RRAM-based Compute In-Memory Engine for Alphabet Classification

Course Project
E6347 : Devices for AI and Neuromorphic Computing
GitHub

Jay Gupta (AE21B026)
Vaibhav Krishna Garimella (EP21B040)

Dataset Creation

The goal of this project was to perform alphabet classification using an In-Memory Computing engine using Non-Volatile Memories. The alphabets A, T, V and X were to be fit in 4x4 grids. We used 2 approaches for the same, one for the training set and the other for the testing set. After making arrays of the ideal letters as shown in [1], we introduced a noise factor, which would pick a random pixel out of the 16 and make it 1. This would ensure that we would have ideal images as well as some noisy images. For the training set, we used a noise factor of 1 and for the testing set, we used a noise factor of 2. This was to see if the model is able to predict noisier images well enough. Some samples of training and testing images are shown in Figure 1 and Figure 2 respectively.

Code for Noise Addition

```python
def add_noise(grid, noise_pixels):
    noisy_grid = grid.copy()
    pixels = [(i, j) for i in range(grid.shape[0]) for j in range(grid.shape[1])]
    # Shuffle zero positions to select random ones
    random.shuffle(pixels)
    # Make the specified pixels ones
    for x, y in pixels[:noise_pixels]:
        noisy_grid[x, y] = 1
    return noisy_grid
```

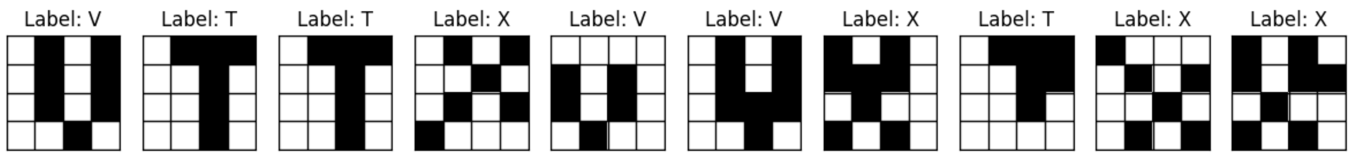


Figure 1: Training Samples (shown 8/500)

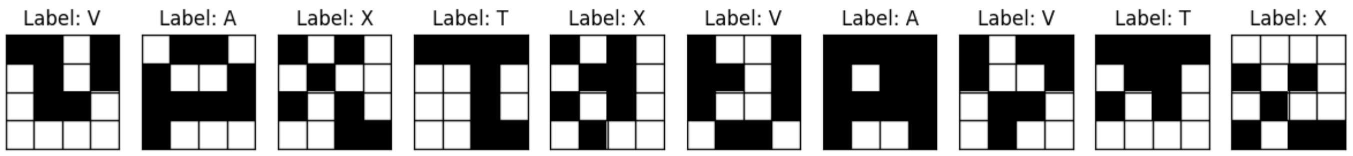


Figure 2: Testing Samples (shown 8/100)

Hardware-Aware Training

The main challenge during training was to account for the fact that weights and biases were to be binarized to program RRAM cells. Vanilla training code followed by binarization based on the mean threshold yielded very poor results. So, the first change we made was the initialization range of the weights and biases.

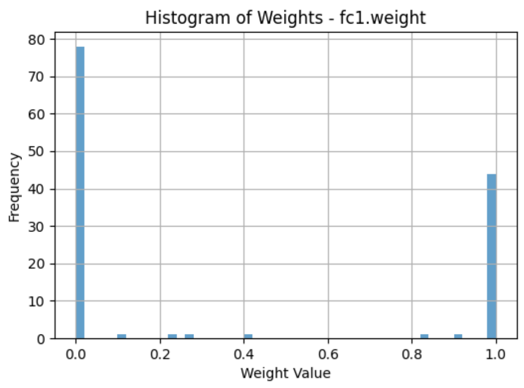
```
def initialize_binary_friendly_weights(layer):
    nn.init.uniform_(layer.weight, 0.0, 1.0)
    nn.init.constant_(layer.bias, 0.5)

initialize_binary_friendly_weights(model.fc1)
initialize_binary_friendly_weights(model.fc2)
```

