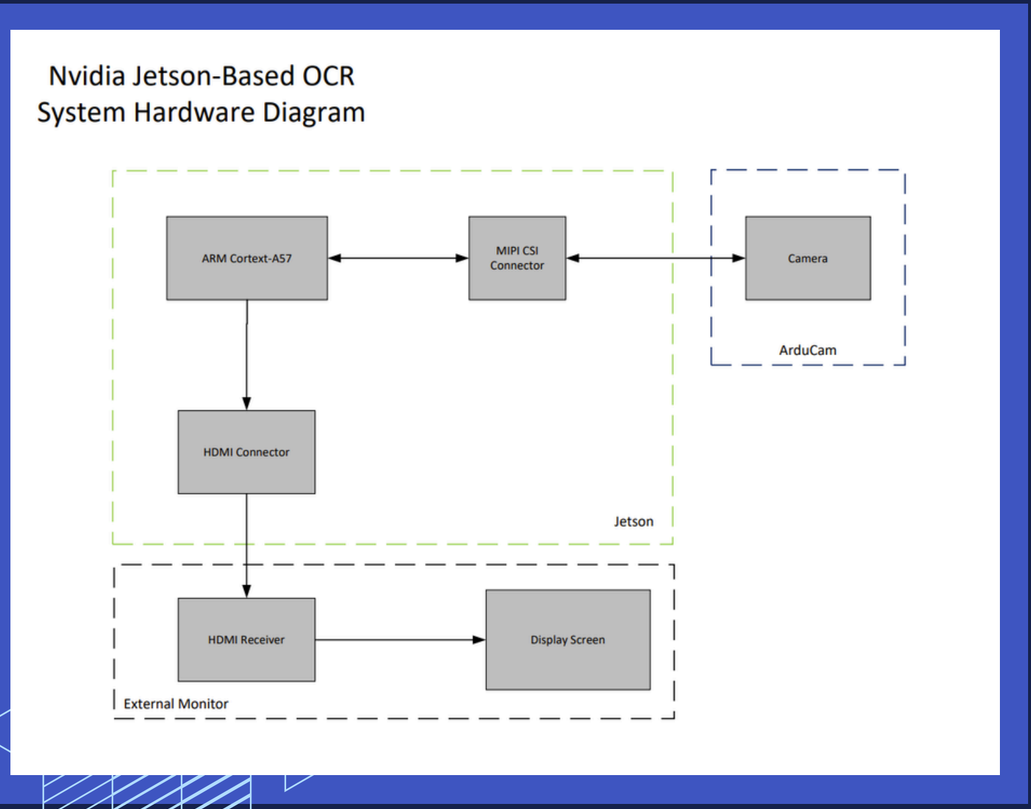
Revised training process and optimized CUDA code for maximum performance

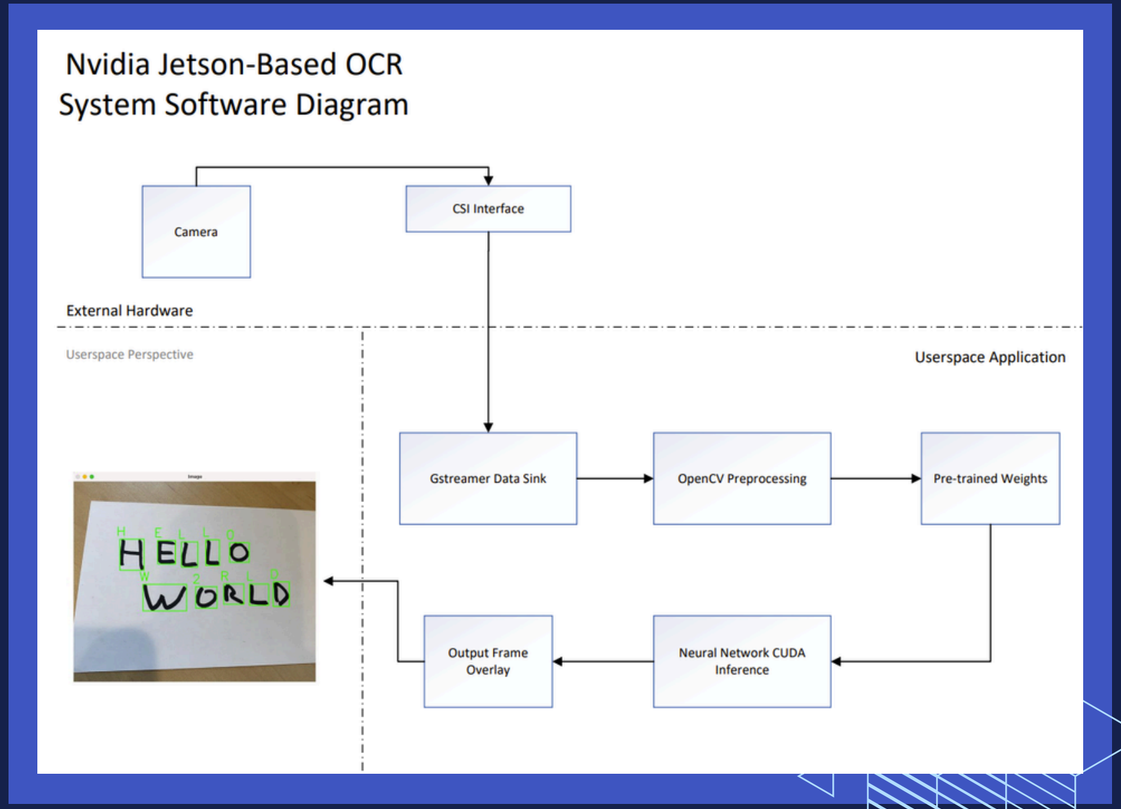


FINAL OUTPUT AND PRESENTATION

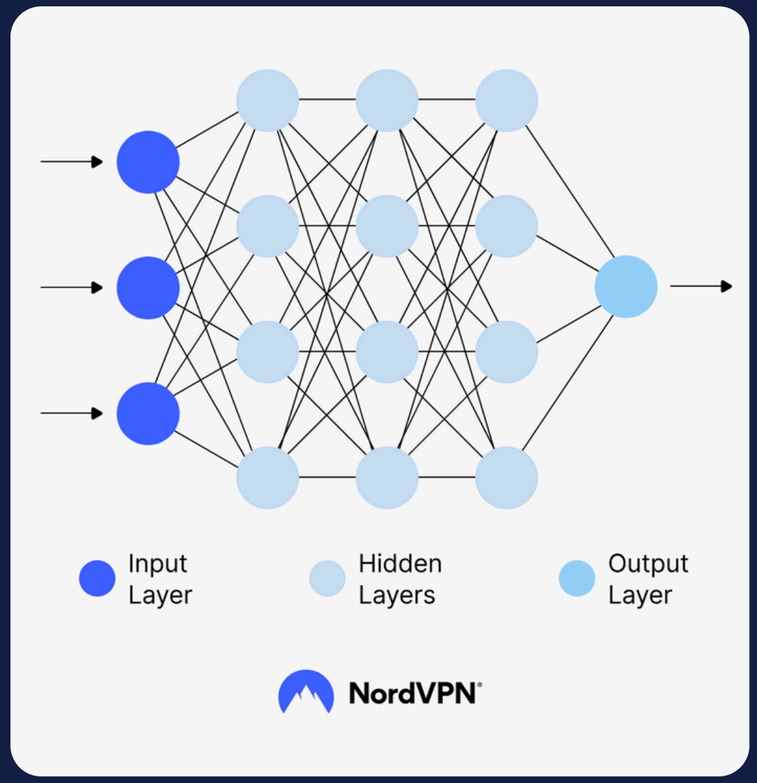
Synthesized project development and summarized educational experience

