Designing a cat detector with resistive memory

Resistive memory Neural Network idealization based on ex-situ training

Gustavo Duarte Tomaz de Sá

March 10, 2025

Contents

1 Introduction

This project aims to study a Resistive Memory implementation of In-Memory Computing for Neural Networks. The application chosen for this study is a simple Cat Detector based on a supervised model(labeled pre-annotated data). Firstly, we will study a purely computational approach, underlining training methodology and results. Then, we will explore a Physical-Aware implementation based on Memristors, since many research suggests that these implementations with Resistive Memories are highly energy efficient. In the latter the study will include a circuit to implement it and analysis of noise effects on the implementation. All data and implementations are done in Python using only numpy for calculations and some other libraries for data extraction.

2 A one-layer cat detector

A simple architecture of a neural network implements a Perceptron(abstracting from the mandatory usage of Heaviside as activation function), an algorithm that tries to mimic the neuron by following mathematical models defined around 1950 to describe a neuron. Following this algorithm with, of course, some changes in the initial proposed Perceptron by Frank Rosenblatt, we can easily implement classification and categorization for linearly separable data, where as for non-linearly separable data, the Universal Approximation Theorem states that a large enough DNN can approximate any function.

With this background in mind, we shall use one classical problem for Neural Networks, binary classification, to explore this later and further translate it to an hardware equivalent with Memristors. The proposed problem is to use a dataset distributed by Andrew Ng containing many cat and non-cat pictures, where we want to train the network to classify, when given new data, wether the picture inputted is a cat or not. The architecture proposed for this activity can be seen in the Figure 1.

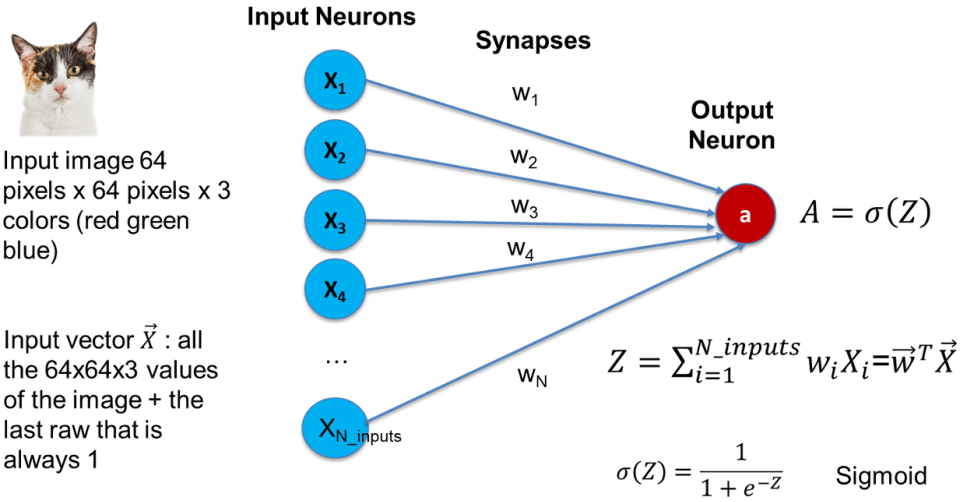


Figure 1: Proposed Architecture for the one-layer cat detector

Pay attention that, since this is a binary classification network, when generalizing, i.e. showing the network images that were not part of the train dataset, it won't be able to classify as anything other than cat or non-cat.

2.1 Dataset and architecture exploration

The dataset is composed by two HDF5 formatted data files, where one file contains the training dataset and the other the test dataset. These two are characterized by:
Train dataset:

- 209 images with 64x64 pixels, represented in RGB(shape : (209,64,64,3)). So each image has $64 \times 64 \times 3 = 12288$ useful data.
- 209 labels, called ground truth, that states if the picture is a cat (Y=1) or non-cat (Y=0).

Test dataset:

- 50 images with 64x64 pixels, represented in RGB(shape : (50,64,64,3)). So each image has $64 \times 64 \times 3 = 12288$ useful data.
- 50 labels, called ground truth, that states if the picture is a cat (Y=1) or non-cat (Y=0).

One usual approach when working with Neural Networks is to have the input layer to be as big as all the useful information one data point(in this case, an image) have. As we have seen, for each image in both training and test dataset, we have 12288 useful information for each image, each one representing the singular value RGB of a given pixel. A representation of one image from the dataset can be seen below in Figure 2, where matplotlib was used to show all 64x64 pixels of the image.

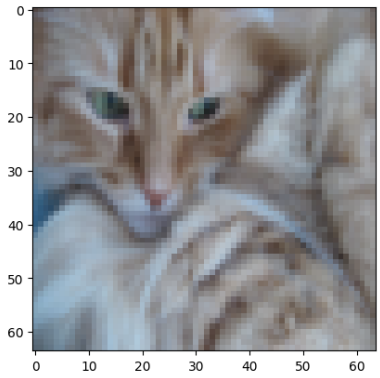


Figure 2: Plot of one picture from the dataset

Knowing that each data point presents 12288 useful information, we flatten the training dataset into a vector X, which will be as the same size of the input layer of the neural network. In fact, it will be flattened to a matrix with 12288 values for each image, so (12288,209).

With an understatement of the dataset, we can finally shape the architecture of this NN. As seen in Figure 1, where the architecture is proposed, the number of input neurons is, as discussed, 12288, that is the size which represents all values from the image. However, one input neuron will be added to account for a bias, which includes some complexity to the Network and, consequentially, improving performance. Then, the Input Neurons layer will have 12288 + 1 neurons.

The next step is to define and initialize the weights for this matrix. Knowing that we have 12289 input neurons and that the output neuron is computed as the sum of the weighted inputs, the weight vector has to have the same size as the input layer, hence we create and initialize as 0 a weight vector with 12289 elements.

Knowing that the output is computed as $Z = \sum_{i=1}^{N_{inputs}} w_i X_i$ and $A = \sigma(Z)$, so no initialization is needed. However, defining functions such as one that compute the sigmoid will be useful

for next steps. The function $\sigma(x)$ is called sigmoid activation function. Activation functions are used in Neural Networks to add non-linearity in the system, enabling it to account for more complex nuances in the input data that a linear combination wouldn't. Also, the whole architecture wouldn't be justifiable(for the usual applications of NN's) if, in the end, it was just a weighted sum to compute the output(linear operations). The sigmoid will shrink the output values to values between 0 and 1.

At last, we can understand the value of A, when the NN is presented an image, as the "certainty" it has if this image is a cat or not, measured in a value between 0 or 1; or even the probability(given the limitations of the architecture) that this picture is a cat. As an arbitrary threshold, the decision boundary is such that:

- If the output neuron has value $A > 0.5$, the image is classified as a cat.
- If $A \leq 0.5$, the image is classified as non-cat.

With the prepared architecture, we can present whatever image that follows our data constraints(64x64 pixels RGB), and it will output a value between 0 and 1, which we translate as cat or non-cat. However, this NN is not trained yet, so we shall still adjust the weights of the network so it is able to better classify images.

2.2 Network training

To train the network, the only tunable parameter we see in the proposed architecture are the weights w_i. Then, we shall apply one optimization method to properly tune these weights achieving good results in this network. For this, we still need a metric to ensure that the classification is right. For this, since we are working with a supervised model, both training and test datasets include a ground truth for each image. Having this ground truth and the output from the network, we can apply an Loss Function to measure for a given image and a given set of weights, the error of the model prediction.

Then, an obvious approach to tune the weights would be minimize the loss function based on weight changes, i.e. calculate $\frac{\delta L}{\delta W}$. For this optimization, we will use the Gradient Descent algorithm, since analytical solutions aren't usually feasible for these problems. Detailing on the calculation of this derivative and discussions around the proof of convergence of the method can be seen in Appendix A. In summary, the optimization will rely on the computation of:

$$\frac{\partial L_i}{\partial w_i} = \frac{\partial L_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_i} = (a_i - y_i)x_i$$

Where i is a given image from the training dataset.