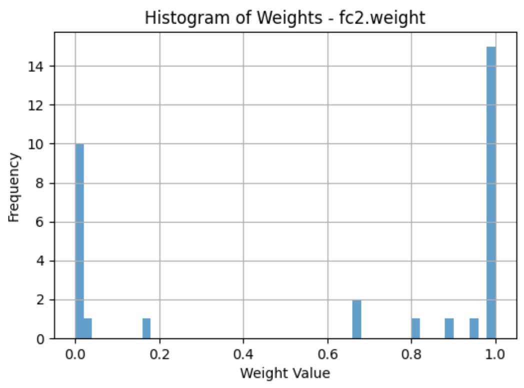
But, just setting the initial range wasn't sufficient since we needed the weights to be close to 0 or 1 in order to minimise the binarization error. For this, we added an extra loss term that penalized any weights that strayed away from the extremes 0 and 1.

```
def weight_regularization_loss(model):
    reg_loss = 0.0
    for name, param in model.named_parameters():
        if 'weight' in name:
            reg_loss += torch.sum(torch.minimum(param ** 2, (1 - param) ** 2))
    return reg_loss
...
...
classification_loss = criterion(outputs, labels)
reg_loss = weight_regularization_loss(model)
total_loss = classification_loss + lambda_reg * reg_loss
```

These modifications to the vanilla training code gave us very good inference results. The final weights' histogram below show that they have been approximately binarized.



(a) Layer 1 Weights



(b) Layer 2 Weights

Figure 3: Weight Histograms of both layers indicating approximate binarization

Inference

Software Inference

Plain software inference gave an accuracy of 71% on the testing dataset. But, to account for the hardware during inference, the following modifications were made to the forward pass:

```python
def forward(self, x, min_val=0.0, max_val=1.0):
    # Threshold and binarize weights for fc1
    fc1_weight = self.fc1.weight.data
    fc1_bias = self.fc1.bias.data
    threshold_1 = torch.mean(fc1_weight).item()
    binarized_fc1_weight = torch.where(fc1_weight > threshold_1, max_val, min_val)
    binarized_fc1_bias = torch.where(fc1_bias > threshold_1, max_val, min_val)

    # Threshold and binarize weights for fc2
    fc2_weight = self.fc2.weight.data
    fc2_bias = self.fc2.bias.data
    threshold_2 = torch.mean(fc2_weight).item()
    binarized_fc2_weight = torch.where(fc2_weight > threshold_2, max_val, min_val)
    binarized_fc2_bias = torch.where(fc2_bias > threshold_2, max_val, min_val)

    # Compute forward pass with binarized weights
    x = (torch.matmul(x, binarized_fc1_weight.T) + binarized_fc1_bias)
    x = torch.clamp(x, min=0.0, max=1.0)
    # x now contains only 0s and 1s
    threshold_x = torch.mean(x).item()
    x_prog = torch.where(x > threshold_x, torch.tensor(1.0), torch.tensor(0.0))
    x = torch.matmul(x_prog, binarized_fc2_weight.T) + binarized_fc2_bias
    return x
```

The weights and biases have been binarized based on the mean threshold. It is important to note the **ABSENCE** of an activation function during inference. We noticed that since we have to binarize the outputs yet again to be fed into the second RRAM array, a squeezing function of sorts is incorporated. This modified inference resulted in an accuracy of **58%**

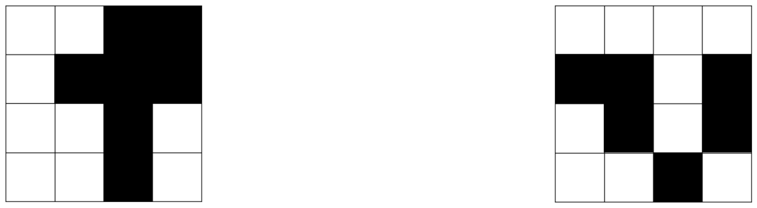


Figure 4: Two accurately predicted images (T & V) by the hardware-aware inference