System Block Diagrams





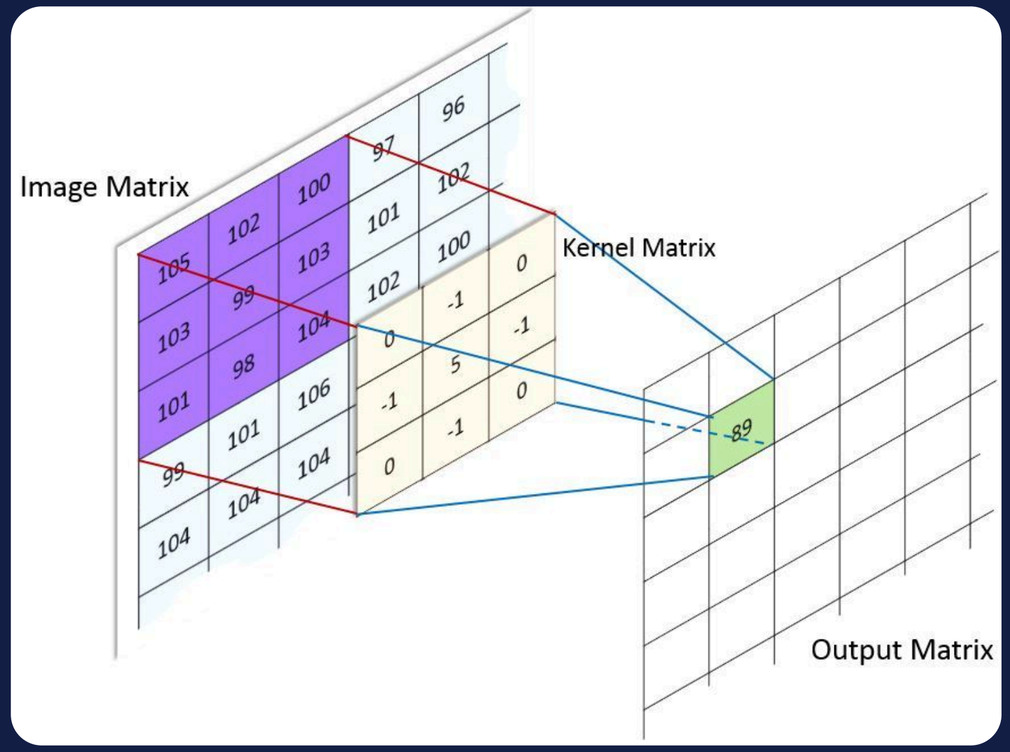
Neural Network Architecture



N-Layer (Fully Connected) Neural Network

- Input Layer: Number of neurons correspond to the number of features in the input data (pixels)
- Images are represented in a 1D format corresponding to the individual pixels i.e 40x60, 2400 pixels
- Going from layer-to-layer involves multiplying by the weights, adding bias, & applying activation function
- Output Layer: Represents the final prediction from the proceeding computations

Neural Network Architecture



Convolutional Neural Network

- Work by convoluting (sliding) a kernel matrix over an image's pixels to compute a numerical representation (feature map)
- Different filters detect different features, such as edges, corners, or textures
- Convolutional Neural Networks are often superior for image recognition tasks
- Sophistication in exchange for its complexity at a low level

Neural Network Architecture

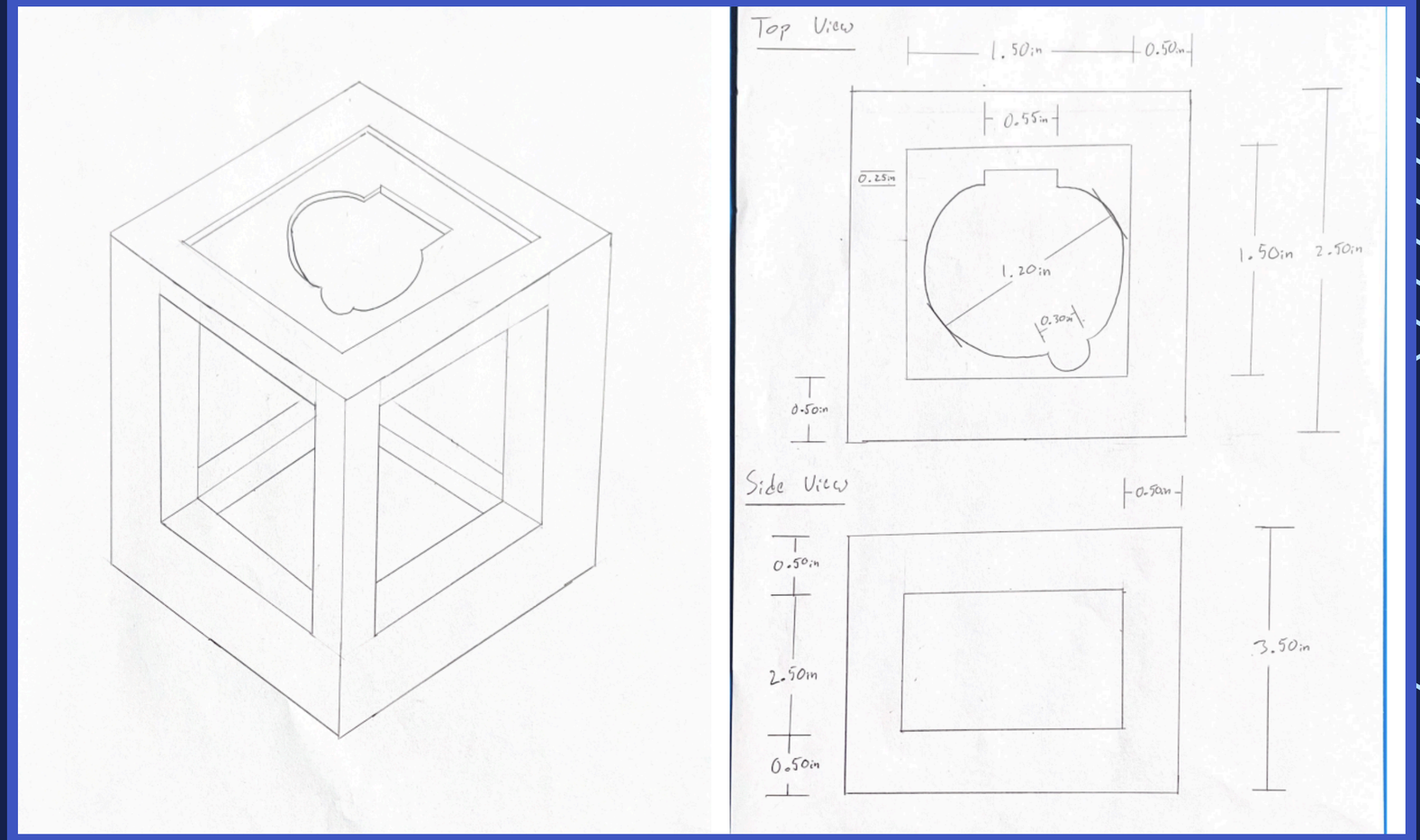
N-Layer vs. CNN

- Ultimately, the N-Layer was decided on for simplicity sake and to allow computation to be easily parallelized
- CNNs are often used in collaboration with fully connected neural networks for classification tasks
- Possibility of implementation in a future version of the program

Hardware Setup

- NVIDIA Jetson Orin Nano Developer Kit
  - Up to 40 TOPS of AI performance
  - GPU w/ 512 CUDA cores
- ArduCam 12MP IMX477 Camera
  - 1920x1080 @ 60fps
- Microscope USB LED Ring Light
- Custom 3D modeled camera stand

