SNN for detecting static object

John Harding
Volgenau School of Engineering ECE
George Mason University
Fairfax, VA USA
jhardi8@gmu.edu

Hao Wan
Volgenau School of Engineering ECE
George Mason University
Fairfax, VA USA
hwan5@gmu.edu

Juan Saavedra
Volgenau School of Engineering ECE
George Mason University
Fairfax, VA USA
jsaaved3@gmu.edu

Oliver Ward
Volgenau School of Engineering ECE
George Mason University
Fairfax, VA USA
oward3@gmu.edu

***Abstract*—Spiking Neural Networks (SNNs) [1] offer a biologically inspired approach to object detection, yet their application to static images remains underexplored compared to dynamic inputs. This project investigates the use of SNNs for static object detection, leveraging the SpikingJelly framework[2] to build and evaluate models on augmented datasets, including MNIST, Fashion-MNIST, and Animal Faces. To address challenges such as angle variation, noise, and adversarial attacks, we integrate data augmentation, convolutional self-attention auto-encoders (CSAAEs) [3], and hardware optimizations. Our results demonstrate that SNNs achieve 92.75% accuracy on MNIST and 87.21% on Fashion-MNIST with augmentation, while CSAAEs enhance robustness against adversarial perturbations (e.g., 80.27% accuracy on Animal Faces under FGSM attacks). Additionally, a custom C-based SNN framework reduces latency by replacing floating-point operations with fixed-point arithmetic and bit-shift optimizations, achieving significant performance gains. This work highlights the potential of SNNs for efficient, noise-resilient static object detection in real-world applications.**

I. Introduction

There has been rapid growth of computer vision systems and applications and this shows no signs of slowing down. Healthcare, robotics, security, automated inspection and facial recognition are all examples where being able to detect features from an image is very useful.

With SNNs, most existing solutions focus heavily on the dynamic video based inputs, while static image detection is often overlooked. As SNNs do naturally fit with time-based problems, due to the way spikes are time-based, this may be a good reason for this tendency. For this project, the aim is to explore building a robust and efficient static image detection system using SNNs that could be used in real world situations, variability and constraints.

One of the challenges with detecting objects from a static image is angle variation. The same object can appear very differently depending on the orientation it's being viewed from. This issue also exists for the scale of objects in an image. This distance an object is from the viewpoint changes its scale which could affect how well it is detected. Varying lighting conditions also present similar challenges. With moving images, there's opportunity to get more detail on an object from consecutive frames, which is not available with static images.

Images often contain noise from different sources. The ability to be noise resilient is very essential in real-world applications. We also want to be mindful of latency in detecting objects. A model that could be used real-time would be more useful.

Our starting place was to test an SNN model using the original MNIST dataset and the fashion MNIST dataset to see how it performs. Then enhance the datasets and compare how the SNN then performs. To enhance the datasets we augmented the data to provide more examples for the model to work with.

The data was augmented by taking the original images, rotating, flipping or zooming in or out on them, and adding them back into the dataset. This was to try and simulate some of the real-world angle and scale variability.

An SNN model was built using the SpikingJelly framework. This is a deep learning framework for spiking neural networks based on PyTorch. The original model was first trained on the original MNIST and fashion MNIST datasets and evaluated. The SNN was then trained and tested on the augmented data and evaluated. Our initial testing showed accuracy before the data augmentation of 47% on the MNIST dataset and about 36% on the fashion MNIST dataset. After augmenting the data the accuracy is increased to about 93% on the MNIST dataset and about 87% on the MNIST dataset.

In our planning for testing static object detection across a number of different datasets, our quorum continued to ask the question: "Is using Python, SNNTorch [4], and other associated frameworks really the most computationally efficient way to simulate a spiking neural network?" For the sake of studying neural networks and performing high-level prototyping, these frameworks are immensely helpful— but for detecting objects with as little computational power as possible, more optimized frameworks were needed. As a result, we chose to seek out aggressive optimization tactics via the development of our own SNN framework in the language C.

II. Background

A. *Why Develop an SNN Framework?*

Our initiative to develop our own SNN Framework was driven by three foundational beliefs and findings:

1) More Performance is Better
    a) Despite the scientific nature of this paper, one of our foundational reasons for the development of this framework was the sheer desire to see if we can simulate a SNN in a magnitude less time than it would take the "industry-standard" frameworks that are PyTorch and SNNTorch [4]. While this provides a number of benefits as highlighted in the next two examples, the driving factor for performance was the excitement of seeing if we could truly bring something new and impactful to the world of SNN development.