This derivative shows the how the Loss changes with respect the weights, then, we update or weights w with the negative of this derivative(then, taking the opposite sense that would increase the loss):

$$w = w - \alpha(A - Y)X$$

Where α is the learning rate.

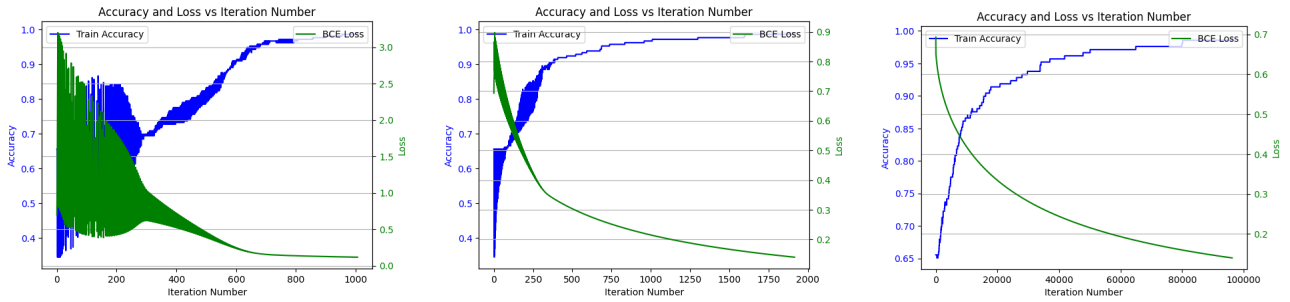
With everything stated until now(the constructed architecture, how to present an image to the network and how to train the network), we finally get to train it. For this, we use the sketch method seem in the code snippet below:

```
def train_model(weight_matrix, train_set_x_flatten, train_set_y,acc_target,
    alpha, max_iterations=10000):
     weight_matrix = np.zeros((train_set_x_flatten.shape[0],1))
     step_number = 0
     accuracy = 0
```

```
while accuracy < acc_target and step_number < max_iterations:
    Z = np.dot(weight_matrix.T, train_set_x_flatten)
    A = sigmoid(Z)

    dw = -alpha *(1 / train_set_x_flatten.shape[1])*
       np.dot(train_set_x_flatten, (A - train_set_y).T)


    # Update weights as specified
    weight_matrix = weight_matrix + dw

    # BCE loss calculation
    loss = -np.mean(train_set_y * np.log(A + 1e-9) + (1 - train_set_y)
       * np.log(1 - A + 1e-9))

    accuracy = calculate_accuracy_2(A, train_set_y)
    step_number += 1
```

This code snipped is just a representation of a way to do it. Since in the first step we want to get a touch on the sensitivity of the model, we have two stopping conditions. The loss is being calculated to further plot but is not really needed for the computation, etc. Look at it on a basis on how to implement, not a final version.

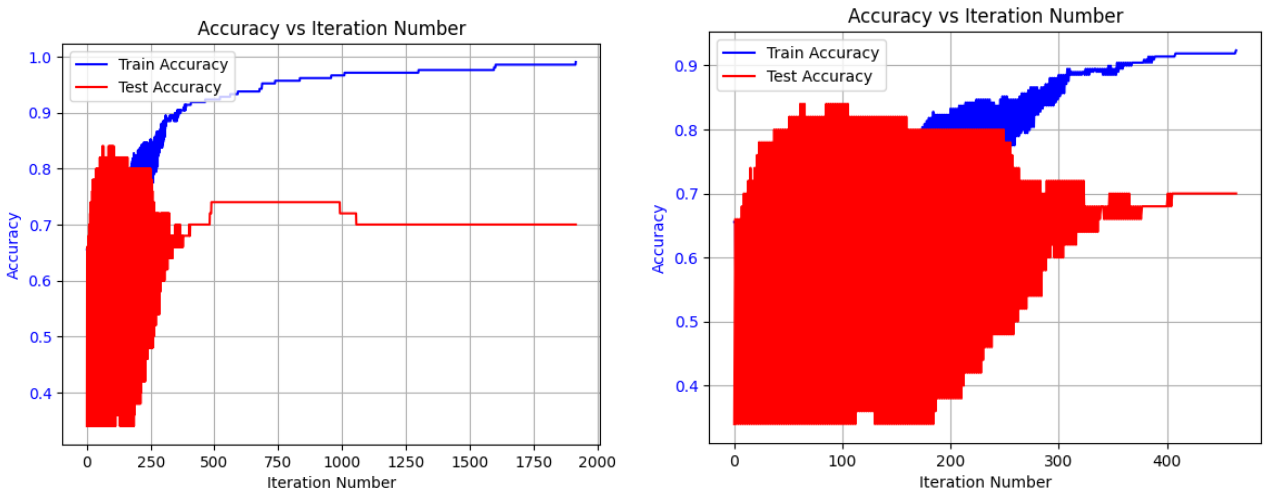
With this approach, we can observe the behavior of the accuracy with respect to the number of iterations and also observe if the loss function is really being minimized. Initially, the subject of the guided work suggested a learning rate of 0.005 and, when training, to pay particular attention on how many iterations are needed to reach "almost perfect" accuracy on the training dataset. Pay attention that, while updating the weights, we are calculating the mean update value(since we run train_model matricially with all images at once), dividing the update by the number of images in the training set(the sole difference is that, without this factor, convergence will be faster but more unstable). Below, one can see the number of iterations needed to achieve training accuracy of 0.99.



(a) Learning rate 0.01 (b) Learning rate 0.005 (c) Learning rate 0.001

Figure 3: Loss function and learning rate w.r.t. number of iterations on training

In reality, in most cases we do not want to achieve 100% of training accuracy, since this will lead to overtrainingoverfitting and further lack of generalization. Since with a learning rate of 0.005 we converge sufficiently fast and without a lot of "noise" during training, we will pick this learning rate for next analysis. In the Figure 4 we can see how the test accuracy behaves in comparison with training accuracy. In reality, this is not done to train NN's, since it doesn't make sense to use your test dataset(responsible to attest the quality of your Network to generalization) to fine tune the early stopping. However, since our dataset is small, it quickly overfits, and as an exercise to better understand the overfitting behavior, it is understandable to train vs test accuracy behaves during training.



(a) Train and test accuracy full view

(b) Train and test accuracy with early stopping

Figure 4: Early stopping effect on NN training

2.3 Optimizing

To avoid using the information gathered from test accuracy computing shown in Figures 4, some validation methods were test. The first of them is separating the training dataset into training and validation (with 0.8 to 0.2 proportion) and implementing a "patience" gauge, meaning that, if validation doesn't improve for a given number of iterations, we early stop the execution to avoid overfitting. It wasn't really useful since the validation set wasn't translating to any capability of generalization of the NN. Since the dataset is too small, the validation set is composed by only around 40 images, while the training is reduced to around 160 images, being insufficient for proper training. As a result, the validation with patience didn't reach any sufficient better results than early stopping with reasonable train accuracy(arbitrarily stopping at 90% accuracy or even 80% accuracy).

Other methods such as mini-batch(SGD) and cross-validation were tried, but they didn't present any particular improvement too. Hence, we will choose to arbitrarily stop at 94% of training accuracy aiming to reduce overfitting where training has a plateau(less chance of have a spike in test accuracy). Again, we could force a 84% test accuracy, but this result wouldn't make sense and neither would the methodology.