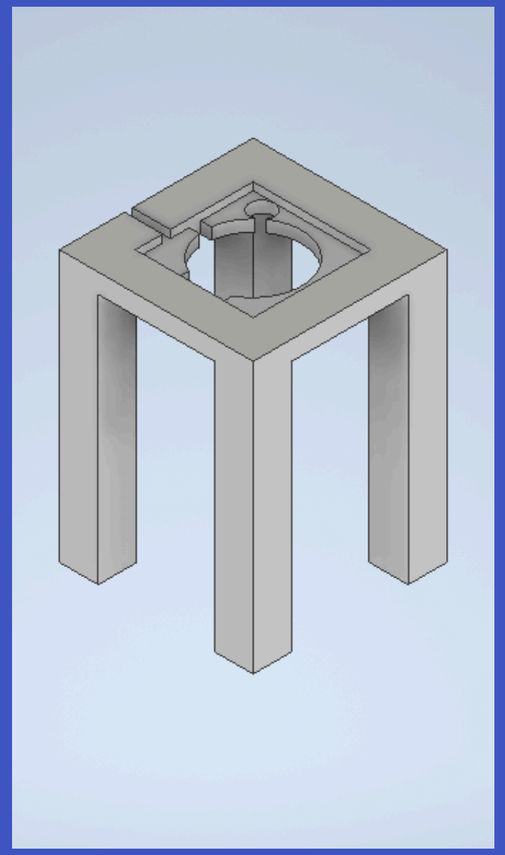
Custom Modeled Camera Stand

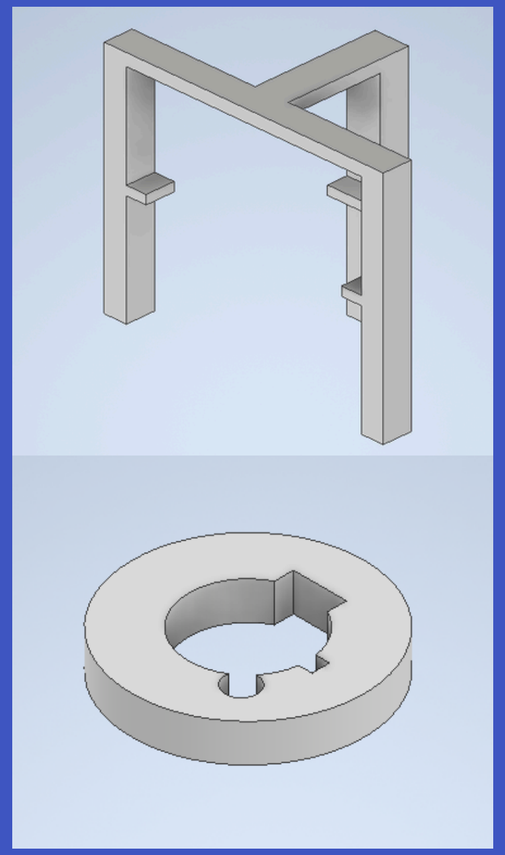


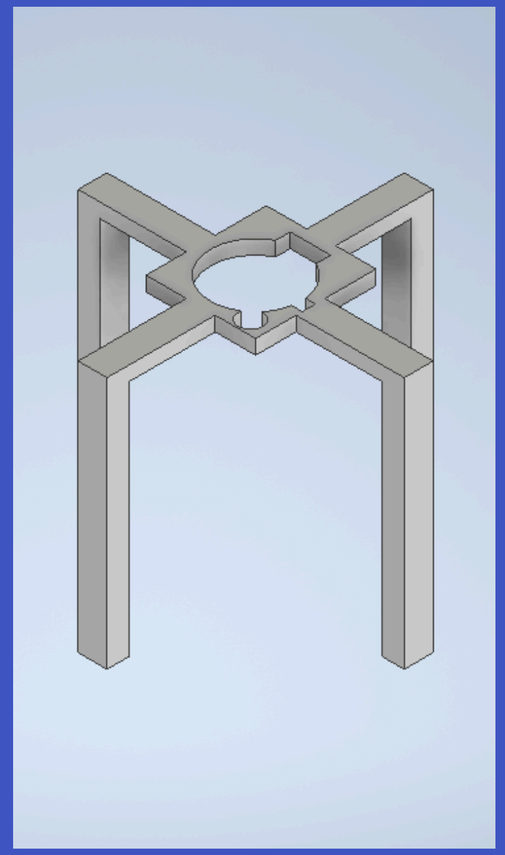
- Keyed recess to allow the camera to sit flush
- Allows for consistent, stable image and video capture
- PCB board 3.5in (9 cm) away from page
- Camera lens 1.75in (4.5cm) away from page

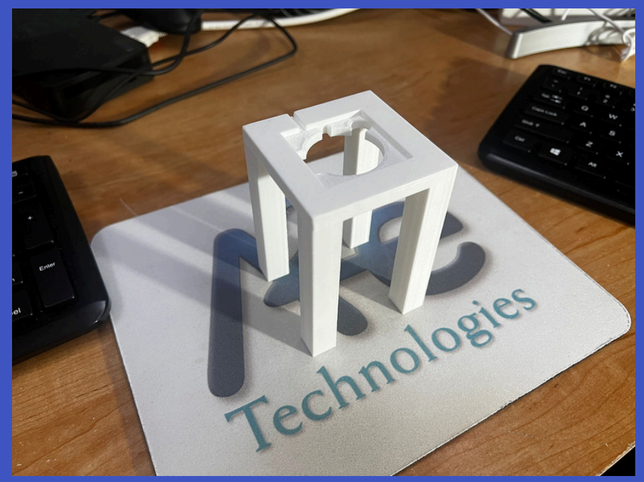
Custom Modeled Camera Stand

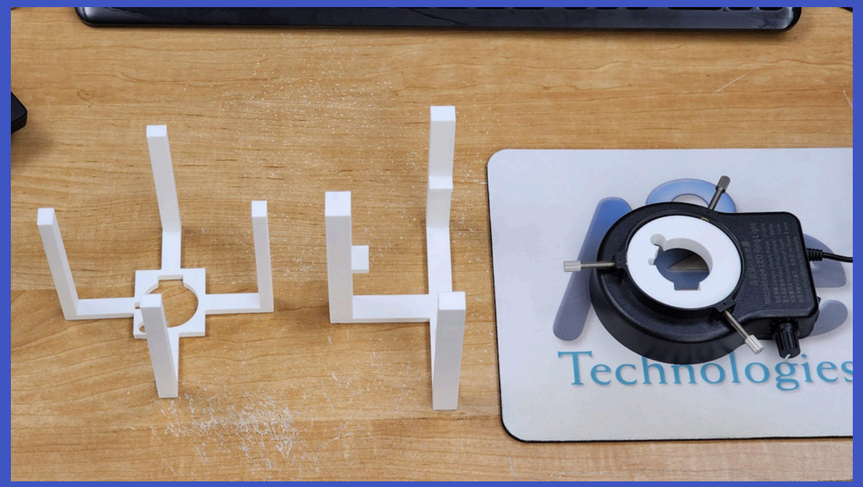
- Designed using using Autodesk Inventor
- V1: Camera intended to sit in the recess
- V2: Frame would hold the light from below
- Camera would sit in the light adapter
- V3: Camera sits in the keyed pattern
- Light sits on top of stand



