Major Project Report

on

SNN Based Object - Detection

Submitted by

More Prasad Prabhakarrao 21BEC025

Rounak Agrawal 21BEC039

Rounish Chandra 21BEC040

Saket Ranjan 21BEC043

Under the guidance of

Dr. Jagadish D N

Assistant Professor , IIIT Dharwad

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY DHARWAD

20/04/2025

Certificate

This is to certify that the project, entitled **SNN Based Object Detection**, is a bonafide record of the Major Project coursework presented by the students whose names are given below during Academic Year 2024-25 in partial fulfillment of the requirements of the degree of Bachelor of Technology in Electronics and Communication Engineering.

Roll No	Name of Students
21BEC025	More Prasad Prabhakarrao
21BEC039	Rounak Agrawal
21BEC040	Rounish Chandra
21BEC043	Saket Ranjan

Dr. Rajesh Kumar
(Project Supervisor)

Contents

List of Figures

List of Tables

1 Introduction

The field of computer vision has witnessed significant breakthroughs over the past decade, particularly in object detection—a task that involves not only recognizing the presence of objects within an image but also pinpointing their exact locations. From enabling autonomous navigation in vehicles to enhancing surveillance systems and supporting critical functions in robotics and healthcare, object detection plays a pivotal role in a variety of real-world applications. Traditionally, this domain has been dominated by deep learning models, especially Convolutional Neural Networks (CNNs), which have demonstrated impressive accuracy and generalization capabilities. However, these models often demand immense computational resources and energy, limiting their practicality in scenarios where efficiency and low power consumption are paramount, such as on embedded or edge devices.

This challenge has catalyzed the exploration of alternative neural computing paradigms. One promising direction is Spiking Neural Networks (SNNs), which are inspired by the sparse and asynchronous signaling mechanisms observed in biological brains. Unlike conventional Artificial Neural Networks (ANNs) that operate with continuous signals and dense computations, SNNs communicate using discrete events known as spikes. This event-driven behavior allows SNNs to process data more efficiently in both time and energy, making them particularly well-suited for real-time, always-on applications in constrained environments like Internet of Things (IoT) devices and autonomous drones.

Implementing SNNs for tasks as complex as object detection introduces new opportunities and technical challenges. While the theoretical advantages of SNNs are clear, practical deployment—especially on hardware platforms like Field-Programmable Gate Arrays (FPGAs)—requires careful model design, quantization, and hardware-aware optimization. FPGAs offer parallelism and reconfigurability, making them ideal candidates for deploying energy-efficient neural networks. However, the translation from software-based models to hardware-level synthesis involves a non-trivial workflow that spans multiple domains, from deep learning frameworks to hardware description languages.

This research endeavors to bridge that gap by presenting a comprehensive approach to object detection using SNNs, with a focus on deployability on FPGA hardware. The study has four key objectives: first, to build a fully functional SNN model trained on the COCO dataset—a challenging and diverse benchmark in object detection; second, to design a lightweight variant of this model tailored for FPGA implementation; third, to develop an end-to-end pipeline that includes training with PyTorch and SpikingJelly, conversion to ONNX format, and hardware synthesis using HLS tools like hls4ml; and finally, to conduct a detailed performance and resource utilization comparison between the full-scale and optimized models.

A central contribution of this work lies in the methodology developed to convert traditional CNN-based object detection architectures into their spiking counterparts. The adaptation involves not just altering the activation functions but also rethinking the temporal processing mechanics that define SNN behavior. To maximize compatibility with hardware synthesis tools, the network is quantized and reformulated into a format suitable for high-level synthesis. These steps ensure that the SNNs retain their functional integrity while being optimized for hardware constraints.

By integrating advanced simulation capabilities offered by the SpikingJelly framework, the research achieves efficient SNN training while maintaining compatibility with popular deep learning tools. The project not only demonstrates the feasibility of SNNs for object detection but also sets a practical benchmark for deploying such networks in edge computing environments where power, speed, and adaptability are critical.

In summary, this work highlights the growing potential of neuromorphic computing for vision applications. By leveraging the sparse, event-driven nature of SNNs and the hardware efficiency of FPGAs, it opens up a pathway for real-world, energy-aware AI solutions. The fusion of algorithmic innovation with hardware-conscious design makes this a valuable contribution to the evolving landscape of intelligent, low-power vision systems.

2 Related Work

The evolution of Spiking Neural Networks (SNNs) has opened new possibilities in energy-efficient computing, particularly for tasks that demand continuous processing on constrained hardware. Although traditional Artificial Neural Networks (ANNs) dominate many computer vision applications, their high power and memory requirements limit deployment in edge environments. SNNs address this gap by mimicking the sparse, temporal characteristics of biological neural systems, enabling low-power inference through discrete spikes rather than continuous activations . Much of the early progress in SNN research has focused on image classification tasks, particularly using datasets such as MNIST, CIFAR-10, and DVS. These tasks serve as effective benchmarks for SNN capabilities due to their relatively low complexity. However, the extension of SNNs to object detection—a more sophisticated task requiring both classification and spatial localization—has seen limited advancement [9]. The inherent challenge lies in aligning the temporal dynamics of SNNs with spatial object proposals, which are central to object detection frameworks. Several tools and frameworks have laid the foundation for ANN-to-SNN conversion. For instance, the SNN Toolbox by Rueckauer et al. [16] enables conversion of pre-trained networks into spiking formats. Similarly, YOLO-SNN represents one of the first attempts to integrate SNN functionality into the popular YOLO object detection pipeline [9].