Hardware Inference

Schematic Design

Our choice of Non-Volatile Memory was a Resistive RAM (RRAM) since it is easily programmable. Two arrays were designed, one for the hidden layer (17x8) and one of the output layer (9x4). The biases have been accounted for in the arrays. Figures 5 and 7 show the schematics for the two. The Multiply and Accumulate (MAC) output of the first layer is passed through a CCVS (so as to not disturb the RRAM read) and then compared with $0 + \delta$V (Figure 6). This essentially binarizes the non-zero MAC values to 1 and zero MAC values to 0 which replicates the software code shown above. The comparator output high is made to be 0.5V so that it can be used to perform the read operation in the second array. Note that NMOSes have been used as pass transistors to switch between programming and reading in both the arrays.

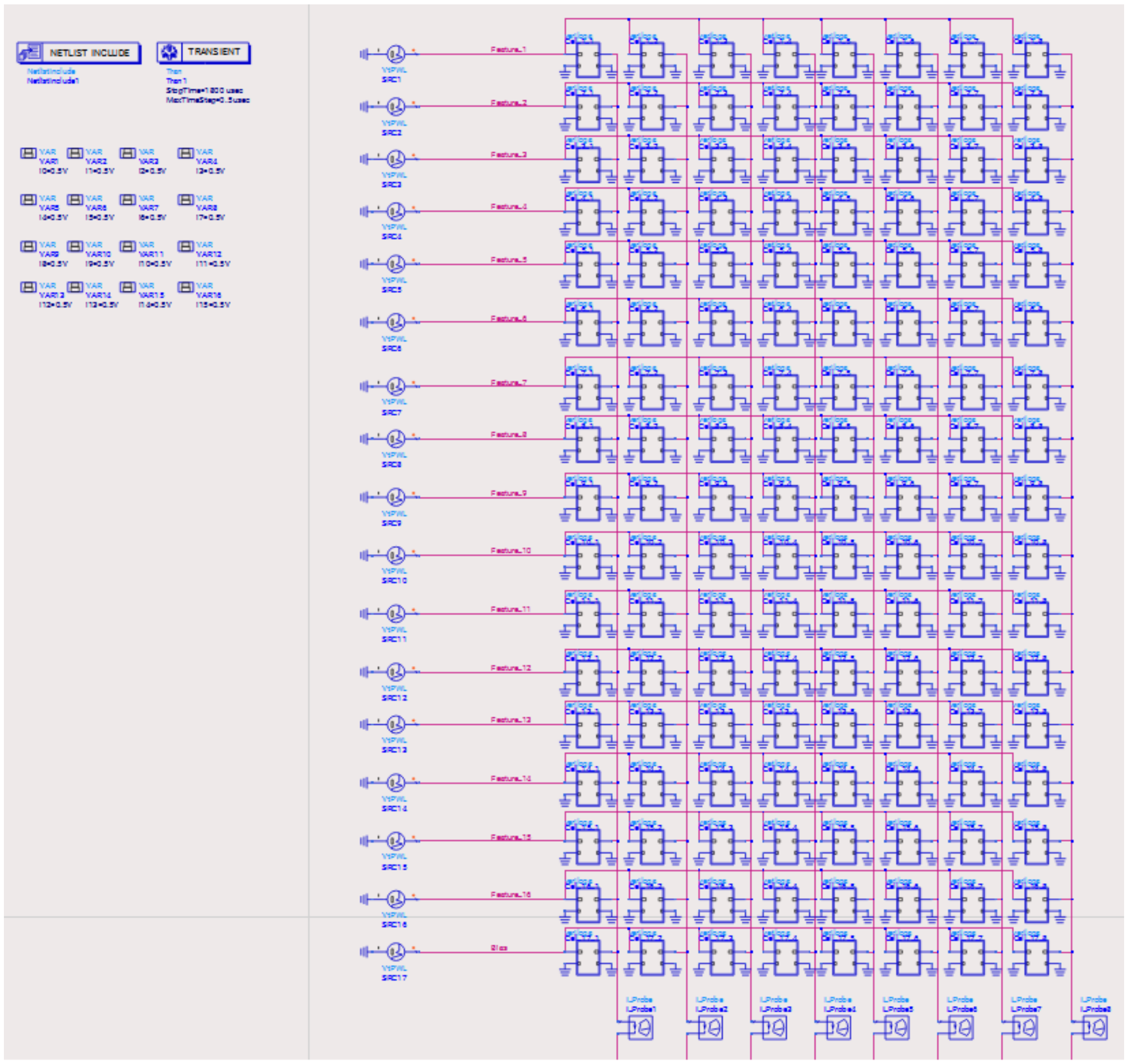


Figure 5: Schematic of Array 1