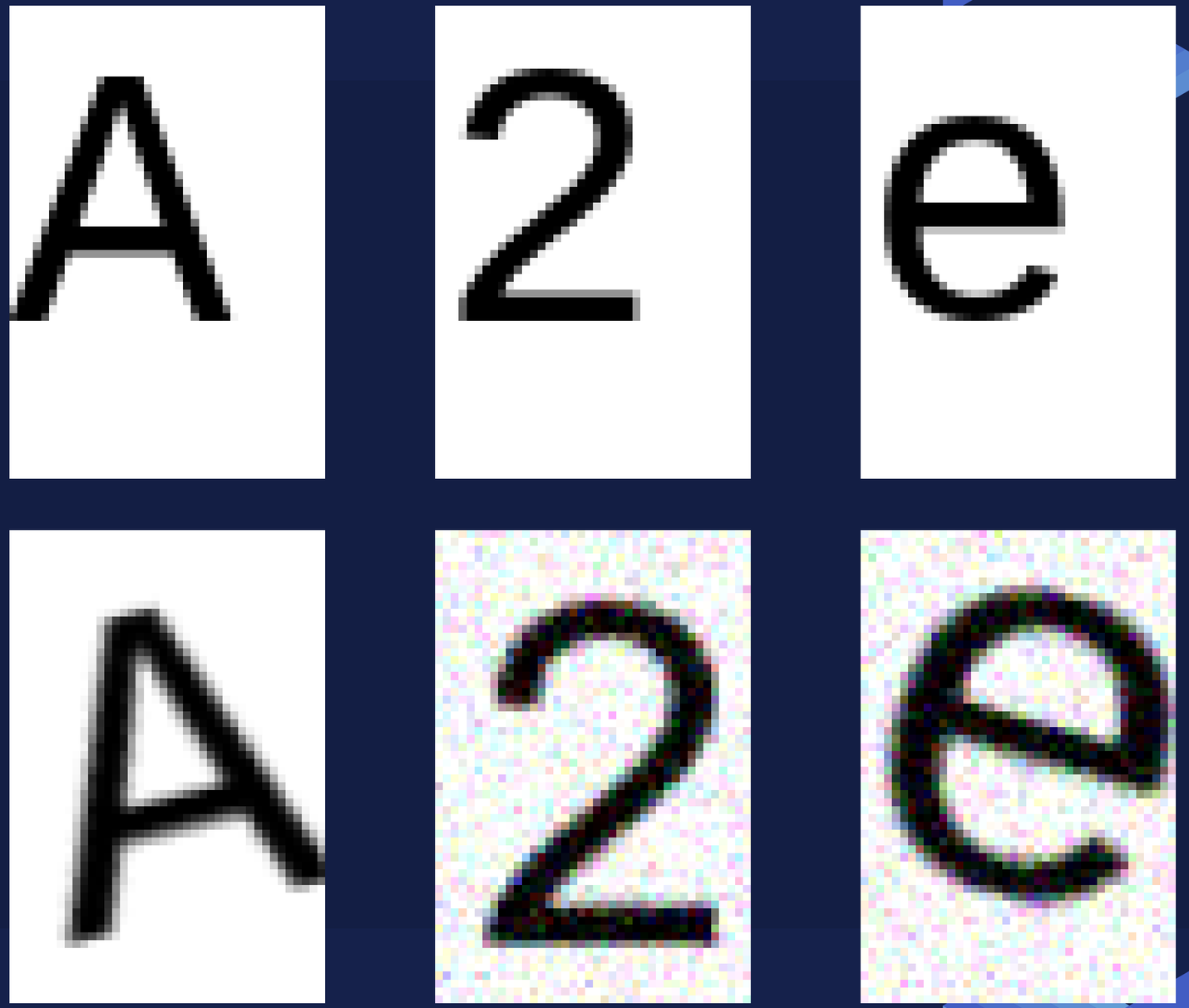








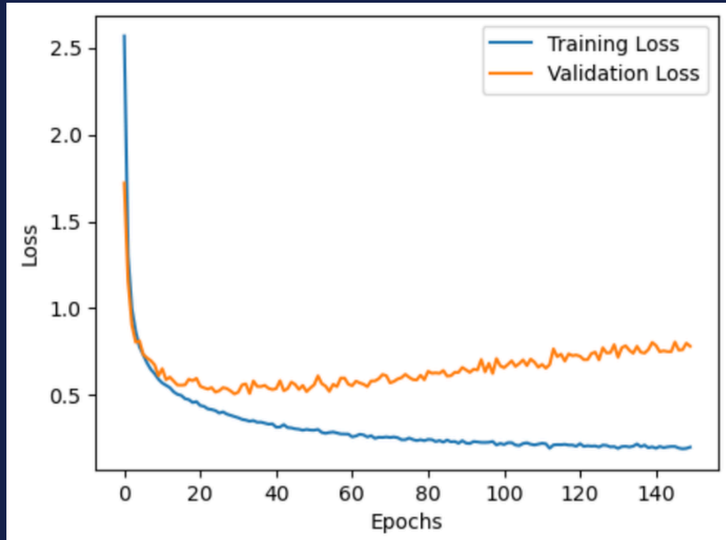
Training Dataset (40x60)



Training Metrics

Full Alphabet

					Training		Validation		Layer Architecture (Number of neurons)					
Test Run	Batch Size	# Layers	Optimizer	Learning Rate	Epochs	Accuracy - T	Loss - T	Accuracy - V	Loss - V	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6
18	512	5	Adam	0.001	150	**0.9435**	0.1839	**0.8724**	0.848	2400	512	512	256	62	
14	512	5	Adam	0.001	100	**0.9285**	0.2322	**0.8753**	0.6783	2400	512	512	256	62	
13	512	5	Adam	0.001	60	**0.9163**	0.2695	**0.8758**	0.5521	2400	512	512	256	62	
10	512	5	Adam	0.001	30	**0.8923**	0.3656	**0.8639**	0.5283	2400	512	512	256	62	
11	1024	5	Adam	0.001	30	**0.8923**	0.4134	**0.8557**	0.5319	2400	512	512	256	62	
12	512	6	Adam	0.001	30	**0.8911**	0.3641	**0.8568**	0.5436	2400	512	512	256	256	62
7	512	5	Adam	0.001	30	**0.8845**	0.3977	**0.8596**	0.5443	2400	512	256	256	62	
9	512	5	Adam	0.001	30	**0.8785**	0.4129	**0.8427**	0.5929	2400	1024	512	256	62	
8	512	6	Adam	0.001	30	**0.8757**	0.4243	**0.8538**	0.5416	2400	512	256	256	128	62
6	256	5	Adam	0.001	15	**0.8581**	0.5034	**0.8376**	0.6147	2400	512	256	256	62	
5	128	5	Adam	0.001	15	**0.8545**	0.5047	**0.8323**	0.6304	2400	512	256	256	62	
17	512	5	Adam	0.0001	30	**0.8544**	0.5631	**0.8433**	0.6136	2400	512	512	256	62	
15	512	5	RMSprop	0.001	30	**0.8483**	0.5331	**0.8308**	0.6458	2400	512	512	256	62	
2	32	5	Adam	0.001	15	**0.8144**	0.6613	**0.7805**	0.8473	2400	512	256	128	62	
3	32	6	Adam	0.001	15	**0.7926**	0.7286	**0.7916**	0.7896	2400	1024	512	256	128	62
16	512	5	SGD	0.001	30	**0.5287**	2.2844	**0.5346**	2.2586	2400	512	512	256	62	
4	32	4	Adam	0.001	15	**0.0174**	4.4311	**0.0172**	4.1512	2400	512	128	62		
1	32	5	Adam	0.001	10	**0.0164**	4.1272	**0.0155**	4.1274	2400	256	128	64	62	



