Extending Forward Forward Algorithm Experimenting with Activations and CIFAR dataset

Except where reference is made to the work of others, the work described in this project is our own or was done in collaboration with my advisory committee. Further, the content of this group project is truthful in regards to our own work and the portrayal of others' work. This project does not include proprietary or classified information.

Venkatesh Tantravahi	Balaji Thota	Akhil Reddy Katti

Certificate of Approval:

Tarannum Shaila Zaman Assistant Professor Department of Computer Science	Jorge Novillo Professor Department of Computer Science

Extending Forward Forward Algorithm Experimenting with Activations and CIFAR dataset

Venkatesh Tantravahi Balaji Thota Akhil Reddy Katti

A Project
Submitted to the
Graduate Faculty of the
State University of New York Polytechnic Institute
in Partial Fulfillment of the
Requirements for the
Degree of Master of Science

Utica, New York
Dec 12, 2024

Extending Forward Forward Algorithm Experimenting with Activations and CIFAR dataset

Venkatesh Tantravahi Balaji Thota Akhil Reddy Katti

_____
Signature of Author

_____
Signature of Author

_____
Signature of Author

_____
Date of Graduation

Extending Forward Forward Algorithm Experimenting with Activations and CIFAR dataset

Project Abstract:

The Forward-Forward (FF) algorithm offers a groundbreaking alternative to the traditional backpropagation method for training neural networks. This study explores the practical implementation and performance of the FF algorithm, extending its applicability to more complex datasets such as CIFAR-10. Although the original FF implementation showed promise on simpler datasets, such as MNIST(Modified National Institute of Standards and Technology), adapting it to CIFAR-10 required significant tweaks, including architectural adjustments and parameter fine-tuning. Furthermore, addressing the future work proposed in the original article, we systematically evaluated the performance of the FF algorithm on various activation functions to assess its adaptability and efficiency.

To provide a robust comparison, we conducted experiments using both the FF algorithm and backpropagation across two datasets. CIFAR-10, known for its high visual complexity, and Fashion MNIST(Modified National Institute of Standards and Technology), a simpler benchmark. Our results highlight that the FF algorithm excels in less complex datasets, outperforming backpropagation in certain configurations. However, it faces challenges when applied to more intricate datasets such as CIFAR-10 due to the increased demands of feature extraction and representation learning. These findings not only validate the viability of the FF algorithm in specific contexts but also identify areas for optimization and further research.

By implementing these enhancements and conducting a comprehensive experimental analysis, this work sheds light on the strengths and limitations of the FF algorithm. It provides valuable insights for its broader adoption and future refinement, contributing to the ongoing exploration of alternative learning paradigms in neural networks.

Acknowledgments

We, the authors of this report, would like to collectively express our heartfelt gratitude to our Assistant Professor, Dr. Zaman, for her invaluable guidance, encouragement, and support throughout this project. Her expertise and insights have been instrumental in shaping the direction and outcomes of our work.

We extend our sincere thanks to each other for the unwavering collaboration and teamwork demonstrated throughout the research and implementation phases. This project has been a testament to our collective dedication and shared vision.

We would also like to thank the Department of Computer Science for providing a conducive environment and the necessary resources to undertake this research, which enabled us to explore innovative ideas and achieve meaningful results.

Special acknowledgment goes to the vast and generous open-source community, including platforms such as Stack Overflow, Google, and other issue-tracking forums, for offering solutions and insights that helped us overcome technical challenges during this journey.

Our work has been greatly inspired by the pioneering contributions of Geoffrey Hinton, whose preliminary investigation of the Forward-Forward algorithm in MATLAB served as a significant reference and motivation for this study. His innovative approach to exploring alternative neural network training paradigms continues to inspire our research.

Thank you all for contributing to the success of our work.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Why do we need alternatives to backpropagation?

Backpropagation, the algorithm underpinning most modern neural networks, relies on gradient descent to minimize a loss function. At its core, backpropagation computes the gradient of the loss with respect to network parameters by applying the chain rule across multiple layers. This approach, while powerful, is not without limitations. One of the most significant challenges is the issue of vanishing or exploding gradients. In deep networks, the gradients propagated through the layers can become exceedingly small or large, especially when activation functions like sigmoid or tanh are used. This results in slower convergence or instability during training, making optimization difficult for very deep architectures.

Another limitation stems from the requirement of differentiable activation functions. Gradient-based optimization inherently relies on differentiability, which restricts the choice of activation functions to smooth, continuous mappings. This excludes a wide range of potentially useful non-differentiable or discrete functions that could better model certain problems or domains. For example, ReLU (Rectified Linear Unit), while commonly used, introduces its own set of challenges, such as dead neurons where gradients vanish entirely.

Backpropagation also suffers from computational inefficiency. The repeated computation of gradients for every parameter, combined with memory-intensive storage of intermediate activations during forward and backward passes, makes backpropagation resource-intensive, particularly for large-scale networks. Moreover, the sequential nature of the algorithm—where errors must propagate backward layer by layer—hinders parallelization, further slowing down training on modern hardware.

These issues motivate the search for alternative training paradigms that do not rely on backpropagated gradients. An ideal alternative would be robust to vanishing or exploding gradients, allow the use of non-differentiable activation functions, and exploit parallelism more effectively. The Forward-Forward (FF) algorithm offers a novel solution by discarding gradient descent altogether and instead optimizing a layer-wise goodness function.

1.2 What is the Forward-Forward algorithm, and how does it work?

The Forward-Forward algorithm fundamentally reimagines the training process by replacing backpropagation with a goodness optimization framework. Each layer is trained independently to maximize a "goodness score" that measures its ability to differentiate between positive and negative data. The goodness for a given layer is defined as:

$$G = \sum_i \sigma(Wx + b)_i^2,$$

where W is the weight matrix, x is the input, b is the bias vector, and σ is the activation function applied element-wise. Positive data corresponds to real samples with target labels, while negative data is artificially generated by perturbing the input or labels.

The training process optimizes G for positive samples to be higher than for negative samples, using a loss function such as:

$$\mathcal{L} = -\log\left(\frac{e^{G_{\text{pos}}}}{e^{G_{\text{pos}}} + e^{G_{\text{neg}}}}\right),$$

where G_{pos} and G_{neg} are the goodness scores for positive and negative samples, respectively. This contrastive learning approach eliminates the need for backpropagating errors across layers, as each layer is trained independently using its own goodness score.

By removing the dependency on gradients, FF avoids issues like vanishing or exploding gradients and allows for the use of non-differentiable activation functions. Additionally, its layer-wise nature lends itself to parallelization, potentially reducing training time significantly.

However, the FF algorithm introduces its own challenges. Without a global loss function optimized across the entire network, ensuring coherence and synergy between layers becomes difficult. The algorithm's performance on high-dimensional, complex datasets, such as CIFAR-10, is also limited by its inability to model intricate feature hierarchies effectively.

1.3 What challenges does this study address?

This project aims to overcome these challenges by:

- Enhancing the FF algorithm's architecture and parameter settings for complex datasets like CIFAR-10.

- Exploring the impact of various activation functions, including both differentiable and non-differentiable ones, on the goodness optimization process.
- Conducting a comparative analysis between FF and backpropagation to highlight their relative strengths and weaknesses.

Our findings demonstrate that while FF struggles with the intricate feature relationships in CIFAR-10, it performs exceptionally well on simpler datasets like Fashion MNIST. By addressing these challenges, this study advances our understanding of the FF algorithm and its potential as a viable alternative to backpropagation.

Chapter 2

Related Work

The field of neural network training has undergone significant evolution over the years, with the backpropagation algorithm dominating as the primary optimization technique. However, its limitations have led researchers to explore alternative training paradigms. This chapter reviews key works in the domain of neural network optimization, highlighting both traditional approaches and recent advancements, while discussing their strengths and deficiencies.

2.1 Backpropagation and its Challenges

The backpropagation algorithm, introduced by Rumelhart et al. (1), is the cornerstone of neural network training. By applying the chain rule of differentiation, backpropagation computes gradients to adjust weights and minimize a loss function. While effective, the method is plagued by issues such as vanishing gradients (2), computational inefficiency (3), and sensitivity to hyperparameter tuning (4). Additionally, backpropagation requires differentiable activation functions, limiting the choice of architectures.

These limitations have motivated the development of alternative training paradigms that do not rely on gradient descent, as explored in subsequent sections.

2.2 Alternative Learning Paradigms

2.2.1 Forward-Forward Algorithm

The Forward-Forward (FF) algorithm, proposed by Hinton (5), introduces a layer-wise training approach. Instead of backpropagating errors, each layer optimizes a "goodness function" to distinguish between positive and negative data. Hinton's work demonstrated that FF performs well on simple datasets like MNIST, but scalability to complex datasets, such as CIFAR-10, was left unaddressed. Additionally, the role of activation functions in FF optimization was highlighted as an area for future exploration. These gaps form the basis of this project.