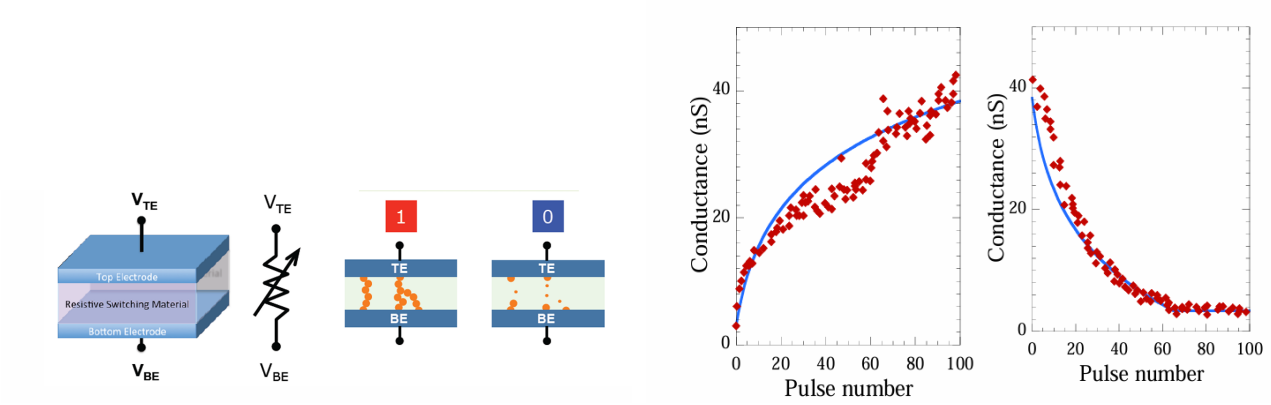
2.4 Final results

With the optimization selected in the previous subsection 2.3, we've reached a 74% accuracy on the test dataset. Some of the mislabeled images can be seen below in Figure 5. Prediction True means that the Network classifies the given image as a cat, ground truth is true when it is a cat. It is difficult to find the pattern on why these pictures were misclassified, however, by deepening the NN with hidden layers, we might improve this accuracy, since the network will be able to find more complex patterns that describes the data.

Figure 5: Failed predictions during generalization

3 Oxide-based resistive memory cat-detector

Oxide-based resistive memories(Memristors) are nonvolatile memory devices that store information by changing their resistance(or conductance). The principle of operation is based on applying a sufficiently high voltage(bigger than the device threshold) that can induce the formation or rupture of conductive filaments within the device(based on the polarity of this excitation). When used as memory, its state will either be 1 (LOW-RESISTANCE, filaments formed) or 0 (HIGH-RESISTANCE, small quantity of filaments), which is measured by the current measured when a readout voltage(smaller than the threshold) is applied to the device. These states and filaments can be seen in Figure 6a. Even if the device suffers from drift over time, it is a NON-VOLATILE memory, which means that a formed state tends to stay as put for a long time.



(a) Binary behavior of Memristor [1]

(b) Analog behavior of Memristor [2]

Figure 6: Analog and binary behavior of oxide-based resistive memory

Due to their scalability, low power consumption, and Non-volatility, oxide-based resistive memories are well-suited for in-memory computing and neuromorphic systems. Apart from the possibility to set HIGH and LOW states to the memory, it is also possible to set analog low resistance values as seen in Figure 6b, which enables them to mimic synaptic behavior in neural networks, supporting parallel data. Below, in Figure 7, one can see how a Memristor array implements very naturally the multiply and accumulate operations from NN's.

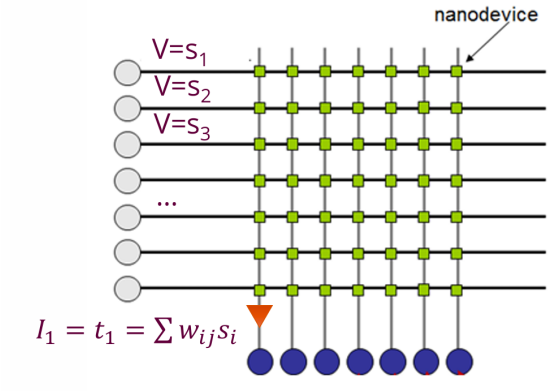


Figure 7: The blue dots(weighted sum of inputs) are inferred naturally by Kirchoff laws.

Each green dot is equivalent to one Memristor, where each weight is set as its conductance. The input neurons are the voltages applied in the net, as seen in the leftmost column. As

a counterfeit, these devices present drift, are relatively hard to program to a specific wanted resistance or conductance(high variability), and present a notable variance/noise within the set conductance value, however, NN are usually robust to variance/noises, which doesn't limit the usage of Memristors for neuromorphic systems.

However, one can quickly notice that this array isn't capable of well representing the weights from Neural Networks, since the weights contain both negative and positive values and there is no such thing as a negative conductance. To solve this, many research studies rely on a differential Memristor pair to represent one weight. In our case, we will rely on a 1T1R differential architecture to each weight, hence, we will need 12289 * 2 Memristors to physically implement this Neural Network.

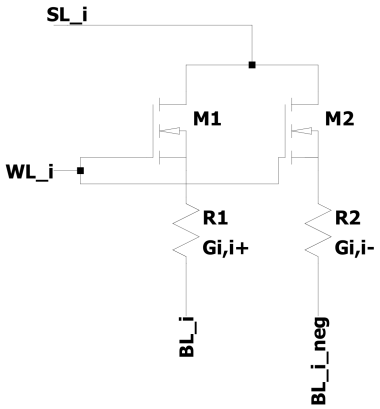


Figure 8: Differential 1T1R weights representation

Each cell above will represent one green dot seem in the image 7. With this, we can implement both positive and negative weights from our original Neural Network. Of course, it lacks the measuring and writing circuit, which will implement differential measurements of each Bit Line. Then, now we have to figure out on how to translate these weights to conductance in a way that we ensure the original behavior of the Neural Network for generalization.

3.1 Translating weights to conductances

Since each weight will be represented as the sum of the two differential conductances proposed before, we must then adjust our weights to this representation. First of all, the device used will have a limit of 110 μS of conductance. Then, the operation to translate a weight into a pair of conductances is simple:

$$G_i = G_i^+ - G_i^- = w_i \frac{G_{max}}{max|W|}$$

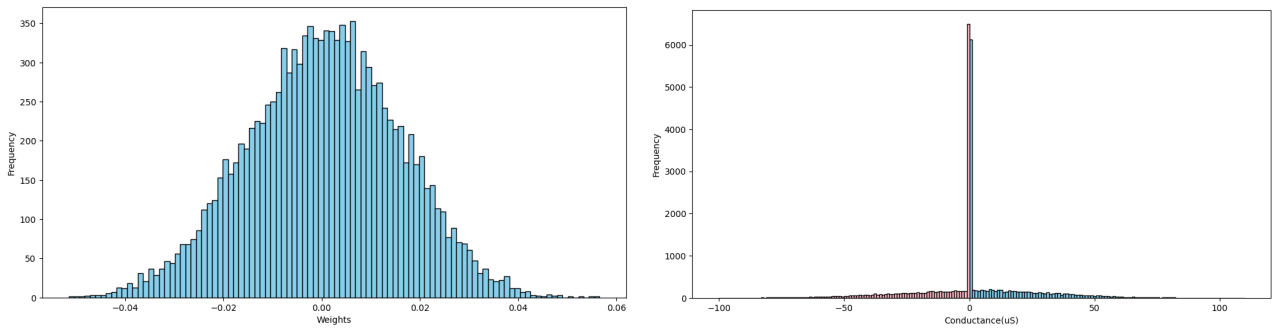
Where $G_i \in [-110\mu S, 110\mu S]$ is the conductance i with positive and "negative" differential components $G_i^+ \in [0, 110\mu S], G_i^- \in [0, 110\mu S]$ and $max|W|$ is the biggest numerical value of the weights.