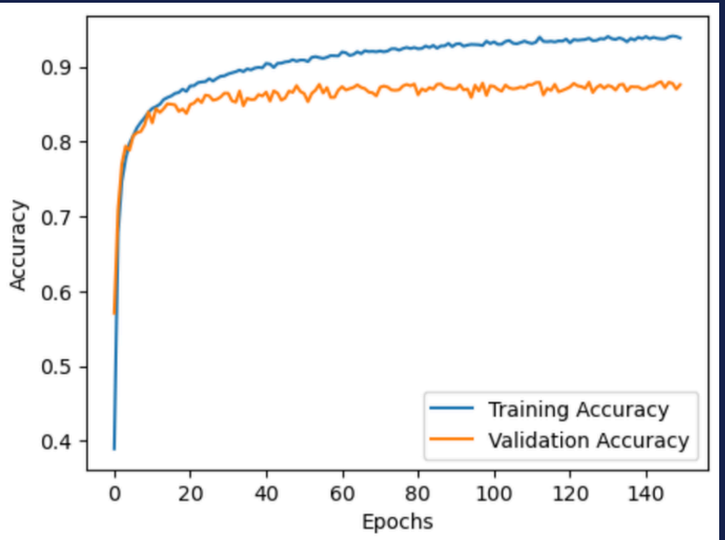
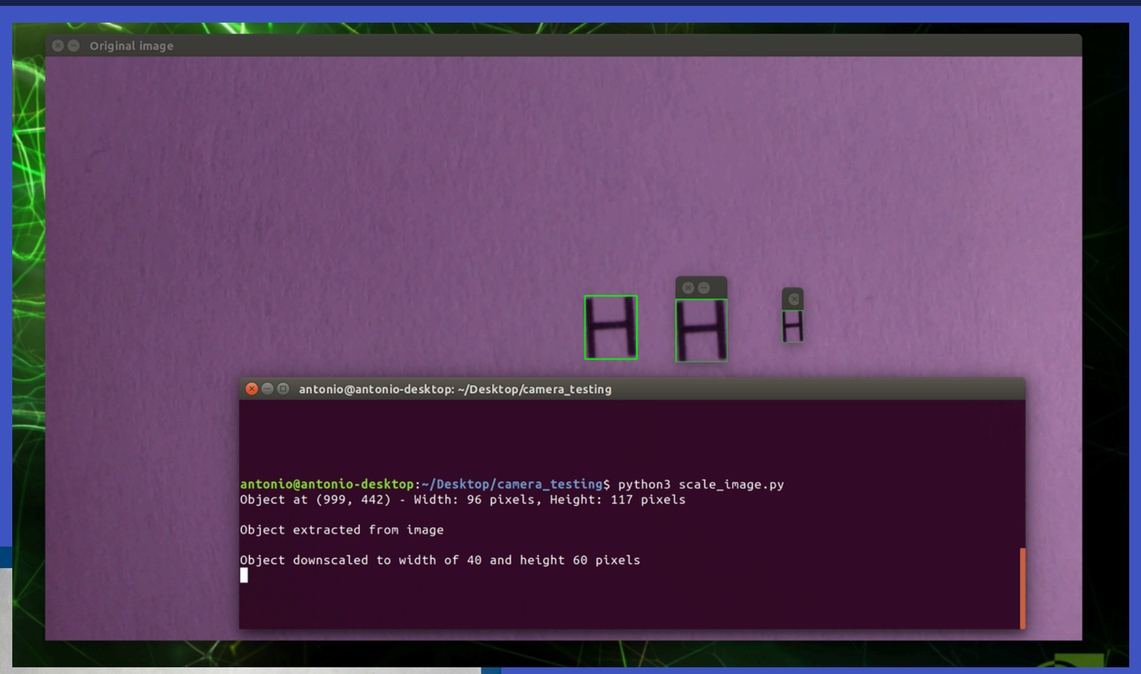
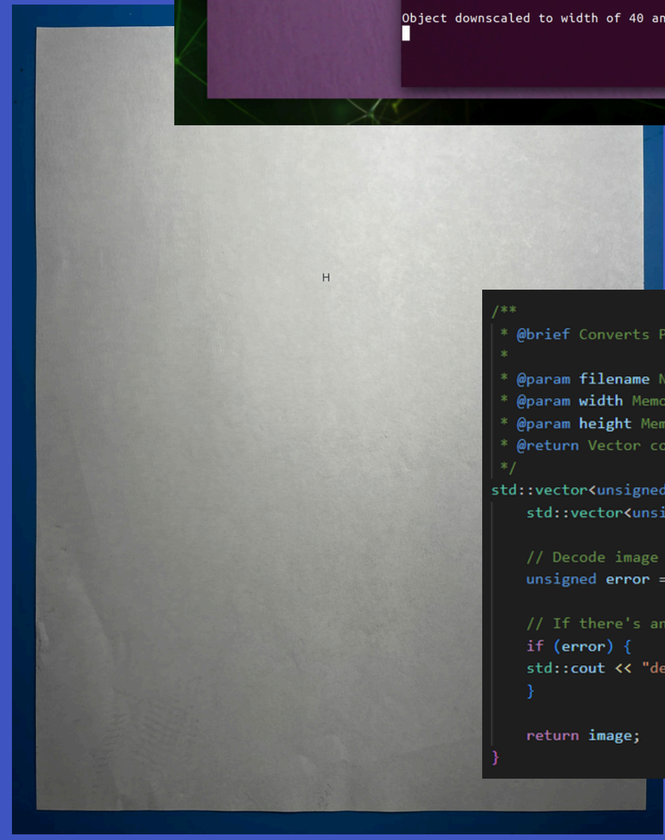


Image Preprocessing

- Camera captures image and saves it as *captured_image.png*
- OpenCV algorithm runs to detect objects and saves their coordinates
- Object extracted from *captured_image.png*, downscaled to 40x60, and saved as *extracted_image.png*
- *extracted_image.png* is passed to the *png2raw* function to save its raw pixel data in a 1D array
- Image array is grayscaled & normalized and used as input layer to NN



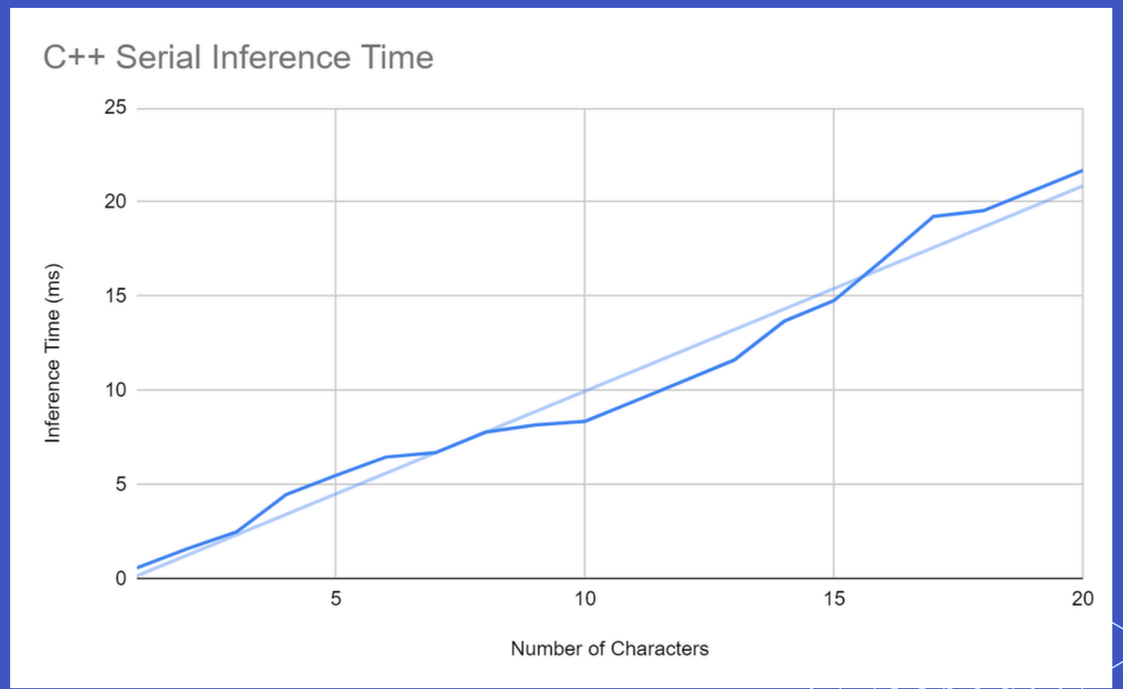



```cpp
/**
 * @brief Converts PNG file from disk to raw pixel data in memory. Used only for still images.
 *
 * @param filename Name of image to process.
 * @param width Memory address of the image width.
 * @param height Memory address of the image height.
 * @return Vector containing raw pixel data stored represented as an unsigned char.
 */
std::vector<unsigned char> png2raw(const char* filename, unsigned& width, unsigned& height) {
    std::vector<unsigned char> image; // used to store the raw pixels

    // Decode image
    unsigned error = lodepng::decode(image, width, height, filename);

    // If there's an error, display it
    if (error) {
    std::cout << "decoder error " << error << ": " << lodepng_error_text(error) << std::endl;
    }

    return image;
}
```