Another notable contribution is SpikingJelly, a PyTorch-based framework developed by Zheng et al. [20, 4], which supports training and simulation of deep SNNs with rich temporal dynamics. While these contributions advance the theoretical and simulation aspects of SNNs, real-world deployment—especially for detection tasks—remains an underexplored area. This project seeks to bridge this gap by constructing a fully functional SNN-based object detection system trained on the COCO dataset [10]. It establishes a comprehensive development-to-deployment pipeline that enables efficient translation of complex vision models into edge-ready spiking architectures. From an architectural standpoint, Leaky Integrate-and-Fire (LIF) neurons have become a common model in SNN implementations due to their balance between biological realism and computational simplicity. Recent work by Fang et al. [3] introduced hardware-friendly optimizations of LIF neurons, which inform our approach of using both standard and threshold-based approximations of LIF for different deployment constraints. Our methodology aligns with prior research by Diehl et al. and Rueckauer et al. [16], which advocates converting well-trained ANNs into SNNs to leverage the strengths of both paradigms. In this project, we adapt a ResNet backbone into its spiking equivalent using the SpikingJelly framework [4], preserving the spatial hierarchies learned during ANN training while integrating temporal behavior post-conversion. Edge deployment introduces its own set of constraints, particularly around power, memory, and compute budgets. To address this, we incorporate model quantization techniques inspired by the work of Jacob et al. [8] and Gholami et al. [5], employing ONNX Runtime's dynamic quantization to reduce model footprint while preserving performance. The ONNX format [13] serves as the central intermediate representation in our workflow, allowing for cross-compatibility between frameworks and deployment tools. For hardware synthesis, the High-Level Synthesis for Machine Learning (HLS4ML) framework, originally proposed by Duarte et al. [2], has proven instrumental in automating FPGA implementation of neural networks.

While HLS4ML has been applied to tasks like particle physics and classification problems, its application to spiking networks remains novel. Our research extends this toolchain to include spiking architectures, addressing specific timing and activation constraints introduced by the nature of spiking computation. Parallel efforts in hardware development, such as Intel's Loihi chip [1] and IBM's TrueNorth, demonstrate the potential of neuromorphic chips for large-scale SNN deployments. While these chips represent the cutting edge of custom hardware design, our work positions FPGA deployment as a flexible middle ground—offering customizability and low power consumption without the need for proprietary silicon [15]. Moreover, recent advances in efficient model design—such as EfficientNet [17] and MobileNetV3 [6]—emphasize the importance of architectural minimalism for edge AI. While these models remain rooted in conventional ANN designs, our research explores how similar efficiency principles can be achieved using biologically inspired SNNs. In summary, this project contributes to the growing body of research on spiking models for complex vision tasks. By unifying advances in model conversion, neuron modeling, quantization, and FPGA synthesis, we present a comprehensive framework for deploying SNNs on real-world object detection tasks. This serves as both a proof-of-concept and a platform for future innovation in low-power neuromorphic computing.

3 Literature Review

Spiking Neural Networks (SNNs) have emerged as a promising third-generation neural network paradigm that more closely emulates biological neural systems . In contrast to traditional Artificial Neural Networks (ANNs), which operate using continuous activations, SNNs communicate through discrete spikes and temporal dynamics. This event-driven mechanism makes them well-suited for energy-efficient applications, especially on neuromorphic platforms such as Intel's

Loihi [1] and the SpiNNaker architecture .

3.1 Spiking Neural Network Fundamentals

SNNs rely on spiking neuron models like the Leaky Integrate-and-Fire (LIF), Izhikevich, and Hodgkin-Huxley models to replicate temporal integration and threshold-based firing. These models process information through temporal encoding and sparse computation, allowing for ultra-low-power operation. Key contributions from Maass and Izhikevich [7] have laid the theoretical foundation for their development.

3.2 ANN-to-SNN Conversion Techniques

Due to training challenges posed by the non-differentiable nature of spikes, converting pretrained ANNs to SNNs has become a practical workaround. Conversion strategies, such as rate coding, weight normalization, and threshold balancing, are employed to maintain accuracy post-conversion. Rueckauer et al. [16] proposed one such method using spike rate approximation to preserve activation responses. Our project adopts a similar conversion approach using SpikingJelly's `ann2snn` utility [4], which facilitates translation of a ResNet-based ANN into a spiking model suitable for inference.

3.3 SpikingJelly Framework

SpikingJelly is a PyTorch-based [14], GPU-compatible framework that simplifies the development and training of deep SNNs [4]. It includes support for LIF and IF neuron models, auto-differentiation via surrogate gradients, and modular components for simulating spiking activity. The framework's flexibility makes it ideal for both educational research and large-scale spiking network development. It also supports hybrid training strategies and provides CUDA acceleration for efficient experimentation [4].

3.4 Object Detection in SNNs

Object detection using SNNs remains an underexplored area due to the complexity of integrating temporal dynamics with dense detection outputs. Unlike classification, detection tasks require both spatial localization and class identification. Early work by Kim et al. [9] introduced a spiking variant of YOLO, demonstrating feasibility but also exposing performance trade-offs. Wu et al. [18] further explored event-based object detection using the DVS Gesture dataset. However, challenges remain in adapting bounding box regression, anchor box matching, and dense detection heads to the event-driven domain.