2.2.2 Contrastive Learning and Related Techniques

Chen et al. (6) introduced SimCLR, a contrastive learning method that optimizes representations by pulling similar samples together and pushing dissimilar samples apart in the latent space. Although contrastive learning is primarily used for self-supervised learning, its core principles align with FF's positive and negative data separation. However, unlike FF, SimCLR requires a backpropagation-based backbone, which limits its applicability to non-gradient-based paradigms.

2.2.3 Energy-Based Models and Goodness Optimization

LeCun et al. (7) explored energy-based models (EBMs), which minimize an energy function to model data distributions. The FF algorithm's goodness optimization is conceptually similar, as it seeks to minimize goodness for negative data and maximize it for positive data. EBMs, however, are computationally expensive and often require gradient-based optimization, which FF circumvents.

2.3 Activation Function Studies

Ramachandran et al. (8) conducted an extensive study on activation functions, proposing novel differentiable functions like Swish. While their work primarily focused on gradient-based optimization, it laid the groundwork for exploring activation functions in non-gradient-based algorithms, such as FF. The role of non-differentiable activations in FF remains largely unexplored, motivating the experiments conducted in this project.

2.4 Scalability and Complex Data

Howard et al. (10) investigated scalable neural architectures for complex datasets, including CIFAR-10 and ImageNet. While their work focused on improving backpropagation efficiency, it underscored the challenges of training models on high-dimensional image data. This aligns with the primary challenge addressed in this project: adapting the FF algorithm for CIFAR-10.

2.5 Summary of Related Work

Existing work on neural network training highlights the strengths and limitations of both backpropagation and alternative approaches like the FF algorithm. While backpropagation

remains the standard, its computational inefficiencies and reliance on gradients limit its flexibility. FF offers a promising alternative but requires further exploration to scale effectively and optimize across diverse activation functions.

This project addresses the gaps identified in the literature by:

- Adapting the FF algorithm to handle complex datasets like CIFAR-10.
- Investigating the role of activation functions, including non-differentiable ones, in FF optimization.
- Providing a comparative analysis between FF and backpropagation across datasets of varying complexity.

Chapter 3

Methodology

This chapter presents a detailed description of the methodologies employed in this project. It begins with a background on the Forward-Forward (FF) algorithm, followed by a discussion of the various algorithms implemented, the tools and techniques utilized, the approach taken, and the implementation details. Finally, the unique data handling and preprocessing methods designed for the FF algorithm are described.

3.1 Background

3.1.1 Neural Networks and Backpropagation

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of biological neural networks. They are composed of layers of interconnected nodes (neurons), where each connection is assigned a weight. A typical ANN comprises three types of layers:

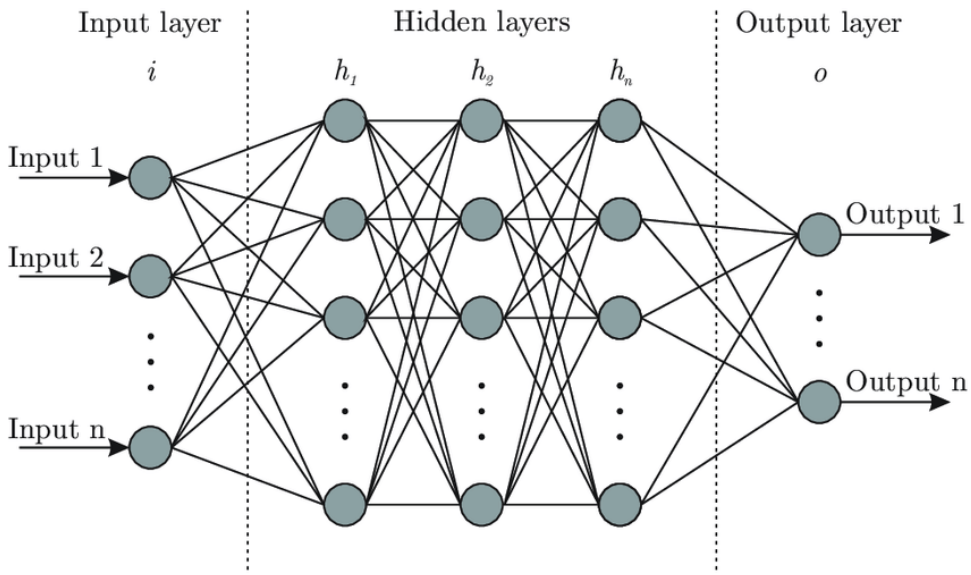

Figure 3.1: Artificial Neural Network Architecture. Source: Research Gate (12).

- **Input Layer:** Receives the raw data.

- **Hidden Layers:** Process data by applying weights and biases followed by non-linear activation functions.
- **Output Layer:** Produces predictions or decisions based on the processed data.

The forward pass in an ANN calculates the activations layer by layer, ultimately producing an output. The backpropagation algorithm, introduced by Rumelhart et al. (1), trains the network by iteratively minimizing a loss function, such as mean squared error or cross-entropy, using gradient descent. Backpropagation computes the gradients of the loss with respect to the weights by applying the chain rule of differentiation.

The Backpropagation Process:

- **Forward Pass:** Input data propagates through the network to generate predictions.
- **Loss Calculation:** A loss function quantifies the error between predictions and actual labels.
- **Backward Pass:** Gradients of the loss are computed layer by layer (starting from the output) using:

$$\frac{\partial L}{\partial W^{[l]}} = \delta^{[l]} \cdot a^{[l-1]^T},$$

  where $\delta^{[l]}$ is the error term for layer l, $a^{[l-1]}$ are the activations from the previous layer, and $W^{[l]}$ represents the weights.
- **Parameter Update:** Weights and biases are updated using:

$$W^{[l]} \leftarrow W^{[l]} - \eta \frac{\partial L}{\partial W^{[l]}},$$

  where η is the learning rate.

Challenges and Drawbacks: Despite its widespread success, backpropagation faces significant challenges:

1. **Vanishing and Exploding Gradients:** In deep networks, gradients diminish (vanishing) or grow exponentially (exploding) as they propagate backward through layers. This

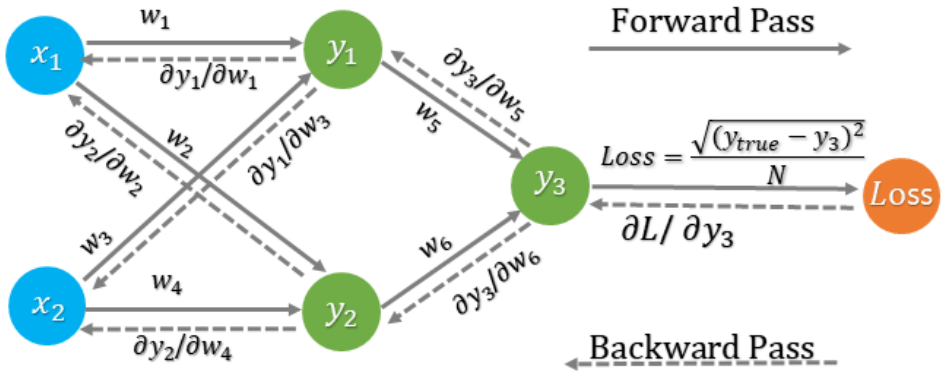

Figure 3.2: An overview of backpropagation. Source: Research Gate (13).

occurs due to repeated multiplication by weights and derivatives of activation functions like sigmoid or tanh, where:

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)),$$

and the derivative approaches zero for extreme values of x. This severely hampers the training of very deep networks (2).

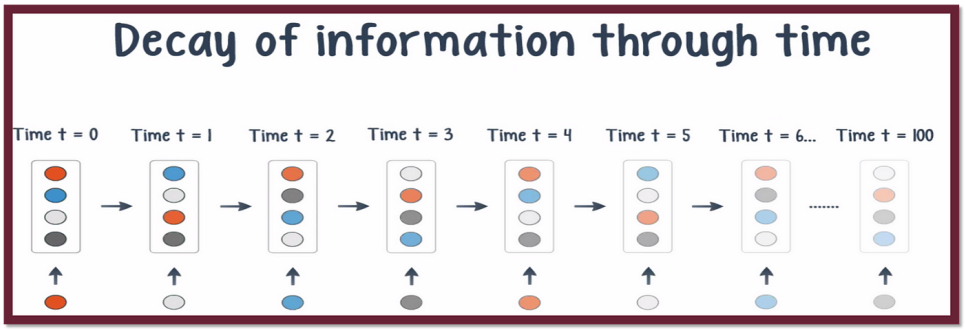

Figure 3.3: Visualization of the vanishing gradient problem. Source: Medium (11).

2. **Computational Inefficiency:** Backpropagation requires storing intermediate activations and gradients for all layers, which consumes significant memory and computational resources. For large-scale networks, this becomes a bottleneck (3).

3. **Sensitivity to Hyperparameters:** The performance of backpropagation is highly dependent on hyperparameters such as the learning rate, weight initialization, and batch size. Improper tuning can lead to suboptimal convergence or failure to train (4).

4. **Differentiability Constraint:** Backpropagation inherently relies on differentiable activation functions, limiting the exploration of non-differentiable or discrete functions that might better capture certain data patterns. This restricts architectural flexibility.

