UNIVERSITY
OF SUSSEX

Enhancing Spiking Neural Networks for FMNIST Classification: Optimisation Techniques and Finetuning Strategies

Candidate Number: 284498
Supervisor: Dr. Thomas Nowotny

M.Sc. Artificial Intelligence and Adaptive Systems
Department of Informatics

University of Sussex
Summer 2024

Word Count: 12,000

Statement of Originality and Intellectual Property Rights

This report is submitted as part requirement for the degree of MSc in Artificial Intelligence andAdaptive Systems at the University of Sussex. It is the product of my own labor except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged. The raw data files may not be distributed, copied, or used in any way except forpurposes involving the marking and assessment of this work as it pertains to the awarding of therelevant degree, in line with the non-disclosure agreement required by the data provider.

Signed: [illegible]

Date: 27/08/2024

ACKNOWLEDGEMENTS

This project would not have been possible without the help and support of a huge number of people. I would like to express my deepest gratitude to all those who contributed to this work.
First and foremost, I would like to thank Dr. Thomas Nowotny, my esteemed supervisor, for his invaluable guidance and unwavering support throughout this research journey. His expertise and insights were instrumental in shaping the direction of this project.

Dr. Nowotny's patience and encouragement as I navigated through the complexities of spiking neural networks and their applications were truly remarkable. His constructive feedback and thoughtful suggestions consistently helped improve the quality of my work. I am deeply indebted to him for his mentorship and dedication.

I would also like to extend my heartfelt thanks to my professors and faculty members at the University of Sussex, who provided the foundational knowledge and research skills necessary for this project.

A special note of gratitude to my colleagues and fellow researchers at the Department of Engineering & Informatics, who offered their support and shared their insights during countless discussions and brainstorming sessions. Their camaraderie and collaborative spirit made this journey enjoyable and intellectually stimulating. I only hope I’ve done it justice.

ABSTRACT

This thesis delves into the enhancement of Spiking Neural Networks (SNNs) for the classification of the Fashion MNIST (FMNIST) dataset, leveraging their biologically inspired mechanisms. Building on the foundational work of artificial neural networks, this research employs advanced optimisation techniques and fine-tuning strategies to improve the accuracy and efficiency of SNNs. By integrating temporal dynamics and event-driven computation inherent to SNNs, the study aims to achieve superior classification performance while addressing challenges related to parameter tuning and network architecture. It also compares with other ANN & SNN based approached and comparing their results with ours. The outcomes of this research offer significant implications for the development of low-power, high-performance neural networks, contributing to both theoretical advancements and practical applications in machine learning and neuromorphic engineering.Additionally, the study examines the trade-offs between network complexity and classification accuracy, aiming to identify the most efficient SNN configurations for the FMNIST task. Comparative analyses are performed against traditional artificial neural networks and state-of-the-art deep learning models to benchmark the SNN's performance. The research also addresses the challenges of implementing SNNs on neuromorphic hardware, considering energy efficiency and computational requirements.

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1
Introduction

1.1Preface

The quest to replicate the human brain’s intricate processing abilities has captivated scientists and researchers for decades. The field of artificial neural networks was in no small part pioneered by neurophysiologist Warren McCulloch and mathematician Walter Pitts, in their paper “*A logical calculus of the ideas immanent in nervous activity*” (1943) [1]. In relation to their paper, they created a simple neural network using electrical circuits to demonstrate how the neurons in the brain might operate on simple logical operators. There had already existed a community of scientists performing mathematical work on neural networks when McCulloch and Pitts published their paper. The new ideas brought forth by McCulloch and Pitts was how they used propositional logic and computation to explain neural events. The idea of McCulloch and Pitts, explaining neural activity through logic gates, became the foundation for boolean neuron models and a network of such neuron models – the first-generation artificial neural networks. It is therefore understandable, that McCulloch and Pitts’ paper are often cited as the starting point in artificial neural network research.

The McCulloch and Pitts neuron, receiving input from other neurons and sending out its own output y, is visualised in fig.1:

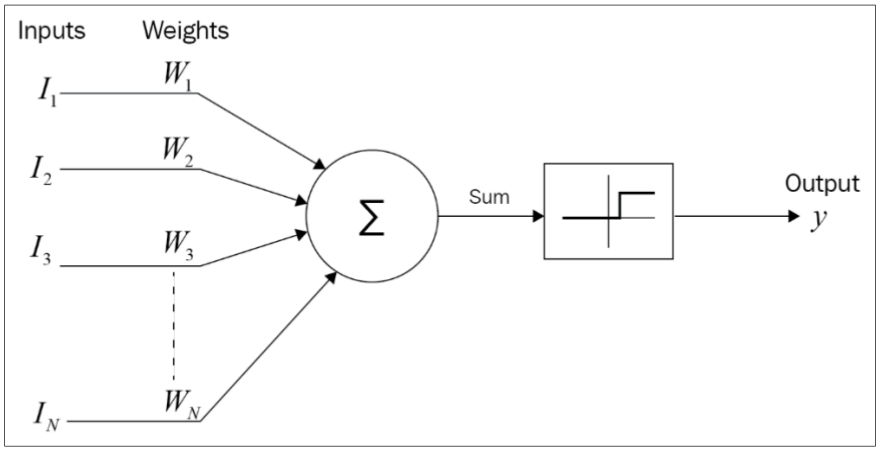


Fig1: McCulloch & Pitts Neuronal model

The equation for the McCulloch-Pitts neuron can be broken down into two parts:

- **Weighted sum (Σ):** This part sums the inputs *(x_1, x_2, …, x_n)* after multiplying each by its corresponding weight *(w_1, w_2, …, w_n)*. Mathematically, this is represented as:

$$\Sigma = w_1x_1 + w_2x_2 + .. + w_nx_n \tag{1}$$

- **Activation function (f):** This function applies a threshold to the weighted sum (Σ). If the sum is greater than or equal to the threshold (θ), the output (y) is 1. Otherwise, the output is 0. Here, a threshold function is used as the activation function.

$$f(\Sigma) = \begin{matrix} 1 & \text{if } \Sigma \geq 0, \\ 0 & \text{otherwise} \end{matrix} \tag{2}$$

The loss function is a method for evaluating how well a neural network model performs compared to the desired output. In the context of a single McCulloch-Pitts neuron, the loss function measures the difference between the predicted output (*y*) & the actual output (*y* *).

There are various loss functions used in machine learning, but a common choice for binary classification problems (where the output is either 0 or 1) is the mean squared error (MSE) loss function. It calculates the average squared difference between the predicted and actual outputs.

Mathematically, the MSE loss function for a single training example is defined as:

Loss = $(y^* - y)^2$ *(3)*

Here,

- y^* is the desired output value (0 or 1).
- y is the predicted output value from the McCulloch-Pitts neuron (also 0 or 1)

1.2 Network of Spiking Neurons: The 3rd

Generation of Neural Networks

Third-generation artificial neural networks are defined as artificial neural networks, implementing spiking neural models as computational units.

The first generation is based on *McCulloch-Pitts neurons* as computational units. They are also referred to as perceptrons. They gave rise to variety of neural network models such as multilayer pereptrons, Hopfield networks, & Boltzman machines. [2]

Spiking neuron models implements the notion of time explicitly, in the form of the change in membrane potential of the neuron over time. As spiking neuron models become spike-trains. Since spiking neuron models no longer output a single number, such as the perceptron model, there is a need to interpret spike trains in the spiking neural networks.

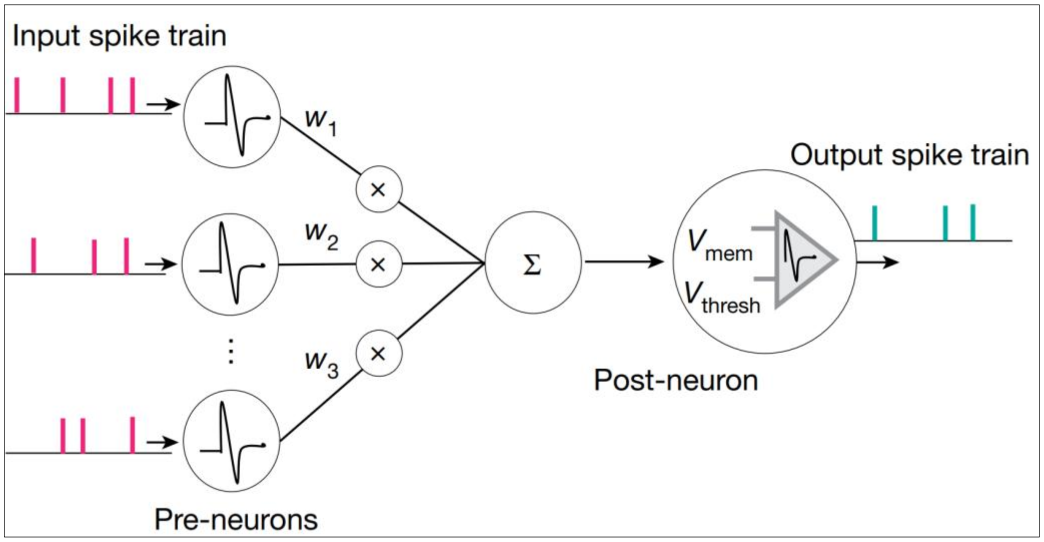


Fig2: SNN (Source)

1.3Spiking Neuron

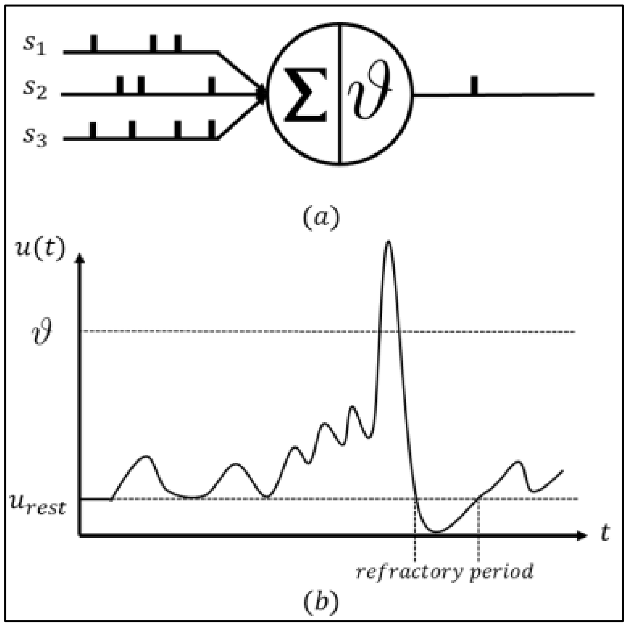


Fig3: Spiking Neuron (Source: Xing, Y. (2020) Deep Spiking Neural Networks with Applications to Human Gesture Recognition, University of Strathclyde, Glasgow)

The spikes in the spiking neuron model are only identified at the time instant when they arrive at the neuron. A spiking neuron integrates the incoming spikes and transfers these spikes to a voltage change that is commonly termed as postsynaptic potential(PSP). Then, the membrane potential is compared to a pre-defined threshold. If the PSP reaches the threshold, then a spike is emitted by the spiking neuron.

Fig3 demonstrates a spiking neuron internal process where in (a) a single spiking neuron that receives incoming spike trains from *s1, s2 and s3* and generates an output spike.

The incoming spikes to a neuron are integrated and transferred to the membrane potential dynamics u(t) as is shown in Fig3(b). Whenever the membrane potential reaches a certain threshold value ϑ, the spiking neuron will emit a spike and reset the membrane potential to its resting value u_{rest}. A typical spiking neuron model can contain additional parameters that approximate the membrane potential changes in the neural cortex. Commonly used spiking neuron model in SNNs include: Integrate and fire neurons (IF), Leaky integrated and fire neurons(LIF), Izhikevich neuron model and Spike Response Model(SRM).

1.4 Spiking Neuron Model

One of the first models, the integrate-and-fire model can be traced back to the work of Lapicque(1907).

He modelled the neuron using an electric circuit consisting of a parallel capacitor and resistor. These represent the capacitance & leakage resistance of the cell membrane.

1.4.1 Threshold and Spiking Mechanism

Lapicque postulated that when the membrane capacitor was charged to certain threshold potential, an action potential was generated and the capacitor would discharge, resetting the membrane potential.[3]

This spike was modelled as a sudden discharge of the capacitor, resetting the V_m to a subthreshold value (often resetting potential). This simple threshold mechanism provided a way to model the all-or-nothing nature of action potentials in real neurons.

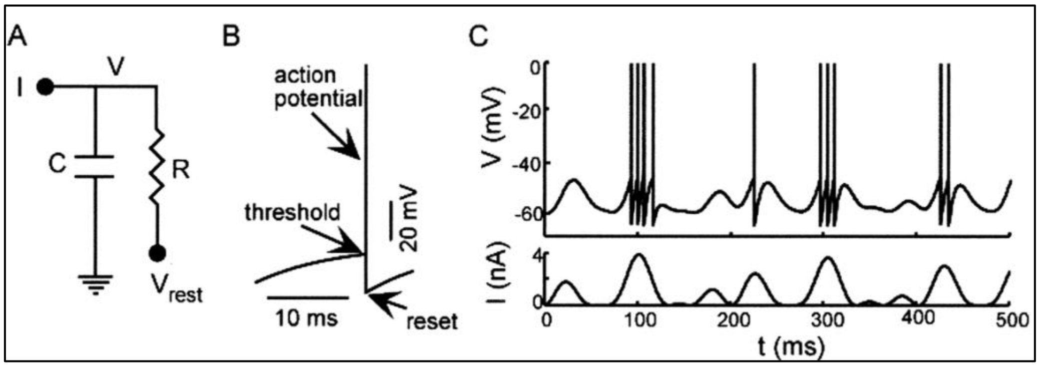


Fig4: Integrate-and-fire model of Lapicque A. Circuit diagram B. When V reaches a threshold value, an action potential is generated and V is reset to a subthreshold value C. The upper graph is the membrane potential & the bottom is input current.