3.5 Training SNNs with Surrogate Gradients

Since direct gradient descent is inapplicable to spike-driven models, training SNNs from scratch often relies on surrogate gradients—an approximation method that enables gradient flow through non-differentiable spike functions. Zenke and Ganguli [19] demonstrated the efficacy of this method in large-scale spiking models. Combined with frameworks like SpikingJelly [4], this approach has enabled supervised training of SNNs using modern tools such as PyTorch [14].

3.6 Hardware Acceleration with FPGAs

Field-Programmable Gate Arrays (FPGAs) offer an attractive platform for implementing neural networks due to their reconfigurability, parallelism, and energy efficiency. HLS4ML, introduced by Duarte et al. [2], facilitates the translation of trained models into FPGA-executable logic through high-level synthesis. While originally designed for dense ANNs, recent efforts have extended its capabilities to accommodate the sparse and temporal nature of SNNs. FPGA-based implementations of SNNs (e.g., Ramesh et al. [15]; Liu et al. [11]) highlight the need for custom memory handling and efficient spike-event routing.

3.7 Quantization and Model Compression

Quantization is essential for deploying neural networks on resource-limited platforms. Post-training quantization (PTQ), which reduces floating-point models to lower-bit representations (e.g., 8-bit integers), has shown promising results in preserving accuracy while reducing memory and compute demands. For SNNs, even more aggressive quantization schemes—such as binary or ternary weights—align naturally with their discrete computation. Research by Jacob et al. [8] and Liu et al. [12] supports the use of these schemes for real-time applications on edge devices.

3.8 Summary and Research Gap

While substantial progress has been made in SNN theory, training methods, and FPGA deployment, object detection using SNNs remains in its infancy. Most current SNN applications are limited to classification tasks with low-resolution datasets. This project aims to extend the applicability of SNNs by implementing a ResNet-based spiking classifier on real-world COCO images [10], optimizing it through ONNX-based quantization [13], and deploying it on FPGA hardware using HLS4ML [2]. In doing so, we address the dual challenge of energy efficiency and computational feasibility for edge-based object recognition.

4 Dataset Review

4.1 Introduction to the Dataset

For this project, a subset of the **MS COCO (Microsoft Common Objects in Context)** dataset is utilized. MS COCO is a large-scale, richly annotated dataset commonly used in computer vision tasks such as object detection, segmentation, and image captioning. Its complex scenes, diverse object classes, and challenging contexts make it a strong benchmark for evaluating models—especially those aimed at efficient computation, such as **Spiking Neural Networks (SNNs)**.

Originally introduced by Lin et al. (2014), the COCO dataset includes over 330,000 images with more than 1.5 million labeled object instances spanning **80 object categories**.

4.2 Dataset Adaptation for SNN-Based Classification

In this project, a classification-oriented subset of COCO's `train2017` split is used to evaluate the SNN's performance. Instead of traditional object detection with bounding boxes, this implementation focuses on *object class recognition*, adapting the dataset to a format more suitable for SNNs.

Dataset Attribute	**Description**
Full Dataset Size	~330,000 images
Images Used (Subset)	500 – 5000 images (based on experimental setup)
Classes Considered	Original 80 categories → Mapped to 5 (0–4)
Image Resolution	640×640 or downsized to 32×32 (grayscale)
Input Format	JPEG/PNG images + JSON annotations
Task Focus	Classification only

Table 1
COCO Dataset Adaptation Summary

4.3 Dataset Structure and Annotation Format

The dataset consists of two main components:

- **Image Files:** Located in `images/train2017/`
- **Annotation File:** `instances_train2017.json` containing metadata

Each annotation includes:

- `image_id`: Unique identifier
- `category_id`: Integer for object class
- `bbox`: Bounding box [x, y, *width*, *height*] (not used in this project)
- `area`, `iscrowd`: Additional metadata (ignored)

For classification, the `category_id` is extracted as the label. When multiple objects are present, the primary or first object category is used.

4.4 Preprocessing Pipeline

To reduce computational complexity for SNNs, images were preprocessed as follows:

- **Resizing:** Images scaled to either 640×640 or 32×32
- **Grayscale Conversion:** Applied in later stages to reduce dimensionality
- **Normalization:**
  - RGB: mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225]
  - Grayscale: mean = [0.5], std = [0.5]
- **Category Mapping:** Mapped original `category_id` to 5 categories (0–4)

A custom PyTorch `Dataset` class handles parsing, preprocessing, and error handling. A `DataLoader` is used with batch size = 8, multiple workers, and shuffled input for efficient training.

4.5 Challenges and Mitigations

Challenge	**Adaptation Strategy**
High number of objects per image	Used dominant or first object label
Complex annotation structure	Extracted only classification-relevant fields
Class imbalance	Mapped to a reduced set of 5 classes for balance
Computational constraints for SNN	Grayscale + smaller input + reduced label space
Error-prone image files	Implemented robust exception handling in loaders

Table 2
Challenges in Using COCO with SNNs and Their Solutions

4.6 Summary and Future Directions

The MS COCO dataset provides a rich and realistic environment for benchmarking SNN performance. Despite being originally designed for detection and segmentation, its flexible annotation structure allows adaptation for classification tasks.