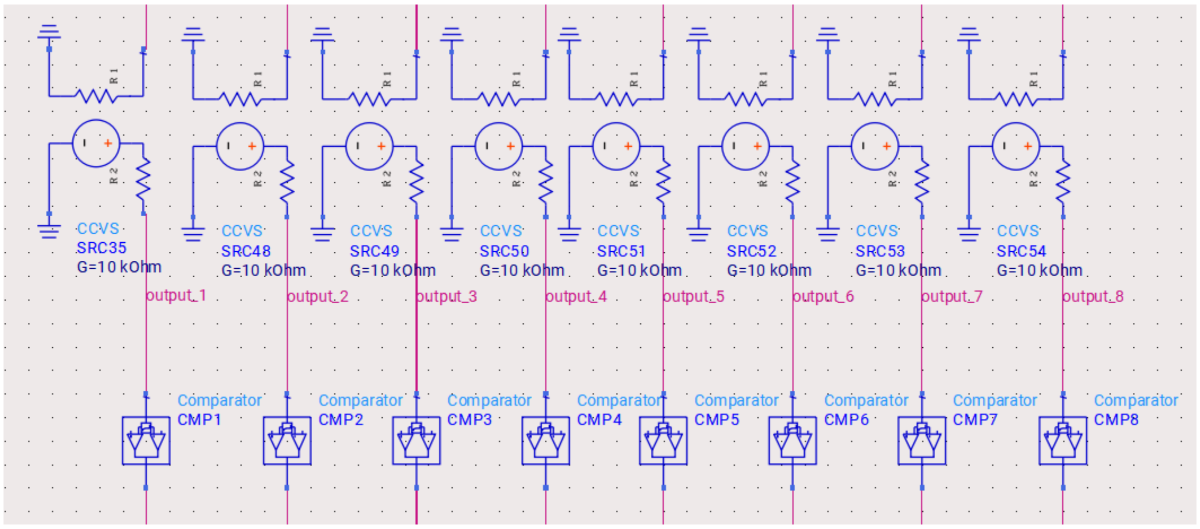


Figure 6: CCVS & Comparator Stage

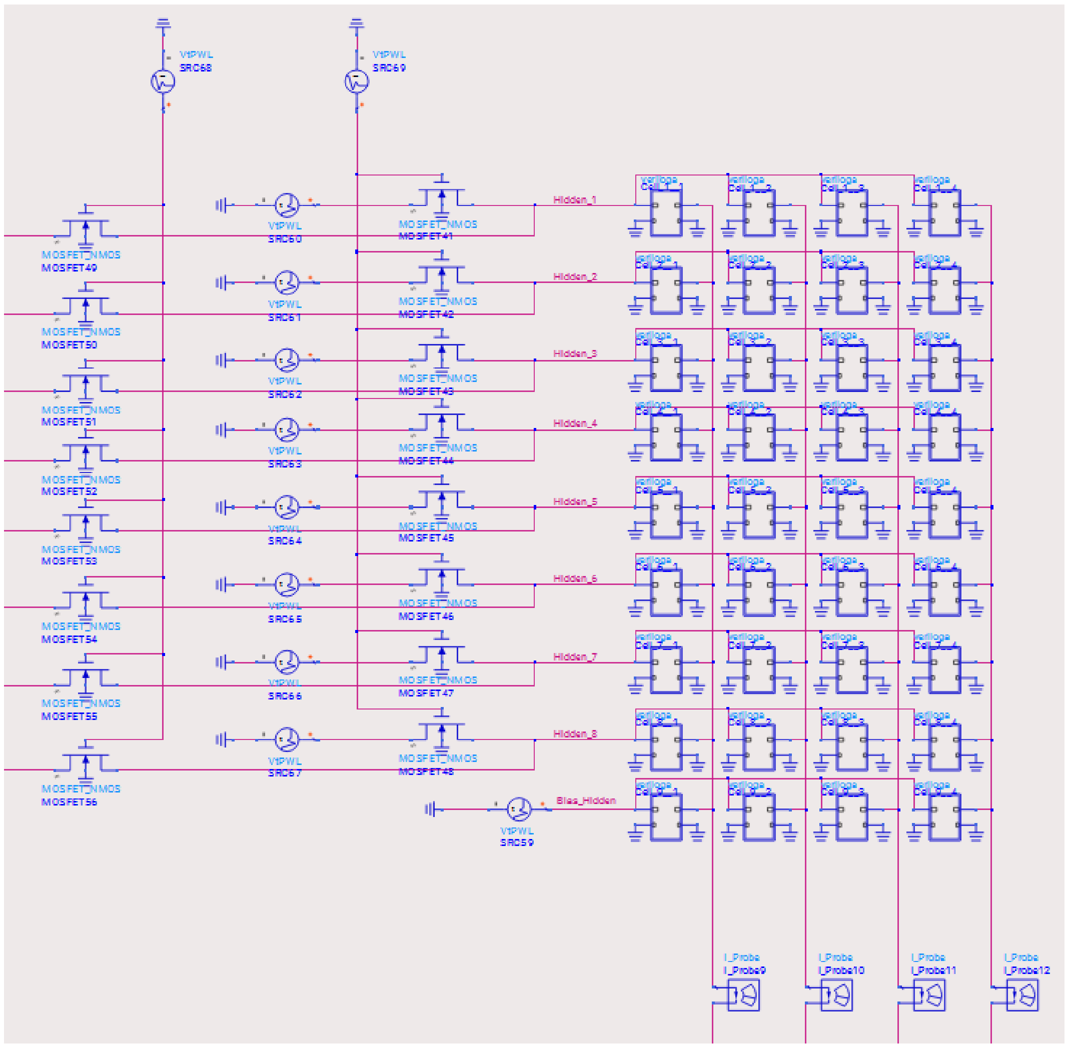


Figure 7: Schematic of Array 2

Simulation Results

We have used the V/3 scheme for programming with a programming voltage of 3V. Shown below are the Word Line and Bit Line voltages of the first array. The input combination is for the letter "X". Observe the Word Line voltage for the last 100us. This is the read voltage (input bit) for that particular row.