With this scaling, the maximum weight will be mapped to the maximum conductance and all the rest will be scaled proportionally to the scale factor $\frac{G_{max}}{max|W|}$. No centering is needed, since the weights are already centered around 0. The code snippet below shows the Python implementation of this conductance transformation.

```
def conductance_calc(w, Gmax=110E-6):

    max_w = np.max(np.abs(w))
    scale_factor = Gmax / max_w
```

```
    Gplus  = (w > 0) * (w * scale_factor)
    Gminus = (w < 0) * (-w * scale_factor)

    if(np.max(Gplus) > Gmax):
        print("error: Conductance above physical maximum")
        print("Gplus", np.max(Gplus)*1e6)

    if(np.max(Gminus) > Gmax):
        print("error: Conductance above physical maximum")
        print("Gminus",np.max(Gminus)*1e6)

    return Gplus.T, Gminus.T
```

To attest if the conversion was successful and understand the behavior/organization of the array, we can plot an frequency map of the weights after the one-layer NN training and after the transformation shown in the snippet above. Figure 9a shows how weights were mapped after training and Figure 9b shows the mapped conductances after transformation, using the scaling method, the boundaries of the weights don't matter to map the conductances, since we are applying a scaling.



(a) NN weights after training (b) Conductances after transformation

Figure 9: Weights and conductances after training and transformation

Figure 9b shows a concentration of conductances with value 0. This happens because, since we are operating with a differential mapping for conductances, one way to represent a given positive weight, for example, is by setting a positive value for G_i^+ and setting no value for G_i^-. This is actually a good way to approach the problem, since by doing this, we have a sparse matrix of conductances, which imply that we can either don't put some devices in the array(corresponding to 0 values) or not program then during training. Since programming is costly in energy and brings more variability to the system, having an sparse array is a positive thing(even though it increases space usage).

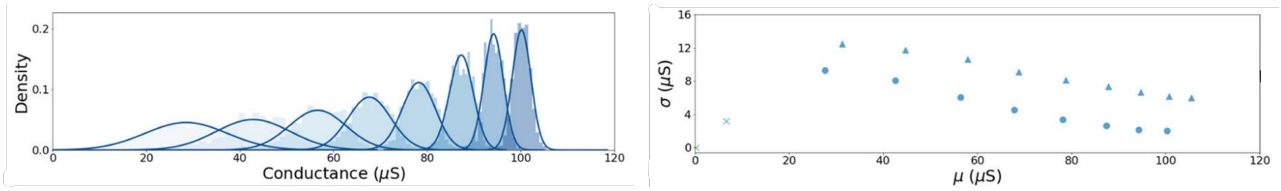
With the weights mapped to respective conductances, now the inference is simple. As seen in Figure 7, the weighted sum is naturally obtained, accounting for the differential behavior. Now, we just have to apply the transformation function to ensure our physical network can also detect complex patterns in the images. To do this, we implement the following method, which would translate to a sigmoid calculation cell in hardware:

```
def memres_inf(x, Gplus, Gminus):
    I_out  = np.dot((Gplus-Gminus),x)

    A = 1.0/(1.0 + np.exp(-I_out))
    return A
```

This resulted in the same accuracy of 0.74 obtained before.

3.2 Variability issue

As introduced before, one of the problems related to the usage of Memristors is the variability while programming the device for a given conductance. Experimentally, oxide-based resistive memories present the behavior seen in Figure 16. However, to understand the resilience of our neuromorphic system we will first study how the system reacts to a naive variability approach, that is, applying a variance dG_i to all set conductances G_i, that is independent of the value of G, even though we see in the figure that it does depend on the set value.



(a) Probability densities of 2,048 filamentary memristors programmed with eight different programming current values

(b) Domain of experimental Gaussian distributions. Triangles represent one shot programming, dots represent iterative programming.

Figure 10: Experimental variability of oxide-based when set to different conductances. [3]

To add this noise to the system, the following method was created:

```
def noise_adding(sigma,Gplus,Gminus):
    Gplus_cpy = np.copy(Gplus)
    Gminus_cpy = np.copy(Gminus)

    Gplus_cpy, Gminus_cpy = (Gplus > 0) * (Gplus +
       np.random.normal(0,sigma,Gplus.shape)), (Gminus > 0) * (Gminus +
       np.random.normal(0,sigma,Gplus.shape))

    return  np.clip(Gplus_cpy,  0, 110E-6),np.clip(Gminus_cpy,  0, 110E-6)
```

It first copies both vectors to avoid editing the non-noisy defined conductances, then add a noise to every non-null conductance value based on a normal distribution with centered at 0 with an arbitrary variance, at last, it clips the noise to limit the values to the maximum possible conductance of $110\mu S$. We do not add any noise to the null conductances since, as stated before, we won't program these devices, as an advantage of the sparse vector. Then, to test the noise limit with this naive approach, a noise sweep was done, starting from $1\mu S$ and going to $40\mu S$.

```
noise_vec = np.linspace(1E-6, 40E-6, int(1E+3))  # Avoid incrementing sigma
   manually (gpt orientation to accelerate method)
acc_vec = np.zeros(len(noise_vec))                #GPT suggestion (leaves a
   memory space created, accelerating)

for i in range(len(noise_vec)):
   Gplus_noisy, Gminus_noisy  =
      noise_adding(noise_vec[i],Gplus_noise_free,Gminus_noise_free)
   A_out = memres_inf(test_set_x_flatten, Gplus_noisy, Gminus_noisy)
   acc_vec[i] = (calculate_accuracy_2(A_out,test_set_y))
```

The plot of this accuracy vector `acc_vec`, seen in the Figure 11 shows at which point the noise overcomes the resilience of the system.

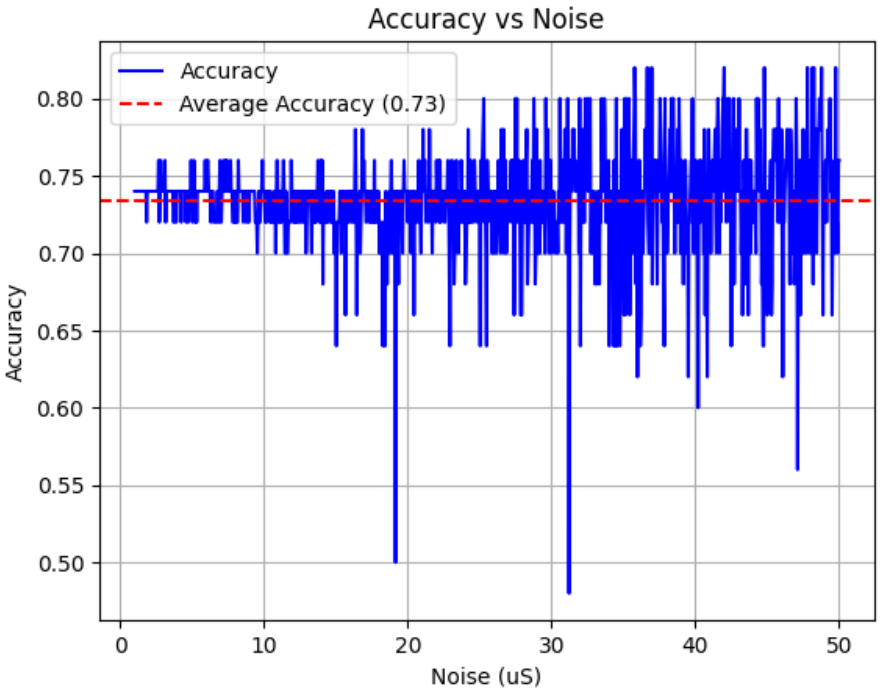
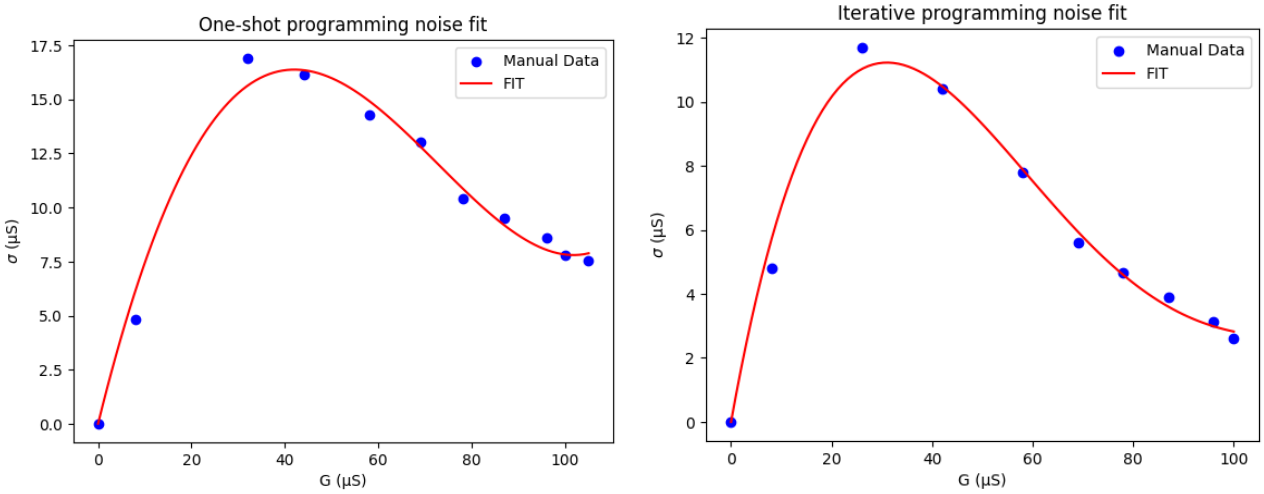