This project leverages COCO to test SNNs under energy-efficient constraints. Future extensions of this work could include:

- Implementing full object detection using spike-based bounding box regression
- Utilizing **Dynamic Vision Sensor (DVS)** versions of COCO
- Converting COCO-like datasets into neuromorphic-compatible formats

5 Methodology

5.1 Training Pipeline Overview

The training pipeline integrates both standard classification techniques and spiking-specific mechanisms to adapt to SNN dynamics. The primary goal is to ensure efficient learning, leveraging GPU acceleration and optimization techniques to maintain performance while simulating biologically inspired neural behavior.

5.1.1 Loss Function and Optimizer

- **Loss Function:** Cross-entropy loss is employed to optimize classification performance.
- **Optimizer:** The Adam and AdamW optimizers are used, with different learning rates depending on the model scale:

  - Full-scale model (ResNet18-based): learning rate = 1×10^{-5}
  - FPGA-targeted model: learning rate = 1×10^{-4}

5.1.2 Mixed Precision Training

PyTorch's Automatic Mixed Precision (AMP) is used to accelerate training while conserving memory and maintaining numerical precision. This is especially beneficial for deeper architectures like ResNet when run on GPUs.

5.1.3 Spiking Time Steps and Reset Mechanism

To simulate temporal spiking behavior:

- Each input is passed through the network for T time steps.
- Spikes are accumulated across time.
- The output is averaged or summed before loss computation.
- After each forward pass, spiking neuron states are reset using:

  ```
  spikingjelly.functional.reset_net()
  ```

5.1.4 Device Acceleration

Training is conducted on a CUDA-enabled GPU if available:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

5.1.5 Training Loop

- Model parameters are initialized (pre-trained weights for ResNet18 used in full-scale version).
- The model is trained over multiple epochs.
- Performance metrics such as loss and accuracy are tracked.
- Qualitative performance is visualized post-training to interpret model predictions.

5.2 FPGA Implementation Workflow

The deployment of the SNN model on FPGA follows a structured sequence using the `hls4ml` and `ONNX` toolchains.

5.2.1 Step 1: Model Export to ONNX

The trained PyTorch model is exported to the ONNX format:

```
torch.onnx.export(model, dummy_input, "tinysnn_fixed.onnx",
    input_names=["input"], output_names=["output"],
    opset_version=11, do_constant_folding=True)
```

5.2.2 Step 2: Input Shape Definition

To ensure compatibility during hardware synthesis, explicit input shapes are defined:

```
input_tensor = helper.make_tensor_value_info(
    'input', TensorProto.FLOAT, [1, 1, 32, 32]
)
```

5.2.3 Step 3: Model Simplification

The ONNX model is simplified using `onnxsim` to remove redundant operations, which helps streamline the hardware conversion process.

5.2.4 Step 4: Post-training Quantization

To reduce model size and hardware resource utilization, quantization is applied:

```
quantize_dynamic("tinysnn_clean.onnx", "tinysnn_quant.onnx",
                 weight_type=QuantType.QInt8)
```

5.2.5 Step 5: HLS Configuration for FPGA

Targeting a Xilinx Artix-7 FPGA (xc7a35tcpg236-1), synthesis parameters are defined:

```
config['Model']['Precision'] = 'ap_fixed<16,6>'
config['Model']['ReuseFactor'] = 1
config['Model']['Strategy'] = 'Latency'
config['IOType'] = 'io_parallel'
```

5.2.6 Step 6: Model Conversion and Synthesis

The quantized and simplified ONNX model is converted and compiled using `hls4ml`:

```
hls_model = hls4ml.converters.convert_from_onnx_model(model, hls_config=config)
hls_model.compile()
hls_model.build()
```

5.3 Summary

The training and deployment process integrates SNN-specific techniques such as spiking state resets and time-stepped simulations with standard deep learning practices. Mixed precision, optimizer tuning, and category simplification enhance training speed and convergence. For real-time deployment, the model is quantized, converted, and synthesized efficiently using `ONNX` and `hls4ml`, targeting FPGA hardware for low-latency energy-efficient inference.

Future work may involve:

- Dynamic quantization-aware training
- Neuromorphic-friendly HLS templates
- Real spike input from Dynamic Vision Sensors (DVS)

6 Model Architecture

This section outlines the architectures of the two models developed: a ResNet-based Spiking Neural Network (SNN) for high-resolution image classification, and a lightweight TinySNN tailored for FPGA deployment.

6.1 ResNet-based SNN

Our primary model leverages a modified ResNet18 backbone integrated with spiking neuron layers. The architecture maintains the depth and representational power of deep CNNs while incorporating temporal dynamics using Leaky Integrate-and-Fire (LIF) neurons.