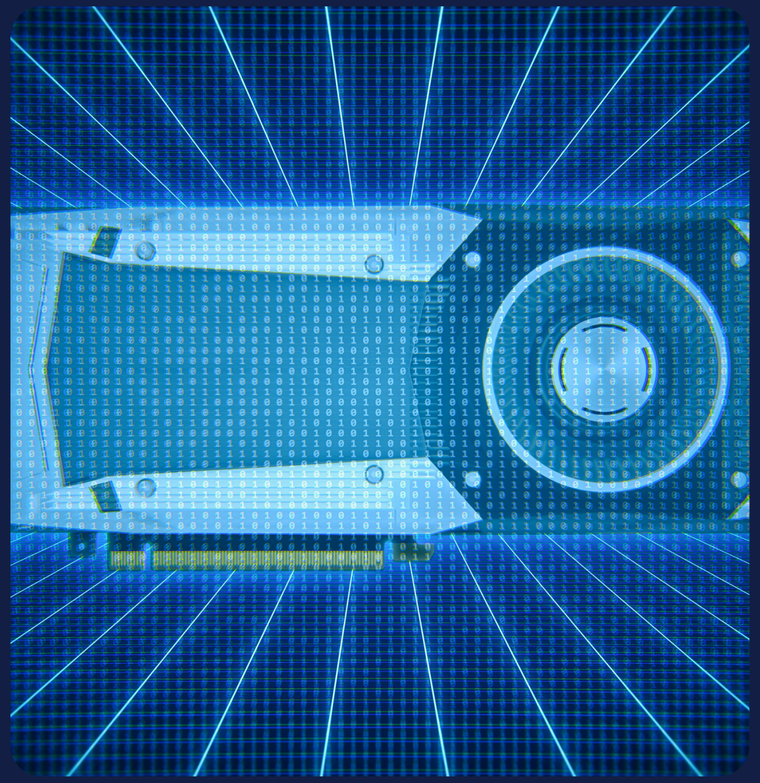
Performance Metrics

Initial C++ Program With Still Images

	Input	Output	% Certanity
Version 1			
2	t	43.35
4	4	74.56
B	t	45.19
H	y	82.6
S	p	37.41
W	y	43.93
p	t	55.74
r	S	58.91
t	4	40.55
y	4	41.25



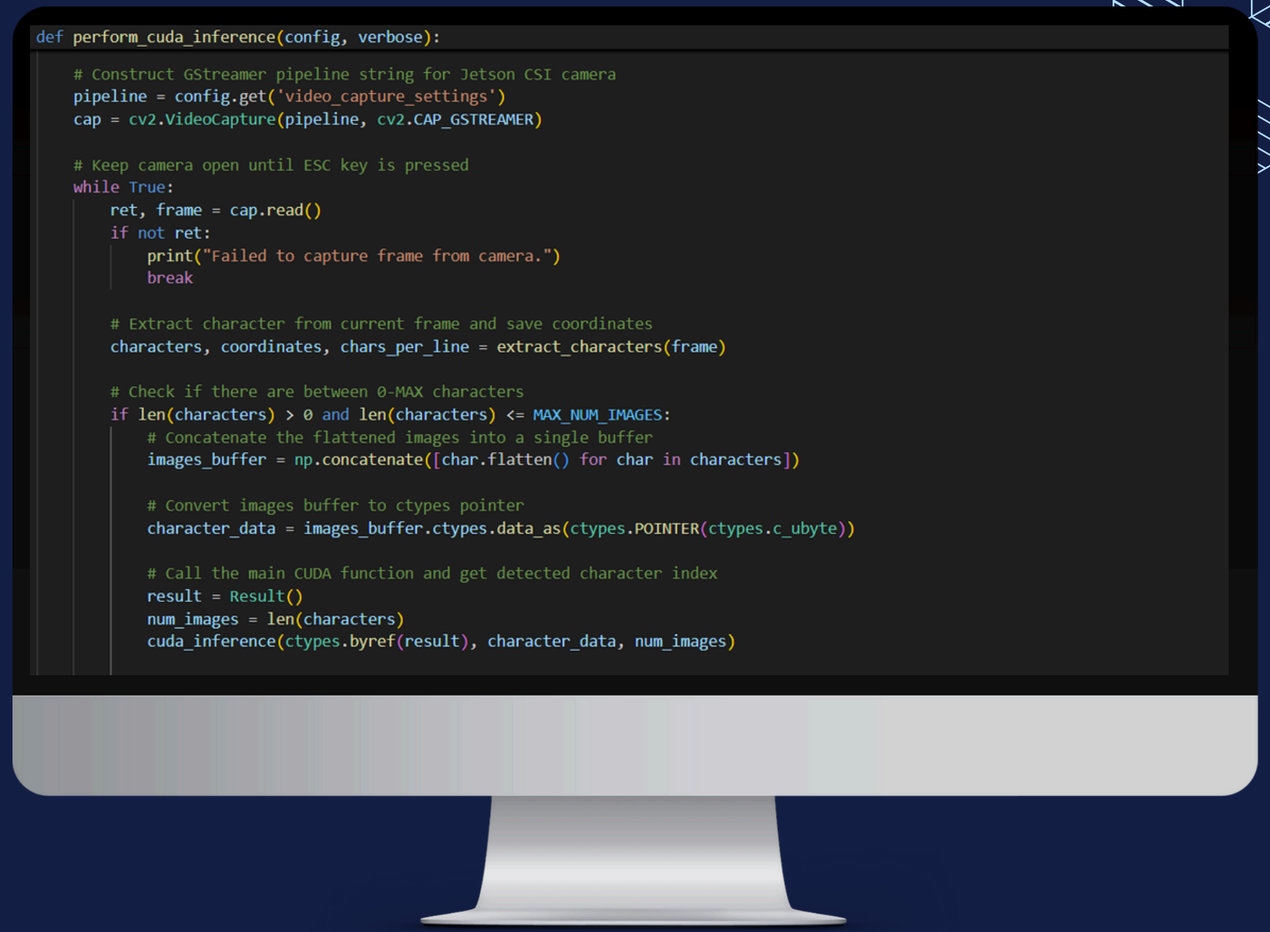
GPU Optimizations



- NN make predictions by performing matrix-vector multiplications using weights and biases to transform input data through each layer
- Made use of the CUDA programming language to parallelize NN computations
- Each CUDA thread computes a single row-column pair in parallel accelerating the multiplication process
- Implemented an initialization function to allocate all necessary device memory before runtime