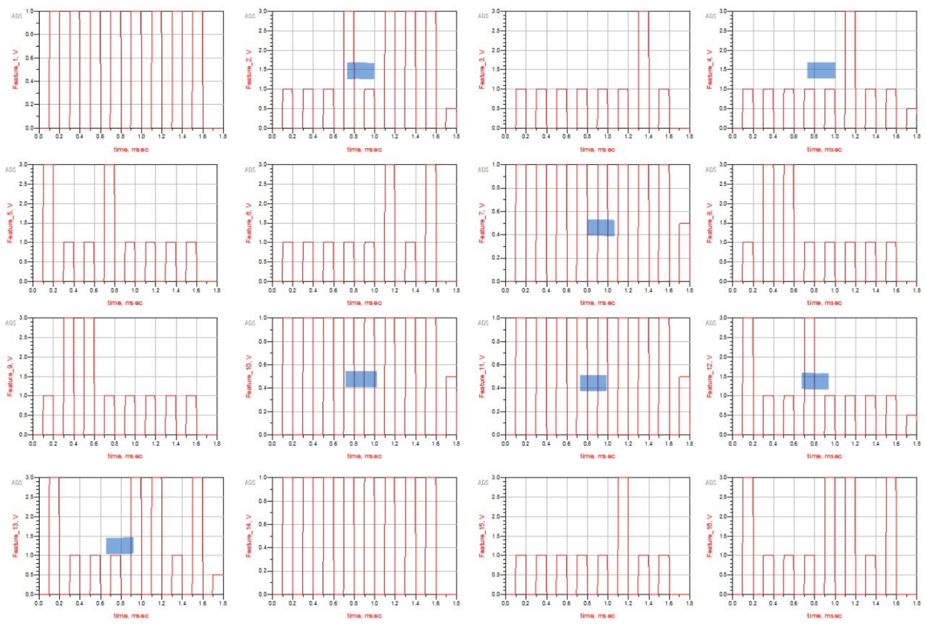


Figure 8: Word Line Voltages : Input X (marked)

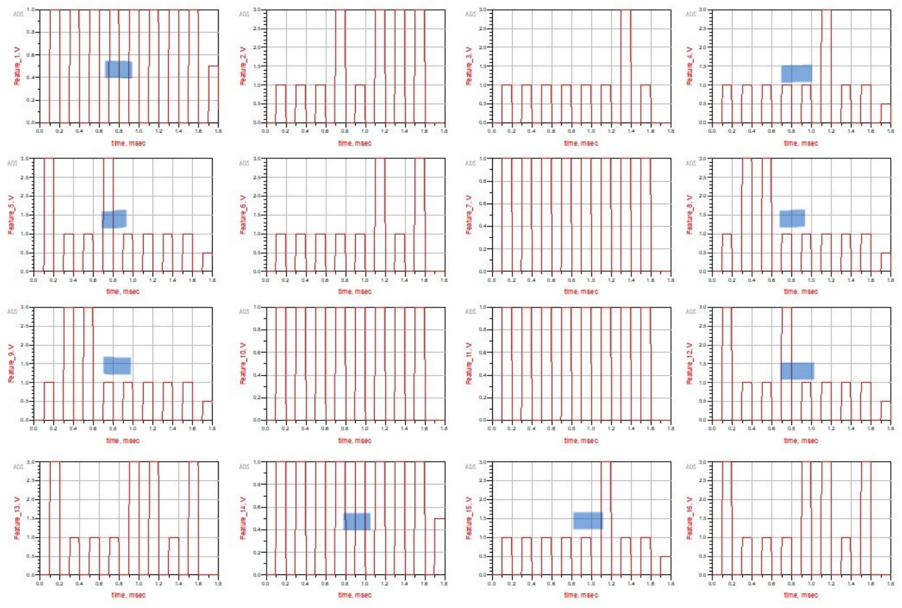


Figure 9: Word Line Voltages : Input V (marked)