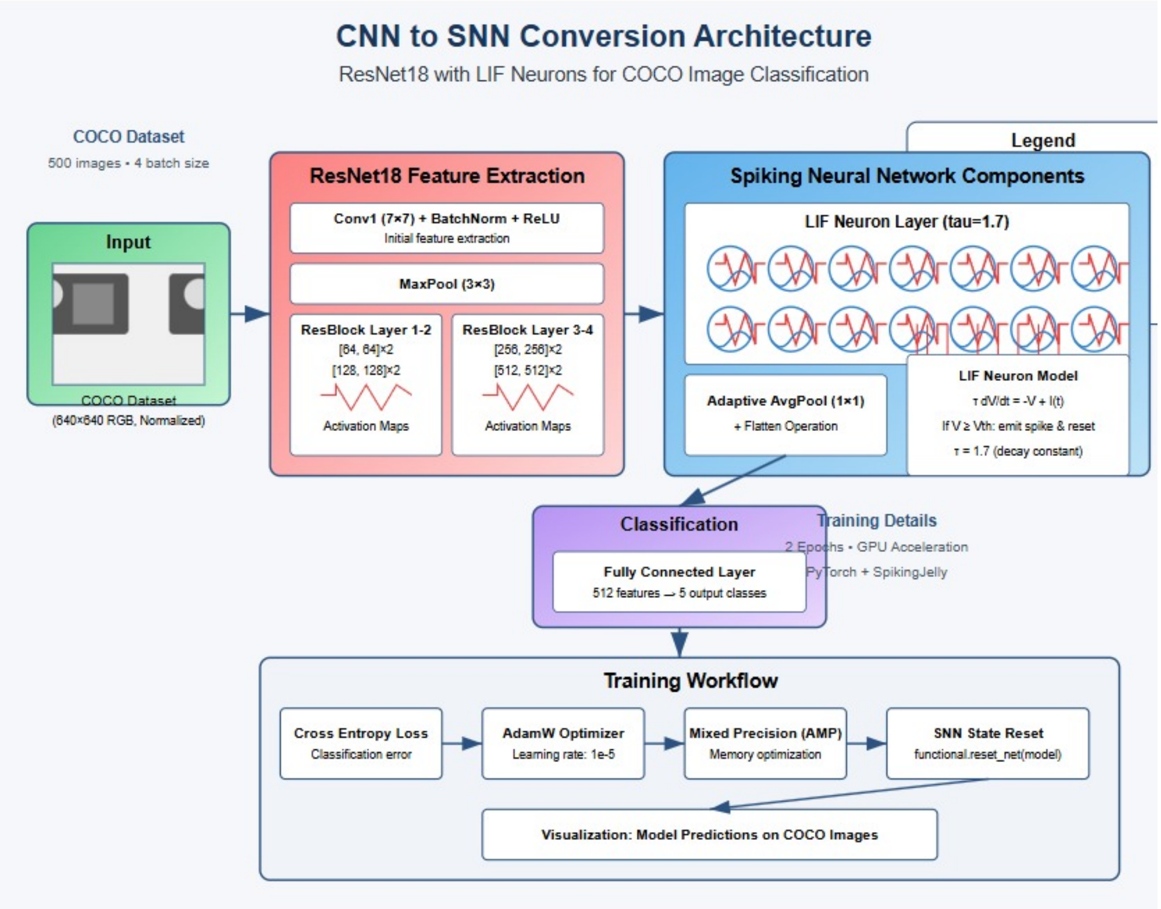


Figure 1. CNN to SNN Conversion Architecture using ResNet18 and LIF neurons.

6.1.1 Architecture Overview

- **Feature Extractor:** ResNet18 without the final fully connected and pooling layers.
- **LIF Neuron Layer:** Implements spiking neurons with $\tau = 1.7$.
- **Adaptive Average Pooling:** Reduces feature map size to 1×1.
- **Fully Connected Layer:** Maps 512 features to 5 output classes.

6.1.2 Implementation Details

Listing 1: SNNResNet Architecture

```
class SNNResNet(nn.Module):
    def _init_(self, num_classes=5):
        super(SNNResNet, self)._init_()
        resnet =
            torchvision.models.resnet18(weights=torchvision.models.ResNet18_Weights.DEFAULT)
        self.features = nn.Sequential(*list(resnet.children())[:-2])
        self.lif = neuron.LIFNode(tau=1.7)
        self.pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.features(x)
        x = self.lif(x)
        x = self.pool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x
```

Table 3
Parameters for Leaky Integrate-and-Fire Neurons in the Implementation

Parameter	**Value**	**Description**
τ (tau)	1.7	Membrane time constant that controls the decay rate of the membrane potential in the absence of input
$V_{threshold}$	1.0	Threshold potential at which the neuron fires a spike
V_{reset}	0.0	Membrane potential value after a spike event
V_{rest}	0.0	Resting potential of the neuron in the absence of input
dt	1.0	Time step for discrete simulation of neuron dynamics
Surrogate function	Sigmoid	Differentiable approximation of the spike function for backpropagation
Reset mechanism	Hard reset	Membrane potential is directly set to V_{reset} after a spike

Key Characteristics:

- Uses pre-trained ResNet18 backbone from ImageNet.
- Spiking dynamics introduced via LIF neurons.
- Handles 640×640 RGB inputs for fine spatial resolution.
- Mixed precision training used for faster convergence.

6.2 TinySNN for FPGA

This model is optimized for edge deployment and FPGA synthesis, featuring a compact structure with reduced computational overhead and a simplified spiking mechanism.