Live Video Integration

- Program would run continuously on frames as they would come in
- CTypes library was used to facilitate the data exchange between the Python OpenCV program and the CUDA program
- Python program passes an image buffer of the extracted images directly, removing the need to write/read files

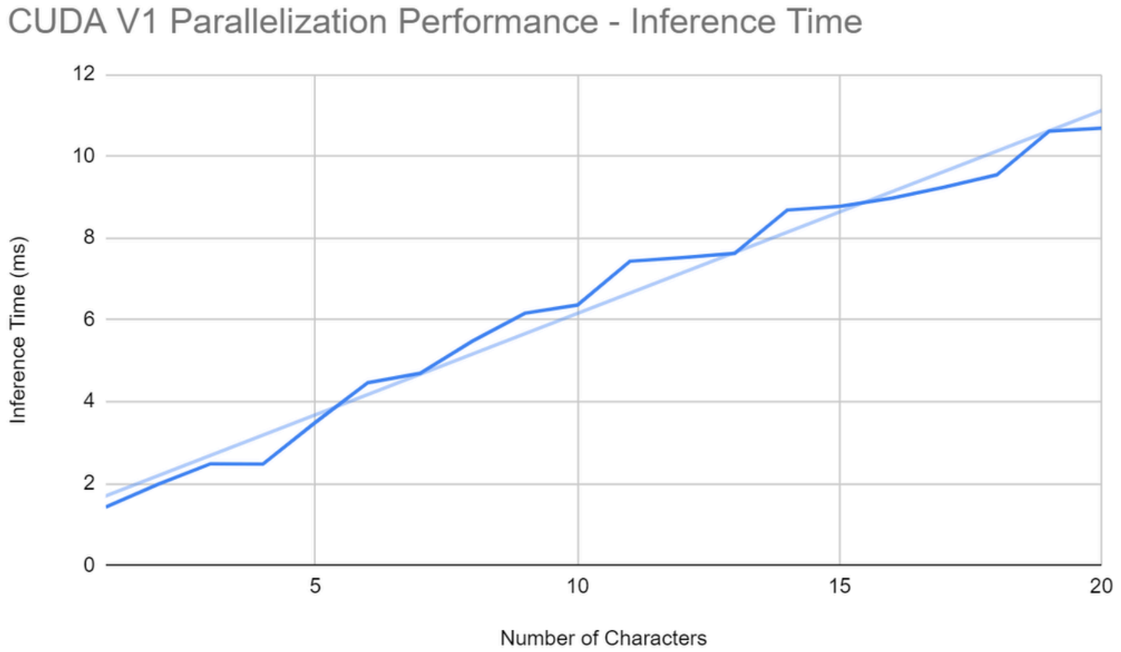


```python
def perform_cuda_inference(config, verbose):

    # Construct GStreamer pipeline string for Jetson CSI camera
    pipeline = config.get('video_capture_settings')
    cap = cv2.VideoCapture(pipeline, cv2.CAP_GSTREAMER)

    # Keep camera open until ESC key is pressed
    while True:
        ret, frame = cap.read()
        if not ret:
            print("Failed to capture frame from camera.")
            break

        # Extract character from current frame and save coordinates
        characters, coordinates, chars_per_line = extract_characters(frame)

        # Check if there are between 0-MAX characters
        if len(characters) > 0 and len(characters) <= MAX_NUM_IMAGES:
            # Concatenate the flattened images into a single buffer
            images_buffer = np.concatenate([char.flatten() for char in characters])

            # Convert images buffer to ctypes pointer
            character_data = images_buffer.ctypes.data_as(ctypes.POINTER(ctypes.c_ubyte))

            # Call the main CUDA function and get detected character index
            result = Result()
            num_images = len(characters)
            cuda_inference(ctypes.byref(result), character_data, num_images)
```

Performance Metrics

CUDA Program With Live Video

	Input	Output	% Certanity
Version 2			
4	4	4	100
2	2	2	92.79
Y	Y	Y	99.81
L	L	L	99.71
7	7	7	100
U	U	U	56.93
6	6	6 or 8	61.35
B	B	b	95.72
W	W	N	82.6
p	p	S	37.41
r	r	M	43.93
G	G	U	55.74



CUDA Optimizations

- Utilized CUDA Streams so that characters would be processed in parallel
- Employed asynchronous CUDA memory copies to transfer data back to the host while other characters were still processing
- Implemented Contrast Stretching to improve image clarity
- Added white space around detected character to prevent clipping

```cpp
// Grayscale / Normalize the captured images and save it into device memory asynchronously
std::vector<std::vector<float>> h_input_layers(num_images);
for (int i = 0; i < num_images; i++) {
    h_input_layers[i] = grayscale_and_normalize(image_data, (i * (WIDTH * HEIGHT * 3)));
    cudaMemcpyAsync(characters[i].d_input_layer, h_input_layers[i].data(),
    ROWS_LAYER1 * sizeof(float), cudaMemcpyHostToDevice, characters[i].stream);
}

// Perform feed forward operation and time the duration
auto start = std::chrono::high_resolution_clock::now();
// Launch CUDA kernels for each character
for (int i = 0; i < num_images; i++) {
    // Layer 1
    int blocks_per_grid = (COLS_LAYER1 + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    matrix_vector_multiplication<<<blocks_per_grid, THREADS_PER_BLOCK, 0, characters[i].stream>>>
    (d_weights[0], characters[i].d_input_layer, characters[i].d_results[0], ROWS_LAYER1, COLS_LAYER1);
    add_bias<<<blocks_per_grid, THREADS_PER_BLOCK, 0, characters[i].stream>>>
    (characters[i].d_results[0], d_biases[0], COLS_LAYER1);
    relu<<<blocks_per_grid, THREADS_PER_BLOCK, 0, characters[i].stream>>>(characters[i].d_results[0], COLS_LAYER1);

    // Layer 2
    blocks_per_grid = (COLS_LAYER2 + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    matrix_vector_multiplication<<<blocks_per_grid, THREADS_PER_BLOCK, 0, characters[i].stream>>>
    (d_weights[1], characters[i].d_results[0], characters[i].d_results[1], ROWS_LAYER2, COLS_LAYER2);
    add_bias<<<blocks_per_grid, THREADS_PER_BLOCK, 0, characters[i].stream>>>
    (characters[i].d_results[1], d_biases[1], COLS_LAYER2);
    relu<<<blocks_per_grid, THREADS_PER_BLOCK, 0, characters[i].stream>>>(characters[i].d_results[1], COLS_LAYER2);
```