2) Enable Edge-Deployed SNNs Without the use of IoT
    a) While we would expect that a large amount of SNNs will be run on high-performance equipment, we also believe that edge devices (small devices with built-in computing) will benefit the most from the optimized use of SNNs. Devices that traditionally don't have a significant amount of computing power (Ring Doorbells, security cameras, digital cameras) and perform high amounts of data ingest are those that are best suited to do real-time analytics, but often lack the power to do so. If a heavily optimized SNN could be implemented on these devices for real-time analytics, this would introduce a significant new device market. Furthermore, deploying an SNN at the edge would mean that these devices would not have to rely on the Internet-of-Things and tremendous amounts of bandwidth to have this data processed elsewhere.

3) Neuromorphic Unavailability
    a) As of April 2025, neuromorphic hardware [5] is largely considered to be unavailable to the consumer market, and only available in restricted settings for industry partners. Due to the high processing requirements of SNN simulation, and not knowing when this hardware will surface for consumer use, this offers us another reason to develop a hyper-optimized SNN framework for deployment on traditional, openly available Von-Neumann architecture-based equipment.

B. *Adversarial Attacks*

Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD) are two of the most widely studied attack methods. FGSM generates adversarial samples by adding perturbations aligned with the gradient of the loss. PGD builds upon FGSM with iterative steps and projection into a valid perturbation space.

Fast Gradient Sign Method (FGSM): FGSM is a simple and fast method to generate adversarial examples—inputs slightly modified to fool a neural network. It works by adding a small perturbation in the direction of the gradient of the loss. FGSM attacks can be described by the following equation [10]:

$$x_{\mathrm{adv}} = x + \epsilon \cdot \mathrm{sign}(\nabla_x J(\theta, x, y))$$

Where:

x is the original input.

$x_{\mathrm{adv}}$is the adversarial input.

ε is the perturbation size.

$\nabla_x J(\theta, x, y)$ is the gradient of the loss function J.

sign( ) is a sign function.

Projected Gradient Descent (PGD): PGD is a stronger attack that applies multiple FGSM steps, each time projecting the result back to an allowed region (usually an epsilon-ball around the original input). It's like an iterative FGSM, with clipping to keep perturbation small. PGD attacks can be described by the following equation [11]:

$$x_{\mathrm{adv}}^{t+1} = \Pi_{\mathcal{B}_\epsilon(x)}\left(x_{\mathrm{adv}}^{t} + \alpha \cdot \mathrm{sign}(\nabla_x J(\theta, x_{\mathrm{adv}}^{t}, y))\right)$$

Where:

x_{adv}^{t} is the adversarial input at step t.

$x_{\mathrm{adv}}^{t+1}$is the adversarial example at step t + 1.

ε is the perturbation size.

$\nabla_x J(\theta, x_{\mathrm{adv}}^{t}, y)$ is the gradient of the loss function J.

sign( ) is a sign function.

$\Pi_{\mathcal{B}_\epsilon(x)}()$ is the projection onto the epsilon-ball around x.

C. *Convolutional Auto-Encoders (CAEs)*

Convolutional Auto-Encoders (CAEs) [12] are a type of autoencoder that uses convolutional layers instead of fully connected layers, making them particularly effective for image data. Like traditional autoencoders, CAEs consist of two main parts:

Encoder: Compresses the input image into a smaller representation (latent space) using convolutional and pooling layers.

Decoder: Reconstructs the original image from the compressed latent representation using transposed convolutions (a.k.a. deconvolutions or upsampling layers).

CAEs can capture spatial hierarchies in images and it's better than fully connected autoencoders. Also, it can help us to compress high-dimensional image data into a lower-dimensional latent space.

D. *Self-Attention Mechanisms*

Self-attention [13] is a mechanism that allows a model to focus on different parts of the same input when processing each element. Originally popularized in the Transformer architecture, it's now widely used in various tasks including computer vision.

To illustrate with a simple example: imagine our input is a sequence of words: ["The", "cat", "sat", "on", "the", "mat"].

When processing the word "cat", self-attention asks a critical question: "Which other words in this sentence are most important for understanding 'cat'?" The model might determine that "sat" and "mat" provide more contextual relevance, and will therefore weigh these connections more heavily when creating the representation for "cat".