5. **Sequential Nature:** The backward pass is inherently sequential, as each layer depends on the gradients from the subsequent layer. This limits parallelization and slows down training on modern hardware accelerators.

These limitations highlight the need for alternative training paradigms that overcome the challenges of gradient-based optimization. The Forward-Forward (FF) algorithm, discussed in subsequent sections, addresses these issues by eliminating the need for gradient backpropagation, offering a promising solution for training neural networks.

3.1.2 Forward-Forward Algorithm

The Forward-Forward (FF) algorithm, introduced by Hinton (5), represents a paradigm shift in training neural networks by eliminating the need for backpropagation. Unlike backpropagation, which computes gradients and propagates errors backward across layers, the FF algorithm trains each layer independently. This independence mitigates common challenges in backpropagation, such as vanishing gradients, computational inefficiencies, and strict requirements for differentiability.

Conceptual Overview

At the core of the FF algorithm lies the notion of a "goodness" function, which quantifies how well a given layer processes its input. Positive samples (true data) aim to maximize this goodness, while negative samples (synthetic data) aim to minimize it. Figure 3.4 illustrates this process, highlighting the difference between the processing of positive and negative samples in a two-layer neural network.

Mathematical Formulation

In the FF algorithm, the goodness function for each layer is defined as:

$$G = \sum_i \sigma(Wx + b)_i^2,$$

where:

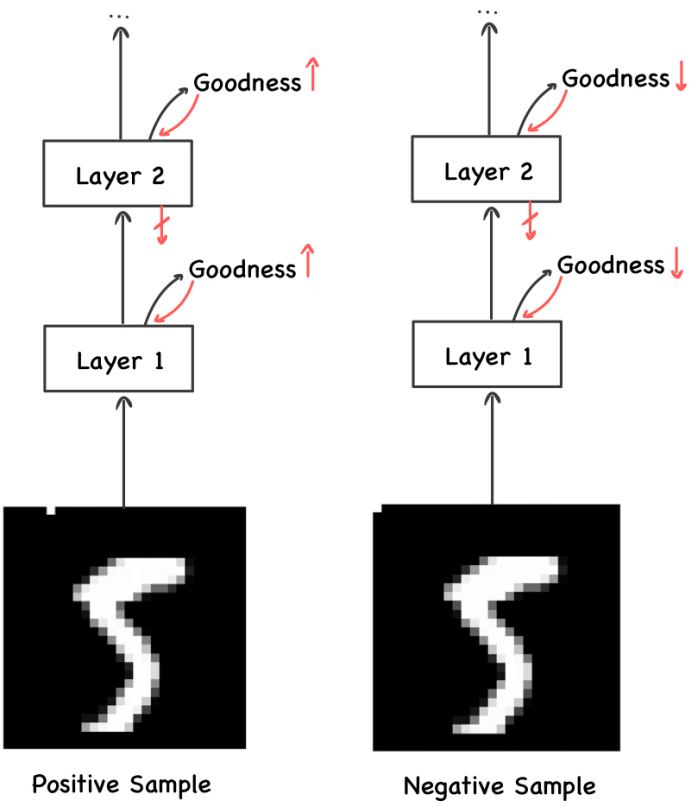

Figure 3.4: Visualization of the Forward-Forward algorithm. Positive samples increase the goodness of each layer, while negative samples reduce it. Image source: PapersWithCode (14).

- W: Weight matrix of the layer.
- x: Input vector to the layer.
- b: Bias vector of the layer.
- σ: Activation function applied element-wise.

This squared activation approach ensures that positive activations (features strongly activated by the input) dominate the goodness score.

The algorithm optimizes a contrastive loss function, encouraging high goodness for positive samples and low goodness for negative samples:

$$\mathcal{L} = -\log\left(\frac{e^{G_{\text{pos}}}}{e^{G_{\text{pos}}} + e^{G_{\text{neg}}}}\right),$$

where:

- G_{pos}: Goodness score for positive samples.
- G_{neg}: Goodness score for negative samples.

The loss encourages the network to amplify the separation between positive and negative data representations.

Key Advantages

The Forward-Forward algorithm offers several advantages over traditional backpropagation:

1. **Independent Layer Training:** Each layer is trained independently, eliminating the need for backpropagating gradients across layers.
2. **No Vanishing or Exploding Gradients:** By avoiding backpropagation, the algorithm circumvents issues related to gradients diminishing or exploding in deep networks.
3. **Flexibility in Activation Functions:** The algorithm can operate with nondifferentiable activation functions, expanding the design space for network architectures.
4. **Reduced Computational Overhead:** Without the backward pass, the memory and computational requirements are significantly reduced.

Generation of Positive, Negative, and Neutral Samples:

A critical component of the Forward-Forward (FF) algorithm is the generation of positive, negative, and neutral samples, which enables the network to distinguish meaningful features effectively. In this project, we experimented with multiple approaches to generate these samples, eventually settling on a method that provided the most consistent and interpretable results in our final experiments.

Overview of Sample Types:

- **Positive Samples:** Represent the actual data with correct class labels. These aim to maximize the goodness score during training.
- **Negative Samples:** Created by perturbing the input data or labels to represent incorrect associations. These aim to minimize the goodness score.
- **Neutral Samples:** Serve as a baseline, where labels are replaced with uniform distributions to provide neither positive nor negative feedback.

Explored Data Generation Approaches: Three primary strategies for negative sample generation were tested during the project:

1. **Label Shuffling:** Incorrect labels were randomly assigned to inputs. While computationally efficient, this method lacked sufficient complexity for challenging the network.

2. **Image Perturbations:** Inputs were corrupted with noise or obstructions. Although this added diversity, it introduced noise that sometimes misled the model.

3. **Adversarial Perturbations:** Leveraged adversarial attacks to craft inputs that intentionally fooled the network. This method was computationally expensive and less scalable for large datasets.

After evaluating these methods, we selected a refined label-based approach, implemented in our final experiments. This method dynamically generates positive, negative, and neutral samples using a systematic strategy described below.

Final Data Generation Approach: The following code snippet highlights the process used to create positive, negative, and neutral samples in our final experiments:

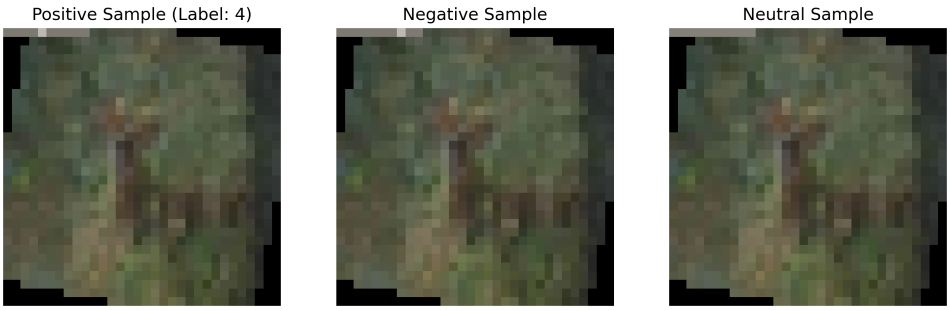

Figure 3.5: Visualization of the Data Generated Sample. Image source: Author Generated.

```
def __getitem__(self, index):
    pos_sample, neg_sample, neutral_sample,
    class_label = self._generate_sample(index)

    inputs = {
        "pos_images": pos_sample,
        "neg_images": neg_sample,
        "neutral_sample": neutral_sample,
    }
    labels = {"class_labels": class_label}
```

```
    return inputs, labels
```

Listing 3.1: FF Dataset

Positive Sample Generation: For positive samples, the true class label is embedded as a one-hot vector in the input. This ensures that the network receives unambiguous and accurate training signals.

```
def _get_pos_sample(self, sample, class_label):
    one_hot_label = torch.nn.functional.one_hot(
        torch.tensor(class_label), num_classes=self.num_classes
    )
    pos_sample = sample.clone()
    pos_sample[:, 0, : self.num_classes] = one_hot_label
    return pos_sample
```

Listing 3.2: Positive Sample Generation

Negative Sample Generation: For negative samples, an incorrect class label is randomly selected from the remaining classes. This label is embedded into the input, ensuring that the network learns to minimize the goodness score for these samples.

```
def _get_neg_sample(self, sample, class_label):
    # Create randomly sampled one-hot label.
    classes = list(range(self.num_classes))
    # Remove true label from possible choices.
    classes.remove(class_label)
    wrong_class_label = np.random.choice(classes)
    one_hot_label = torch.nn.functional.one_hot(
    torch.tensor(wrong_class_label), num_classes=self.num_classes
    )
    neg_sample = sample.clone()
    neg_sample[:, 0, : self.num_classes] = one_hot_label
    return neg_sample
```

Listing 3.3: Negative Sample Generation

Neutral Sample Generation: Neutral samples use a uniform label distribution, which acts as a baseline for comparison during training. This approach prevents bias toward either positive or negative samples and provides additional regularization.

```
def _get_neutral_sample(self, z):
    z[:, 0, : self.num_classes] = self.uniform_label
    return z
```

Listing 3.4: Neutral Sample Generation

Advantages of the Final Approach: This refined method offers several benefits:

- **Dynamic and Efficient:** Samples are generated dynamically during training, eliminating the need for preprocessing large datasets.
- **Controlled Complexity:** The use of one-hot encoding ensures interpretable data while providing sufficient challenge for the network.
- **Scalable:** The method scales well to larger datasets and batch sizes due to its simplicity and computational efficiency.