Contrast Stretching Formula:

$$P_{out} = (P_{in} - c)\left(\frac{b-a}{d-c}\right) + a$$

a = lowest possible value: 0

b = largest possible value: 255

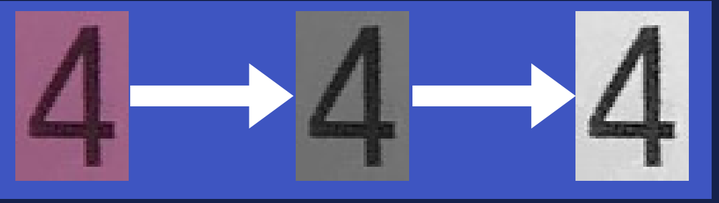
c = image minimum value

d = image maximum value

P_{out} = output pixel

P_{out} = input pixel

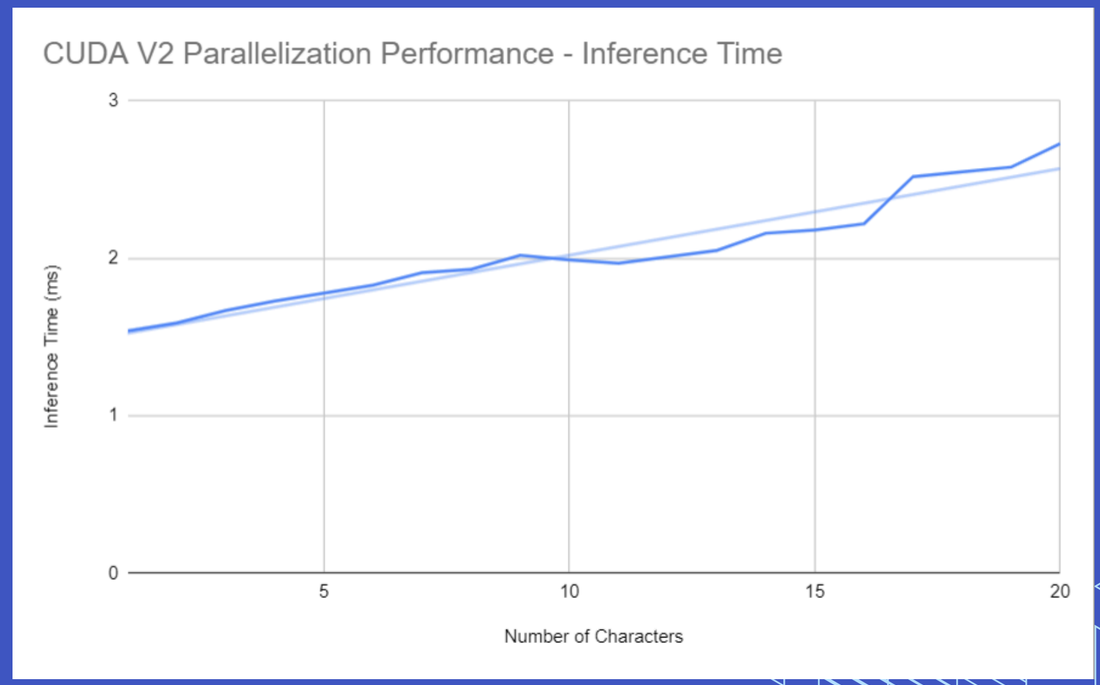
```cpp
// Contrast stretch the image then normalize to the range of (0, 1)
// Pout = (Pin - c)(b - a/d - c) + a
// a = 0, b = 255, c = min pixel value, d = max pixel value
// Find min and max
auto min_max = std::minmax_element(grayscale_image.begin(), grayscale_image.end());
float c = *min_max.first;
float d = *min_max.second;

// Contrast stretch the image
for (unsigned y = 0; y < HEIGHT; y++) {
    for (unsigned x = 0; x < WIDTH; x++) {

        // Contrast stretch formula then normalize to the range of (0, 1)
        grayscale_image[y * WIDTH + x] = ((grayscale_image[y * WIDTH + x] - c) *
        (255.0f / (d - c))) / 255.0f;
    }
}
return grayscale_image;
```



Performance Metrics

Optimized CUDA Program
With Live Video

	Input	Output	% Certanity
Version 3			
4	4	100	
2	2	92.79	
Y	Y	99.81	
L	L	99.71	
7	7	100	
U	U	88.3	
X	X	90.98	
3	3	80.49	
8	8	82.6	
J	J	99.99	
V	V	99.82	
O	O	100	



Program Demo

Activities Terminal Jul 25 10:53 15W

agonzalez@ubuntu: ~/Desktop/demo/jetson-ocr/code/inference

```
agonzalez@ubuntu:~/Desktop/demo/jetson-ocr/code/inference$ python3 perform_inference.py
GST_ARGUS: Creating output stream
CONSUMER: Waiting until producer is connected...
GST_ARGUS: Available Sensor modes :
GST_ARGUS: 4032 x 3040 FR = 21.000000 fps Duration = 47619048 ; Analog Gain range min 1.000000, max 2
2.250000; Exposure Range min 13000, max 683709000;

GST_ARGUS: 3840 x 2160 FR = 29.999999 fps Duration = 33333334 ; Analog Gain range min 1.000000, max 2
2.250000; Exposure Range min 13000, max 683709000;

GST_ARGUS: 1920 x 1080 FR = 59.999999 fps Duration = 16666667 ; Analog Gain range min 1.000000, max 2
2.250000; Exposure Range min 13000, max 683709000;

GST_ARGUS: Running with following settings:
   Camera index = 0
   Camera mode  = 2
   Output Stream W = 1920 H = 1080
   seconds to Run    = 0
   Frame Rate = 59.999999
GST_ARGUS: Setup Complete, Starting captures for 0 seconds
GST_ARGUS: Starting repeat capture requests.
CONSUMER: Producer has connected; continuing.
[ WARN:0@2.656] global cap_gstreamer.cpp:1728 open OpenCV | GStreamer warning: Cannot query video pos
ition: status=0, value=-1, duration=-1
```

Open

cuda_inference.cu perform_inference.py

Single Character

```
                    // Grayscale formula
                    float gray = 0.299f * image_data[index] + 0.587f * image_data[index + 1]
+ 0.114f * image_data[index + 2];
                    grayscale_image[y * WIDTH + x] = gray;
                }
            }

            // Contrast stretch the image then normalize to the range of (0, 1)
            // Pout = (Pin - c)(b - a/d - c) + a
            // a = 0, b = 255, c = min pixel value, d = max pixel value
            // Find min and max
            auto min_max = std::minmax_element(grayscale_image.begin(),
grayscale_image.end());
            float c = *min_max.first;
            float d = *min_max.second;

            // Contrast stretch the image
            for (unsigned y = 0; y < HEIGHT; y++) {
                for (unsigned x = 0; x < WIDTH; x++) {

                    // Contrast stretch formula then normalize to the range of (0, 1)
                    grayscale_image[y * WIDTH + x] = ((grayscale_image[y * WIDTH + x] - c) *
(255.0f / (d - c))) / 255.0f;
                }
            }

            return grayscale_image;
        }

        /**
         * @brief Performs the Softmax activation function to calculate probability
distribution.
         *
         * @param output_layer 1D array (vector) representing the output layer of the neural
network.
         * @return A probability distribution representing the network's prediction of the given
image.
         */
        std::vector<float> softmax(const std::vector<float>& output_layer) {
            std::vector<float> probability_distribution(output_layer.size());
            float max_value = *max_element(output_layer.begin(), output_layer.end()); // find
the max value in the layer

            // Compute the exponential of each element minus the maximum input value
            float sum = 0.0;
            for (size_t neuron = 0; neuron < output_layer.size(); neuron++) {
                probability_distribution[neuron] = std::exp(output_layer[neuron] - max_value);
                sum += probability_distribution[neuron];
            }

            // Normalize the output to get probabilities
            for (size_t neuron = 0; neuron < probability_distribution.size(); neuron++) {
```

CUDA Tab Width: 8 Ln 223, Col 13 INS

Proposed Optimizations

- Transition from a Fully Connected Neural Network to a Convolutional Neural Network
- Implementation of a Hidden Markov Model with a Viterbi Algorithm to increase word and sentence detection accuracy
- Refinement of dataset and training process to better differentiate between characters of similar appearance
- Make use of shared memory to further accelerate inference time

Learning Takeaways

- Low-Level Neural Network Implementation and Structure
- Memory Management Techniques
- CUDA Programming and GPU Architecture
- Hardware Debugging Skills
- Driver Compatibility and System Stability
- Foundations of Computer Vision
- Digital Image Representation
- Interoperability and Data Exchange

THANK YOU

A2e Technologies

By Antonio Gonzalez