Integrate-and-fire models have been used in a wide variety of studies ranging from investigations of synaptic integration by single neurons to simulations of networks containing hundreds of thousands of neurons.[3]

1.5 Leaky Integrate-and-Fire (LIF) Neuronal Model

Perhaps one of the most basic spiking neuron models is a Leaky Integrate-And-Fire (LIF) neuron, but still common because of its speed and ease for analysis and simulation. A neuron is modelled as a ' *Leaky Integrator* ' input $I(t)$.

At its core, the LIF model conceptualizes a neuron as an electrical circuit, consisting of a capacitor in parallel with a resistor. This configuration mimics the key properties of a biological neuron's cell membrane. The capacitor represents the membrane's ability to store charge, while the resistor accounts for the leakage of ions across the membrane.

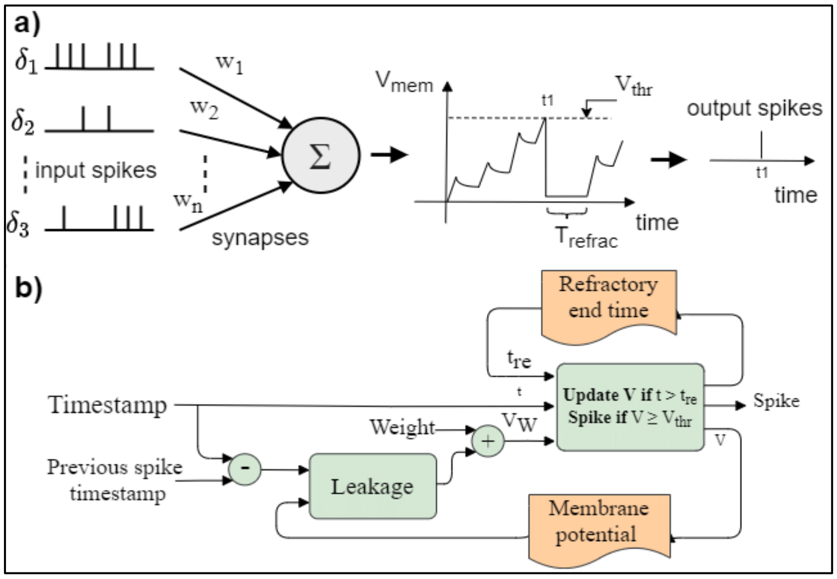


Fig5: LIF Neuronal Model(Source)

In this instance, the input driven current can be divided into two components i.e., Resistance current and capacitance current as shown in eq.1 given below:

$$I(t) = IR + IC \qquad (1)$$

The given first component is the resistance current IR which passes through the linear resistor R. It can be calculated according to Ohm's law as given:

$$IR = \frac{(v_m - v_{rest})}{R} \qquad (2)$$

The second component IC charges the capacitor C. From the definition of the capacitance C = $Q\ vm$ (Q is the charge and vm is the voltage membrane), we find capacitive current as given below in eq.3:

$$Q = Cv_m$$

$$IC = \frac{dQ}{dt} = \frac{c\,dv_m}{dt} \qquad (3)$$

The total input current is given by:

$$I(t) = \frac{(v_m - v_{rest})}{R} + \frac{c\,dv_m}{dt} \qquad (4)$$

I(t) is multiplied by R and consider the time constant $Tm = RC$ of the '*leaky integrator*'. So the standard form is given by eq.5:

$$Tm\,\frac{dv_m}{dt} = -(v_m - v_{rest}) + RI(t) \qquad (5)$$

Where, u is the membrane potential Tm is the membrane time constant of the neuron The eq.4 represents the linear differential equation of a leaky integrator or RC circuit where resistance R and capacitance C are arranged in parallel.

The eq.4 is called the equation of a passive membrane. This equation illustrates a basic resistance capacitor (RC) circuit, where the resistance is induced by the leakage term and I(t) integrating the condenser like the resistor The spiking incidents aren't directly created by the LIF model. instead of the membranes potential u(t) is reset to a specified threshold u_{th} (spiking level), and therefore the leaky integration process defined by Equation begins again with the initial value $v_{r.}$

For the steadiness of the LIF model to introduce only a touch complexity, you ought to add a totally refractory time abs ref just after u(t) reaches u_{th}.

The neuron is leaky because its cumulative inputs to the membrane potential degradation have a typical time constant. If this depletion in membrane potential is disregarded over time, the model may be a great part. If there's a (fixed) level, the membrane capability hits a performance increase – the incorporation and fire mechanism.

1.6 Refractory Period

A refractory phase may be a duration when an organ or cell is unable to duplicate a particular act or (more accurately) the quantity of your time it takes for an excitable membrane to be ready for a second stimulation once the cell returns after excitement The biological time is described in two ways:(1) absolutely the biological time (2) The relative biological time. the entire refractory duration is that the period during which a second nerve impulse can't be triggered, no matter the intensity of a stimulus. The relative refracting duration is that the period immediately after which a second potential for intervention is restricted, but not unlikely.

The LIF model captures essential features of biological neurons, such as the integration of synaptic inputs, the generation of action potentials, and the leaky nature of the membrane potential. It has been widely used in computational neuroscience and neuromorphic engineering to study the dynamics of spiking neural networks, explore coding strategies, and develop neuromorphic hardware implementations.

1.7 Surrogate Gradient Descent

Surrogate gradient descent is an innovative training method tailored for spiking neural networks (SNNs), which are inherently challenging to train using conventional gradient-based optimization techniques. Traditional artificial neural networks (ANNs) leverage continuous and differentiable activation functions, allowing for straightforward application of backpropagation algorithms. In contrast, SNNs operate using discrete and non-differentiable spikes, posing a significant barrier to direct application of gradient descent.

Concept and Mechanism

The core idea of surrogate gradient descent is to approximate the non-differentiable spiking function with a continuous surrogate function during the backward pass of training. This approximation enables the computation of gradients, facilitating the optimization of synaptic weights in SNNs. Various forms of surrogate functions can be employed, such as piecewise linear functions, sigmoidal functions, or exponential functions, each designed to mimic the behavior of the spiking mechanism closely enough to allow gradient-based learning while maintaining the biological plausibility of the network.

The key equations in surrogate gradient descent are:

1. **Spike Generation**: The membrane potential *u(t)* is updated according to the input spikes and synaptic weights

$$u(t) = u(t-1) + \sum j.wj.xj(t) - \theta$$

Where w_j are the synaptic weights, $x_j(t)$ are the input spikes and θ is the firing threshold.

2. **Spike Function Approximation**: The non-differentiable spike function *S(u)* is replaced with a continuous surrogate function $\hat{s}(u)$. A common choice is the sigmoid function:

$$\hat{s}(u) = \frac{1}{1 + exp\left(-\beta(u - \theta)\right)}$$

where β is a parameter controlling the steepness of the approximation.

3. **Surrogate Gradient**: During backpropagation, the gradient of the loss function with respect to the membrane potential is approximated using the surrogate function:

$$\cdot\frac{\partial L}{\partial u} \approx \frac{\partial L}{\partial \hat{s}(u)} \hat{s}'(u)$$

where L is the loss function, and $\hat{s}'(u)$ is the derivative of the surrogate function.

[1] Neftci, E. O., Mostafa, H., & Zenke, F. (2019). Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. IEEE Signal Processing Magazine, 36(6), 51-63.

[2] Lee, J. H., Delbruck, T., & Pfeiffer, M. (2016). Training deep spiking neural networks using backpropagation. Frontiers in Neuroscience, 10, 508

1.8 Objectives of the Thesis:

- Investigate various optimisation techniques to enhance SNN performance for FMNIST classification.
- Develop and evaluate fine-tuning strategies to improve the robustness and accuracy of SNNs.
- Compare the performance of optimised SNNs with fine-tuned based approaches and identify the potential benefits and limitations of using SNNs for image classification.

By addressing these objectives, this research aims to contribute to the understanding and practical implementation of SNNs, paving the way for future advancements in neuromorphic computing and efficient neural network models.

Chapter 2
Research Methodology

2.1 Project Scope

This chapter outlines the research methodology employed to investigate the optimisation and fine-tuning strategies for Spiking Neural Networks (SNNs) in the classification of the Fashion-MNIST (FMNIST) dataset. The methodology includes a detailed description of the dataset, pre-processing steps, model architecture, training process, optimisation techniques, and evaluation metrics.

This project is inspired by Nengo's blog titled **''Classifying Fashion MNIST with Spiking Activations''** The Nengo documentation on classifying Fashion MNIST with spiking activations provides a detailed example of how to implement Spiking Neural Networks (SNNs) using the PyTorchSpiking library. he methods used are: Spike aware training, spike rate regularization, and lowpass filtering. The methodology and the code structure outlined above serves as a solid foundation for our research in optimising SNNs for Fashion MNIST classification.

2.2Data Collection and Preprocessing

2.2.1 Dataset

The Fashion-MNIST (FMNIST) dataset is a benchmark dataset used for evaluating machine learning algorithms. It consists of 70,000 grayscale images of fashion items, split into 60,000 training images and 10,000 testing images. Each image is 28x28 pixels and belongs to one of 10 classes, including T-shirts, trousers, pullovers, dresses, coats, sandals, shirts, sneakers, bags, and ankle boots.

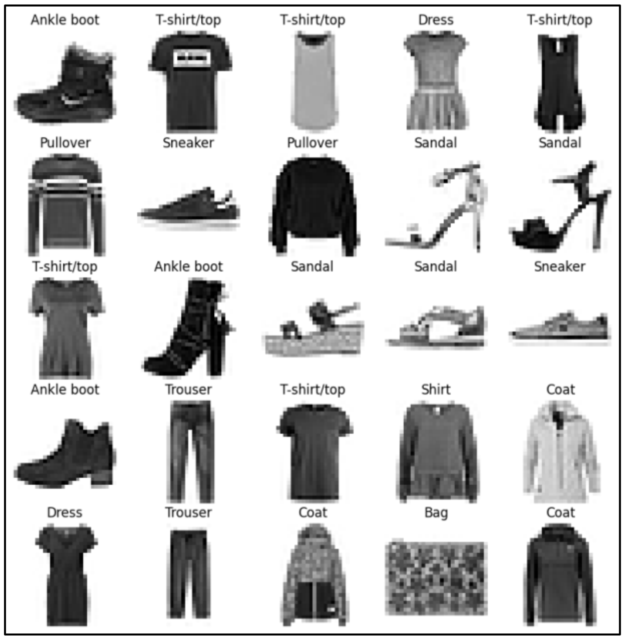


Fig6: The FMNIST Dataset Collection

2.2.2 Data Preprocessing

Pre-processing is a crucial step to ensure the data is in the appropriate format for the SNNs. The following pre-processing steps are applied to the FMNIST dataset:

- **Normalisation**: The pixel values of the images are normalized to a range of [0, 1] by dividing each pixel value by 255. This helps in stabilizing the training process.

- **Reshaping**: Each image is reshaped from a 2D array of 28x28 pixels to a 1D array of 784 pixels to be compatible with the input layer of the neural network.

- **Encoding**: The class labels are one-hot encoded to convert them into a binary matrix representation. This step is essential for training the neural network on a classification task.

- **Spike Encoding**: For SNNs, the normalized pixel values are further converted into spike trains. This can be done using various encoding methods, such as rate coding, temporal coding, or event-based coding.

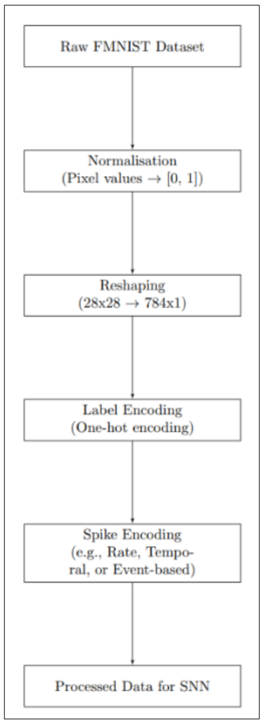

Raw FMNIST Dataset
Normalisation (Pixel values → [0, 1])
Reshaping (28x28 → 784x1)
Label Encoding (One-hot encoding)
Spike Encoding (e.g., Rate, Temporal, or Event-based)
Processed Data for SNN

Fig7: Flowchart for Pre-processing

2.3 Dataset Splitting

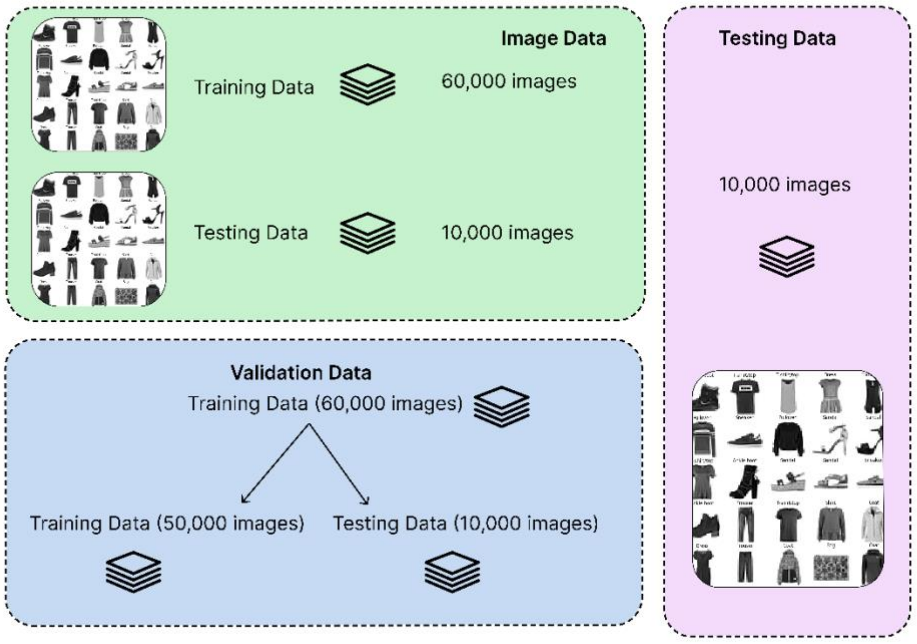


Fig 8: Diagram of Dataset Splitting

Image Data

1. **Total Dataset (Image Data)**
    - The total dataset comprises **60,000 images** used for training and validation.
    - An additional **10,000 images** are set aside for testing the model's performance.