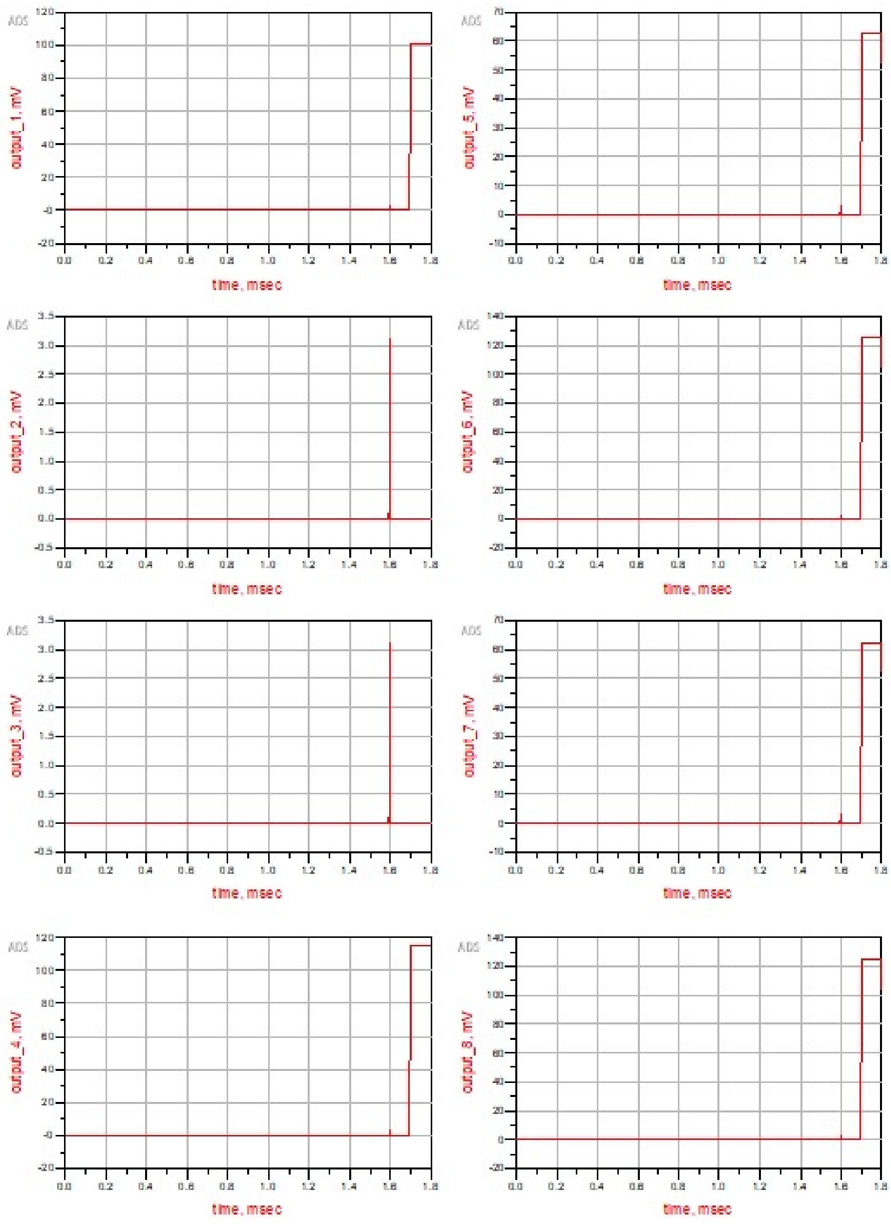


Figure 10: Layer 1 Output for X input : Matches the expected binary string '10011111'

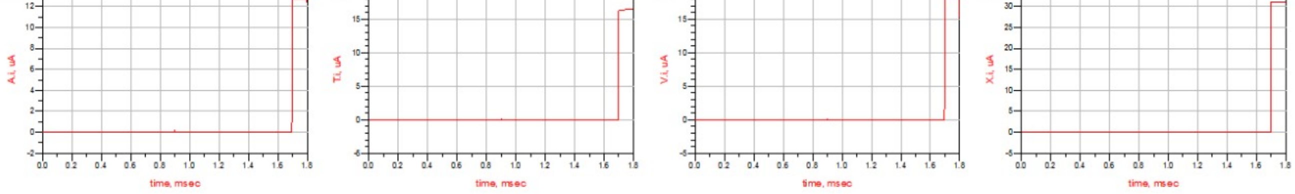


Figure 11: Layer 2 Output for X input : Predicted Class 3 (X) has the highest current output