Figure 11: Naive approach to extract system resilience to noise. The noise is applied as an independent random variable w.r.t. the conductance G.

However, as seen in Figure 16, in reality the variability isn't independent and, as suggested by the results extracted from the noise sweep, the system presents only about 2% of variability in accuracy when disturbed with normal distributions with deviation of $10\mu S$. In general, there are some outliers such as σ around 20 μS, but the system seems to hold wekk ebiugh until $30\mu S$. This hints that our system is resilient when simulated with a hardware-aware(accounting for experimental noise) implementation.

3.3 Hardware-aware noise limit

The first step was to create fit the experimental data presented in such way that we can adjust the variability of the conductance w.r.t. its own conductance. The first approach was to create a polynomial fit, one for the one-shot programming extracted data and another for iterative programming data. As a conservative step, trying to account for human measurement mistake(from the graph observed in [3]), the Y axis was multiplied by an error factor of 1.1. The fit with this "safety factor" can be seen in Figure 12.



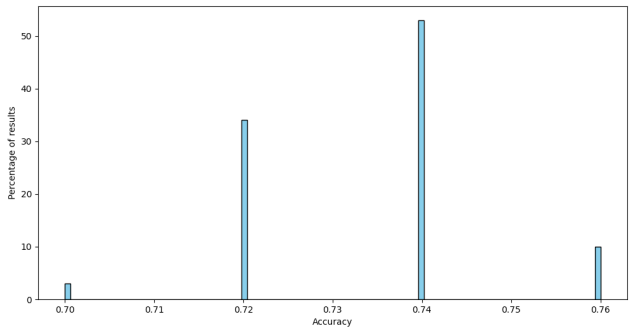
(a) One-shot programming fit with 1.1 factor. (b) Iterative programming fit with 1.1 factor.

Figure 12: Experimental data fit for hardware-aware simulation

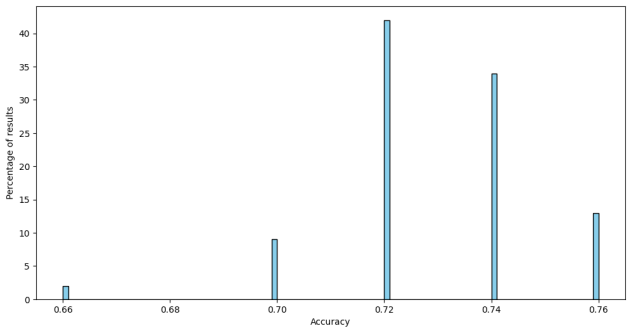
Now, with the fit of both data in hands, we just have to add this noise to the calculated conductances. For this, the python method seem below was used. It relies on adding a normal distribution centered at 0 with variance extracted from the fits. By doing this, each G_i will have the corresponding real hardware noise related to it, all the rest is similar to the previously explained method of noise adding.

```
def noise_adding_fit(Gplus, Gminus):
Gplus_copy = np.copy(Gplus)
Gminus_copy = np.copy(Gminus)

for i in range(Gplus_copy.shape[1]):
    Gplus_copy[0][i] = Gplus_copy[0][i] + np.random.normal(0,
       poly_func(Gplus_copy[0][i]))
    Gminus_copy[0][i] = Gminus_copy[0][i] + np.random.normal(0,
       poly_func(Gminus_copy[0][i]))

return Gplus_copy, Gminus_copy
```

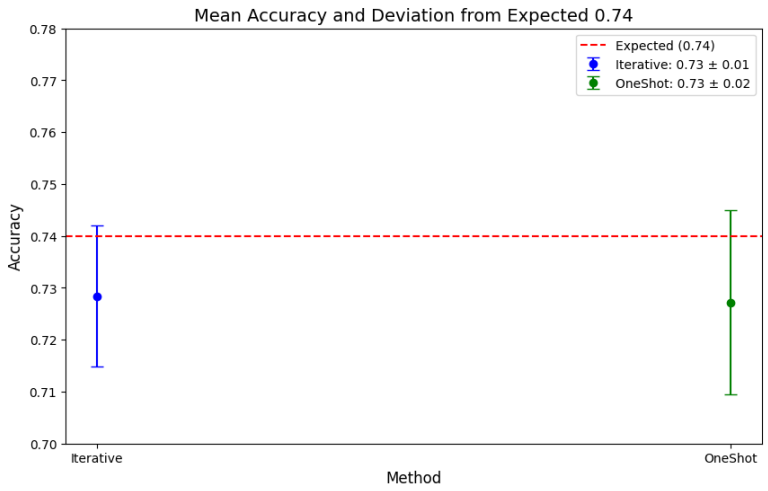
After adding noise to both vectors `Gplus` and `Gminus`, we can extract the inference from the hardware-aware model and see if the accuracy remains the same. For both cases(iterative and one-shot), there was no variation in the accuracy after adding noises, accordingly with what was expected before with the naive approach. However, one iteration isn't enough to attest the resilience of the system, them, we iterate 100 times of noise adding to the system and observe how it deviates from the original accuracy previously found.



(a) Iterative programming accuracy under 100 executions.



(b) One-shot programming accuracy under 100 executions.



(c) One-shot and Iterative programming error under 100 executions.

Figure 13: Mean accuracy and deviation for both one-shot and iterative programming under 100 executions.

There is no significative difference in the mean accuracy(difference in the third decimal point) degradation when using one-shot or iterative programming techniques. This happens because the system is particularly resilient to noise, so the experimental variability between the two programming methods doesn't deeply affect the system. However, it is important to notice that, for implementations that are more noise-sensitive there is a tradeoff to be considered when choosing which programming method to use, since iterative method presented closer mean to the expected 0.74 accuracy obtained in the pure programming NN. At worse, we observe a 1% error with iterative method while one-shot method presents 2% error.

If the system need constant weight updates on flight, for example, in systems that reprogram themselves as a re alimentation process, one-shot might be adequate since it is faster to settle. However, if the system doesn't need any reprogrammability on run, such as our cat-detector, iterative programming is probably a best option to choose, since research such as [4] suggests that it not only reduces variability, but also reduces drift effects over time.