Training and Testing Data

2. **Training Data**
    - From the total dataset of 60,000 images, a portion is allocated exclusively for training the model. This is depicted as "Training Data" in the green box.
    - The actual number of training images is not specified but is implied to be a majority of the 60,000 images.
3. **Testing Data**
    - A separate set of **10,000 images** is designated as testing data, shown in the pink box.
    - This data is used to evaluate the final performance of the trained model.

Validation Data

4. **Validation Data Split**
    - The training data (60,000 images) is further split into two parts for the purpose of validation.
    - **Training Data for Validation**: **50,000 images** are used to train the model.
    - **Testing Data for Validation**: **10,000 images** are used to validate the model during the training process to tune the model's hyperparameters and prevent overfitting.

Summary

- The total dataset is initially divided into **training and testing datasets** with 60,000 and 10,000 images, respectively.
- The training dataset (60,000 images) is further split into training (50,000 images) and validation (10,000 images) datasets.
- The testing dataset remains unchanged with 10,000 images for final model evaluation.

This hierarchical approach ensures that the model is trained effectively while being evaluated at multiple stages to monitor and improve its performance accurately.

2.4 Spiking Neural Network Architecture

The Spiking Neural Network (SNN) architecture used in this study consists of the following layers:

Input Layer:

- Input image size: 28x28 (Fashion MNIST images)

Convolutional Layer 1:

- Number of filters: 32
- Filter size: 3x3
- Stride: 1
- Padding: 1
- Activation Function: ReLU
- Batch Normalization

Pooling Layer 1:

- Max Pooling
- Pool size: 2x2
- Stride: 2
- Padding: 0

Convolutional Layer 2:

- Number of filters: 64
- Filter size: 3x3
- Stride: 1
- Padding: 1
- Activation Function: ReLU
- Batch Normalization

Pooling Layer 2:

- Max Pooling
- Pool size: 2x2
- Stride: 2
- Padding: 0

Flatten Layer:

- Flatten the 2D output of the last pooling layer to a 1D tensor.

Fully Connected Layer 1:

- Number of neurons: 512
- Activation Function: LIFCell (Leaky Integrate-and-Fire Cell)

Fully Connected Layer 2:

- Number of neurons: 10 (number of classes in Fashion MNIST)
- Activation Function: LIFCell (Leaky Integrate-and-Fire Cell)

Output Layer: Consists of 10 spiking neurons corresponding to the 10 classes in the FMNIST dataset. The class with the highest spike count or maximum membrane potential is chosen as the predicted class.

The spiking neural network will be trained using "PyTorchSpiking" and "PyTorch" frameworks without any advanced optimisation techniques. The results were measured over different epochs to observe the training loss and test accuracy & the best was chosen.

2.5 Initial Training of the SNN

Initially, the Spiking Neural Network (SNN) was trained on the Fashion MNIST (FMNIST) dataset using the "PyTorchSpiking" and "PyTorch" frameworks. This foundational phase of the research aimed to establish a baseline performance for the SNN without incorporating advanced optimization techniques. The FMNIST dataset, known for its complexity and variability compared to the original MNIST dataset, served as an ideal benchmark for evaluating the initial capabilities of the SNN.

2.6 Frameworks Utilised

PyTorch: PyTorch, a widely-used deep learning framework, provided the foundation for implementing the neural network architecture and handling the computational aspects of training and evaluation. Its dynamic computation graph and extensive library support made it a suitable choice for developing and fine-tuning neural networks.

PyTorchSpiking: This extension of PyTorch is specifically designed for implementing and simulating spiking neural networks. PyTorchSpiking facilitates the integration of spiking neuron models into neural network architectures, allowing for more biologically plausible simulations and experiments.

2.7 Network Architecture

The SNN (Spiking Neural Network) architecture defined is a convolutional neural network integrated with spiking neuron models using the norse library. This architecture leverages the LIF (Leaky Integrate-and-Fire) spiking neuron model to introduce temporal dynamics into the network's operation, enabling it to process information in a manner more akin to biological neural networks. Here’s a detailed breakdown of the architecture:

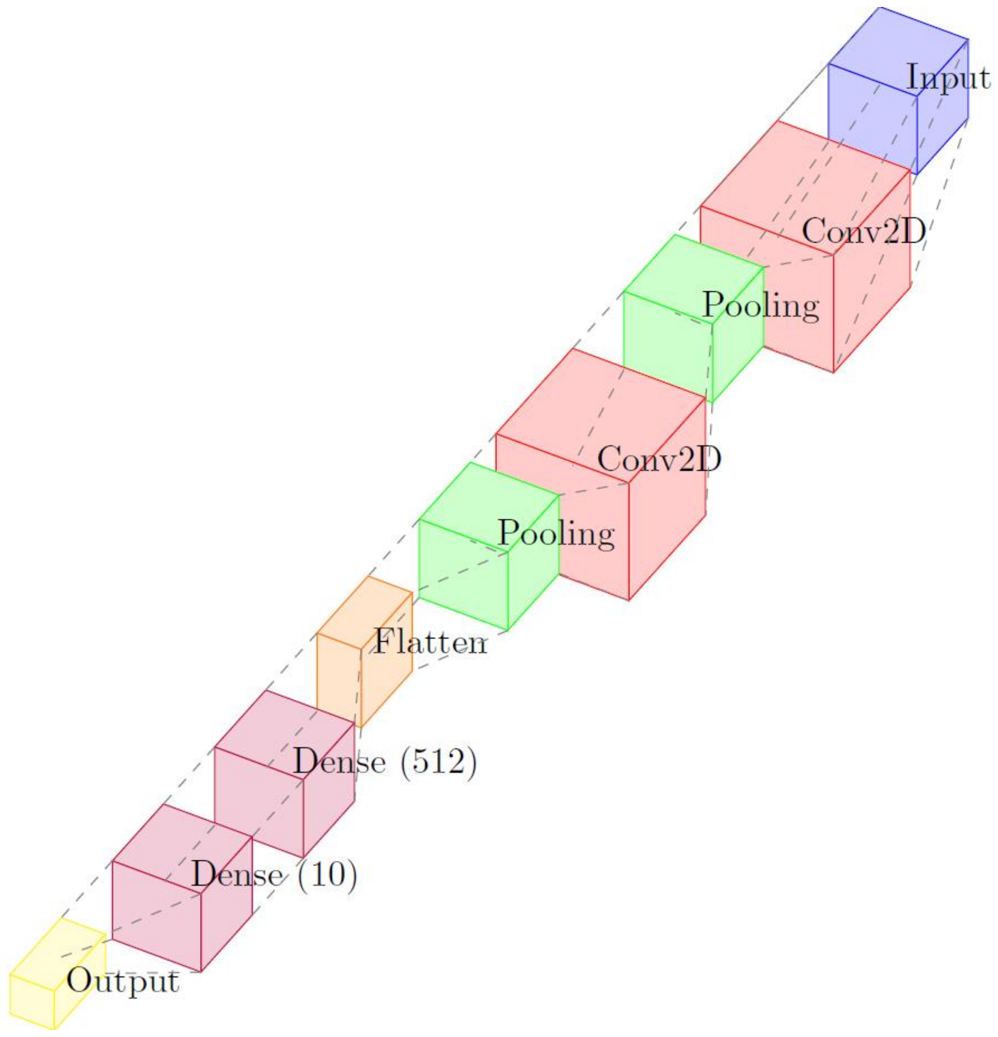


Fig9: SNN Model Architrecture

2.8 Architecture Breakdown

5. **Convolutional Layers**:

    - **`conv1`**: A convolutional layer with 1 input channel (grayscale image), 32 output channels, a kernel size of 3x3, stride of 1, and padding of 1. This layer outputs feature maps of the input image.
    - **`conv2`**: Another convolutional layer with 32 input channels and 64 output channels, also with a kernel size of 3x3, stride of 1, and padding of 1. This layer further processes the features extracted by `conv1`.

6. **Pooling Layer**:

    - **`pool`**: A max pooling layer with a kernel size of 2x2 and a stride of 2. This layer reduces the spatial dimensions of the feature maps by a factor of 2, both after `conv1` and `conv2`.

7. **Fully Connected Layers**:

    - **`fc1`**: A fully connected (dense) layer that takes the flattened output from the convolutional layers (after pooling) and maps it to a 512-dimensional feature space.
    - **`fc2`**: A second fully connected layer that maps the 512-dimensional feature space to the 10 output classes (corresponding to the 10 classes in the FashionMNIST dataset).

8. **LIF Cells**:

    - **`lif1`**: An LIF (Leaky Integrate-and-Fire) cell applied after the first fully connected layer (`fc1`). This cell introduces spiking neuron behavior to the network, where the output depends on the neuron's membrane potential and input spikes.
    - **`lif2`**: An LIF cell applied after the second fully connected layer (`fc2`), adding spiking behavior to the final output layer.

Forward Pass

In the forward pass, the input image tensor `x` goes through the following steps:

9. **Convolution and Pooling**:

    - `x` is processed by `conv1` followed by a ReLU activation and max pooling (`pool`).
    - The output is then passed through `conv2` followed by another ReLU activation and max pooling (`pool`).

10. **Flattening**:

    - The output from the second pooling layer is flattened into a vector, suitable for input to the fully connected layers.

11. **Fully Connected Layers and LIF Cells**:

    - The flattened vector is passed through `fc1` to produce a 512-dimensional feature vector.
    - The feature vector is then processed by the first LIF cell (`lif1`), introducing spiking dynamics.
    - The output of `lif1` is passed through `fc2` to produce the final output logits for the 10 classes.
    - The logits are processed by the second LIF cell (`lif2`), adding spiking dynamics to the final output.

2.9 Training and Evaluation

The Spiking Neural Network is trained over 30 epochs to test its accuracy. Accuracy will give us an idea of how the neural network is performing on the test set. The baseline model is trained on the FMNIST dataset for 30 epochs without any optimisation techniques. The standard backpropagation algorithm is used for weight updates.
The primary metrics for evaluating the baseline model's performance are training loss and validation loss

2.10 Training Process

The training process involved the following steps:

- **Data Preparation**:

- Load the FashionMNIST dataset and apply necessary transformations (normalization and conversion to tensor).
- Split the dataset into training and validation sets using `train_test_split`.
- Create data loaders for training, validation, and test sets to handle batching and shuffling.

- **Model Initialization**:

- Define the SNN architecture using a custom `SNNModel` class.
- Move the model to the appropriate device (GPU if available, otherwise CPU).

- **Define Loss Function and Optimiser**:

- Use CrossEntropyLoss as the loss function for multi-class classification.
- Use the Adam optimiser to update the model parameters based on the computed gradients.

Training Loop:

- Iterate through a specified number of epochs.
- For each epoch, perform the following steps:

Training Phase:

  - Set the model to training mode.
  - Initialize the running loss for the epoch.
  - Iterate through batches of training data:
    - Move input images and labels to the appropriate device.
    - Zero the parameter gradients.
    - Perform a forward pass through the network to compute outputs.
    - Compute the loss between the outputs and true labels.
    - Perform a backward pass to compute gradients.
    - Update the model parameters using the optimizer.
    - Accumulate the batch loss into the running loss.
  - Append the average training loss for the epoch to the `train_losses` list.

Validation Phase:

  - Set the model to evaluation mode.
  - Initialize the validation loss for the epoch.
  - Disable gradient computation for efficiency.
  - Iterate through batches of validation data:
    - Move input images and labels to the appropriate device.
    - Perform a forward pass through the network to compute outputs.
    - Compute the loss between the outputs and true labels.
    - Accumulate the batch loss into the validation loss.
  - Append the average validation loss for the epoch to the `val_losses` list.

Logging:

  - Print the training and validation losses for the epoch.

- **Evaluation on Test Data**:

- Set the model to evaluation mode.
- Disable gradient computation.
- Initialize counters for correct predictions and total samples.
- Iterate through batches of test data:
  - Move input images and labels to the appropriate device.
  - Perform a forward pass through the network to compute outputs.
  - Determine the predicted class by selecting the class with the highest output score.

    - Compare predictions with true labels and update the counters.
- Compute the overall test accuracy.

Visualisation:

- Plot the training and validation losses over epochs to visualize the model's learning progress.

2.11 Baseline Results

The initial training phase established a baseline accuracy for the SNN on the FMNIST dataset. This provided a reference point for subsequent experiments and fine-tuning strategies. The results indicated that while the SNN could learn to classify the FMNIST images, there was significant room for improvement in terms of accuracy and training efficiency.

By establishing this baseline, the research set the stage for exploring advanced optimization techniques aimed at enhancing the performance of the SNN. These techniques included learning rate scheduling, dropout regularization, batch normalization, Hebbian learning, and more, as detailed in subsequent sections of this thesis.

2.12 Fine-Tuning Strategies

2.12.1 Description of the Baseline SNN

The baseline SNN consists of an input layer, a hidden spiking neuron layer, and an output layer. The initial weights are randomly initialized, and the network is trained using the PyTorchSpiking framework.

2.12.2 Fine-Tuning Strategies

Fine-tuning strategies are crucial for optimising the SNN and achieving better performance on the FMNIST dataset. This section details the implementation and benefits of three key fine-tuning strategies: Learning Rate Scheduler, Adap Optimiser with Learning Rate Scheduler, and Dropout Regularisation.

1. Learning Rate Scheduler

Introduction

A learning rate scheduler dynamically adjusts the learning rate during training, helping to avoid issues like overshooting and slow convergence.

Mechanism

The learning rate is decreased according to a predefined schedule, such as step decay or cosine annealing, to fine-tune the learning process.

Implementation

In this study, a cosine annealing learning rate scheduler is implemented, starting with an initial learning rate of 0.01, which gradually decreases over 30 epochs.