In our image defense context, this translates to identifying relationships between distant pixels. For instance, when reconstructing an adversarially perturbed digit or fashion item, self-attention helps our model maintain coherence across the entire image, not just locally. This global awareness is crucial for restoring the original patterns disrupted by adversarial noise.

E. *Market Survey*

In our pursuit of optimizing static object detection (and more broadly, SNNs), we first surveyed the online landscape to see if other researchers had pursued similar efforts. To our surprise, many had— with cuSNN by Federico Paredes-Valles offering a generally optimized C++ framework for the creation of large-scale SNNs and Spyker by Shahriar Rezghi offering a similar C framework with C++ and Python interfaces [6] [7]. While both of these frameworks were promising, research showed that there was no thorough documentation of what optimizations were implemented, nor was there any immediate evidence of optimizations in the code. Our team determined that authors calling these frameworks "optimized" may have done so on the notion that C/C++ is considered more efficient than Python alone, and not performed many actual low-level optimizations. As a result, we pursued the creation of a (Minimum Viable Product) optimized C framework for the simulation of Spiking Neural Networks.

III. RELATED WORKS

Prior research in SNNs has largely prioritized dynamic inputs, with limited focus on static image detection. Frameworks like SNNTorch [4] and SpikingJelly [2] provide high-level tools for prototyping but incur computational overhead. Recent work by Paredes-Valles (cuSNN) and Rezghi (Spyker) [6] [7] explores optimized C++ implementations, though their documentation lacks details on low-level optimizations. Our work bridges this gap by introducing a lightweight C framework with fixed-point arithmetic and event-driven computation.

In adversarial defense, auto-encoders [12] have proven effective for CNNs but are less studied in SNNs. We adapt CSAAEs with self-attention mechanisms to preserve global context during reconstruction, outperforming traditional methods under PGD attacks. Hardware optimizations, such as replacing exponential decay with bit-shifts, draw inspiration from embedded systems literature, though our application to SNNs is novel.

IV. METHODOLOGY

This study want to build a spiking neural networks (SNNs) to successfully detect the static objects. We use images to replace the static object and will show the performance on several datasets. However, there are some challenges we need to face:

1: Angle Variation: The same object can appear drastically different when viewed from different angles. The system must be invariant to such transformations to ensure consistent detection.

2:Image Noise: Real-world images often contain various types of noise that degrade detection performance. Robust preprocessing and noise-resilient models are essential.

3: Latency: For real-time or near real-time applications, reducing detection latency is crucial. Optimizing performance without compromising accuracy is a key goal.

To deal with those challenges, we will use Image Augmentation to make our model can detect the same image with different angles. We will integrate convolutional self-attention mechanisms into auto-encoders to improve the robustness of spiking neural networks against adversarial attacks. The proposed framework is evaluated on multiple datasets using standard spiking neural networks (SNNs) and auto-encoder architectures, with a focus on classification accuracy to deal with Image Noise. We also will do hardware optimization to solve the latency problem.

A. *Dataset*

The proposed methodology is implemented using three publicly available, pre-processed datasets: the MNIST database of handwritten digits, the Fashion-MNIST database of Zalando product images and the Animal Faces dataset of real animal face.

The MNIST dataset contains 60,000 training images and 10,000 test images, each representing a handwritten digit. These grayscale images are 28x28 pixels in size and come with labels corresponding to one of ten digit classes. The dataset is a curated subset from a larger collection by NIST, with images normalized in size and centered within a fixed frame.

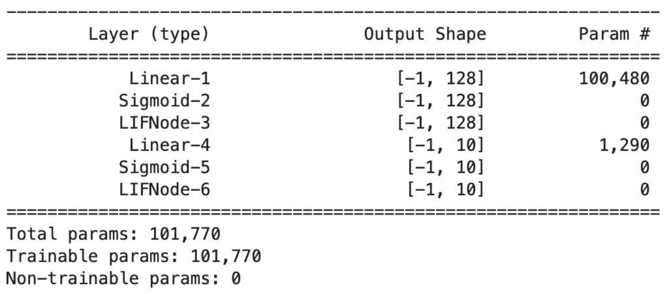
Fashion-MNIST was created to serve as a drop-in replacement for the original MNIST dataset in machine learning benchmarks. It shares the same image dimensions, data format, and training/testing splits, allowing for easy substitution and consistent comparison of algorithm performance.