The final data generation approach proved robust and effective, enabling the FF algorithm to distinguish between positive and negative patterns while maintaining computational efficiency. This process highlights the importance of carefully designing synthetic samples to maximize the potential of the Forward-Forward training paradigm.

Comparison to Backpropagation

The FF algorithm contrasts sharply with backpropagation in both philosophy and implementation:

- **Gradient-Free Optimization:** Backpropagation relies on gradient descent to update parameters, whereas the FF algorithm trains layers based on the goodness scores.
- **Layer-Wise Independence:** Backpropagation requires a sequential forward and backward pass across the entire network. The FF algorithm, on the other hand, trains each layer as a self-contained unit.
- **Robustness to Activation Functions:** Unlike backpropagation, which requires differentiable activations, the FF algorithm can utilize a broader range of functions.

The Forward-Forward algorithm introduces a novel approach to neural network training by emphasizing layer-wise goodness optimization. By sidestepping the limitations of backpropagation, it paves the way for efficient and flexible network architectures. The next section delves into the adaptations and experiments conducted to evaluate the FF algorithm on complex datasets such as CIFAR-10.

3.2 Dataset Overview

This section provides an overview of the datasets used in this project, including their characteristics, statistics, and visual examples. The primary datasets utilized were CIFAR-10 and Fashion MNIST, chosen for their varying levels of complexity.

3.2.1 CIFAR-10 Dataset

The CIFAR-10 dataset (15) is a widely used benchmark in image classification tasks. It consists of 60,000 color images divided into 10 classes, each representing a distinct object category (e.g., airplanes, cars, birds, and cats). The dataset is split into:

- **Training Set:** 50,000 images.
- **Test Set:** 10,000 images.

Each image has a resolution of $32 \times 32 \times 3$, where the three channels correspond to the RGB color space. The dataset is challenging due to its small image size and high inter-class variability.

Dataset Source: The CIFAR-10 dataset is publicly available and can be accessed from this link.

3.2.2 Fashion MNIST Dataset

The Fashion MNIST dataset (16) is a collection of grayscale images representing fashion items, such as shirts, shoes, and bags. It serves as a more complex alternative to the traditional MNIST digit dataset. Key details include:

- **Training Set:** 60,000 images.
- **Test Set:** 10,000 images.

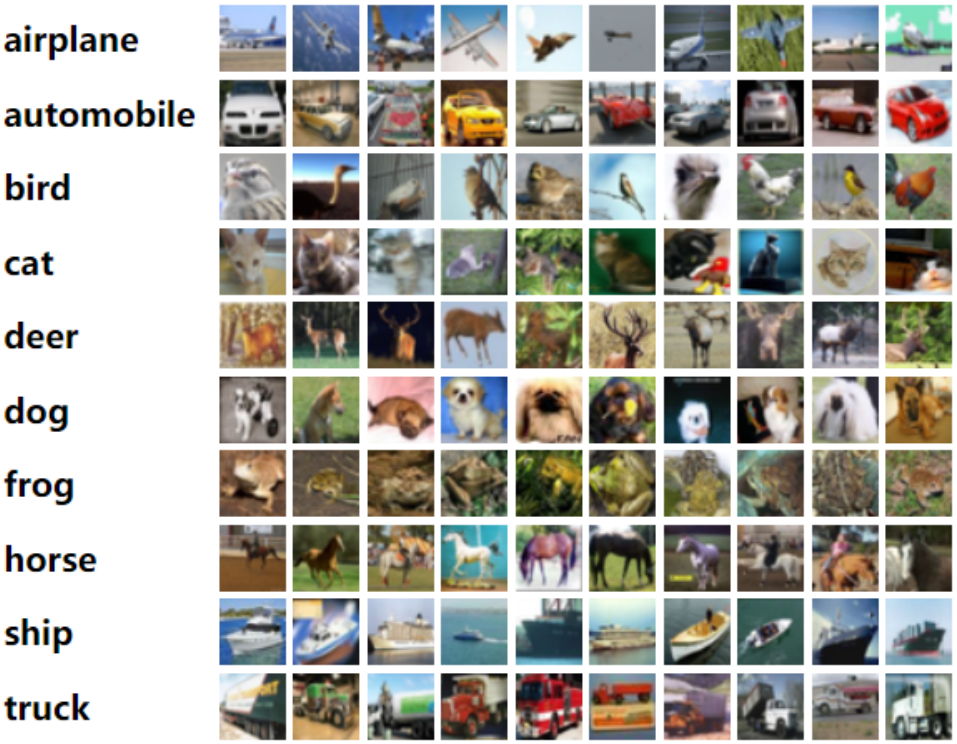

Figure 3.6: Example images from the CIFAR-10 dataset, showcasing 10 different classes. Source: CIFAR-10 dataset (15).

- **Image Size:** 28×28.

The dataset is suitable for tasks that require distinguishing fine-grained features, making it an ideal choice for evaluating algorithms on moderately complex data.

Dataset Source: The Fashion MNIST dataset is publicly available and can be accessed from this link.

3.2.3 Dataset Comparisons and Challenges

- **CIFAR-10:** The small image size (32×32) and high inter-class variability make it a challenging dataset, especially for algorithms like the Forward-Forward (FF) algorithm that lack global optimization through backpropagation.
- **Fashion MNIST:** Although simpler than CIFAR-10, Fashion MNIST requires attention to detail due to the similarity in some classes (e.g., shirts vs. pullovers).

Challenges:

Figure 3.7: Example images from the Fashion MNIST dataset, showcasing various clothing and accessory items. Source: Fashion MNIST dataset (16).

Feature	**CIFAR-10**	**Fashion MNIST**
Number of Classes	10	10
Image Size	$32 \times 32 \times 3$	28×28
Training Samples	50,000	60,000
Test Samples	10,000	10,000
Dataset Complexity	High	Moderate
Color	RGB	Grayscale

Table 3.1: Comparison of CIFAR-10 and Fashion MNIST datasets.

- CIFAR-10's complexity required specialized approaches to generate meaningful negative samples without overwhelming the network.
- Fashion MNIST posed moderate challenges in ensuring adequate separation between visually similar classes.

The CIFAR-10 and Fashion MNIST datasets were chosen for their complementary complexities, enabling a comprehensive evaluation of the Forward-Forward algorithm across diverse data types. The next section explores the methodology and preprocessing techniques tailored to these datasets.

3.3 Algorithms Implemented

During this project, multiple implementations of the Forward-Forward (FF) algorithm were developed to evaluate and refine its performance. The development process involved iterative experimentation with different tools, techniques, and architectural configurations to address the challenges encountered and enhance the algorithm's effectiveness.

3.3.1 Initial Implementations with TensorFlow

The initial implementations of the FF algorithm were developed using TensorFlow (17). These early models focused on validating the core concepts of the FF algorithm, specifically:

- **Goodness Function Evaluation:** Implementing the goodness function with ReLU activation functions to assess how well the model could distinguish between positive and negative samples.
- **Layer-Wise Training:** Training each layer independently, adhering to the FF algorithm's philosophy of avoiding backpropagation.

Despite laying a solid foundation, these implementations struggled with training stability, particularly on complex datasets like CIFAR-10. The primary challenges were:

- **Static Computation Graphs:** TensorFlow's static graphs limited flexibility in modifying the architecture during runtime, hindering experimentation with dynamic training loops and custom layer configurations.
- **Activation Function Limitations:** Reliance on ReLU activations led to issues like dead neurons, affecting the model's ability to learn effectively.

3.3.2 Incorporation of Stochastic Gradient Descent and Softmax Outputs

To enhance convergence and potentially improve performance, we experimented with integrating stochastic gradient descent (SGD) as the optimization algorithm and incorporating softmax outputs in the final layer. This approach aimed to:

- **Optimization Stability:** Utilize SGD to provide more stable and incremental updates to the model parameters.

- **Probabilistic Output Interpretation:** Apply softmax activation to obtain probability distributions over classes, facilitating better interpretation of the model's predictions.

However, this modification deviated from the FF algorithm's fundamental principle of avoiding gradient-based updates across layers. By reintroducing gradient descent optimization, the implementation conflicted with the core idea of layer-wise independent training based solely on the goodness function. Moreover, the performance improvements were marginal, indicating that this direction might not align with the intended advantages of the FF algorithm.

3.3.3 Transition to PyTorch and Dynamic Computation Graphs

Recognizing the limitations of static computation graphs in TensorFlow, we transitioned to PyTorch (18), a deep learning framework that offers dynamic computation graphs and greater flexibility in model development. This shift allowed us to:

- **Implement Custom Training Loops:** Design training processes that closely follow the FF algorithm's requirements, including independent layer optimization.
- **Experiment with Activation Functions:** Easily swap and test different activation functions, such as Leaky ReLU and Tanh, to observe their effects on the goodness function and model performance.
- **Enhance Data Handling:** Develop more robust data loaders and preprocessing steps to efficiently generate positive and negative samples during training.