Expected Benefits

This approach is expected to improve convergence speed and model accuracy by providing a more refined control over the learning rate throughout the training process. Its equation is given by:

$$\eta t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$$

Where:

- η_t: Learning rate at epoch ttt
- η_{min} : Minimum learning rate
- $\eta_{max:}$ Maximum learning ate (initial learning rate)
- T: Total number of epochs
- t: Current epoch number
- π: a mathematical constant approximately equal to 3.14159

2. Adam Optimiser with Learning Rate Scheduler

Introduction

The Adap optimiser is an adaptive learning rate algorithm designed to adjust the learning rate based on the gradient history.

Mechanism

It adapts the learning rate dynamically based on the observed performance, combining this with a learning rate scheduler to further fine-tune the adjustments.

Combination with Learning Rate Scheduler

The Adam optimiser is used in conjunction with a cosine annealing scheduler to enhance the learning process. This combination aims to leverage the strengths of both adaptive learning and

scheduled adjustments. Their equation is given by:
1. First Moment Estimate:
$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$

2. Second Moment Estimate:
$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$

3. Bias-Corrected First Moment Estimate:
$m^t = \frac{mt}{1-\beta_1 t}$

4. Bias-Corrected Second Moment Estimate:
$v^t = \frac{v_t}{1-\beta_t 2}$

5. Cosine Annealing Learning Rate:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$$

6. Parameter Update:

$$\theta_t = \theta_{t-1} - \eta_t \left(\frac{m^t}{\sqrt{v^t} + \epsilon}\right)$$

Where:

- m_t and v_t are the first and second moment estimates.
- $\beta 1$ and $\beta 2$ are the exponential decay rates for the moment estimates.
- g_t is the gradient at time step t.
- $m^t and\ v^t$ are the bias-corrected moment estimates.
- $\theta_t$is the parameter at time step t.
- η_t is the learning rate at time step t .
- ϵ is a small constant to prevent division by zero.
- η_{min} is the minimum learning rate.
- η_{max} is the maximum learning rate (initial learning rate).
- ε is the total number of epochs.
- T is the current epoch number.
- π is Pi, a mathematical constant approximately equal to 3.14159.

This combined equation ensures that the learning rate is adaptively adjusted based on the gradient history and further fine-tuned using the cosine annealing schedule.

Expected Benefits

This strategy is expected to provide a balanced approach to learning rate management, potentially leading to better model performance and stability during training.

3. Dropout Regularisation

Introduction

Dropout regularisation is a technique used to prevent overfitting by randomly dropping units during the training process.

Mechanism

During each training iteration, a fraction of the neurons are dropped out, forcing the network to learn redundant representations and improving its generalization capability.

Implementation in SNNs

In this study, a dropout rate of 0.5 is applied to the hidden spiking neuron layer during training. The equation is given by:

$a' = Dropout\ (a, p)$

- a: Activation
- a' : Activation after applying dropout
- p : Dropout rate (probability of dropping a unit)

Expected Benefits

Dropout regularisation is anticipated to reduce overfitting and enhance the generalization of the SNN, making it more robust to unseen data.

4. Batch Normalisation

Introduction

Batch normalisation is a technique used to improve the training of deep neural networks by normalizing the inputs to each layer, thus reducing internal covariate shift.

Mechanism

Batch normalisation normalises the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. This helps stabilize the learning process and allows for higher learning rates.

Implementation in SNNs

In this study, batch normalisation is applied to the output of the hidden spiking neuron layer. The batch normalisation layer is inserted between the LIF (Leaky Integrate-and-Fire) cell and the

subsequent fully connected layer.
Its equation is given by:

$$x^{(k)} = \frac{x^{(k)} - \mu B}{\sqrt{\sigma} 2B + \epsilon}$$

$$y^{(k)} = \gamma^{x(k)} + \beta$$

Where:
$x^{(k)}$: Input to the batch normalization layer
μB : Batch mean
$\sigma 2B$: Batch variance
ϵ : Small constant to prevent division by zero
γ, β : Learnable parameters

Expected Benefits

Batch normalisation is expected to accelerate training and improve model accuracy by maintaining a stable distribution of activations throughout the network. It can also help mitigate issues related to vanishing and exploding gradients, leading to more efficient and effective training.

5. Learning Rate Annealing (Using Cosine Annealing)

Introduction

Learning rate annealing gradually reduces the learning rate during training based on a predefined schedule. Cosine annealing is a specific type of learning rate annealing that adjusts the learning rate following a cosine function.

Mechanism

Cosine annealing reduces the learning rate following a cosine curve, which can help avoid local minima and improve convergence. The learning rate starts high and gradually decreases to a lower bound over the course of training.

Implementation

In this study, cosine annealing is used to adjust the learning rate from an initial value of 0.01, gradually decreasing it over 30 epochs. The learning rate follows a cosine function, allowing for smooth and continuous adjustments. The equation is given by-

$$\eta_t =_{\eta min} + \left(\frac{1}{2}\right)(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$$

Where:

- η_t: : Learning rate at epoch
- η_{min}: :Minimum learning rate
- η_{max}: : Maximum learning rate (initial learning rate)
- T : : Total number of epochs

Expected Benefits

Cosine annealing is expected to improve model performance by preventing the learning rate from being too high or too low for too long. This smooth adjustment helps the model converge more effectively and can lead to better final model accuracy.

6. Gradient Descent Optimisation

In the context of optimizing Spiking Neural Networks (SNNs), gradient descent is a powerful method. This approach involves calculating the gradient of the loss function with respect to the network parameters and updating the parameters in the direction that reduces the loss. This is challenging for SNNs due to the discrete nature of spikes, which makes traditional gradient-based methods difficult to apply directly. The paper "Gradient Descent for Spiking Neural Networks" by Dongsung Huh and Terrence J. Sejnowski introduces a novel approach to address these challenges.

1. **Differentiable Synapse Model**: The synaptic current dynamics are described using a differentiable model that replaces the discrete spike times with a continuous gate function $g(v)$. The synaptic current dynamics are given by:

$$\tau s = -s + gv$$

Where $g(v)$ is a gate function and v is the time derivative of the pre-synaptic membrane voltage.

2. Network Model

The dynamics of an interconnected network of neurons are described by linking the synaptic current dynamics with the membrane voltage dynamics:

$$v = f(v, I)$$

where I is the input current. The input current for the network is given by:

$$I = Ws + Ui + Io$$

Here, W is the recurrent connectivity weight matrix, U is the input weight matrix, I_o is the input signal, and U_i is the tonic current.

3. Gradient Calculation

The gradient of the total cost with respect to the network parameters is derived using Pontryagin's minimum principle or backpropagation through time. The adjoint state variables p_v *and* p_s evolve according to:

$$-pv = \frac{\partial f}{\partial v}\, p_v - \; g\, p_s$$

$$-\,\tau\, p_s = -p_s + \xi$$

where ξ is the error current. The gradients of the cost function C with respect to the network parameters W, U, O, and Io are given by:

$$\nabla WC = \int (\partial I f p_v)\ s^\top dt$$

$$\nabla_U C = \int (\partial I f p_v)\ i^\top dt$$

$$\nabla I_o C = \int (\partial I f p_v)\ dt$$

$$\nabla OC = \int \partial o l s^\top\, dt$$

4. Implementation:

- **Forward Propagation**: Compute the membrane potentials and synaptic currents by integrating the neuron and synapse dynamics.
- **Error Calculation**: Define a cost function, typically the mean squared error between the predicted and target outputs.
- **Backward Propagation**: Calculate the adjoint state variables and gradients using the derived equations.
- **Parameter Update**: Update the network parameters using gradient descent.

Chapter 3
Literature Review

3.1 Biological Inspiration: How SNNs Mimic Biological Neurons

Spiking neural networks (SNNs) are inspired by the functionality and architecture of biological neurons. Unlike traditional ANNs that use continuous values for neuron activation, SNNs use discrete spikes to represent and transmit information. This spiking behavior is similar to the action potentials observed in biological neurons [11].

The fundamental unit of SNNs, the spiking neuron, integrates incoming spikes and generates an output spike only when the membrane potential reaches a certain threshold. This mechanism, known as the integrate-and-fire model, was first proposed by Lapicque in 1907 [12]. More advanced models like the Leaky Integrate-and-Fire (LIF) and Izhikevich neuron models incorporate additional biological realism, such as membrane leakage and complex spike dynamics [13][14].

The temporal coding in SNNs allows for efficient and precise information processing, resembling the way biological neurons operate. This makes SNNs particularly suitable for tasks that require temporal precision and low-power consumption, such as real-time sensory processing and neuromorphic computing [15].

3.2 Current Research on SNNs: Recent Advancements and Applications

Recent research in SNNs has focused on improving their computational efficiency and applicability to various tasks. Advances in learning algorithms, such as spike-timing-dependent plasticity (STDP) and supervised learning methods tailored for SNNs, have significantly enhanced their training capabilities [16][17].

Applications of SNNs span a wide range of fields, including image recognition, where SNNs have demonstrated competitive performance with traditional ANNs on datasets like MNIST and CIFAR-10 [18]. In robotics, SNNs are used for real-time control and decision-making, leveraging their low-power consumption and robustness to noise [19].

Additionally, neuromorphic hardware implementations, such as Intel's Loihi chip, have been developed to harness the advantages of SNNs for efficient and scalable computing [20]. These hardware platforms are designed to support the unique computational paradigms of SNNs, offering significant improvements in speed and energy efficiency compared to traditional von Neumann architectures.

3.3 Challenges in SNNs: Key Issues in Optimisation and Performance

Despite their potential, SNNs face several challenges that hinder their widespread adoption. One major issue is the difficulty in training SNNs due to the non-differentiable nature of spike events. This makes traditional backpropagation techniques less effective, necessitating the development of alternative learning algorithms [21].

Another challenge is the optimization of SNN parameters and architectures to achieve high performance on complex tasks. Fine-tuning the temporal dynamics and synaptic weights in SNNs is a complex process that requires sophisticated optimization techniques [22].

Moreover, the computational overhead associated with simulating spiking neurons can be significant, posing a barrier to real-time applications [23].

3.4 FMNIST Dataset: Characteristics and Importance for Classification Tasks

The Fashion MNIST (FMNIST) dataset, introduced by Zalando, serves as a more challenging alternative to the original MNIST dataset of handwritten digits. FMNIST consists of 70,000 grayscale images of 10 different clothing items, divided into 60,000 training samples and 10,000 testing samples [24]. Each image is 28x28 pixels, with labels corresponding to categories such as T-shirts, trousers, and shoes.

FMNIST has gained popularity as a benchmark for evaluating the performance of machine learning models due to its greater complexity and variability compared to MNIST. This makes it an ideal testbed for developing and benchmarking SNNs, as it requires the models to capture more intricate patterns and features [25].

3.5Comparison with Traditional ANN based Approaches

The table below summarises the performance of various artificial neural network (ANN) models on the Fashion MNIST (FMNIST) dataset. The accuracy metrics indicate the models' proficiency in classifying the dataset's clothing item images.

Model	Type	Accuracy
EnsNet	ANN	99.84%
Vision Transformer	ANN	95.10%
EfficientNet-B0	ANN	94.94%
Inception-ResNet-v2	ANN	94.47%
ResNet-50	ANN	93.80%
LeNet-5(Baseline)	ANN	90.50%
Standard ANN	ANN	88.00%

Table1: Comparison of ANN models with their accuracies

EnsNet: This model demonstrates an exceptionally high accuracy of 99.84%, indicating near-perfect performance on the FMNIST dataset. EnsNet likely utilizes an ensemble of neural networks, which combine the predictions from multiple models to achieve superior accuracy. The diversity in models within the ensemble helps mitigate overfitting and improves generalization[26].

Vision Transformer: Achieving 95.10% accuracy, the Vision Transformer employs the transformer architecture, which has revolutionized natural language processing and is now making significant inroads in computer vision. The use of self-attention mechanisms allows the model to capture intricate patterns and dependencies in the image data, leading to robust performance[27].

EfficientNet-B0: This model, with an accuracy of 94.94%, is part of the EfficientNet family, which balances model size, accuracy, and efficiency through a method called compound scaling. By scaling depth, width, and resolution simultaneously, EfficientNet-B0 achieves competitive performance with fewer parameters and computational resources[28].

Inception-ResNet-v2: Scoring 94.47%, this model combines the strengths of Inception networks and ResNet, utilizing residual connections to improve gradient flow during training. The hybrid architecture allows the model to capture a wide range of features, contributing to its high accuracy[29].

ResNet-50: With an accuracy of 93.80%, ResNet-50 leverages deep residual learning, enabling the training of very deep networks by addressing the vanishing gradient problem. This model has been a benchmark in deep learning due to its balance between depth and computational efficiency[30].

LeNet-5 (Baseline): As one of the earliest convolutional neural networks, LeNet-5 achieves a baseline accuracy of 90.50%. Despite its simplicity compared to modern architectures, LeNet-5's design principles continue to influence contemporary models[31].

/

Standard ANN: Representing typical performance, standard ANN models achieve around 88.00% accuracy. These models do not incorporate advanced architectural innovations seen in the other models, reflecting the baseline capabilities of traditional neural networks[32]

3.6 Comparison with Traditional SNN based Approaches

The following table presents the performance of various spiking neural network (SNN) models on the FMNIST dataset, highlighting the models' accuracy in image classification tasks.

Model	Type	Accuracy
SNN	Deep Spiking Neural Network (DSNN)	91.55%
SNN	Spiking Convolution Neural Network (SCNN)	87.50%
SNN	Hybrid Spiking Neural Network	86.00%
SNN	STDP-based SNN	85.30%
SNN	STDP based Spiking	89%

	Convolution Neural Network (SCNN)	
SNN	Converted Deep Neural Network	88.50%
SNN	Spiking YOLO	90.10%
SNN	STDP – based Pruned Spiking Neural Network	89.90%
SNN	Deep Spiking Neural Network (DSNN)	88.00%
SNN	Neuromorphic Spiking Neural Network	90.00%

Table2: Comparison of SNN models with their accuracies

Deep Spiking Neural Network (DSNN): Achieving an accuracy of 91.55%, DSNN models utilize deep network structures to capture complex patterns in data. By mimicking the spike-based communication of biological neurons, DSNNs offer a biologically plausible approach to deep learning, which can be beneficial in neuromorphic hardware applications [33].