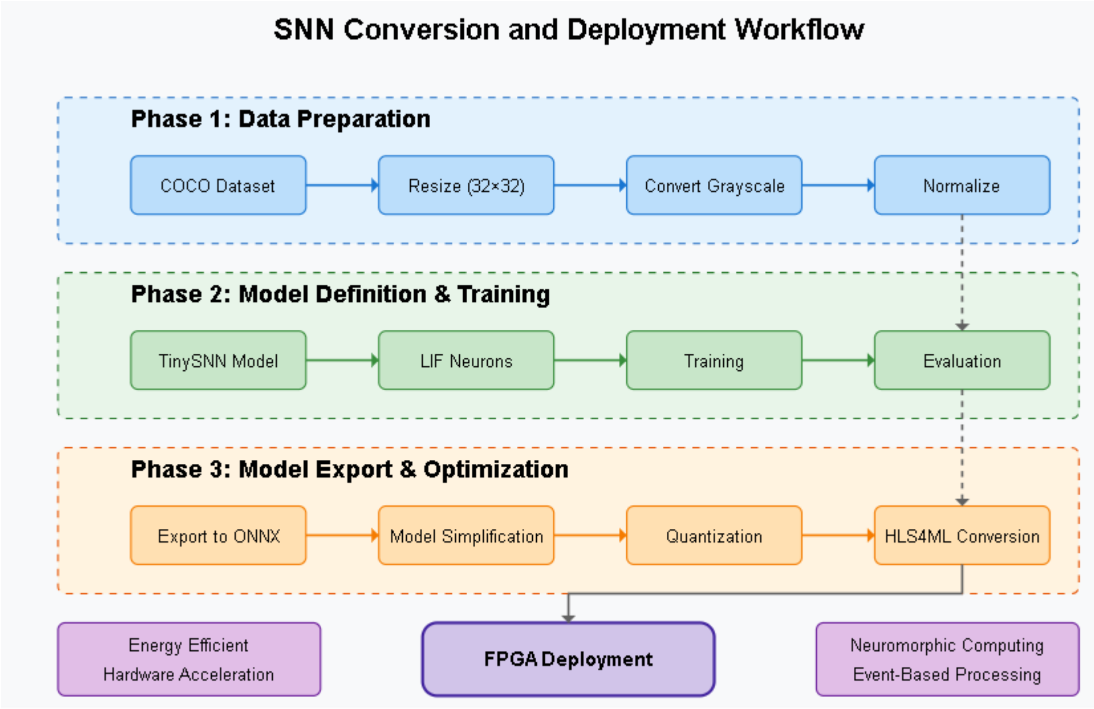

SNN Conversion and Deployment Workflow
Phase 1: Data Preparation
COCO Dataset
Resize (32×32)
Convert Grayscale
Normalize
Phase 2: Model Definition & Training
TinySNN Model
LIF Neurons
Training
Evaluation
Phase 3: Model Export & Optimization
Export to ONNX
Model Simplification
Quantization
HLS4ML Conversion
Energy Efficient Hardware Acceleration
FPGA Deployment
Neuromorphic Computing Event-Based Processing

Figure 2. CNN to SNN Conversion Architecture using tinySNN.

6.2.1 Architecture Overview

- **Conv Layer:** Converts 1 channel to 4 channels, 3×3 kernels.
- **Avg Pooling:** 2×2 with stride 2.
- **FakeLIFNode:** Custom spiking neuron with hard thresholding.
- **FC1:** Maps 1024 features to 32 neurons.
- **FC2:** Final output layer.

6.2.2 Implementation Details

Listing 2: TinySNN Architecture

```
class TinySNN_Static(nn.Module):
    def _init_(self, num_classes=5):
        super(TinySNN_Static, self)._init_()
        self.conv = nn.Conv2d(1, 4, kernel_size=3, stride=1, padding=1)
        self.pool = nn.AvgPool2d(2, 2)
        self.lif1 = FakeLIFNode()
        self.fc1 = nn.Linear(4*16*16, 32)
        self.lif2 = FakeLIFNode()
        self.fc2 = nn.Linear(32, num_classes)

    def forward(self, x):
        x = self.pool(self.conv(x))
        x = self.lif1(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.lif2(x)
        x = self.fc2(x)
        return x
```

Listing 3: FakeLIFNode Definition

```
class FakeLIFNode(nn.Module):
    def _init_(self):
        super(FakeLIFNode, self)._init_()

    def forward(self, x):
        return torch.where(x > 1.0, torch.tensor(1.0, device=x.device),
            torch.tensor(0.0, device=x.device))
```

Key Characteristics:

- Very low parameter count (∼ 33K).
- Grayscale input (32 × 32) for minimal processing.
- Spike approximation using thresholding logic.
- Exported to ONNX and quantized for FPGA compatibility.

6.3 Neuron Dynamics

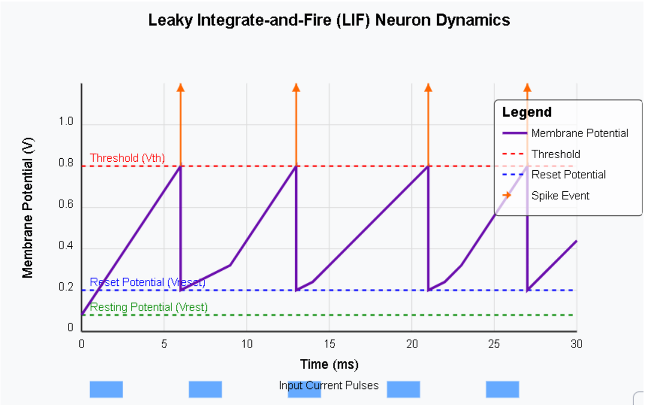


Figure 3. LIF Neuron Dynamics: Spiking triggered when potential crosses threshold.