The Animal Faces dataset [14] contains high-quality, labeled images of animal faces from various species, with consistent formatting to support model training and evaluation. Each image in the dataset is centered on the animal's face and typically resized to a fixed resolution—commonly 224x224 or 128x128 pixels—ensuring uniformity across samples. The dataset includes multiple classes, with each class representing a different animal species or breed. Labels are provided for supervised learning tasks, making it suitable for both classification and feature extraction experiments.

B. SNN models

- Simple SNN: We created our own SNN model and used it on MNIST and Fashion-MNIST to avoid overfitting. The specific SNN architecture is shown in fig 1.

- Spiking ResNet18 [15]: Consists of 18 layers with learnable parameters—16 convolutional layers and 2 fully connected layers. All convolutional layers use 3x3 kernels with stride 1 and padding 1, and shortcut connections are used to directly pass feature maps between layers, promoting easier gradient flow and stable training. The model processes input images of size 224×224×3 across multiple discrete time steps using spiking neurons such as leaky integrate-and-fire (LIF) units. Spiking activity replaces traditional ReLU activations, enabling event-driven, sparse computation. Outputs are aggregated over time and passed through a final softmax layer to produce class probabilities. Known for its balance of depth, efficiency, and biological plausibility, Spiking ResNet18 is widely used in tasks like image classification, object recognition, and neuromorphic computing.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 128]         100,480
           Sigmoid-2                  [-1, 128]               0
           LIFNode-3                  [-1, 128]               0
            Linear-4                   [-1, 10]           1,290
           Sigmoid-5                   [-1, 10]               0
           LIFNode-6                   [-1, 10]               0
================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
```

Fig. 1. SNN Architecture Details

C. Proposed Defense Architecture

We came up with a new convolutional self-attention auto-encoder by modifying the traditional convolutional auto-encoder with self-attention mechanism. The framework's details are shown in Fig 2.

The structure follows a U-shaped design. On the left side, we have the encoding pathway that processes the adversarial image through multiple convolutional layers with GELU activation functions.

What distinguishes our model is the yellow arrows representing self-attention, which we've added to create connections between distant features in the feature maps. This helps the model maintain global context even as the image is compressed.

At the bottleneck, we flatten the representation to a 1x288 vector - our latent space. Here, we introduce random Gaussian noise, which serves a crucial purpose: it makes our model more robust against adversarial examples by teaching it to reconstruct clean images even from slightly perturbed latent representations. This mirrors how adversarial examples themselves are created but uses this technique as a defense mechanism.

The right side shows our decoding pathway, which mirrors the encoder but uses transpose convolutions to progressively expand the feature maps back to the original image dimensions. Again, we've incorporated self-attention at the beginning of the decoder to ensure global context is maintained during reconstruction.

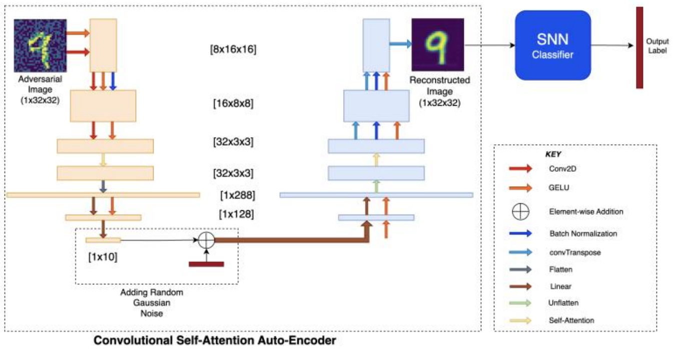


Fig. 2. Proposed Defense framework with a U-shaped Convolutional Self-attention Auto-Encoder and a SNN Classifier

Once processed through the entire network, the output is a reconstructed 28x28 image that closely resembles the original clean image, effectively removing the adversarial perturbations. This reconstructed image is then passed to a SNN classifier for final classification.We chose GELU activation functions throughout the network as they provide smoother gradients than ReLU, helping to avoid the dying ReLU problem and facilitating more effective learning across our deep architecture.

D. FGSM and PGD algorithm

We wrote an algorithm to combine our SNN model with FGSM and PGD attacks. It can depends on the adversarial type we put in to run the related codes automatically. Also, we divided the whole formula into several parts to get the middle variables which helps us to check whether our code is right or not.

Algorithm 1: SNN with FGSM and PGD