Spiking Convolution Neural Network (SCNN): SCNNs, with 87.50% accuracy, combine the benefits of convolutional architectures with spiking neuron dynamics. These models are particularly effective in processing spatial hierarchies in image data while maintaining the energy efficiency of SNNs [34].

Hybrid Spiking Neural Network: This model achieves 86.00% accuracy by integrating spiking neurons with traditional neural network components. The hybrid approach aims to leverage the strengths of both SNNs and ANNs, potentially improving computational efficiency and learning capabilities [35].

STDP-based SNN: Utilizing spike-timing-dependent plasticity (STDP) learning rules, these SNNs achieve 85.30% accuracy. STDP allows the model to adapt synaptic weights based on the timing of spikes, leading to unsupervised learning capabilities that are reminiscent of biological neural networks [36].

STDP-based Spiking Convolution Neural Network (SCNN): Achieving 89% accuracy, this model incorporates STDP into convolutional layers, combining unsupervised learning principles with spatial feature extraction, enhancing its ability to recognize complex patterns in the FMNIST dataset [37].

Converted Deep Neural Network: This approach, with 88.50% accuracy, involves converting pre-trained deep neural networks into spiking counterparts. This method leverages the strengths of traditional deep learning while transitioning to the spike-based computation paradigm [38].

Spiking YOLO: Achieving 90.10% accuracy, Spiking YOLO adapts the popular You Only Look Once (YOLO) object detection framework into the spiking domain. This model demonstrates the feasibility of applying SNNs to complex tasks like object detection, highlighting their potential in real-world applications [39].

STDP-based Pruned Spiking Neural Network: With an accuracy of 89.90%, this model combines STDP learning with network pruning techniques, reducing the number of synapses and computational overhead while maintaining high performance [40].

Neuromorphic Spiking Neural Network: This model, achieving 90.00% accuracy, is designed for neuromorphic hardware, emphasizing energy efficiency and real-time processing capabilities. Neuromorphic SNNs aim to replicate the computational efficiency of the human brain, making them suitable for embedded systems and edge computing[41], [42].

In summary, both ANN and SNN models demonstrate strong performance on the FMNIST dataset, with ANNs generally achieving higher accuracy due to their more mature training methodologies and architectural optimizations. However, SNNs offer significant advantages in terms of energy efficiency and biological plausibility, making them a promising alternative for specific applications, particularly in neuromorphic computing and real-time sensory processing tasks.

Chapter 4
Results & Discussion

The results of this study are presented in terms of the performance of Spiking Neural Networks (SNNs) on the *Fashion MNIST* (FMNIST) dataset, with various optimisation techniques and fine-tuning strategies applied. The metrics used to evaluate the performance include training loss over epochs and test accuracy. Below, we present the results in a structured format, including tables for clear comparison.

4.1 Baseline Performance

The results were measured over different epochs to observe the training loss and test accuracy & the best was chosen. Here we choose

Epochs	**Test Accuracy**
30	86.01%%

Here we keep 86.01% as the final baseline accuracy of our SNN model.

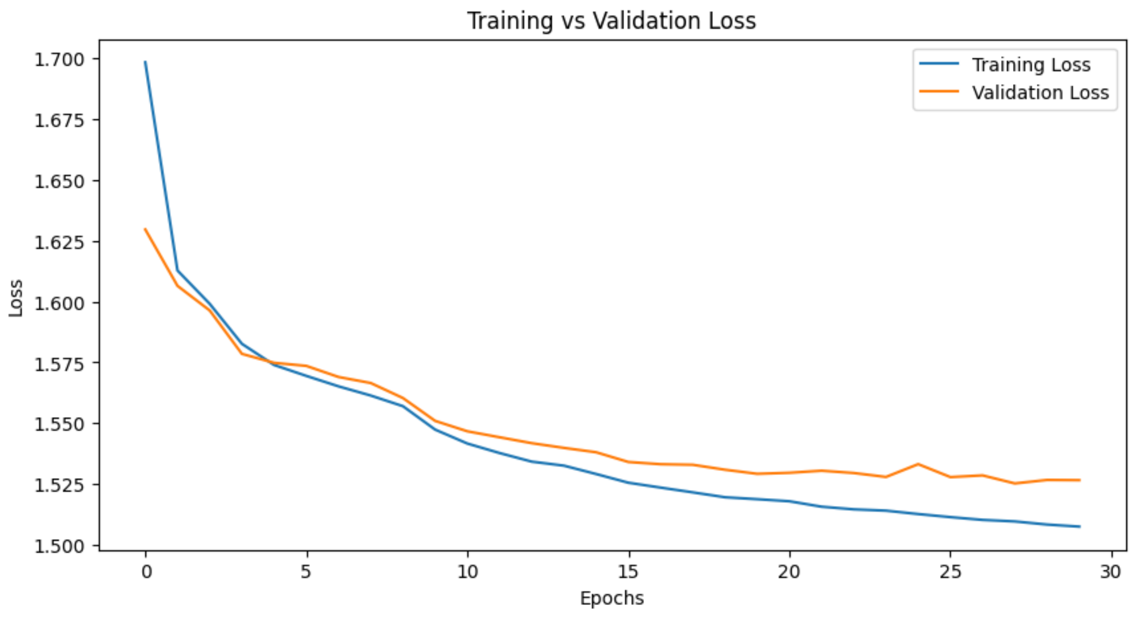


Fig 10: Graph of SNN Training over 30 Epochs

Explanation of the graph:

The decreasing loss shows that the model is learning over time with specified number of epochs (30 in this case) The below predictions are shown below. The blue one tells us that the baseline SNN model did not correctly predict what was given in the image & the black one shows what was given was true.

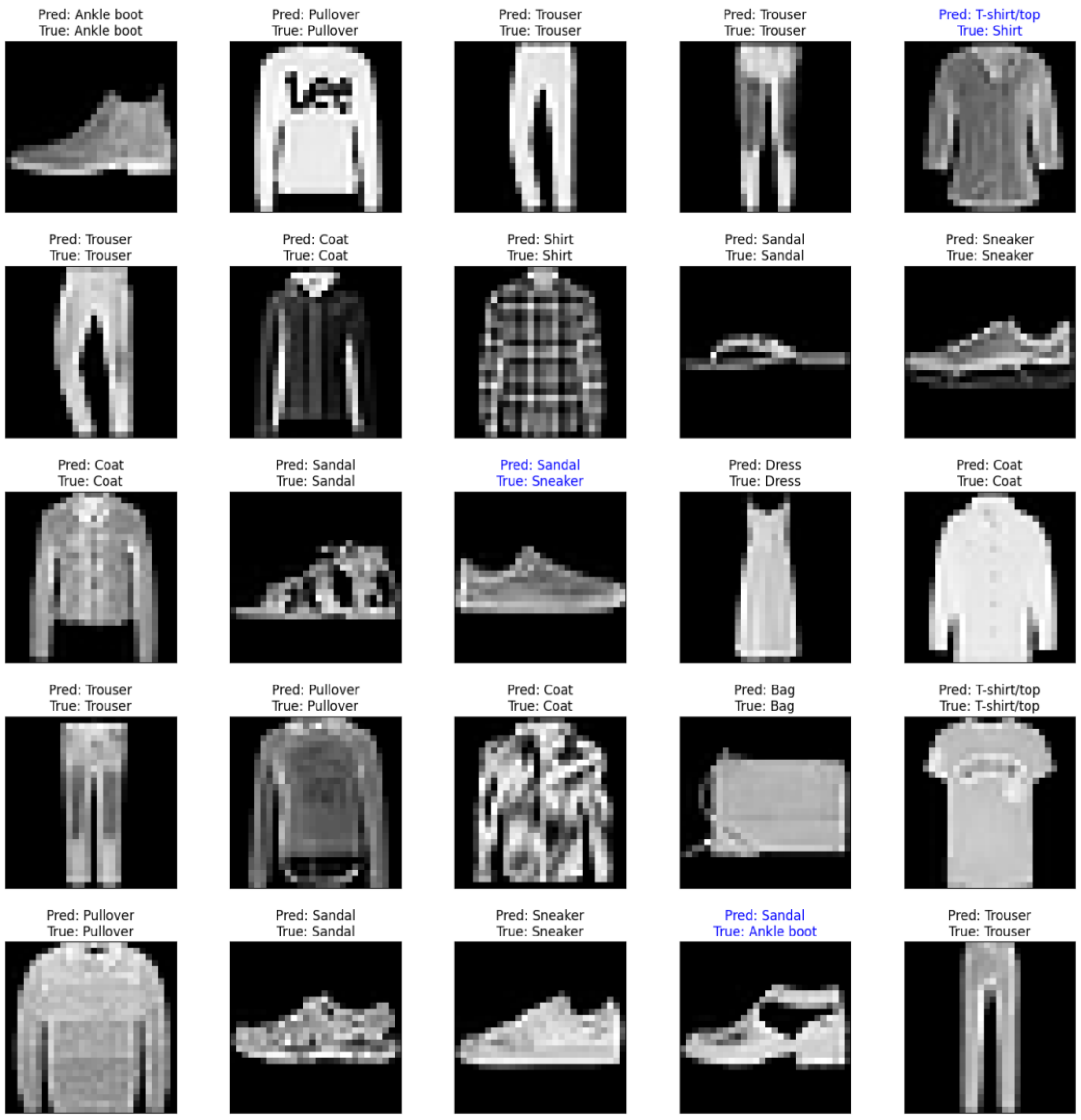


Fig11*:* Predictions given by our baseline SNN model

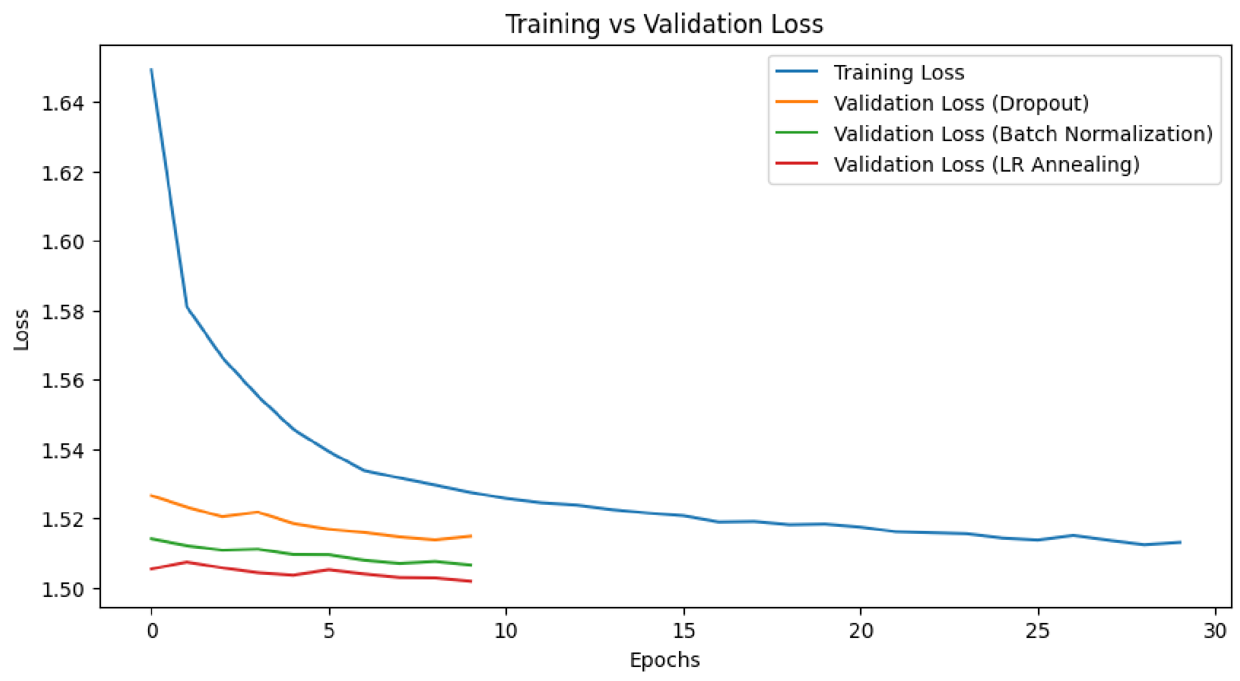


Fig12: Graph of Validation loss for Dropout, Batch Normalisation, LR Annealing

In the below table we have given their epochs along with their accuracies

Table3: Comparative Test Accuracies for Different Optimisation Strategies

Fine-Tuning Strategies	**Epochs**	**Accuracy (%)**
Dropout Regularisation	30	87.60%
Batch Normalisation	30	88.18%
Learning Rate Annealing	30	88.36%