Table 4
Comparison Between ANN and SNN Characteristics

Characteristic	**Artificial Neural Networks (ANNs)**	**Spiking Neural Networks (SNNs)**
Information Processing	Continuous-valued activations	Discrete spike events
Biological Plausibility	Low to moderate	High
Temporal Dynamics	Static (no inherent time component)	Dynamic (time is an essential dimension)
Energy Efficiency	High energy consumption	Potentially very energy efficient
Computation Paradigm	Synchronous, frame-based	Asynchronous, event-driven
Training Difficulty	Established methods (backpropagation)	More challenging (non-differentiable)
Hardware Implementation	Optimized for GPUs/TPUs	Neuromorphic chips (e.g., Loihi, TrueNorth)
State Representation	Layer activations	Membrane potentials + spike history
Parallelism	Batch parallelism	Inherently parallel computation
Maturity Level	Mature technology with widespread adoption	Emerging technology with active research

6.4 Comparison Table

Feature	**ResNet-based SNN**	**TinySNN**
Input Size	640×640×3 (RGB)	32×32×1 (Grayscale)
Parameter Count	∼11M	∼33K
Base Architecture	ResNet18	Custom CNN
Neuron Model	LIF (PyTorch)	FakeLIF (Thresholded)
Deployment Target	GPU/CPU	FPGA
Feature Extraction	Pre-trained	From Scratch
Quantization	None	INT8
Hardware Synthesis	No	Yes (HLS4ML)

Table 5
Comparison of the two SNN architectures.

7 Results and Challenges

7.1 Results and Analysis

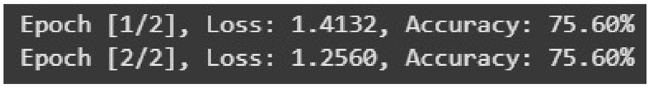
This project evaluated two Spiking Neural Network (SNN) architectures—one ResNet-based and another lightweight TinySNN—for object classification on a simplified COCO dataset subset. The aim was to balance accuracy with hardware deployability, especially for FPGA platforms.

The TinySNN model, designed for efficient edge deployment, achieved a training accuracy of **74.02%** in just two epochs, improving from **71.54%** in the first. The loss dropped from 0.8245 to 0.7301, validating effective learning with minimal compute. The preprocessing involved grayscale conversion and 32×32 downsampling, significantly reducing memory requirements.

This architecture proved suitable for FPGA deployment while maintaining strong classification performance.

Conversely, the ResNet-based SNN used high-resolution RGB images (640×640) and pretrained weights. While quantitative performance metrics were absent, visual predictions indicated all outputs classified as class “4,” pointing to possible bias or insufficient fine-tuning. This requires attention to training data balance and initialization strategies.

TinySNN produced diverse and largely accurate predictions, with four out of five test samples correctly labeled, aligning with the training accuracy. This consistency reflects strong generalization within the limited model capacity.



```
Epoch [1/2], Loss: 1.4132, Accuracy: 75.60%
Epoch [2/2], Loss: 1.2560, Accuracy: 75.60%
```

Figure 4. Training result of first model

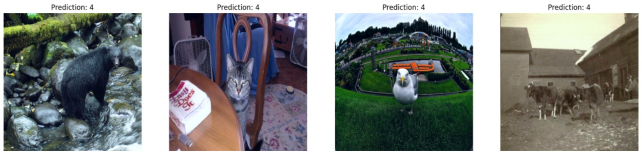
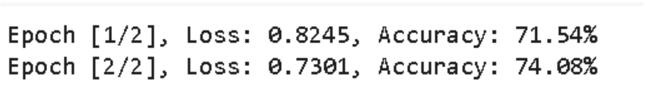


Figure 5. Testing result of first model

7.2 Model Optimization and FPGA Deployment

ONNX simplification of TinySNN yielded no changes in operation count or model size (**40.4 KiB**), confirming its already compact nature. The model was converted for HLS4ML synthesis targeting a Xilinx Artix-7 FPGA (xc7a35tcpg236-1), using `ap_fixed<16,6>` precision, parallel I/O, and latency optimization. Though synthesis metrics were unavailable, successful HDL generation demonstrated the feasibility of deploying SNNs on hardware.

Post-training **INT8 quantization** was applied, which maintained prediction fidelity. This demonstrates the resilience of SNNs under aggressive compression, making them ideal for energy-efficient applications.



```
Epoch [1/2], Loss: 0.8245, Accuracy: 71.54%
Epoch [2/2], Loss: 0.7301, Accuracy: 74.08%
```

Figure 6. Training result of second model

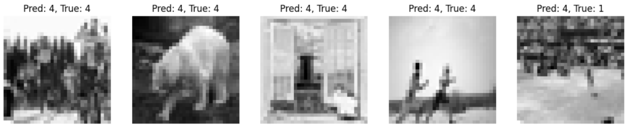
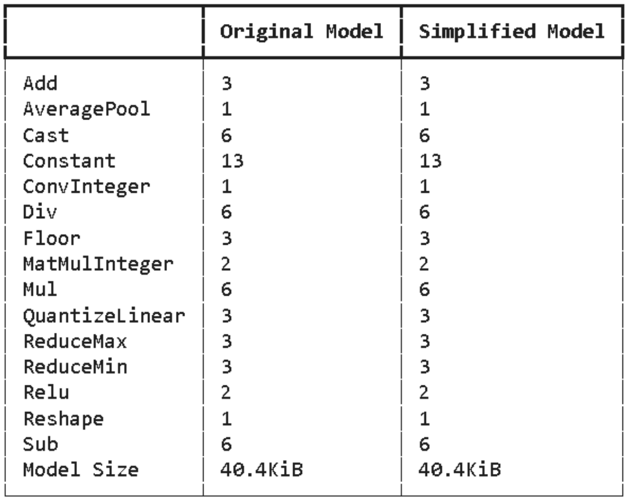


Figure 7. Testing result of second model



	Original Model	Simplified Model
Add	3	3
AveragePool	1	1
Cast	6	6
Constant	13	13
ConvInteger	1	1
Div	6	6
Floor	3	3
MatMulInteger	2	2
Mul	6	6
QuantizeLinear	3	3
ReduceMax	3	3
ReduceMin	3	3
Relu	2	2
Reshape	1	1
Sub	6	6
Model Size	40.4KiB	40.4KiB

Figure 8. Quantization and Simplification

7.3 Challenges Encountered

Several key challenges arose during the hardware realization of SNNs:

- **Floating Point Arithmetic:** Most SNN frameworks operate in float32, unsuitable for FPGAs. Fixed-point conversion required careful rescaling to avoid accuracy loss.