```
Input: SNN Model (M), Original Sample (X, y), Perturbation
    Strength (ε), Step Size (a), Number of Iterations (N)
Initialize:
I_in = RateEncoder (X)
∇ = 0
$X' = X$ + small random noise
1: if adversarial type is FGSM
2:     Output = M(I_in)
3:     L = Loss(Output, y)
4:     $\nabla = \nabla_x L(M(X), y)$
5:     $X' = X + \epsilon \cdot \mathrm{sign}(\nabla)$
6: else if adversarial type is PGD
7:     for i = 0 to N do
8:     Output = M($X'$)
9:     L = Loss(Output, y)
10:    $\nabla = \nabla_x L(M(X), y)$
11:    $X' = X' + a \cdot \mathrm{sign}(\nabla)$
12:    Project $X'$ back into ε-ball around X
13: end for
14: Return $X'$
```

E. Hardware Latency Design Imperatives

In the development of our framework, we chose to focus on three core optimization opportunities that we believed would provide us with the highest performance impact (given limited time for development). These optimizations were: the deliberate minimization of "high-expense" (read: high processor cycle count) functions, the elimination of floating-point calculations, and championing event-driven architecture where possible.

First, our effort to minimize high cycle-count functions forced us to focus on two core expensive events in the simulation of neurons. These were 1) optimizing the potential decay of neurons after a spike, and 2) minimizing spike detection branching.

Leveraging a standard Leaky, Integrate and Fire neuron, one of our most expensive functions is the exponent function, required to evaluate the decay of the LIF neuron's potential, as seen below. This exponent calculation alone is anticipated to cost thousands of cycles on a non-floating-point device such as the Raspberry Pi Pico (emblematic of the typical cheap, low-power edge device). Furthermore, this calculation needs to be done each time step (1 ms) for each neuron in the network, potentially having catastrophic effects on performance.

$$U_{mem}(t) = U_0 e^{-t/\tau}$$

Fig. 3. LIF Membrane Voltage Function

Leveraging our knowledge of low-level computing and hardware architecture, we recognized that performing a bit-shift may offer us a similar decay as the exponent function while being completed in magnitudes less time. Performing a bit shift, or rather sending the CPU a command to shift bits in memory by X places, offers us the mathematical opportunity to change a number by 2*x (shifted left) or 1/(2*x) (shifted right). However, this bit shift only costs us a single cycle on most architectures— offering magnitudes better performance. In our research, performing a bitwise shift by 7 on Q16 fixed-point arithmetic numbers offers us very similar accuracy to evaluating e^(-1/100) (single 1ms time step, tau = 100), only being +0.22% greater in its final result, while being magnitudes faster. Similarly, we recognized that we could reduce a significant number of pipeline stalls (due to the unpredictability of neuron results) by reducing the number of "if" statements used and instead replacing them with inline bitwise functions for basic threshold calculations. Traditionally, when a pipelined processor prepares for a branch (i.e. jumping to a different set of code if voltage > threshold), it will assume one outcome over the other and be forced to throw out the result and start again if it made an incorrect assumption. Thus, by forcing computation on all values, we can further optimize our core neuron simulation.

Second, we made the core design/optimization choice to eliminate floating-point calculations at the cost of potentially sacrificing accuracy. This design choice was made despite a potential loss in accuracy because a significant amount of cheap, low-power embedded processors do not have Floating Point Units, which means that floating-point-based math calculations would take significantly longer to calculate, having to be processed in software. Our team found this sacrifice to be acceptable because by leveraging Q16 fixed-point arithmetic, we were able to eliminate floating-point numbers while also not having any loss in accuracy. While Q16 only allows us 16 bits of definition compared to more advanced IEEE formats, leveraging Q16 means that we were able to maintain full non-lossy calculations while also simulating neurons with a more memory-efficient number set.

In this one case, preparing to sacrifice accuracy actually offered us two benefits in one!

Lastly, one of our goals (and core design tenets) is to implement event-driven calculations. Namely, we aim to design our neural network so that calculations with an assessed impact of zero are not evaluated. While not implemented yet, we anticipate that this will take the form of our network ignoring synapses with a weight of 0 and not evaluating neurons that have not received a spike and are connected via that 0 weight synapse. While additional work will be needed to ensure that training still functions with this in place and that this does not result in dead neurons, we anticipate that this may help to further optimize "un-eventful" (low-spiking) time steps.