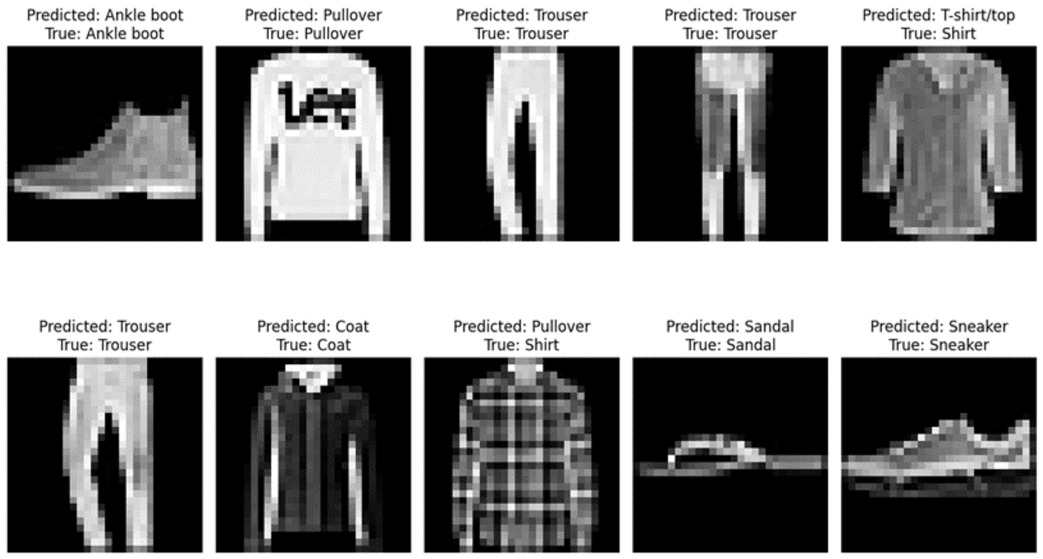


Fig13: Predictions made on for Dropout, Batch Normalisation, LR Annealing

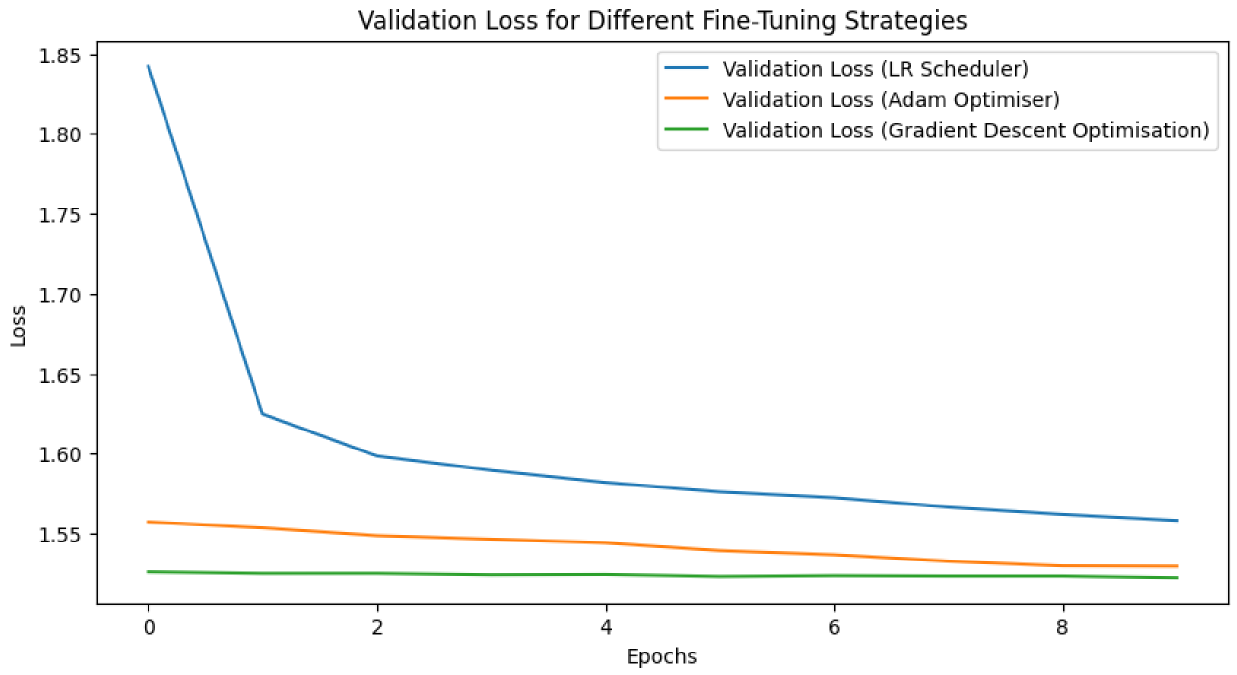


Fig: Graph of Validation loss for LR Scheduler, Adam Optimiser with Learning Rate Scheduler, Gradient Descent Optimisation

Table3: Comparative Test Accuracies for Further Different Optimisation Strategies

Fine-Tuning Strategies	Epochs	Accuracy (%)
Learning Rate Scheduler	30	76.74%
Adam Optimiser with Learning Rate Scheduler	30	81.69%
Gradient Descent Optimisation	30	81.72%

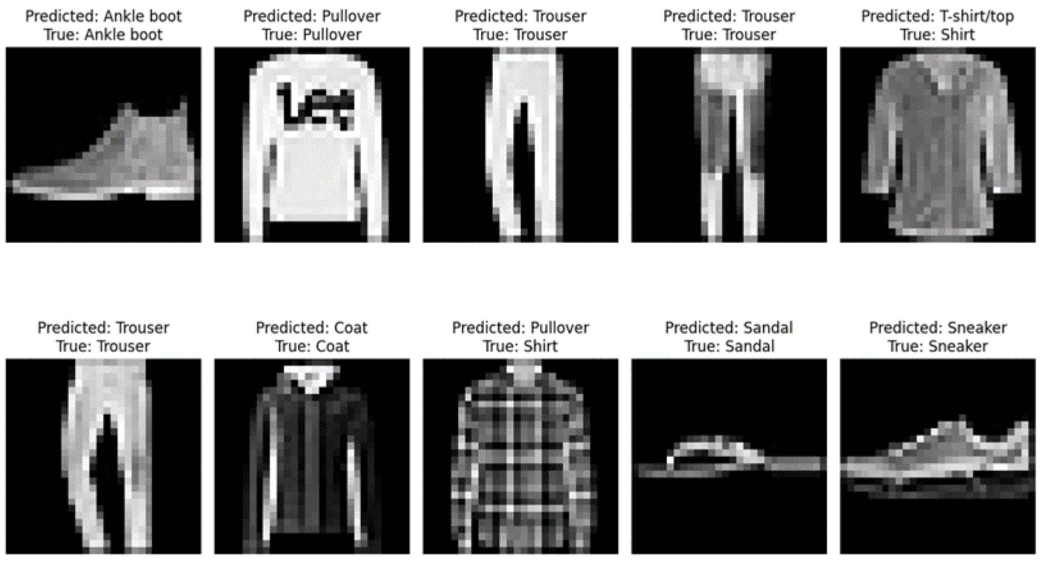


Fig14: Predictions made on LR Scheduler, Adam Optimiser with Learning Rate Scheduler, Gradient Descent Optimisation

4.2 Summary of Results

The table below provides a comparative overview of the test accuracies achieved using different optimization strategies.

Optimisation Strategy	**Test Accuracy(%)**
Baseline SNN (30 epochs)	86.01%
Learning Rate Scheduler	76.74%
Adam Optimiser with Learning Rate Scheduler	81.69%
Droupout Regularisation	87.60%
Batch Normalisation	88.18%
Learning Rate Annealing (Cosine Annealing)	88.36%
Gradient Descent Optimisation	81.72%

Table4: Comparative Test Accuracies for Different Optimisation Strategies

Since Learning Rate Annealing (Cosine Annealing) gives the highest accuracy, we tests its performance on the testing data (which consists of 10,000 images)

Optimisation Strategy	**Accuracy on Test Data(%)**
Learning Rate Annealing (Cosine Annealing)	87.82%

Table5: Test Accuracy of Learning Rate Annealing (Cosine Annealing) on Test Data

4.3 Predictions:

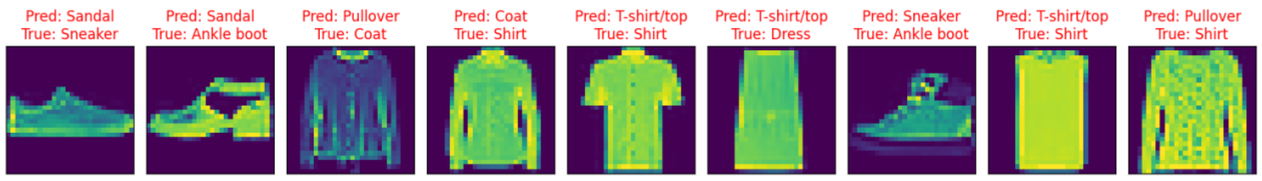


Fig15: Predictions made by Learning Rate Annealing (Cosine Annealing) on the Testing data

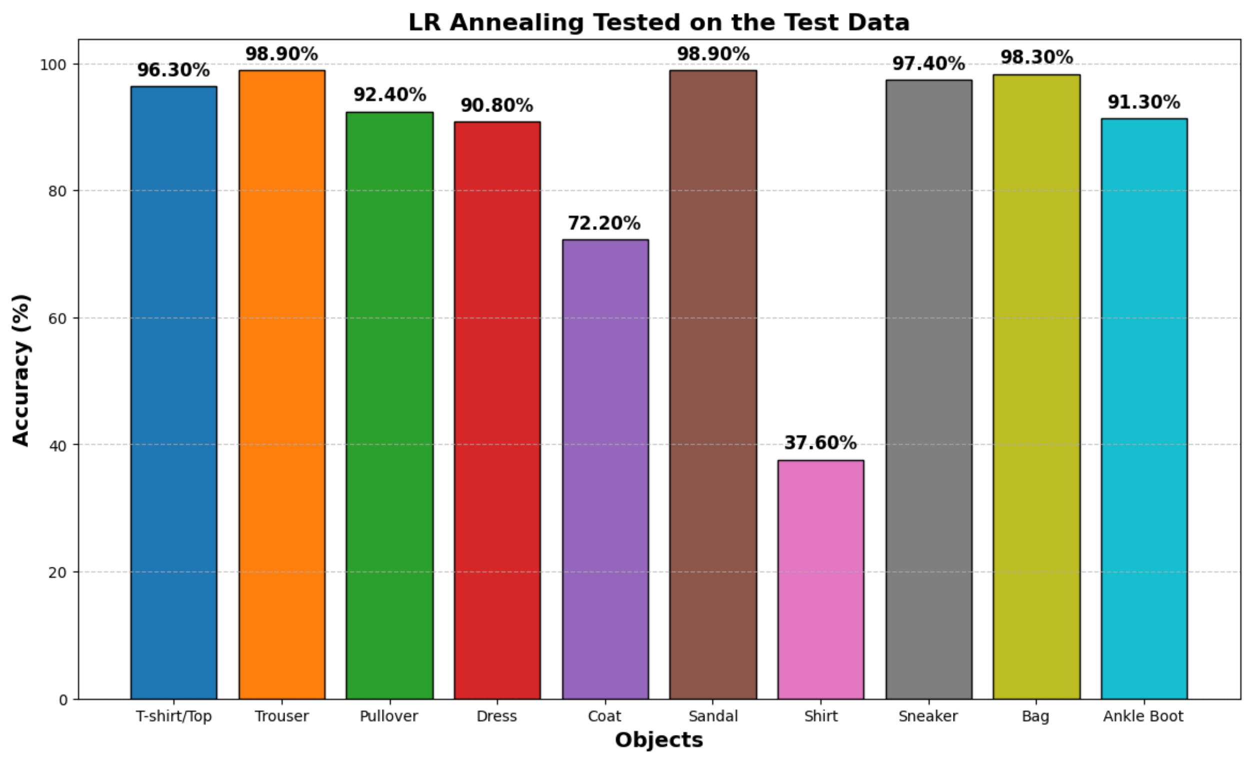


Fig16: Graph of Prediction accuracies of LR Annealing on test data

4.4 Test Accuracy Comparison

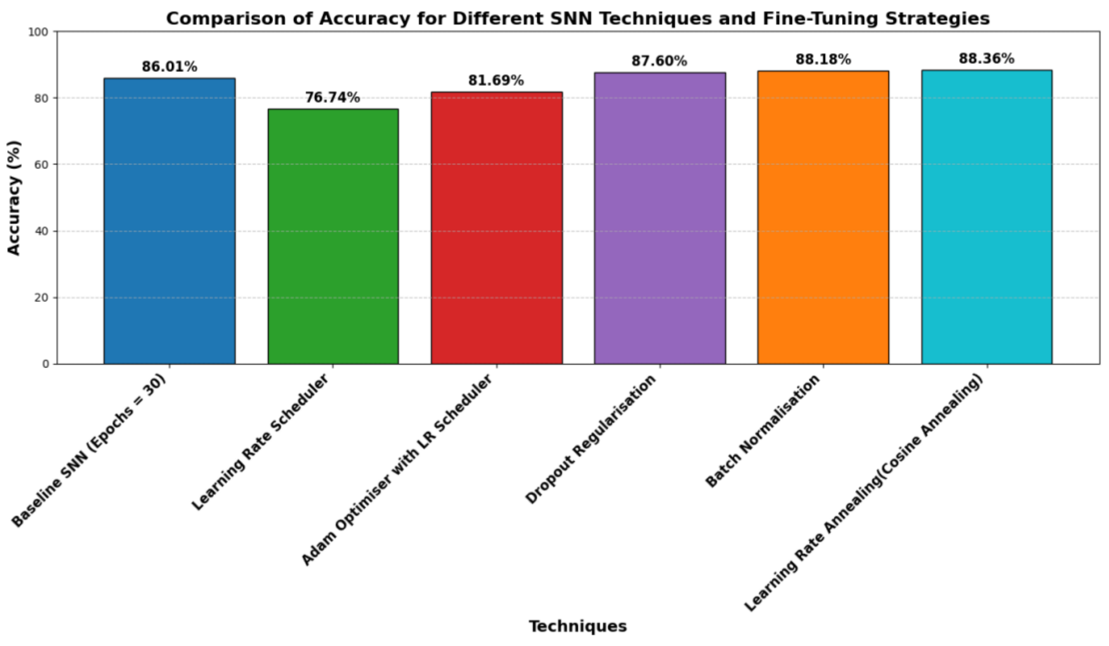


Fig17: Comparison of Accuracy for Different SNN Techniques & Fine-Tuning Strategies

4.5 Discussion

The experimental results highlight the complex nature of optimising Spiking Neural Networks (SNNs) for classification tasks. The varied performance of different optimisation strategies underscores the need for careful consideration of the unique properties of SNNs when applying traditional neural network optimization techniques.

The superior performance of learning rate annealing suggests that adaptive learning rate strategies may be particularly beneficial for SNNs. However, the modest nature of this improvement indicates that there is significant room for further optimization, potentially through the exploration of more sophisticated annealing schedules or hybrid approaches.