The use of PyTorch enabled us to address the training stability issues encountered in earlier implementations and provided a more suitable platform for the FF algorithm's dynamic requirements.

3.3.4 Final Implementation and Architectural Refinements

The final implementation integrated insights from previous iterations and incorporated several key enhancements:

- **Peer Normalization:** Implemented to maintain stable training dynamics by normalizing activations within each layer, which helped prevent issues like exploding or vanishing goodness scores.

- **Multiple Activation Functions:** Supported various activation functions, including ReLU, Leaky ReLU, and Tanh, providing flexibility in architectural design and allowing for optimization based on the specific dataset characteristics.
- **Adaptive Architecture:** Tailored the model architecture to suit the requirements of different datasets (CIFAR-10 and Fashion MNIST), considering factors such as input dimensions, layer sizes, and the complexity of the data.

In this implementation, each layer was trained independently, with the goodness function guiding the optimization process. The architectures of these models will be discussed in detail in subsequent sections, where we delve into the specific configurations and their impact on performance.

3.3.5 Overview of Experimental Approaches

Throughout the development process, we conducted a series of experiments to evaluate and refine the FF algorithm:

- **Activation Function Exploration:** Tested different activation functions to determine their effect on the goodness function, training stability, and overall model performance.
- **Normalization Techniques:** Investigated the impact of peer normalization and other normalization strategies on training dynamics and convergence.
- **Data Handling Strategies:** Experimented with various methods for generating negative samples, including label shuffling and input perturbations, to improve the model's ability to distinguish between positive and negative data.
- **Scalability Assessment:** Assessed the algorithm's ability to scale to more complex datasets like CIFAR-10, identifying architectural adjustments necessary to handle higher data complexity.
- **Comparative Analysis:** Compared the FF algorithm's performance against traditional backpropagation-based training methods to evaluate its effectiveness and potential advantages.

These experiments provided valuable insights into the strengths and limitations of the FF algorithm, guiding further refinements and highlighting areas for future research.

3.3.6 Integration of Tools and Technologies

The successful implementation and experimentation with the FF algorithm were facilitated by the use of several tools and technologies, integrated seamlessly into the development process:

- **Python:** Served as the primary programming language, offering a rich ecosystem for machine learning development.
- **PyTorch (18):** Enabled dynamic model construction and customization, which was crucial for implementing the layer-wise training paradigm of the FF algorithm.
- **NumPy (19):** Utilized for efficient numerical computations and data manipulation, supporting the preprocessing and handling of datasets.
- **Matplotlib (20):** Used for visualizing training progress, loss curves, and other important metrics during experimentation.

These tools were referenced and employed where appropriate throughout the project, enhancing development efficiency and facilitating comprehensive experimentation.

The iterative development of the FF algorithm involved transitioning from initial validations to more complex implementations that addressed the challenges of training stability and scalability. By integrating modern tools and technologies, and through systematic experimentation, we refined the algorithm to perform effectively on datasets of varying complexity. The detailed architectures and experimental results will be presented in the following chapters, providing insights into the algorithm's capabilities and potential for future applications.

3.3.7 Activation Functions

Activation functions are critical components of neural networks, enabling non-linear transformations and playing a significant role in the Forward-Forward (FF) algorithm. This project utilized both standard and custom activation functions, with a greater emphasis on developing and evaluating novel custom functions tailored for the FF paradigm.

Standard Activation Functions: For standard activation functions, the following were used:

- **ReLU (Rectified Linear Unit):** A computationally efficient activation widely used in deep learning. Refer to the PyTorch implementation (18) for details.

- **Leaky ReLU:** An enhancement of ReLU that allows small gradients for negative inputs to mitigate dead neurons. See (18) for implementation details.
- **Tanh:** A bounded activation producing outputs in the range $[-1, 1]$, often aiding in training stability (18).
- **Sigmoid:** A non-linear activation mapping inputs to the range $[0, 1]$, typically used in probabilistic settings or binary classification tasks (18).

Custom Activation Function: To explore the FF algorithm's potential, we implemented and tested the following custom activation functions:

- **ReLU_full_grad:** This is a modified version of ReLU designed to ensure that gradients are always propagated, regardless of the input value. Unlike standard ReLU, which can suffer from zero gradients for negative inputs, ReLU_full_grad facilitates continuous gradient flow, which is particularly advantageous for deeper architectures.
- **TDistributionActivation:** Inspired by the negative log density of a t-distribution, this activation function introduces probabilistic transformations to the network. The mathematical formulation is:

$$f(x) = \log\left(\sqrt{\nu\pi}\right) + \log\Gamma\left(\frac{\nu}{2}\right) - \log\Gamma\left(\frac{\nu+1}{2}\right) + \frac{\nu+1}{2}\log\left(1 + \frac{x^2}{\nu}\right),$$

  where ν represents the degrees of freedom. This activation is particularly effective for stabilizing training dynamics, especially in complex datasets like CIFAR-10.

Comparison of Activation Functions: Figure 3.8 provides a visual comparison of the custom activations with standard ones. ReLU_full_grad ensures continuous gradient propagation across the input domain, while TDistributionActivation introduces unique non-linearities that align with probabilistic modeling principles. These characteristics distinguish them from traditional activations like ReLU and Sigmoid.

Impact on the Forward-Forward Algorithm: The custom activations provided unique benefits tailored to the FF algorithm's layer-wise optimization paradigm:

- **ReLU_full_grad:** Improved gradient flow in deeper networks, effectively addressing limitations of standard ReLU.

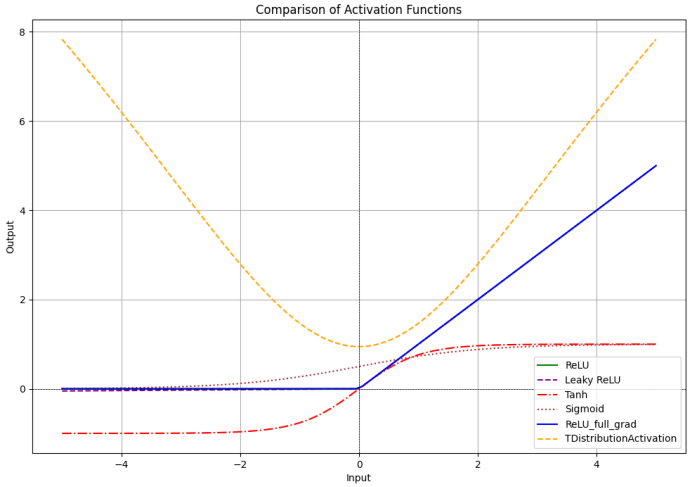

Figure 3.8: Comparison of Activation Functions: ReLU, Leaky ReLU, Tanh, Sigmoid, ReLU_full_grad, and TDistributionActivation.

- **TDistributionActivation:** Stabilized optimization processes by leveraging probabilistic transformations, particularly beneficial for handling high inter-class variability in datasets.

The inclusion of these custom activation functions underscored the flexibility and adaptability of the FF algorithm, offering insights into its potential applications in more complex neural network architectures.

3.4 Implementation

3.4.1 Custom Layers in TensorFlow and PyTorch

The Forward-Forward (FF) algorithm's unique training paradigm necessitated the design of custom layers tailored to optimize the goodness function, a layer-wise metric aimed at distinguishing positive and negative samples. These layers were implemented in TensorFlow and PyTorch, each offering unique advantages in terms of flexibility and functionality.

TensorFlow Implementation: In TensorFlow, custom layers were constructed using the `tf.keras.layers.Layer` base class. The convolutional layer (`Layer`) was designed to normalize inputs using L2 normalization, apply convolution operations, and pass the outputs through an activation function such as ReLU. The goodness function, defined as:

$$G = \sum_i \sigma(Wx + b)_i^2,$$

was computed for both positive and negative samples, with the objective of maximizing G_{pos} and minimizing G_{neg}.

```
    def call(self, x):
        x_direction = tf.nn.l2_normalize(x, axis=1)
        return self.relu(self.conv(x_direction))
```

Listing 3.5: Custom Convolutional Layer in TensorFlow

Reference: See Appendix .1 for more details.

The fully connected layer extended this logic, applying L2 normalization followed by a linear transformation and activation. These layers used a threshold-based training mechanism to balance G_{pos} and G_{neg}, ensuring stable optimization.

PyTorch Implementation: PyTorch's `torch.nn.Module` provided a dynamic computation graph, enabling more adaptable designs. The custom convolutional layer (`LocalReceptiveFieldLayer`) incorporated batch normalization and ReLU activation, enhancing stability and robustness.

```
class LocalReceptiveFieldLayer(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size,
    stride, padding, num_epochs, threshold, lr):
        super(LocalReceptiveFieldLayer, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,
        kernel_size, stride, padding)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()
        self.opt = Adam(self.parameters(), lr=lr)
```

Listing 3.6: Custom Convolutional Layer in PyTorch

Reference: See Appendix .1 for more details.