4 Deep Neural Network

As seem before, the one-layer cat detector fails to completely account for complexity in the training data, implying that some improvement in accuracy can be reached when adding one hidden-layer to the system, supporting ourselves in the Universal Approximation Theorem stated before. Figure 14 shows the architecture choose to implement this DNN.

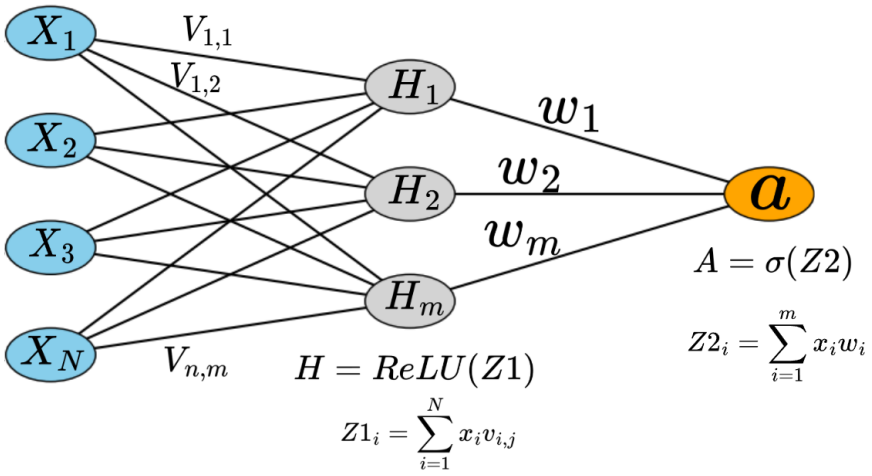


Figure 14: DNN architecture

Notice that ReLu was used as activation function to avoid the problem of the vanishing gradient. It is also stated that ReLU is easily implementable in hardware [5], and has also easily implementable derivative, which is good for possible in-situ training.

4.1 Network training

The approach to train the network is similar to the one used for the single-layer structure. We will use Gradient Descent to optimize the Loss function w.r.t. the weights(both of the hidden layer and input layer). We will call the first line of weights $v_{i,h}$, that are responsible to connect the input layer to the hidden layer H, from H to the output layer, the weights will be called W. Then what we want is to minimize:

$$\frac{\partial L_i}{\partial w_i} \text{ and } \frac{\partial L_i}{\partial v_{i,j}}$$

The development of the terms is found in A, here we will just present the results of this derivation. For $\frac{\partial L_i}{\partial w_i}$ the only change is that now, the "input layer" for the last linear combination is the hidden layer H, then:

$$\frac{\partial L_i}{\partial w_i} = \frac{\partial L_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial h_i} = (a_i - y_i)h_i$$

Now for $\frac{\partial L_i}{\partial v_{i,j}}$:

$$\frac{\partial L_i}{\partial v_{i,j}} = \frac{\partial L_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial h_j} \cdot \frac{\partial h_j}{\partial s_j} \cdot \frac{\partial s_j}{\partial v_{i,j}}$$

Combining these terms, the gradient for the hidden layer weights becomes:

$$\frac{\partial L_i}{\partial v_{i,j}} = (a_i - y_i) \cdot w_j \cdot 1(s_j > 0) \cdot x_i$$

or

$$\frac{\partial L}{\partial V} = (A - Y) \cdot W \cdot 1(S > 0) \cdot X$$

Where $S = V^T X$, $Z1 = \sum_i^N s_i$.

To train as stated, the following methodology was followed(no method was created, since some fine tuning and hyperparametrization was done beforehand):

```
for i in range(max_iteration):
# Forward pass
Z1 = V @ train_set_x_flatten  # Linear transformation for hidden layer
H = np.maximum(0, Z1)  # ReLU activation
Z2 = W @ H       # Linear transformation for output layer
A2 = sigmoid(Z2)   # Sigmoid activation

train_acc = calculate_accuracy_2(A2,train_set_y) # Measuring progression
acc_vec_train.append(train_acc)                  # Measuring progression

# Backward pass
dZ2 = A2 - train_set_y  # Gradient of loss w.r.t. output layer linear
    transformation
dW = (1 / train_set_x_flatten.shape[1]) * dZ2 @ H.T  # Gradient of loss
    w.r.t. output layer weights

dA1 = W.T @ dZ2  # Gradient of loss w.r.t. hidden layer activations
dZ1 = dA1 * (Z1 > 0)  # Gradient of loss w.r.t. hidden layer linear
    transformation (ReLU derivative)
dV = (1 / train_set_x_flatten.shape[1]) * dZ1 @ train_set_x_flatten.T
    # Gradient of loss w.r.t. hidden layer weights

# Update weights
V -= alpha * dV
W -= alpha * dW

Z1test = V @ test_set_x_flatten_test
Htest = np.maximum(0, Z1test)              # ReLU activation
Z2test = W @ Htest
A_test = sigmoid(Z2test)

test_acc = calculate_accuracy_2(A_test,test_set_y)  # Measuring
    progression
acc_vec.append(test_acc)                            # Measuring
    progression

#if(train_acc > 0.83 and i>300):                     # From
    hyperparametrization
#    print(f"Test Accuracy: {test_acc:.4f}","iteration: ",i)
#    break

# Compute BCE loss. GPT suggestion: (add epsilon to avoid log(0))
epsilon = 1e-9
loss = -np.mean(train_set_y * np.log(A2 + epsilon) + (1 - train_set_y)
    * np.log(1 - A2 + epsilon))
loss_vec.append(loss)
```

Since the hyperparametrization of the DNN is computationally heavy, it was done outside and the intermediate results don't interest to the scope of this work. However, it is interesting to notice that, if we ignore the noise of the circuit that the DNN will be soon translated to, we could decide that the hidden layer would contain only 8 neurons, for example. However, after the translation, it is seen that if the hidden layer is too small, the intrinsic noise of the devices will imply in a expressive degradation of the accuracy obtained in the non hardware-aware DNN. In the gridsearch accounting for noise, it was found that, to keep a similar resilience of that found in the single layer NN, we should use at least 256 neurons in the hidden layer (the grid search was done selecting powers of 2 as the size of the hidden layer).

4.2 Optimizing

By training the DNN with the parameters specified above (hidden_size =256, alpha = 0.01, max_iteration = 749) we avoid overfitting while increasing train and test accuracy as well as keeping the system more noise immune. This comes with great computational cost(some steps of the computation required strong parallelization) and space cost(since we are idealizing a circuit equivalent). The latter one shows the impracticality(or no need, given that it is a simple neural network) of this approach, taking as an example HERMES from IBM [6], the net is able to store $64 \times 256 \times 256$ or 4,194,304 parameters in a reasonable size. The neural network with 12288 input neurons and 256 hidden neurons would need, in the weight set V, $12288 \times 256 = 3.145.728$ parameters. As a rule of thumb, we will shrink the size of our network to be able in a reasonable PIM(In-Memory computing) unit, such as HERMES, then, our hidden_layer size will be limited to 32 neurons at maximum, aiming for a 256x256 1T1R net with multiple cores.

By following the tradeoff, the final implementation is done with parameters (hidden_size =32, alpha = 0.01, max_iteration = 720), which gives the result seem in the Figure 15 and 0.8 accuracy, higher than the simple NN.