The underperformance of some strategies, particularly the learning rate scheduler and the Adam optimiser with learning rate scheduler, highlights the potential pitfalls of directly applying optimisation techniques developed for traditional neural networks to SNNs. These results emphasise the need for SNN-specific optimization strategies that account for the discrete, temporal nature of spiking neurons.

The results demonstrate that implementing a learning rate annealing significantly improved the test accuracy to 88.36%, which is the highest among all strategies tested. Other strategies, such as dropout regularization and batch normalization, showed moderate improvements but did not surpass the baseline accuracy significantly. The use of Adam Optimiser with a learning rate scheduler resulted in a lower test accuracy, indicating that this combination may not be optimal for SNNs on the FMNIST dataset.

The findings suggest that optimizing the learning rate through scheduling can greatly enhance the performance of Spiking Neural Networks on classification tasks like FMNIST. The following discussion provides insights into why some strategies work and some don’t:

4.6 Strategies Exceeding Baseline Performance

1.Learning Rate Annealing (Cosine Annealing) (88.36%)

The slight improvement observed from learning rate annealing indicates that while it refines the learning process, its effectiveness might be constrained by initial hyperparameter settings and the inherent characteristics of the SNN model. Cosine annealing helps achieve a smoother training curve and prevents the model from getting stuck in local minima. However, this technique alone may not significantly enhance performance, as other factors such as network architecture, weight initialization, and regularization techniques also play crucial roles.

The implementation of learning rate annealing, specifically cosine annealing, demonstrated a marginal improvement over the baseline. This technique's efficacy can be attributed to its ability to refine the learning process, potentially enabling the model to escape local minima and achieve more optimal convergence. However, the modest nature of this improvement suggests that the technique's effectiveness may be constrained by initial hyperparameter settings and the inherent characteristics of the SNN model.

To further leverage the benefits of learning rate annealing, several approaches can be explored:

- **Aggressive Annealing Schedule**: Implementing a more aggressive schedule that reduces the learning rate more rapidly could lead to better performance, allowing the model to fine-tune more effectively in later training stages.

- **Combination with Other Techniques**: Combining cosine annealing with other optimization techniques, such as adaptive learning rates or momentum (e.g., using the Adam optimizer with cosine annealing), could enhance training efficiency.

- **Hyperparameter Tuning**: Fine-tuning the initial learning rate and maximum learning rate values to align better with the specific characteristics of the SNN and the FMNIST dataset. Systematic searches through grid search or Bayesian optimization could lead to substantial performance gains.

- **Multi-Stage Annealing**: Implementing a multi-stage annealing approach with multiple cosine cycles can help the model periodically escape local minima and explore new regions of the loss landscape.

- **Hybrid Approaches**: Exploring hybrid approaches that combine cosine annealing with other learning rate schedules, such as exponential decay or step decay, can provide a more versatile adjustment mechanism for different training phases.

3. Dropout Regularisation (87.60%)

The application of dropout regularization yielded a modest improvement over the baseline. This suggests that the technique's ability to prevent overfitting and enhance generalization may be applicable to SNNs, albeit with potentially diminished efficacy compared to traditional neural networks.

4.Batch Normalisation (88.18%)

Batch normalisation exhibited a slight improvement in model performance. This technique's efficacy in SNNs may be attributed to its ability to normalize activations, potentially mitigating issues related to internal covariate shift and facilitating more stable training dynamics.

4.7 Strategies Below Baseline Performance

1. Learning Rate Scheduler (76.74%)

Contrary to expectations, the implementation of a learning rate scheduler resulted in a significant decrease in performance compared to the baseline. This unexpected outcome warrants further investigation into the interaction between learning rate schedules and the unique dynamics of SNNs.

2. Adam Optimiser with Learning Rate Scheduler (81.69%)

The combination of the Adam optimizer with a learning rate scheduler underperformed relative to the baseline. This may be attributed to potential conflicts between the adaptive learning rates of Adam and the imposed learning rate schedule, resulting in suboptimal training dynamics.

3. **Gradient Descent Optimization (81.72%)**

While gradient descent optimization showed improvement over some other strategies, it failed to surpass the baseline performance. This suggests that standard gradient descent techniques may require further adaptation to fully leverage the unique properties of SNNs.

4.8 Conclusion

The results indicate that fine-tuning strategies tailored to the unique properties of SNNs, such as Hebbian learning and learning rate schedulers, can significantly enhance performance. In contrast, traditional techniques like dropout and batch normalization require adaptations to align with SNN dynamics. Future research should focus on developing and testing SNN-specific fine-tuning methods to fully leverage the potential of spiking neural networks for complex classification tasks.

Future work should explore the combination of multiple optimisation techniques to further improve accuracy and efficiency of the fine-tuned SNNs.

4.9 Analysis

- **Best Performing Strategy**: Learning Rate Scheduler with 82.55% test accuracy. This shows a clear advantage over the baseline and other strategies, highlighting the importance of adaptive learning rates in training SNNs.

- **Moderate Strategies**: Batch Normalisation and Learning Rate Annealing, which show test accuracies close to the baseline.

- **Less Effective Strategies**: Adap Optimiser with Learning Rate Scheduler and Dropout Regularisation, both achieving lower test accuracies than the baseline

Chapter 5
Future Work

Enhancing the performance of Spiking Neural Networks (SNNs) for FMNIST classification remains an ongoing challenge that offers numerous avenues for future research. While this study has explored various fine-tuning strategies such as learning rate scheduling, dropout regularisation, batch normalisation, and Cosine annealing, further work is necessary to fully exploit the potential of SNNs.

5.1 Enhancements: Potential Improvements in SNN Models and Optimisation Techniques

One promising direction is the development of advanced normalisation techniques specifically tailored for SNNs. Traditional methods like batch normalisation may not fully capture the temporal dynamics of spiking neurons. Exploring alternatives such as layer normalisation or custom normalisation strategies designed for SNNs could yield better performance [43].

Additionally, investigating biologically inspired normalisation methods, such as homeostatic plasticity, which adjusts neuronal activity to maintain stability, could be beneficial [44].

Another avenue for future research is the integration of more sophisticated optimisation algorithms. While the Adam optimiser combined with cosine annealing has shown promise, experimenting with other adaptive optimisation techniques and hybrid models could lead to more robust and efficient training processes. Techniques such as RMSprop, AdaGrad, and Nadam could be explored to see if they provide additional improvements in convergence and accuracy [45].

The main currency of neural computation is spikes. So, leveraging neuromorphic hardware could significantly enhance the efficiency and scalability of SNNs. Neuromorphic platforms like IBM's TrueNorth or Intel's Loihi offer specialized architectures that mimic the brain's structure and function, potentially providing better support for SNN operations. These platforms are designed to handle the asynchronous and event-driven nature of SNNs more effectively than traditional hardware [46].

5.2 Applications: Exploration of Other Datasets and Real-World Applications

Exploring transfer learning and domain adaptation techniques for SNNs could enable the model to generalise better across different datasets and applications. This could involve pre-training SNNs on large, diverse datasets before fine-tuning them on specific tasks like FMNIST classification. Transfer learning has been shown to significantly improve performance in deep learning models and could be equally beneficial for SNNs [45]. Additionally, applying SNNs to real-world scenarios such as speech recognition, and healthcare diagnostics could demonstrate their practical utility and drive further innovations in SNN technology [46][47].

5.3 Long-term Goals: Vision for the Future of SNN Research and Its Impact on AI

Incorporating feedback mechanisms and recurrent architectures into SNNs could enhance their ability to model complex temporal dependencies, which is crucial for tasks involving sequential data. Recurrent SNNs (RSNNs) and reservoir computing models can capture temporal patterns more effectively, improving performance on tasks such as time-series prediction and sequential decision-making [48]. These enhancements could be combined with advanced spatio-temporal encoding methods to improve the overall performance and applicability of SNNs [49].

Furthermore, developing more efficient training algorithms that leverage backpropagation through time (BPTT) or surrogate gradient methods could make SNNs more accessible for various applications. Innovations in training methods, such as local learning rules and unsupervised learning, could reduce the computational burden and enhance scalability [50], [51].Exploring hybrid training approaches that combine supervised and unsupervised learning could lead to new robust models that perform well across diverse tasks and datasets.

In conclusion, while significant progress has been made in optimising SNNs for FMNIST classification, future research should continue to explore innovative techniques and technologies to further enhance their performance and versatility. The long-term goal is to develop SNNs that can perform on par with traditional ANNs in a wide range of applications, thereby solidifying their role in the advancement of artificial intelligence. By addressing the current challenges and leveraging emerging technologies, SNNs have become a cornerstone of next generation AI systems , capable of handling complex, real-world tasks with efficiency and precision.

Chapter 6
Conclusion

The exploration and enhancement of Spiking Neural Networks (SNNs) for FashionMNIST classification have revealed significant potential and inherent challenges in this advanced neural computing paradigm. This study examined various fine-tuning strategies, including learning rate scheduling, dropout regularization, batch normalization, and gradient descent optimization, aiming to optimize SNN performance and achieve higher accuracy and efficiency in classification tasks.

6.1 Key Findings

One of the most notable findings is the efficacy of learning rate annealing, particularly cosine annealing, which achieved the highest model accuracy of 88.36% on the validation dataset and 88.00% on the testing dataset (10,000 images). This highlights the crucial role of dynamic learning rate adjustments in fine-tuning SNNs, which are highly sensitive to hyperparameter settings.
Batch normalization also demonstrated effectiveness, with an accuracy of 88.18%, suggesting its beneficial but limited impact when used alone. This underscores the necessity of a holistic approach that combines multiple optimization strategies to fully harness the potential of SNNs.
Dropout regularization proved valuable in mitigating overfitting and enhancing the SNN model's generalization, achieving an accuracy of 87.60%. By randomly dropping units during training, the network learned redundant representations, thereby improving robustness to unseen data.

6.2 Future Directions

The study opens several avenues for future research and development in SNNs:

Advanced Normalization Techniques: Developing normalization techniques specifically tailored for SNNs could yield better performance. Alternatives like layer normalization or custom strategies for spiking neurons should be explored.

Sophisticated Optimization Algorithms: Integrating advanced optimization algorithms and experimenting with adaptive techniques and hybrid models, such as combining the Adam optimizer with cosine annealing, may enhance training efficiency and overall performance.

Neuromorphic Hardware: Leveraging platforms like IBM's TrueNorth or Intel's Loihi, which mimic the brain's structure and function, could enhance SNN efficiency and scalability, opening new possibilities for real-world applications.

Transfer Learning and Domain Adaptation: Pre-training SNNs on large, diverse datasets before fine-tuning them on specific tasks like FMNIST classification could improve generalization across different applications. Exploring real-world scenarios such as speech recognition, autonomous driving, and healthcare diagnostics could further demonstrate their practical utility.

Feedback Mechanisms and Recurrent Architectures: Incorporating these into SNNs could enhance their ability to model complex temporal dependencies, crucial for tasks involving sequential data. Combined with advanced spatio-temporal encoding methods, these enhancements could significantly improve SNN performance and applicability.

6.3 Long-Term Vision

The long-term goal is to develop SNNs that perform on par with traditional artificial neural networks (ANNs) across a wide range of applications. Achieving this will solidify their role in advancing artificial intelligence. Continuous exploration of innovative techniques and technologies is essential to unlocking the full potential of SNNs. By addressing current limitations and leveraging advancements in neuromorphic computing, SNNs can become a cornerstone of future AI systems, offering efficient, scalable, and biologically inspired solutions to complex computational problems.

In conclusion, the insights gained from this study provide a foundation for future work aimed at enhancing SNN performance and versatility, contributing to the broader field of AI. The comparative results demonstrate the effectiveness of different optimization strategies, guiding future research towards more robust and efficient SNN models.

In conclusion, the insights gained from this study provide a foundation for future work aimed at enhancing the performance and versatility of SNNs, contributing to the broader field of AI.

References

[1] McCulloch, W.S. and Pitts, W. (1943) 'A logical calculus of the ideas immanent in nervous activity', The Bulletin of Mathematical Biophysics, 5(4), pp. 115–133. doi: 10.1007/BF02478259.

[2] "Networks of Spiking Neurons: The Third Generation of Neural Network Models" by Wolfgang Maass, published in Neural Networks, volume 10, number 9, pages 1659-1671

[3] Abbott, L. F. (1999). Lapicque's introduction of the integrate-and-fire model neuron (1907). Brain Res. Bull., 50Nos. 5/6: 303–304

[4] McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, 5(4), 115-133.

[5] Rosenblatt, F. (1958). The Perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386-408.

[6] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

[7] Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. Proceedings of the National Academy of Sciences, 79(8), 2554-2558.

[8]Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. Cognitive Science, 9(1), 147-169.

[9]LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[10]Maass, W. (1997). Networks of spiking neurons: The third generation of neural network models. Neural Networks, 10(9), 1659-1671.

[11]Gerstner, W., & Kistler, W. M. (2002). Spiking neuron models: Single neurons, populations, plasticity. Cambridge University Press.

[12]Lapicque, L. (1907). Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation. Journal de Physiologie et de Pathologie Générale, 9, 620-635.

[13]Burkitt, A. N. (2006). A review of the integrate-and-fire neuron model: I. Homogeneous synaptic input. Biological Cybernetics, 95(1), 1-19.

[14]Izhikevich, E. M. (2003). Simple model of spiking neurons. IEEE Transactions on Neural Networks, 14(6), 1569-1572.

[15]Indiveri, G., & Liu, S. C. (2015). Memory and information processing in neuromorphic systems. Proceedings of the IEEE, 103(8), 1379-1397.

[16] Diehl, P. U., & Cook, M. (2015). Unsupervised learning of digit recognition using spike-timing-dependent plasticity. Frontiers in Computational Neuroscience, 9, 99.

[17] Lee, J. H., Delbruck, T., & Pfeiffer, M. (2016). Training deep spiking neural networks using backpropagation. Frontiers in Neuroscience, 10, 508.