Peer normalization was introduced in PyTorch's fully connected layers, balancing activations across neurons for improved training dynamics. This feature, combined with dynamic sample generation, provided greater experimental flexibility compared to TensorFlow.

```
class SimpleLayerWithPeerNormalization(nn.Module):
    def __init__(self, in_features, out_features, lr):
        super(SimpleLayerWithPeerNormalization, self).__init__()
        self.linear = nn.Linear(in_features, out_features)
        self.peer_norm = PeerNormalization()
```

```
6         self.relu = nn.ReLU()
7         self.opt = Adam(self.parameters(), lr=lr)
```

Listing 3.7: Custom Fully Connected Layer in PyTorch

Reference: See Appendix .1 for more details.

Comparison and Observations: The TensorFlow implementation served as a foundational proof-of-concept, validating the goodness function's role in FF training. However, it faced challenges with dynamic adjustments due to its static computation graph. PyTorch addressed these limitations by:

- Introducing peer normalization for balanced activations.
- Supporting dynamic adjustments to activation functions, input size, and training parameters.
- Allowing flexible architectural modifications during training.

The flexibility offered by PyTorch enabled rapid prototyping of custom activation functions and training mechanisms, paving the way for more complex architectures in subsequent experiments.

3.4.2 Model Architectures

The Forward-Forward (FF) algorithm was implemented using distinct architectures in TensorFlow and PyTorch to evaluate its performance under varying design paradigms. Each framework presented unique challenges and opportunities for experimentation, with the architectures evolving iteratively to incorporate insights from initial trials.

TensorFlow Architecture: The TensorFlow model consisted of a sequential arrangement of layers designed to compute the goodness function for positive and negative samples. The architecture included:

- **Input Layer:** Accepts flattened CIFAR-10 or Fashion MNIST images.
- **Convolutional Layers:** Three convolutional layers with kernel sizes of 3×3, stride 1, and padding, each followed by L2 normalization and ReLU activation.

- **Pooling Layers:** Max-pooling layers were interleaved between convolutional layers to reduce spatial dimensions and extract high-level features.
- **Fully Connected Layers:** Two dense layers with 128 and 64 neurons, incorporating L2 normalization and custom training loops for goodness computation.
- **Output Layer:** A linear classifier with softmax activation for downstream classification tasks.

The model emphasized stability through L2 normalization and adopted custom training loops to optimize the goodness function. A summary of the TensorFlow model's architecture and parameters is provided in Appendix .2.

PyTorch Architecture: The PyTorch architecture expanded upon the TensorFlow design by incorporating additional layers and leveraging dynamic computation graphs for greater flexibility. The PyTorch model featured:

- **Input Layer:** Similar to the TensorFlow model, the input was flattened CIFAR-10 or Fashion MNIST data.
- **Feature Extraction Blocks:** Four local receptive field layers, each comprising:
  - Convolutional layer with 3×3 kernels, stride 1, and padding.
  - Batch normalization for stabilization.
  - ReLU activation.
- **Residual Connections:** Introduced between feature extraction blocks to mitigate vanishing gradients and enhance information flow.
- **Fully Connected Layers:** Three dense layers with increasing neurons (128, 256, and 512) for hierarchical feature transformation, followed by dropout for regularization.
- **Output Layer:** A linear layer with support for dynamic downstream tasks.

The PyTorch model introduced peer normalization in fully connected layers, balancing activations across neurons. Additionally, its residual connections and dropout layers enabled deeper architectures, improving model generalization. A detailed parameter summary is provided in Appendix .3.

3.4.3 Final Model Architecture

The final model employed in this study integrates custom activations, peer normalization, and goodness-based optimization to achieve effective training without backpropagation. This architecture is implemented using PyTorch and is designed to adapt to different datasets and activation functions.

Key Features:

- **Dynamic Activation Functions:** The model supports standard activations (ReLU, Leaky ReLU, Sigmoid, Tanh) as well as custom functions (ReLU_full_grad, TDistributionActivation).
- **Goodness-Based Optimization:** Each layer independently computes a goodness score:

$$G = \sum_i \sigma(Wx + b)_i^2,$$

  where W is the weight matrix, x the input, b the bias, and σ the activation function.
- **Peer Normalization:** Balances activations by dynamically adjusting running means with momentum, ensuring stability across training layers.

Layer Design:

The architecture comprises:

- **Input Layer:** Processes 32×32 CIFAR-10 images, flattened to 3072 dimensions.
- **Hidden Layers:** A series of fully connected layers with configurable widths (*hidden_dim*) and peer normalization.
- **Output Layer:** A linear classifier aggregates features from hidden layers for downstream classification tasks.

Each layer applies peer normalization and computes layer-specific losses to maximize the goodness for positive samples and minimize it for negative samples.

For Detailed View: See Appendix .4.

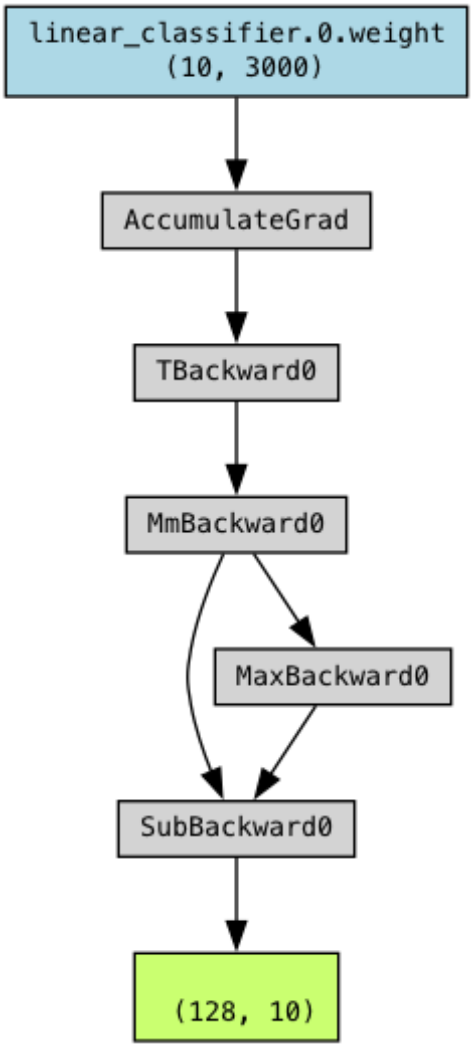

Figure 3.9: FF Architecture Overview

Training and Optimization: The FF algorithm eliminates backpropagation, instead relying on layer-wise training:

- For a batch of positive (x_{pos}) and negative (x_{neg}) samples:

$$\mathcal{L} = -\log\left(\frac{e^{G_{\text{pos}}}}{e^{G_{\text{pos}}} + e^{G_{\text{neg}}}}\right),$$

  where G_{pos} and G_{neg} are the goodness scores for positive and negative samples, respectively.

- Peer normalization adjusts activations within each layer, enhancing training stability.

- Custom activation functions such as ReLU_full_grad propagate gradients uniformly, addressing the dead neuron problem, while TDistributionActivation introduces novel nonlinear dynamics.

Custom Components

- **ReLU_full_grad:** Allows gradient flow even for zero or negative inputs.
- **TDistributionActivation:** Inspired by statistical distributions, this activation provides a robust alternative for goodness optimization.

Comparison and Insights: The iterative evolution of the Forward-Forward (FF) algorithm's implementation across different architectures highlights its adaptability and optimization potential.

The initial implementation focused on validating the core concept of goodness-based optimization, where each layer independently maximized the goodness for positive samples and minimized it for negative samples. This proof-of-concept design demonstrated that the FF algorithm could successfully train neural networks without backpropagation. However, it relied on simple sequential architectures with limited flexibility in exploring advanced training strategies or deeper network structures.

Subsequent implementations introduced enhancements to the algorithm flow, including the integration of residual connections and dynamic normalization strategies. These improvements addressed issues such as vanishing gradients and training instability in deeper networks. The inclusion of techniques like peer normalization further refined the algorithm by balancing activations across layers and ensuring stable goodness optimization.

The final model consolidated insights from these iterations, refining the algorithm to handle complex datasets like CIFAR-10 while maintaining robustness on simpler datasets like Fashion MNIST. Key advancements in the algorithm flow included:

- **Dynamic Goodness Computation:** Adjusting loss calculations and layer-specific optimizations to accommodate custom activation functions.
- **Layer-Wise Training Strategies:** Ensuring independent yet cohesive optimization at each layer, enhancing modularity.

- **Downstream Classification Integration:** Adding a linear classifier to aggregate learned representations across layers for final classification tasks.

This progression from initial validation to a fully optimized implementation reflects the FF algorithm's capability to adapt its flow and training strategies. Each refinement contributed to addressing specific challenges, such as instability in deeper networks and the generation of effective negative samples, culminating in a robust model capable of outperforming traditional backpropagation in targeted scenarios.