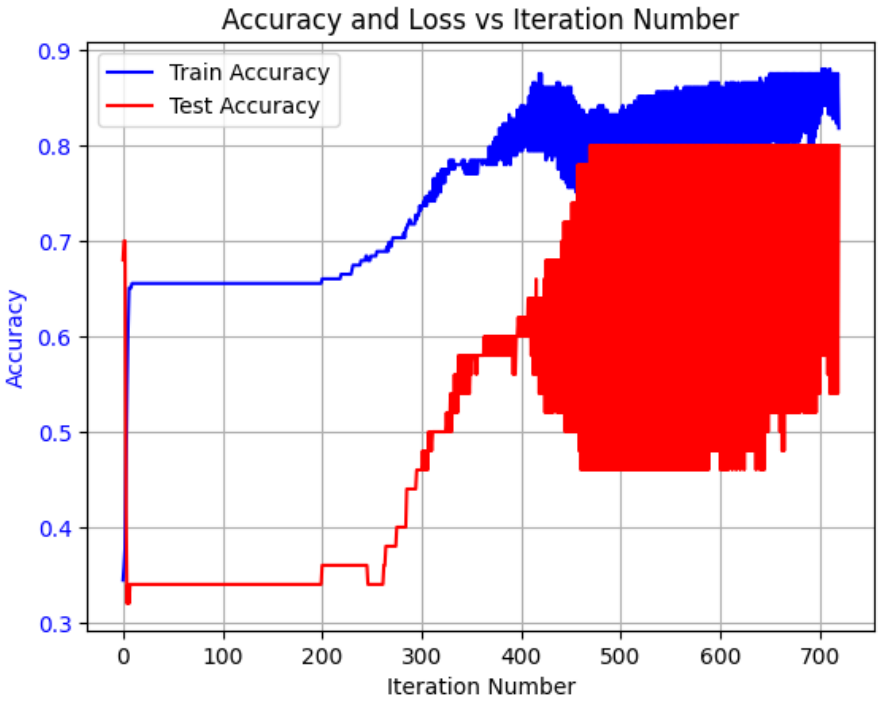
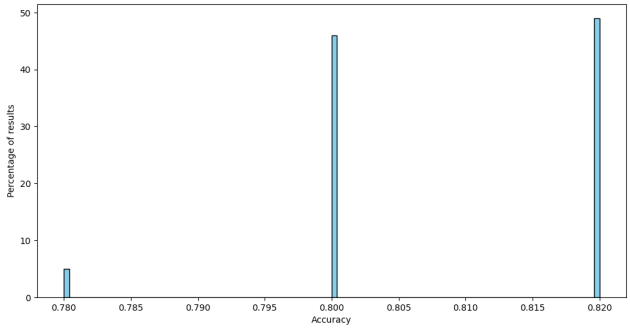
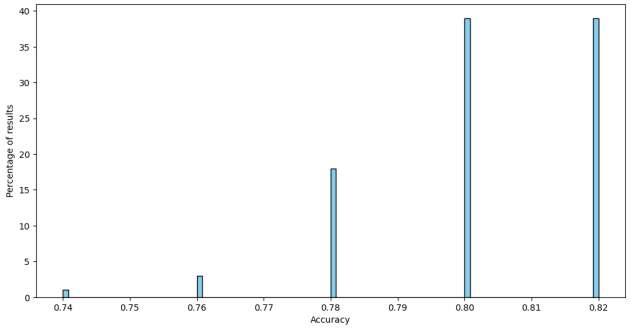


Figure 15: DNN results with final 0.8 accuracy on the test set.

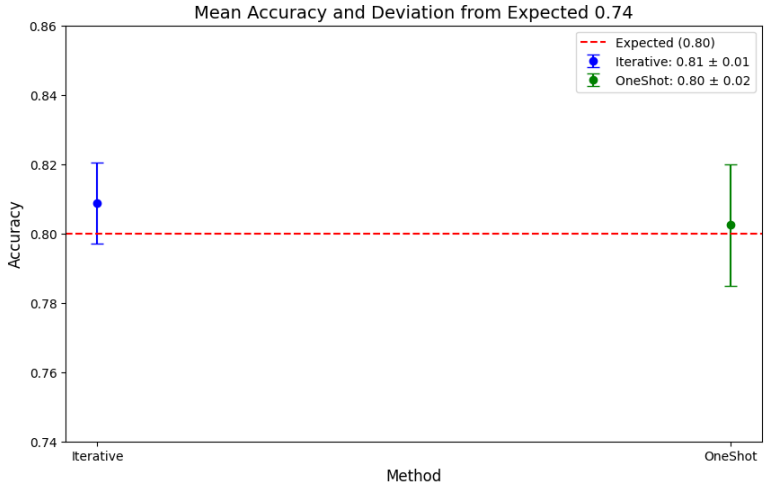
4.3 Final Results



(a) Iterative programming accuracy under 100 executions.



(b) One-shot programming accuracy under 100 executions.



(c) One-shot and Iterative programming error under 100 executions.

Figure 16: Mean accuracy and deviation for both one-shot and iterative programming under 100 executions.

5 Conclusion

This project explored the design and implementation of a cat detector using a neural network, first through a purely computational approach and subsequently via a hardware-aware implementation leveraging oxide-based resistive memories (memristors). The initial one-layer perceptron, trained on a dataset of cat and non-cat images, achieved a test accuracy of 74% after careful optimization to mitigate overfitting, demonstrating the feasibility of simple binary classification with supervised learning. However, its limitations in capturing complex patterns limits its accuracy and shows the need of a Deep Neural Network. The transition to a memristor-based implementation highlighted the potential of in-memory computing for energy-efficient neuromorphic systems. By mapping neural network weights to differential conductance pairs in a 1T1R array, the system retained its 74% accuracy while naturally performing multiply-and-accumulate operations via Kirchhoff's laws. Since we have only 12289 weights, this whole NN could be implemented in a 128x128 1T1R differential array, with 32768 devices RRAM devices in total. Both noise analyses(naive and hardware-aware) confirmed the system's robustness, with no significant accuracy degradation even under realistic conductance variability, showing the robustness of Neural Networks to variability and the robustness of the memristor based In-Memory Computing cell. In conclusion, this work successfully demonstrated the viability of resistive memory for implementing a cat detector, bridging theoretical neural network design with physical constraints. The memristor-based approach offers promising energy efficiency and scalability for neuromorphic applications, though challenges remain in optimizing array size and programming precision. Future work should focus on increasing the depth of the Neural Network, with small number of neurons for each layer. Since this goes beyond the scope of the project, it is a matter of future works to have small enough, resilient enough neural networks. The usage of new memories and In-Memory Processing is a closer approach on how animal brains work, presenting enormous complexity and energy efficiency. This work shows the capabilities of these systems under real conditions, promoting, in an educative approach, the advancement of such necessary evolution towards Energy Efficient NNs.

References

[1] Deleruyelle, Damien. *Nouveaux composants pour les mémoires non volatiles: du stockage de charges vers la commutation de résistance.* Presentation at [Event Name], January 2012.

[2] Jo SH, Chang T, Ebong I, Bhadviya BB, Mazumder P, Lu W. *Nanoscale memristor device as synapse in neuromorphic systems.* Nano Letters. 2010 Apr 14;10(4):1297-301. doi: 10.1021/nl904092h.

[3] Bonnet D, Hirtzlin T, Majumdar A, Dalgaty T, Esmanhotto E, Meli V, Castellani N, Martin S, Nodin JF, Bourgeois G, Portal JM, Querlioz D, Vianello E. *Bringing uncertainty quantification to the extreme-edge with memristor-based Bayesian neural networks.* Nature Communications. 2023 Nov 20;14(1):7530. doi: 10.1038/s41467-023-43317-9.

[4] Esmanhotto E, Hirtzlin T, Bonnet D, Castellani N, Portal J, Querlioz D, Vianello E. *Experimental demonstration of multilevel resistive random access memory programming for up to two months stable neural networks inference accuracy.* Advanced Intelligent Systems. 2022;4(12):2270054. doi: 10.1002/aisy.202270054.