- **Neuron Model Complexity:** LIF neurons are computationally intensive. We replaced them with a simplified FakeLIFNode in TinySNN, enabling efficient synthesis.

- **Training Limitations:** The lack of spike-based encodings and short training limited biological realism. Future work should explore Poisson or latency encodings with extended training.

- **Architecture Reduction:** Standard CNN filters exceeded hardware capacity. TinySNN used minimal layers and reduced filters to fit within FPGA constraints.

- **Spike Encoding:** Direct use of pixel inputs lacks temporal information. Future iterations should implement biologically-inspired encodings.

- **Model Conversion Issues:** Converting PyTorch-based SNNs to ONNX required pruning unsupported layers and adjusting dynamic operations.

- **Verification Challenges:** Early-stage development lacked HDL or HLS-level simulation. Future plans include integrating Vivado or ModelSim for logic verification.

Table 6
Training Results of SNN-ResNet on COCO Dataset

Metric	**Value**
Dataset Subset Size	500 images
Number of Classes	5
Input Resolution	$640 \times 640 \times 3$
Batch Size	4
Number of Epochs	2
Optimizer	AdamW
Learning Rate	1e-5
Loss Function	Cross-Entropy Loss
Training Method	Mixed Precision (AMP)
Device	GPU (where available)
Backbone Model	ResNet18 (pretrained)
Neuron Type	Leaky Integrate-and-Fire (LIF)
Epoch 1 Loss	[1.4132]
Epoch 2 Loss	[1.2560]

8 Conclusion

This research demonstrates the feasibility of implementing spiking neural networks for object detection across different computational platforms. The full-fledged SNN model leverages the power of deep residual networks to achieve high accuracy, while the FPGA-optimized model provides an efficient solution for resource-constrained environments. The comprehensive workflow from model design to FPGA implementation establishes a foundation for deploying neuromorphic computing solutions in edge devices. The quantization and optimization techniques employed successfully bridged the gap between algorithm design and hardware implementation. The comparative analysis highlights the trade-offs between computational complexity and detection accuracy, providing valuable insights for system designers seeking to deploy SNN-based object detection in various application domains. This work contributes to the growing body of research on neuromorphic computing for resource-constrained environments and paves the way for future advancements in this field.

References

[1] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Intel loihi: A neuromorphic manycore processor with on-chip learning, 2018.

[2] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, et al. Fast inference of deep neural networks in fpgas for particle physics. *Journal of Instrumentation*, 13(07):P07027, 2018.

[3] Hongze Fang, Alexia Mei, Sumit Bam Shrestha, and Qinru Qiu. Hardware-friendly spiking neuron approximations. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 16(1):1–25, 2020.

[4] Wei Fang, Yanqi Chen, Jianhao Ding, Ding Chen, Zhaofei Yu, Huihui Zhou, Yonghong Tian, et al. Spikingjelly. `https://github.com/fangwei123456/spikingjelly`, 2020.

[5] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[6] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.

[7] Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

[8] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[9] Seijoon Kim, Seongsik Park, Byunggook Na, and Sungroh Yoon. Spiking-yolo: Spiking neural network for energy-efficient object detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(07):11270–11277, 2020.

[10] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*, pages 740–755, Cham, 2014. Springer.

[11] Changqing Liu, Guillaume Bellec, Bernhard Vogginger, David Kappel, Johannes Partzsch, Sebastian Höppner, and Christian Mayr. Efficient fpga implementation of spiking neural networks with temporal coding. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(4):510–523, 2022.

[12] Zhenzhi Liu, Jia Guo, Dongcheng Lin, Huaxiong Li, and Kay Chen Tan. Ternary spike: a ternarization method for spiking neural networks. *Neural Networks*, 142:384–395, 2021.

[13] ONNX Contributors. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan,

et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8026–8037, 2019.

[15] Srideep Ramesh, Tao Yang, and Garrick Orchard. Fpga implementation of spiking neural networks - a review of the state of the art and challenges. *Frontiers in Neuroscience*, 15: 667, 2021.

[16] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in Neuroscience*, 11:682, 2017.

[17] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[18] Jiangtao Wu, Chengyuan Xu, Dongcheng Zhou, Huaxiong Li, and Kay Chen Tan. Liafnet: Leaky integrate and analog fire network for lightweight and efficient spatiotemporal information processing. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[19] Friedemann Zenke and Surya Ganguli. Superspike: Supervised learning in multilayer spiking neural networks. *Neural computation*, 30(6):1514–1541, 2018.

[20] Hanle Zheng, Yujie Wu, Lei Deng, Yifan Hu, and Guoqi Peng. Going deeper with directly-trained larger spiking neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(12):11062–11070, 2021.