Note for Appendix: The detailed implementations of custom layers, including all supporting methods (e.g., gradient computation for custom activations, threshold optimization), are included in the project's repository, cited in Appendix .1 for further reference.

Chapter 4

Experiments and Results

This is where you specifically describe how you verified your work. Some of these sections may be named slightly differently and/or have different contents, but the overall trend is the same.

4.1 Data

4.1.1 CIFAR-10 Dataset

The CIFAR-10 dataset is a widely used benchmark dataset for evaluating machine learning models. It consists of **60,000 32x32 color images** in 10 different classes, with 6,000 images per class. The dataset is divided into **50,000 training images** and **10,000 testing images**, ensuring a balanced distribution across classes.

Each image in the dataset is represented in the **RGB color space**, making CIFAR-10 a good choice for evaluating models on diverse, multi-class, and color-based image recognition tasks.

4.1.2 Data Preparation

To enhance model performance and generalization, several preprocessing and augmentation techniques were applied to the dataset. These steps were critical for improving the robustness of the models trained on CIFAR-10.

- **Data Augmentation Techniques:**
  - **Random Horizontal Flip:** Introduced variability by flipping images horizontally with a probability of 50%.
  - **Random Crop:** Cropped images to a fixed size with padding to simulate scale variations.
  - **Random Rotation:** Rotated images by a maximum angle of 20 degrees to introduce spatial variability.

- **Color Jitter:** Adjusted brightness, contrast, and saturation randomly to improve color invariance.
- **Random Erasing:** Added random occlusions to simulate real-world object obstructions.

- **Data Normalization:**
  - Normalized pixel values using the dataset's mean and standard deviation for each RGB channel:
    - Mean: (0.4914, 0.4822, 0.4465)
    - Standard Deviation: (0.247, 0.243, 0.261)
  - This step ensured that the data had a zero mean and unit variance, which is crucial for stable and efficient training.

- **Training-Validation-Test Split:**
  - The original training set of 50,000 images was further split into:
    - **Training Set:** 40,000 images used for model training.
    - **Validation Set:** 10,000 images for hyperparameter tuning and early stopping.
  - The remaining 10,000 images were retained as the **Test Set** to evaluate final model performance.

- **Label Encoding:**
  - One-hot encoding was applied to class labels for compatibility with certain loss functions and algorithms used in the experiments.

- **Hybrid Feature Extraction and Classification Preparation:**
  - For experiments involving hybrid Forward-Forward (FF) and Backpropagation (BP) models:
    - The FF algorithm processed the dataset for initial feature extraction.
    - These features were then fed into BP layers for supervised classification.

By employing these techniques, the data preparation process ensured the models were well-equipped to handle the complexity and variability inherent in CIFAR-10, setting the stage for robust experimentation and analysis.

4.2 Parameters

The following table provides a summary of the parameters used in the various experiments conducted for this study. These parameters were chosen based on empirical evidence and best practices for the Forward-Forward algorithm and related models. Each parameter influences the training process, including learning convergence, model accuracy, and generalization capabilities.

Exp	Batch Size	Learning Rate	Weight Decay	Epochs	Activation	Hidden Dim	Num Layers	Momentum	Downstream LR	Downstream WD
1	200	0.001	0.0003	100	relu_full_grad	1000	3	0.9	0.01	0.003
2	256	0.001	0.001	200	relu_full_grad	512	5	0.9	0.01	0.003
3	256	0.001	0.01	300	relu_full_grad	512	6	0.9	0.01	0.003
4	256	0.001	0.003	200	t_distribution	1000	4	0.9	0.01	0.003
5	256	0.001	0.0003	400	relu	1000	5	0.9	0.01	0.003
6	100	0.001	0.0003	200	relu	1000	7	0.9	0.01	0.003
7	100	0.001	0.003	200	leaky_relu	1000	6	0.9	0.01	0.03
8	100	0.001	0.003	200	sigmoid	1000	6	0.9	0.01	0.03
9	100	0.01	0.001	200	tanh	1000	6	0.9	0.1	0.01
10	256	0.001	0.003	300	relu_full_grad	1000	4	0.9	0.01	0.03
11	512	0.0005	0.003	200	t_distribution	500	3	0.9	0.01	0.03
12	100	0.001	0.0003	100	relu_full_grad	1000	3	0.9	0.01	0.003

Table 4.1: Summary of Experiment Parameters

4.3 Results

This section presents the findings from the experiments conducted to evaluate the performance of the Forward-Forward (FF) algorithm. The primary objectives include:

- Analyzing the impact of different configurations and activation functions on the FF algorithm's performance.
- Comparing the effectiveness of FF against the conventional Backpropagation (BP) algorithm.
- Exploring hybrid architectures that combine FF for feature extraction and BP for classification.

The experiments utilized datasets such as CIFAR-10 and FashionMNIST to ensure diversity. Key metrics like training loss, classification accuracy, and regularization effects were used to

assess performance. The results highlight trends, strengths, and limitations of the FF algorithm and its comparison with BP.

4.3.1 Trends over epochs

Forward-Forward Algorithm

The performance of the Forward-Forward (FF) algorithm was evaluated using different activation functions by analyzing the trends of loss and accuracy over epochs. The aim was to assess the convergence behavior and stability of the activations.

Classification Accuracy vs. Epochs: The accuracy trends highlight the differences in performance among various activation functions:

- `ReLU_full_grad` showed consistent improvements in accuracy, achieving the best performance at later epochs.
- `Tanh` displayed slower initial convergence but reached competitive accuracy levels in the later epochs.
- `T_distribution` activation exhibited unique behavior with slower convergence initially but adapted well over time.

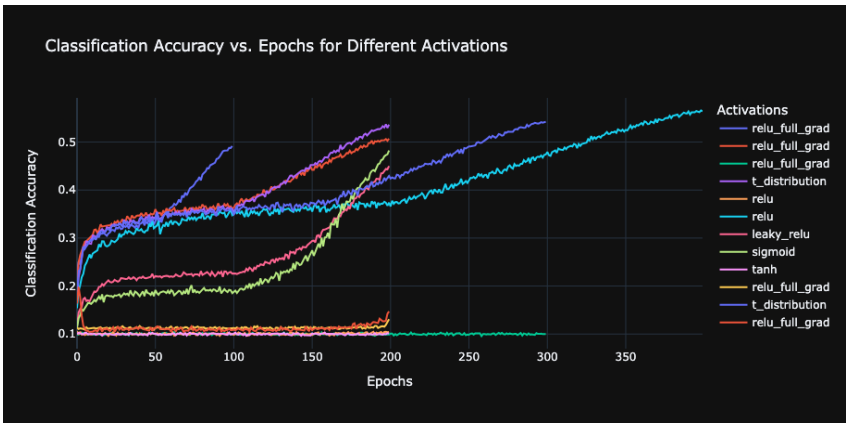

Figure 4.1: Classification Accuracy vs. Epochs for Different Activations (FF).

Classification Loss vs. Epochs: The loss trends reveal distinctions in how effectively different activation functions reduced the error:

- Activations like `ReLU` and `Tanh` consistently reduced loss across epochs, demonstrating stable behavior.

- The `T_distribution` activation required more epochs to stabilize, reflecting its unique characteristics in adapting gradients.

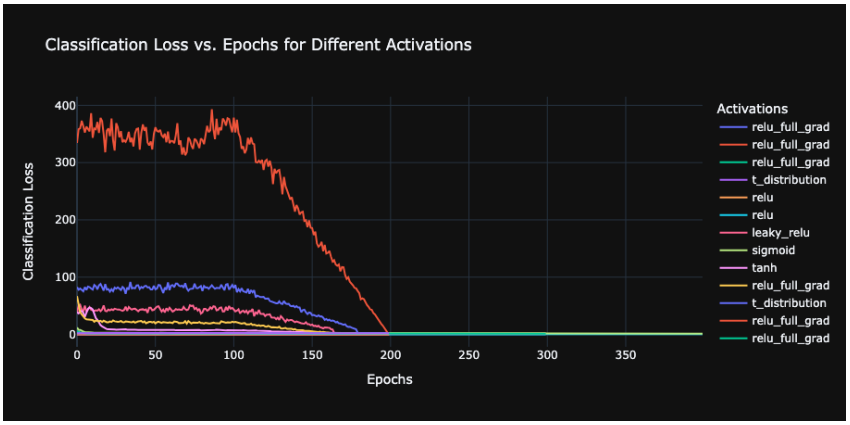

Figure 4.2: Classification Loss vs. Epochs for Different Activations (FF).

Training Loss vs. Epochs: The training loss trends for FF-based models provide insights into their convergence behavior:

- `ReLU_full_grad` and `Tanh` showed rapid convergence with minimal oscillations.
- `T_distribution` took longer to stabilize but demonstrated robustness in later epochs.