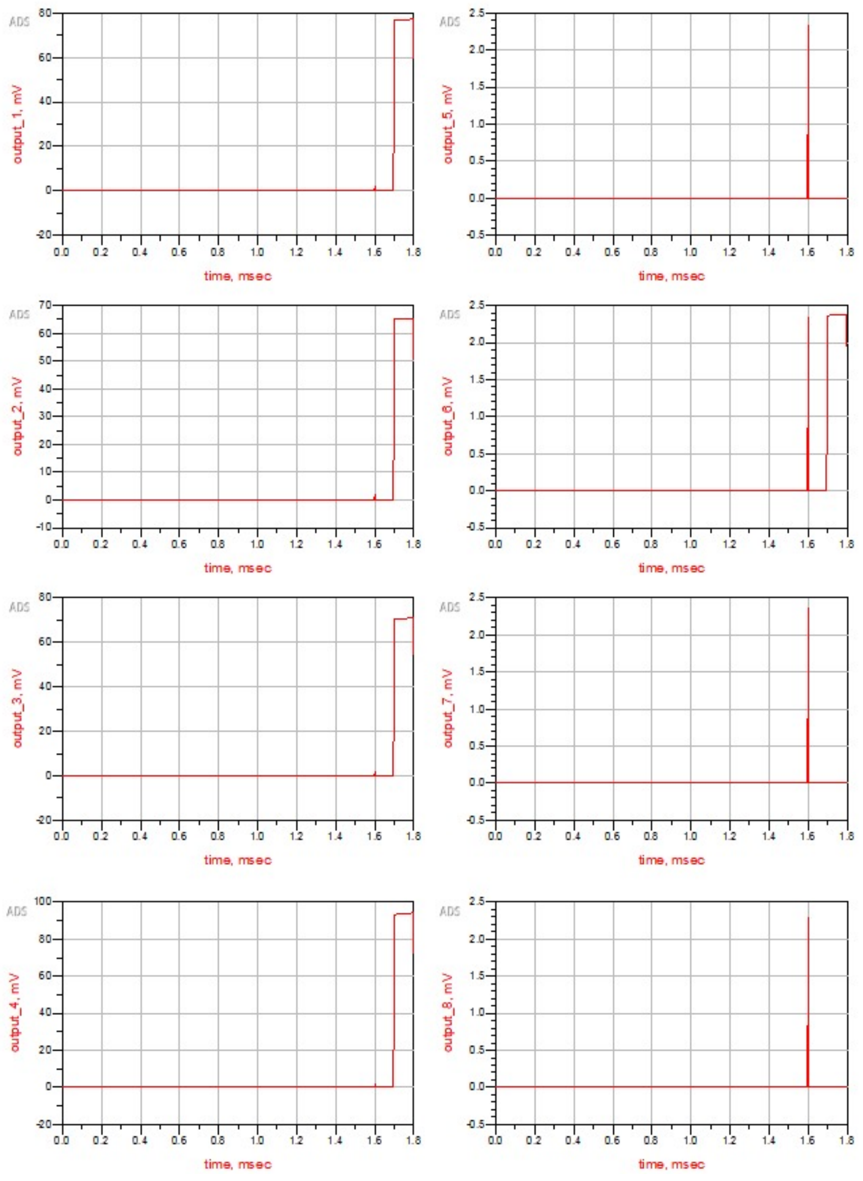


Figure 12: Layer 1 Output for V input : Matches the expected binary string ’11110100’

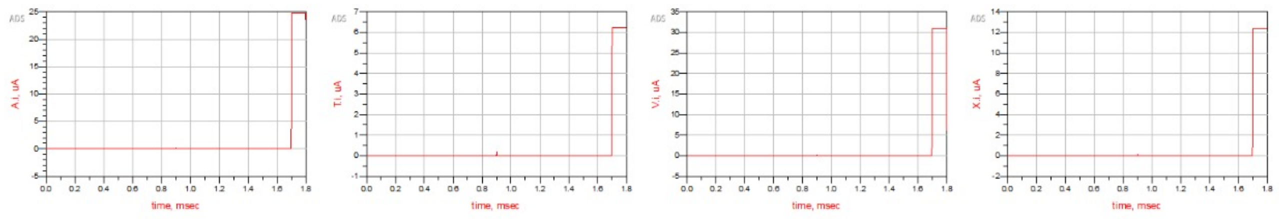


Figure 13: Layer 2 Output for V input : Predicted Class 2 (V) has the highest current output

References

[1] B Chakrabarti *et al*, Implementation of multilayer perceptron network with highly uniform passive memristive crossbar circuits

[2] Kuk-Hwan Kim *et al*, A Functional Hybrid Memristor Crossbar-Array/CMOS System for Data Storage and Neuromorphic Applications