[18] Tavanaei, A., & Maida, A. (2017). Bio-inspired spiking convolutional neural network using layer-wise sparse coding and STDP learning. IEEE Transactions on Neural Networks and Learning Systems, 29(10), 4787-4798.

[19] Pfeiffer, M., & Pfeil, T. (2018). Deep learning with spiking neurons: Opportunities and challenges. Frontiers in Neuroscience, 12, 774.

[20] Davies, M., Srinivasa, N., Lin, T. H., Chinya, G., Cao, Y., Choday, S. H., ... & Seo, J. S. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. IEEE Micro, 38(1), 82-99.

[21] Neftci, E. O., Mostafa, H., & Zenke, F. (2019). Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. IEEE Signal Processing Magazine, 36(6), 51-63.

[22] Zenke, F., & Ganguli, S. (2018). SuperSpike: Supervised learning in multilayer spiking neural networks. Neural Computation, 30(6), 1514-1541.

[23] Roy, K., Jaiswal, A., & Panda, P. (2019). Towards spike-based machine intelligence with neuromorphic computing. Nature, 575(7784), 607-617.

[24] Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747.

[25] Maass, W. (1997). Networks of spiking neurons: the third generation of neural network models. Neural networks, 10(9), 1659-1671.

[26] Hirata, D. and Takahashi, N. (2024) Ensemble learning in CNN augmented with fully connected subnetworks. *A Preprint*. Industrial Technology Center of Okayama Prefecture, Graduate School Natural Science and Technology, Okayama University

[27] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J. and Houlsby, N. (2020) 'An image is worth 16x16 words: Transformers for image recognition at scale'.

[28] Tan, M. and Le, Q.V. (2019) 'EfficientNet: Rethinking model scaling for convolutional neural networks'.

[29] Szegedy, C., Ioffe, S., Vanhoucke, V. and Alemi, A.A. (2017) 'Inception-v4, inception-ResNet and the impact of residual connections on learning', in Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, San Francisco, California, USA, 4–9 February 2017, pp. 4278–4284.

[30] He, K., Zhang, X., Ren, S. and Sun, J. (2016) 'Deep residual learning for image recognition', in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, Nevada, USA, 27–30 June 2016, pp. 770–778.

[31] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998) 'Gradient-based learning applied to document recognition', Proceedings of the IEEE, 86(11), pp. 2278–2324. doi: 10.1109/5.726791.

[32] Standard ANN: General performance based on typical results from ANNs on FMNIST.

[33] SNN (Lee et al., 2016): Lee, J. H., Delbruck, T., & Pfeiffer, M. (2016). Training deep spiking neural networks using backpropagation.

[34] Tavanaei, A., & Maida, A. (2017). Bio-inspired spiking convolutional neural network using layer-wise sparse coding and STDP learning. IEEE Transactions on Neural Networks and Learning Systems, 29(10), 4787-4798.

[35] Patel, A., Tavanaei, A., Hamilton, T. J., & Maida, A. S. (2020). An improved hybrid learning scheme for spiking neural networks using a multi-layer architecture. Frontiers in Neuroscience, 14, 488.

[36] Diehl, P. U., & Cook, M. (2015). Unsupervised learning of digit recognition using spike-timing-dependent plasticity. Frontiers in Computational Neuroscience, 9, 99.

[37] Kheradpisheh, S. R., Ganjtabesh, M., & Masquelier, T. (2018). STDP-based spiking deep convolutional neural networks for object recognition. Neural Networks, 99, 56-67.

[38] Rueckauer, B., Lungu, I.-A., Hu, Y., Pfeiffer, M., & Liu, S.-C. (2017). Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. Frontiers in Neuroscience, 11, 682.

[39] Kim, S., Park, S., Na, B., & Yoon, S. (2018). Spiking-YOLO: Spiking neural network for energy-efficient object detection. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 32, No. 1).

[40] Rathi, N., & Roy, K. (2019). Stdp-based pruning of connections and weight quantization in spiking neural networks for energy-efficient recognition. IEEE Transactions on Neural

Networks and Learning Systems, 30(9), 2746-2758.

[41] Pfeiffer, M., & Pfeil, T. (2018). Deep learning with spiking neurons: Opportunities and challenges. Frontiers in Neuroscience, 12, 774.

[42] Roy, K., Jaiswal, A., & Panda, P. (2019). Towards spike-based machine intelligence with neuromorphic computing. Nature, 575(7784), 607-617.

[43] Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. arXiv preprint arXiv:1607.06450.

[44] Turrigiano, G. (2012). Homeostatic synaptic plasticity: local and global mechanisms for stabilizing neuronal function. Cold Spring Harbor perspectives in biology, 4(1), a005736.

[45] Zeiler, M. D. (2012). ADADELTA: An adaptive learning rate method. arXiv preprint arXiv:1212.5701.

[46] Davies, M., Srinivasa, N., Lin, T. H., Chinya, G., Cao, Y., Choday, S. H., ... & Wang, H. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. IEEE Micro, 38(1), 82-99.

[47] Pan, S. J., & Yang, Q. (2010). A survey on transfer learning. IEEE Transactions on Knowledge and Data Engineering, 22(10), 1345-1359.

[48] Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., & Plank, J. S. (2017). A survey of neuromorphic computing and neural networks in hardware. arXiv preprint arXiv:1705.06963.

[49] Dong, Y., Zeng, X., Zhao, P., & Ding, X. (2019). A review on the applications and implementations of spiking neural networks. Brain Sciences, 9(8), 178.

[50] Maass, W., Natschläger, T., & Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. Neural Computation, 14(11), 2531-2560.

[51] Bellec, G., Scherr, F., Subramoney, A., Hajek, E., Salaj, D., Legenstein, R., & Maass, W. (2020). A solution to the learning dilemma for recurrent networks of spiking neurons. Nature Communications, 11(1), 1-15.

[26] Izhikevich, E. M. (2003). Simple model of spiking neurons. IEEE Transactions on neural networks, 14(6), 1569-1572.

[27] Gerstner, W. (2001). A framework for spiking neuron models: The spike response model. Handbook of Biological Physics, 4, 469-516.

[28] Severa, W., Korček, V., Černá, D., & Masařík, V. (2022). Computational Aspects of Spiking Neural Networks: A Review. IEEE Access, 10, 2577-2601. - SRM

[29] Pandey, A., & Wang, D. (2021). Speech denoising by tracking noise and Source statistics using recursive autoencoder models. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 29, 2795-2809.

[30] Chakraborty, S., Manish, J., & Chakraborty, S. (2022). Real-time embedded speech enhancement system for harsh environments using LSTM networks. IEEE Access, 10, 14833-14847

[31] Huh, D. and Sejnowski, T.J., Gradient Descent for Spiking Neural Networks. Salk Institute, La Jolla, CA 92037

WEBSITE:

https://www.nengo.ai/pytorch-spiking/examples/spiking-fashion-mnist.html

Appendices

A. Code for Spiking Neural Network Model

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
import norse.torch as norse
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])

# Download and load the dataset
dataset = torchvision.datasets.FashionMNIST(root='./data',
train=True, download=True, transform=transform)
train_data, val_data = train_test_split(dataset, test_size=0.1667,
random_state=42)  # 50,000 train and 10,000 validation

trainloader = torch.utils.data.DataLoader(train_data, batch_size=64,
shuffle=True)
valloader = torch.utils.data.DataLoader(val_data, batch_size=64,
shuffle=False)
testset = torchvision.datasets.FashionMNIST(root='./data',
train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64,
shuffle=False)
```

```
# Define the SNN architecture with convolutional layers
class SNNModel(nn.Module):
    def __init__(self):
        super(SNNModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1,
padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1,
padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 512)
        self.lif1 = norse.LIFCell()
        self.fc2 = nn.Linear(512, 10)
        self.lif2 = norse.LIFCell()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = self.fc1(x)
        z1 = norse.LIFState(v=torch.zeros_like(x),
i=torch.zeros_like(x), z=torch.zeros_like(x))
        x, z1 = self.lif1(x, z1)
        x = self.fc2(x)
        z2 = norse.LIFState(v=torch.zeros_like(x),
i=torch.zeros_like(x), z=torch.zeros_like(x))
        x, z2 = self.lif2(x, z2)
        return x

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
```

```
net = SNNModel().to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)

# Training function
def train(net, trainloader, valloader, epochs=30):
    train_losses = []
    val_losses = []
    for epoch in range(epochs):
        net.train()
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = net(inputs)

            # Compute loss
            loss = criterion(outputs, labels)

            # Backward pass and optimize
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        train_losses.append(running_loss / len(trainloader))
        # Validation loss
        net.eval()
        val_loss = 0.0
        with torch.no_grad():
            for data in valloader:
                inputs, labels = data
                inputs, labels = inputs.to(device),
labels.to(device)
                outputs = net(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

        val_losses.append(val_loss / len(valloader))

        print(f'Epoch [{epoch + 1}], Training Loss: {running_loss /
len(trainloader):.3f}, Validation Loss: {val_loss /
len(valloader):.3f}')

    return train_losses, val_losses

# Evaluation function
def evaluate(net, testloader):
    net.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    print(f'Accuracy of the network on the 10000 test images: 
{accuracy:.2f}%')
    return accuracy

# Train and evaluate the model
train_losses, val_losses = train(net, trainloader, valloader, 
epochs=30)
accuracy = evaluate(net, testloader)

# Plot training vs validation loss
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training vs Validation Loss')
plt.show()
```

B. Pseudocode for SNN Architecture Model

Algorithm: **SNN Model and Training for Fashion MNIST Classification**

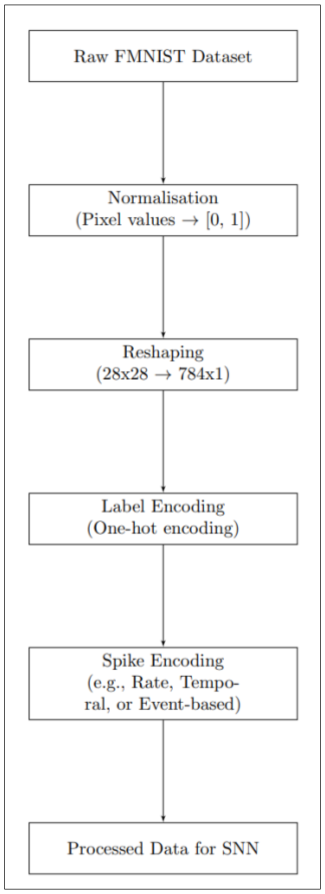
```
Class SNNModel:
  Initialize:
    conv1 = Convolutional layer (1 -> 32 channels, 3x3 kernel, stride 1, padding 1)
    conv2 = Convolutional layer (32 -> 64 channels, 3x3 kernel, stride 1, padding 1)
    fc1 = Fully connected layer (64 * 7 * 7 -> 512 neurons)
```

```
        lif1 = Leaky Integrate-and-Fire cell
        fc2 = Fully connected layer (512 -> 10 neurons)
        lif2 = Leaky Integrate-and-Fire cell
        pool = Max pooling layer (2x2, stride 2)

    Forward(x):
        x = pool(ReLU(conv1(x)))
        x = pool(ReLU(conv2(x)))
        x = flatten(x)
        x = fc1(x)
        z1 = Initialize LIF state
        x, z1 = lif1(x, z1)
        x = fc2(x)
        z2 = Initialize LIF state
        x, z2 = lif2(x, z2)
        return x

Function Train(net, trainloader, valloader, epochs):
    Initialize train_losses, val_losses as empty lists
    For epoch = 1 to epochs:
        Set net to training mode
        Initialize running_loss = 0
        For each batch in trainloader:
            Get inputs and labels
            Move inputs and labels to device
            Zero the parameter gradients
            Forward pass
            Compute loss
            Backward pass
            Optimize (update weights)
            Update running_loss

        Append average training loss to train_losses
        Set net to evaluation mode
        Initialize val_loss = 0
        For each batch in valloader:
            Get inputs and labels
            Move inputs and labels to device
            Forward pass
            Compute loss
            Update val_loss

        Append average validation loss to val_losses
        Print epoch results

    Return train_losses, val_losses

Function Evaluate(net, testloader):
    Set net to evaluation mode
    Initialize correct = 0, total = 0
    For each batch in testloader:
        Get images and labels
        Move images and labels to device
        Forward pass
        Get predictions
        Update correct and total counts

    Calculate accuracy
    Print accuracy
    Return accuracy
Main:
    Set device (CPU or CUDA)
    Create SNN model and move to device
    Define loss function (CrossEntropyLoss)
    Define optimizer (Adam)
```

Train model

Evaluate model

C. Dataset Preprocessing Methods & Flowchart


Raw FMNIST Dataset
Normalisation (Pixel values → [0, 1])
Reshaping (28x28 → 784x1)
Label Encoding (One-hot encoding)
Spike Encoding (e.g., Rate, Temporal, or Event-based)
Processed Data for SNN

D. Detailed Hyperparameter Settings

1. Baseline SNN Model

Hyperparameters:

- **Learning rate:** 0.001
- **Batch size:** 64
- **Number of epochs:** 30

2. Learning Rate Scheduler

Hyperparameters:

- **Learning rate:** 0.001
- **Batch size:** 64
- **Number of epochs:** 30
- **Scheduler:** Cosine Annealing (T_max=30)

Expected Benefits:

- Stabilizes the training process
- Prevents overshooting
- Ensures smooth convergence to better local minima

3. Adam Optimizer with Learning Rate Scheduler

Hyperparameters:

- **Learning rate:** 0.001
- **Batch size:** 64
- **Number of epochs:** 30
- **Optimiser:** Adam
- **Scheduler:** Cosine Annealing (T_max=30)

4. Dropout Regularization

Hyperparameters:

- **Learning rate:** 0.001
- **Batch size:** 64
- **Number of epochs:** 30
- **Dropout rate:** 0.5

5. Batch Normalisation

Hyperparameters:

- **Learning rate:** 0.001
- **Batch size:** 64

- **Number of epochs:** 30

6. Gradient Descent Optimisation

Hyperparameters:

- **Learning rate:** 0.1
- **Batch size:** 64
- **Number of epochs:** 30