```
import time
import torch
import snntorch as snn
import torch.nn as nn
# Simulation Parameters
num_steps = 100
input_size = 784      # MNIST image size: 28x28
hidden_size = 64      # Hidden layer size
output_size = 10      # Number of output classes (digits
0–9)
# Define the Network Model
class SNN(nn.Module):
    def __init__(self): …
```

Fig. 4. SNN Torch MNIST Setup Example [9]

```
void              update_neurons(int32_t
input_current[NUM_NEURONS]) {
    for (int i = 0; i < NUM_NEURONS; i++)
{
        neurons[i].V += input_current[i];
        neurons[i].V  -=  neurons[i].V  >>
7;
        neurons[i].fired = neurons[i].V >
THRESHOLD;
        neurons[i].V   =   (neurons[i].V   >
THRESHOLD) ? RESET_VALUE : neurons[i].V;
}

    //  Process  synaptic  connections  and
apply STDP
    for (int i = 0; i < NUM_NEURONS; i++)
{
        if (neurons[i].fired) {
```

Fig. 5. Custom C SNN Neuron "Update" Function

F. *Data Ingest*

In order to prepare our framework for evaluation with the MNIST dataset, we introduced the framework "MNIST for C" by Takafumi Horiuychi [8]. We chose to use this framework instead of designing a data-ingest mechanism because of the difficulty in doing it ourselves. In order to interpret the MNIST dataset, we would have to leverage the C filesystem, use advanced image frameworks in order to break down each image, do manual array conversions, and prepare that for use with our framework. Additionally, we assessed that our pursuit of optimization would likely not net us any major cost-savings in rewriting the MNIST for C framework, since image ingestion is by default a very cost-intensive task. As a result, we decided to stick with the MNIST for C framework and implement basic rate encoding in order to drive our networks' input.

V. RESULTS

We did several experiments and the following are the results. We will discuss our model's performance under different conditions.

A. *Image Augmentation*

We use some common augmentation techniques [16], eg: Rotation: Randomly rotate images (e.g., -45° to +45°) to simulate angle variations. Flipping: Horizontally or vertically flip images. Scaling & Cropping: Zoom in/out or crop parts of the image. Translation: Shift the image along X or Y axis. to create a new MNIST and Fashion-MNIST dataset. Then we test our model with/without image augmentation to get it's accuracy. The result is showed below. Our model can get 92.75% and 87.21% with augmentation.

Accuracy(w/o Augmentation)		**Accuracy(with Augmentation)**	
MNIST	***Fashion-MNIST***	***MNIST***	***Fashion-MNIST***
0.4721	0.3609	0.9275	0.8721

Table. 1. Accuracy for data augmentation.

B. *Evaluate on the MNIST and Fashion-MNIST dataset*

In order to solve the image noise problem. We combined our Convolutional Self-attention Auto-Encoder (CSAAE) with simple SNN on the MNIST and Fashion-MNIST dataset. For each one, we tested the performance with/without defense under adversarial attacks FGSM where ε = 0.6 and PGD where ε = 0.15.

We can see the data showed in table 2. Our simple SNN framework with defense method can get good accuracy.

Model	**Attack**	**Accuracy(w/o defense)**		**Accuracy(with defense)**	
		MNIST	***Fashion-MNIST***	***MNIST***	***Fashion-MNIST***
CSAAE	FGSM(ε=0.6)	0.2532	0.1648	0.8892	0.8531
CSAAE	PGD(ε=0.15)	0.0327	0.3150	0.9386	0.7699

Table. 2. Accuracy for on MNIST and Fashion-MNIST

C. *Evaluate on the Animal Face dataset*

We also want to know what's the performance on real-world image. So we choose animal face dataset to test. But our simple SNN can't deal so complex dataset, so we choose to use Spiking ResNet18 [15] to replace our simple SNN. We combined our Convolutional Self-attention Auto-Encoder (CSAAE) with *Spiking ResNet18* on the Animal Face dataset [14]. For each one, we tested the performance with/without defense under adversarial attacks FGSM where ε = 0.6 and PGD where ε = 0.15.

Model	**Attack**	**Accuracy(w/o defense)**	**Accuracy(with defense)**
		Animal Face	***Animal Face***
CSAAE	FGSM(ε=0.6)	0.1041	0.8027
CSAAE	PGD(ε=0.15)	0.0498	0.7636

Table. 3. Accuracy for on Animal Face dataset

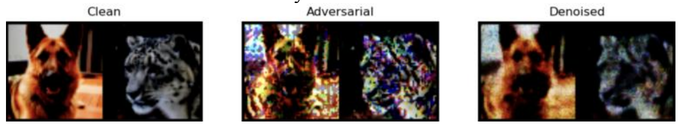


Fig. 6. clear, adversarial and denoise images

We can see the result from table 3. Our model still got good performance on real world image. although it didn't get a very high accuracy compared with MNIST and Fashion-MNIST dataset. It means the complex dataset will limit our model's ability to detect the image, but compared with the accuracy without defense method, our model can remove image noise.

VI. CHALLENGES

