FYP PROGRESS REPORT:

MINIMAL PROTOTYPE OF CNN ACCELERATOR IN MATLAB

GROUP MEMBERS:
ISMAIL KHAN (El-22075)
EMAN JAVED (El-22050)
MUHAMMAD ZAID (El-22066)
SHAHMIR AMIR (El-22123)

1. Introduction

- **State the purpose:** *The goal of this project is to design a CNN accelerator at the RTL level. As a first step, we implemented and analyzed a simple two-layer CNN model in MATLAB to understand the computational flow, data dependencies, and memory requirements.*

- **Mention motivation:** *This prototype serves as a baseline to explore optimization strategies and identify hardware design challenges (compute bottlenecks, memory bandwidth, data reuse).*

2. Methodology

2.1 Model Architecture (Minimal Prototype)

- Input: 8×8 grayscale image.
- Conv1: 3×3 vertical edge detector → 6×6 output.
- ReLU: removes negative values.
- Pool1: 2×2 max pooling → 3×3 output.
- Conv2: 3×3 horizontal edge detector → 1×1 output.
- ReLU: thresholding.
- Fully Connected Layer: maps single feature to 3 class scores.

Key Point: Layer 1 extracts primitive features (vertical edges). Layer 2 builds higher-level patterns (horizontal arrangements of vertical edges). This illustrates how CNNs create feature hierarchies.

2.2 Implementation in MATLAB

To validate the CNN workflow, a two-layer CNN prototype was implemented in MATLAB. The model was coded manually using nested loops and basic arithmetic, without any external deep learning libraries. This allowed step-by-step observation of convolution, ReLU, pooling, and fully connected operations.

- ***Input: 8×8 grayscale image (represented as a numeric matrix).***
- ***Conv1: 3×3 vertical edge detector → produces 6×6 feature map.***
- ***ReLU1: replaces negative values with zero.***
- ***Pool1: 2×2 max pooling → reduces to 3×3.***
- ***Conv2: 3×3 horizontal edge detector → reduces to 1×1.***
- ***ReLU2: threshold applied.***
- ***Fully Connected: maps scalar result into 3 output scores.***

Intermediate results (Conv1 feature map and Pool1 output) were printed and visualized to verify correct layer behavior. Operation counts confirmed that Conv1 dominates computation (~95% of total MACs).

1. **Input Image:**

```
% Step 1: Input image (8x8 matrix)
input_img = [0 0 0 0 0 0 0 0;
             0 9 9 9 0 1 1 0;
             0 9 9 9 0 1 1 0;
             0 9 9 9 0 1 1 0;
             0 0 0 0 0 0 0 0;
             1 1 0 0 2 2 2 0;
             1 1 0 0 2 2 2 0;
             0 0 0 0 0 0 0 0];
```

2. **Kernel and Conv1:**

Below snippet shows how vertical edge detector filter (Kernel) has been defined in MATLAB and the manual sliding of kernel across 3×3 patch of input image by using nested-loops and basic arithmatic operators.

```
% Step 2: Conv1 (3x3 vertical edge filter)
kernel1 = [-1 0 1;
           -1 0 1;
           -1 0 1];
out1 = zeros(6,6); % (8-3+1) = 6

for i = 1:6
    for j = 1:6
        patch = input_img(i:i+2, j:j+2);   % take 3x3 patch
        out1(i,j) = sum(sum(patch .* kernel1)); % dot product
    end
end
```

OUTPUT OF CONV1:

```
Conv1 output (6x6):
    18     0     0     0     2     0
    27     0     0     0     3     0
    18     0     0     0     2     0
     8     0     0     0     1     0
     0     0     4     4     0     0
     0     0     4     4     0     0
```

3. ReLU1 and POOL1:

- ReLU1 is working as an activation function to neglect weak values in the specific region and maintain strong values of the feature map(Conv1 output).
- POOL1 is used to reduce the size of the feature map by looking into 2×2 patches of feature map and choosing the maximum value.

```
%% Step 3: ReLU1
out1(out1 < 0) = 0;
```

```
%% Step 4: Pool1 (2x2 max pooling)
pool1 = zeros(3,3);
for i = 1:3
    for j = 1:3
        block = out1(2*i-1:2*i, 2*j-1:2*j);
        pool1(i,j) = max(block(:));
    end
end
```

OUTPUT:

```
Pool1 output (3x3):
    27     0     3
    18     0     2
     0     4     0
```

4. CONV2 and ReLU2:

Horizontol edge detector (Kernel) is used to figure out horizontol edges from the reduced feature map of layer1 and again ReLU2 is used to neglect weak signals.

```
%% Step 5: Conv2 (3x3 horizontal edge filter)
kernel2 = [-1 -1 -1;
            0  0  0;
            1  1  1];
patch = pool1(1:3,1:3); % whole 3x3
out2 = sum(sum(patch .* kernel2)); % gives 1x1 output
```

```
%% Step 6: ReLU2
if out2 < 0
    out2 = 0;
end
```

OUTPUT:

```
Conv2 output (1x1):
     0
```

5. FULLY CONNECTED LAYER:

The fully connected layer combines the features of the 2 layers and make a final decision based on the features. Weights and bias are randomly chosen to depict its working. In real system the model figure it out itself from the training data.

```
%% Step 7: Fully Connected layer (3 outputs)
% Flatten -> here it's just one number out2
fc_weights = [0.5; -0.2; 0.8]; % random demo weights
fc_bias = [0.1; 0; -0.1];
fc_out = fc_weights * out2 + fc_bias;
```

OUTPUT:

```
Final FC outputs (3 classes):
    0.1000         0   -0.1000
```

3. RESULTS

3.1 Intermediate Outputs:

- **Conv1 feature map (6×6):** highlights vertical edges in input image.
- **Pool1 output (3×3):** compressed representation, retaining strongest activations.
- **Conv2 output (1×1):** single scalar feature capturing higher-level pattern.
- **FC outputs (3 scores):** classification scores; highest score = predicted class.

3.2 Compute Analysis:

- Conv1: (6×6 positions × 9 multiplications) = **324 MACs**.
- Conv2: (1×1 × 9 multiplications) = **9 MACs**.
- FC: (3 multiplications) = **3 MACs**.
- **Total ≈ 336 MACs**.
- Observation: Conv1 dominates computation (>95%).

3.3 Memory Analysis:

- Weights: Conv1 (9), Conv2 (9), FC (3) → ~21 parameters.

- Feature maps: Input (64), Conv1 (36), Pool1 (9), Conv2 (1).
- Memory footprint is small, but relative bandwidth usage highlights importance of reusing Conv1 weights and buffering feature maps.

4. Hardware Perspective

- **Conv1 is the bottleneck:** most of the MACs and data movement occur here → this layer must be optimized with line buffers, weight reuse, and local SRAM.
- **ReLU and Pooling:** cheap operations, simple comparators and max units.
- **Conv2 and FC:** very lightweight in this toy model; in deeper networks, later layers will also grow.
- **Data reuse:** once input rows are fetched into SRAM, sliding kernels should reuse them to reduce DRAM access.
- **Observation:** first prototype confirms that DRAM → SRAM buffering and Conv1 optimization are critical directions for accelerator design.

5. Next Steps

In the next phase, the prototype will be extended by adding multiple filters, using larger and realistic image inputs, expanding layer depth, and logging compute/memory statistics per layer. This will allow us to gradually transition from toy examples toward practical CNN models while still staying within MATLAB for conceptual clarity.