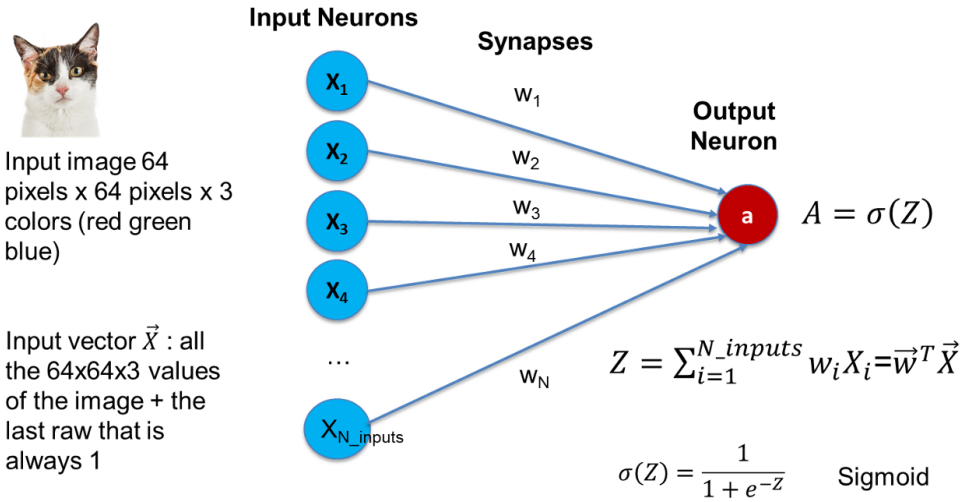
[5] Chen Z, Tong L, Chen Z. *Research and design of activation function hardware implementation methods.* Journal of Physics: Conference Series. 2020 Nov;1684:012111. doi: 10.1088/1742-6596/1684/1/012111.

[6] Le Gallo M, Khaddam-Aljameh R, Stanisavljevic M, Vasilopoulos A, Kersting B, Dazzi M, Karunaratne G, Brändli M, Singh A, Müller SM, Büchel J, Timoneda X, Joshi V, Rasch MJ, Egger U, Garofalo A, Petropoulos A, Antonakopoulos T, Brew K, Choi S, Ok I, Philip T, Chan V, Silvestre C, Ahsan I, Saulnier N, Narayanan V, Francese PA, Eleftheriou E, Sebastian A. *A 64-core mixed-signal in-memory compute chip based on phase-change memory for deep neural network inference.* Nature Electronics. 2023 Aug;6(9):680–693. doi: 10.1038/s41928-023-01010-1.

A Two Layer Neural Network for Cat Classification

*Partially **transcribed** by DeepSeek from my personal notes

A.1 One-layer Gradient Descent



Definitions:

- **Input Size N:** Information carried in one image: $64\ pixels \times 64\ pixels \times 3(RGB)$ (size: 12288)
- **Input:** Flattened vector $x \in R^{n+1}$ where 1 is added to account for biasing (size: $[12889 \times 1]$)
- **Weights:** $w \in R^n$ (size: $[12889 \times 1]$)
- **Output:** $a = \sigma(w^\top x)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$ (size: $[1 \times 1]$)
- **Loss Function:** $L = -y \ln(a) - (1-y) \ln(1-a)$
- **Total loss:** $J = \frac{1}{m} \sum_{i=1}^{m} L^{(i)}$

Step 1: Gradient of L with Respect to w

Using the chain rule:

$$\frac{\partial L}{\partial w} = \underbrace{\frac{\partial L}{\partial a}}_{\text{Step 1}} \cdot \underbrace{\frac{\partial a}{\partial z}}_{\text{Step 2}} \cdot \underbrace{\frac{\partial z}{\partial w}}_{\text{Step 3}}$$

$$\text{Step 1:} \quad \frac{\partial L}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\text{Step 2:} \quad \frac{\partial a}{\partial z} = a(1-a) \quad (\sigma'(z) = \sigma(z)(1-\sigma(z)))$$

$$\text{Step 3:} \quad \frac{\partial z}{\partial w} = x$$

Combining these:

$$\frac{\partial L}{\partial w} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right) \cdot a(1-a) \cdot x = (a-y)x$$

Step 2: Gradient of J with Respect to w

Average over all m training examples:

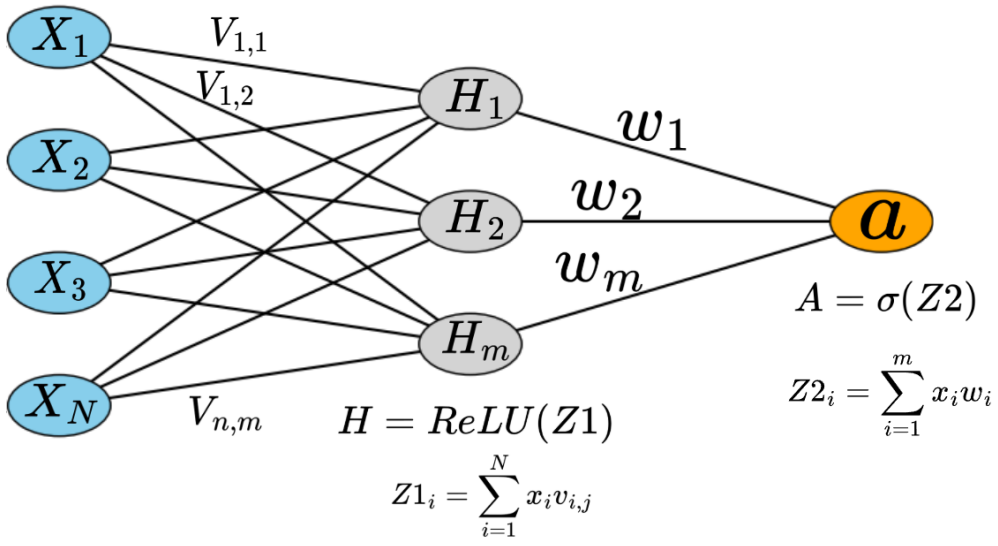
$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial L^{(i)}}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)})x^{(i)}$$

Step 3: Weight Update Rule

Update weights using learning rate η:

$$w \leftarrow w - \eta\frac{\partial J}{\partial w}$$

A.2 Two-layer Gradient Descent



Definitions:

- **Input Size N:** Information carried in one image: $64 \times 64 \times 3$ (RGB) (size: 12889)
- **Input:** Flattened vector $x \in R^N$ with bias term (size: $[12889 \times 1]$)
- **Hidden Weights:** Matrix $V \in R^{N \times M}$ connecting input to hidden layer (size: $[12889 \times M]$)
- **Output Weights:** Vector $w \in R^M$ connecting hidden to output (size: $[M \times 1]$)
- **Hidden Layer:** $h = \text{ReLU}(V^\top X)$ where $h \in R^H$ (size: $[H \times 1]$)
- **Output:** $a = \sigma(w^\top h)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$ (size: $[1 \times 1]$)
- **Loss Function:** $L = -y\ln(a) - (1-y)\ln(1-a)$
- **Total Loss:** $J = \frac{1}{m}\sum_{i=1}^{m} L^{(i)}$

$$\frac{\partial L_i}{\partial v_{i,j}} = \frac{\partial L_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial h_j} \cdot \frac{\partial h_j}{\partial s_j} \cdot \frac{\partial s_j}{\partial v_{i,j}}$$

1. **Output Term**:
$$\frac{\partial L_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial z_i} = (a_i - y_i)$$

2. **Hidden Layer Term**:
$$\frac{\partial z_i}{\partial h_j} = w_j, \quad \frac{\partial h_j}{\partial s_j} = 1(s_j > 0)$$

3. **Input Term**:
$$\frac{\partial s_j}{\partial v_{i,j}} = x_i$$

Combining these terms, the gradient for the hidden layer weights becomes:

$$\frac{\partial L_i}{\partial v_{i,j}} = (a_i - y_i) \cdot w_j \cdot 1(s_j > 0) \cdot x_i$$

B Python repository

Link to Python implementations: https://github.com/guduart02sa/ResistiveMemoryImageRecognition