Throughout the whole project, we also met some challenges. We've completed the evaluation of both the SNN and CNN models under the same experimental conditions. While the CNN demonstrated stronger performance on static image detection—typically achieving 5% to 10% higher accuracy than the SNN depending on the the level of adversarial attacks—the SNN offers interesting insights into event-driven computation and shows potential for further optimization.

Throughout the latency optimization effort, we encountered a number of challenges that delayed the completion of the project. Our simplest challenge to overcome, as previously mentioned, was the ingestion of the MNIST dataset. Initially, we had chosen to try any method for data ingest and evaluated a number of other frameworks, as well as starting to build our own rudimentary method for data ingest. Due to the assessed complexity, this was abandoned, and MNIST for C was selected as our final method for data ingest. While easy to overcome, bringing a significant amount of image data into a low-level language still proved to be a daunting task.

Even more challenging, we faced (and still face) the challenge of spike train navigation. More concisely, navigating the three-dimensional array containing our generated spike train information requires at a minimum 2-3 multiplication operations per neuron per time cycle— a significant computational cost. Although we did not have enough time to resolve this issue in time for this report, one of our major challenges was trying to find a way to optimize out this computational expense. At the recommendation of Dr. Maryam Parsa of George Mason University, we anticipate looking at a number of other SNN frameworks to observe how they ingest MNIST data and more efficiently navigate a spike train.

Lastly, we faced the challenge of trying to characterize the overall accuracy of this model. While it is simple to characterize the accuracy and loss of individual functions (such as our characterization of 0.22% loss during decay functions), we have found it difficult and time-prohibitive to calculate the overall model's loss compared to its SNNTorch equivalent in the first part of this paper. Due to time constraints, we were unable to complete this model, and as such, have not been able to build a 1:1 equivalent to our SNNTorch-based model for static object detection. We anticipate being able to characterize loss in the future with additional time to complete the model.

Despite these hurdles, we believe that with more tuning or adapting the model architecture, the SNN could yield more competitive results in the future and our C implementation of a Spiking Neural Network simulation framework offers impressive performance and significant promise if it can be brought to completion.

VII. FUTURE GOALS

Given more time for the development of this model, we aim to complete a number of additions that promise to bring this SNN framework beyond its state as an MVP to a usable product for use in proprietary solutions or potential release to the public.

As the project has progressed, we now anticipate that the majority of products our SNN will run on the ARM-Cortex architecture (or some variant thereof), since this is the platform that a majority of low- to high-power edge-based devices are based on. With this design change in mind, we hope to further optimize the platform by targeting ARM-specific optimizations. For example, instead of using Q16 fixed-point arithmetic, ARM is optimized for the use of Q15 fixed-point arithmetic, which we intend to take advantage of to maximize memory use and minimize wasted CPU cycles.

In the interest of designing a 1:1 product (compared to other SNN frameworks), we also have the goal of implementing learning and offering the user the opportunity to train their C-based SNNs. Having studied training through a variety of our previous assignments, we aim to implement a basic form of Gradient Descent learning. Since high-level learning does not have to be run on-device, we are considering implementing the learning function on more powerful hardware and transferring the network weights to our deployed platform (to start).

Lastly, and as previously mentioned, we aim to fully characterize the accuracy of our network compared to other SNN frameworks. While we do not have a particular goal in mind, we are hoping, post-training, to be at most 2-3% off the results of more accuracy-driven frameworks. If we can meet this result, there is some promise that the results of our network can be trusted while running significantly (20-30x, in recent testing) faster. Additionally, if we are able to effectively characterize the source of this error, we may also be able to implement neuron-level error correction and reduce that error.

While we were time-constrained in the development of this SNN framework, we are very pleased with the results and believe that it was a meaningful side-goal in the exploration of static object detection with an SNN.

VIII. Conclusion

We created a spiking neuron networks (SNNs) combined with Data Augmentation and Convolutional Self-Attention Auto-Encoder (CSAAE) to enhance the robustness of image reconstruction to solve angle variation and noise problems.

We also developed a C SNN Simulation Framework MVP for embedded hardware to demonstrate the end-to-end accuracy cost of low level code optimizations and achieved significant improvement in reducing hardware latency. Our custom C framework reduces latency through fixed-point arithmetic and hardware-centric optimizations, paving the way for edge deployment without neuromorphic hardware.

Future work includes ARM-specific optimizations, gradient-descent training, and full accuracy characterization against Python frameworks. While challenges remain in spike-train navigation and error correction, our results underscore SNNs' potential as efficient, noise-resilient solutions for real-world applications. By combining biological plausibility with computational efficiency, this work advances the state of SNN-based static image detection.

Addition Information

All code scripts have been made available at https://github.com/prediction-hao/ECE-556-Neuromorphic-Computing/tree/main/Final%20Project

References

[1] Yamazaki, K., Vo-Ho, V.-K., Bulsara, D., & Le, N. (2022). Spiking Neural Networks and Their Applications: A Review. Brain Sciences, 12(7), 863. https://doi.org/10.3390/brainsci12070863

[2] YFang, W., Chen, Y., Ding, J., Yu, Z., Masquelier, T., Chen, D., Huang, L., Zhou, H., Li, G., & Tian, Y. (2023). SpikingJelly: An open-source machine learning infrastructure platform for spike-based intelligence. Science Advances, 9(40), eadi1480. https://doi.org/10.1126/sciadv.adi1480

[3] Wu, P., Gong, S., Pan, K., Qiu, F., Feng, W., & Pain, C. (2021). Reduced order model using convolutional auto-encoder with self-attention. Physics of Fluids, 33(7). https://doi.org/10.1063/5.0051155

[4] Eshraghian, J. K., Ward, M., Neftci, E., Wang, X., Lenz, G., Dwivedi, G., Bennamoun, M., Jeong, D. S., & Lu, W. D. (2023). Training Spiking Neural Networks Using Lessons From Deep Learning. Proceedings of the IEEE, 111(9), 1016–1054. https://doi.org/10.1109/JPROC.2023.3291534

[5] Frontiers in Neuroscience. (2024). A comprehensive review of advanced trends: from artificial synapses to neuromorphic systems. Frontiers in Neuroscience, 18, 1279708. https://doi.org/10.3389/fnins.2024.1279708

[6] Paredes-Valles, Federico. “cuSNN.” *GitHub*, github.com/tudelft/cuSNN. Accessed 25 Apr. 2025.

[7] Rezghi, Shahriar. “Spyker.” *GitHub*, github.com/ShahriarRezghi/. Accessed 26 Apr. 2025.

[8] Horiuchi, Takafumi. “Mnist Dataset Loader for C.” *GitHub*, github.com/takafumihoriuchi/MNIST_for_C. Accessed 26 Apr. 2025.

[9] Eshraghian, Jason K. “Tutorial 3 - A Feedforward Spiking Neural Network.” *Tutorial 3 - A Feedforward Spiking Neural Network - Snntorch 0.9.4 Documentation*, snntorch.readthedocs.io/en/latest/tutorials/tutorial_3.html. Accessed 26 Apr. 2025.

[10] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” in International Conference on Learning Representations (ICLR), 2015

[11] Deng, Y., & Karam, L. J. (2020). Universal adversarial attack via enhanced projected gradient descent. 2022 IEEE International Conference on Image Processing (ICIP), 1241–1245. https://doi.org/10.1109/icip40778.2020.9191288

[12] Mandal, S. (2023). Defense Against Adversarial Attacks using Convolutional Auto-Encoders. arXiv preprint arXiv:2312.03520

[13] Zhao, H., Jia, J., & Koltun, V. (2020). Exploring Self-Attention for Image Recognition. 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). https://doi.org/10.1109/cvpr42600.2020.01009M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.

[14] https://www.kaggle.com/datasets/andrewmvd/animal-faces/data

[15] Fang, W., Yu, Z., Chen, Y., Huang, T., Masquelier, T., & Tian, Y. (2021). Deep Residual Learning in Spiking Neural Networks. Advances in Neural Information Processing Systems, 34, 2102–2114. https://proceedings.neurips.cc/paper_files/paper/2021/hash/afe434653a898da20044041262b3ac74-Abstract.html

[16] Shorten, C., & Khoshgoftaar, T. M. (2022). Data augmentation: A comprehensive survey of modern approaches. Journal of Big Data, 9(1), 1-54. https://doi.org/10.1186/s40537-019-0197-0