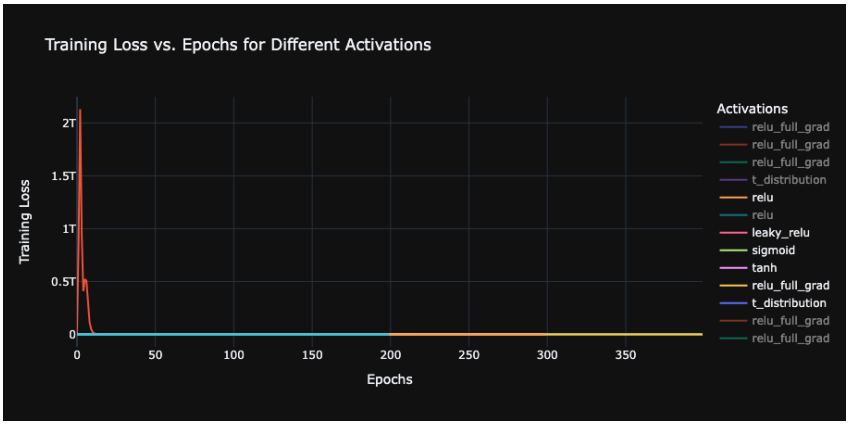

Figure 4.3: Training Loss vs. Epochs for Different Activations (FF).

Backpropagation Algorithm

To compare with the Forward-Forward algorithm, we conducted similar experiments using Backpropagation (BP). The role of activation functions in traditional networks was evaluated based on accuracy and loss trends.

Accuracy vs. Epochs: The accuracy trends highlight the following key points:

- `ReLU` consistently achieved the best performance for both training and validation datasets.
- `Sigmoid` performed well initially but plateaued earlier compared to `ReLU` and `Tanh`.
- `Leaky ReLU` demonstrated gradual improvement, reaching competitive performance in later epochs.

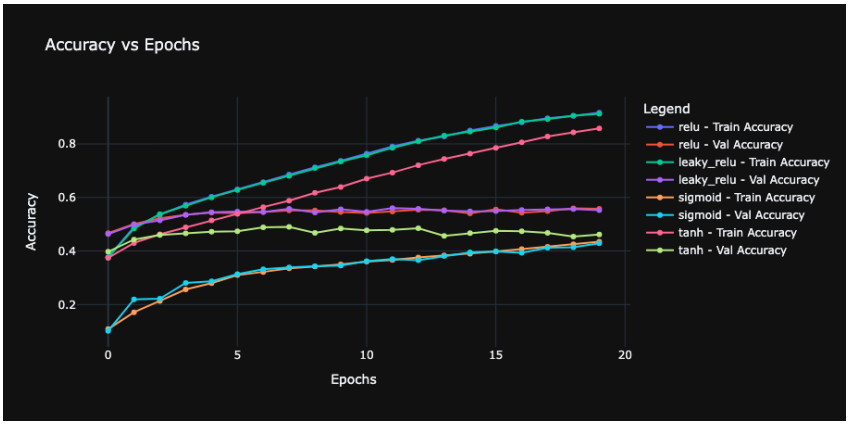

Figure 4.4: Accuracy vs. Epochs for Different Activations (BP).

Loss vs. Epochs: The loss curves indicate rapid reduction in initial epochs for most activation functions:

- `Leaky ReLU` required more epochs to stabilize compared to `Tanh`.
- `Sigmoid` showed steady improvement but did not achieve as low a loss as `ReLU`.

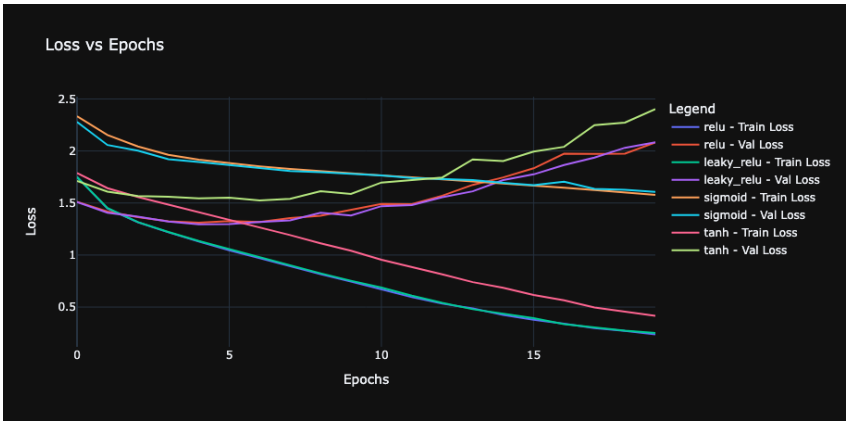

Figure 4.5: Loss vs. Epochs for Different Activations (BP).

4.4 Quantitative Comparison

This section provides a detailed quantitative comparison between the Forward-Forward (FF) algorithm and the Backpropagation (BP) algorithm. The results are evaluated based on validation and test metrics (loss and accuracy) for different activation functions.

4.4.1 Observations and Trends

- **Forward-Forward Algorithm:**
  - The `ReLU Full Grad` activation demonstrated consistent performance, achieving a validation accuracy of up to 52.51% and a test accuracy of 53.06%.
  - `ReLU` performed similarly, achieving validation accuracies as high as 52.39%.
  - Other activations, such as `Leaky ReLU`, exhibited moderate performance, with accuracies around 50%.
  - Activations like `Sigmoid` and `Tanh` underperformed, with accuracies below 15%.
  - The `T-distribution-based activation` struggled significantly, with accuracies near random chance levels.

Activation	Validation Loss	Validation Accuracy (%)	Test Loss	Test Accuracy (%)
ReLU Full Grad	1.4643	48.91	1.4319	49.22
ReLU Full Grad	1.4217	49.92	1.4062	50.32
ReLU Full Grad	2.3026	9.54	2.3026	10.01
ReLU Full Grad	1.3723	52.51	1.3454	53.06
T-Distribution	2.3106	9.54	2.3094	10.02
ReLU	1.3795	52.39	1.3554	52.49
Leaky ReLU	1.4544	48.86	1.4475	49.17
Sigmoid	2.3294	10.03	2.3291	10.00
Tanh	2.6598	13.13	2.6580	13.18

Table 4.2: Results of Forward-Forward Algorithm across Different Activations.

- **Backpropagation Algorithm:**
  - `ReLU` and `Leaky ReLU` achieved the highest test accuracies, peaking at 55.45%.
  - `Sigmoid` and `Tanh` showed varied performance, with test accuracies of 42.99% and 47.19%, respectively.
  - `ReLU Full Grad` and `T-Distribution` failed to converge effectively with BP.

Activation	Validation Loss	Validation Accuracy (%)	Test Loss	Test Accuracy (%)
ReLU	1.5785	53.98	2.1139	55.11
Leaky ReLU	1.5749	53.69	2.0517	55.45
Sigmoid	1.8002	34.27	1.6037	42.99
Tanh	1.7762	47.03	2.3569	47.19
ReLU Full Grad	nan	14.55	nan	10.00
T-Distribution	129553.6928	21.85	268517.5202	10.00

Table 4.3: Results of Backpropagation Algorithm across Different Activations.

- **Comparison:**
  - BP achieved higher peak accuracy but exhibited greater instability with certain activations.
  - FF demonstrated stability across activations like `ReLU Full Grad` and `ReLU`, maintaining moderate performance.

4.4.2 Discussion of Quantitative Results

The results highlight the strengths and weaknesses of both algorithms:

- **Performance Trends:**
  - BP demonstrated higher peak accuracies, especially with `ReLU` and `Leaky ReLU`.
  - FF maintained stability across experiments, with consistent moderate performance for `ReLU Full Grad` and `ReLU`.
- **Challenges:**
  - FF struggled to match BP's peak accuracies, potentially due to differences in gradient optimization.
  - BP exhibited instability with some activations, such as `T-Distribution`.
- **Insights:**
  - BP remains the standard for accuracy but suffers from higher variability.
  - FF offers promise for scenarios prioritizing layer-wise training stability and modular design.
  - Hybrid approaches combining FF for feature extraction and BP for classification may leverage the strengths of both methods.

Chapter 5

Conclusion

In this work, we explored and extended the capabilities of the Forward-Forward (FF) algorithm, an alternative to the widely used Backpropagation (BP) approach for training neural networks. Our primary objective was to assess the performance of the FF algorithm on complex datasets like CIFAR-10 and Fashion MNIST, while also investigating the role of various activation functions within this paradigm.

Accomplishments

- **Extension of the FF Algorithm:** We successfully extended the FF algorithm to handle more complex datasets such as CIFAR-10, which presented a significantly higher degree of inter-class variability compared to simpler datasets like MNIST. Our approach involved architectural refinements, improved data preprocessing, and systematic exploration of activation functions.
- **Activation Function Analysis:** We incorporated and analyzed the impact of standard activation functions (ReLU, Leaky ReLU, Sigmoid, Tanh) as well as custom activations (ReLU Full Grad and T-Distribution Activation) on the goodness-based optimization process of the FF algorithm. Our experiments highlighted the unique behaviors of each activation function and its influence on convergence, loss, and accuracy.
- **Quantitative Comparison with Backpropagation:** A comprehensive comparison between FF and BP was performed using metrics such as validation loss, test loss, validation accuracy, and test accuracy. The analysis revealed that while BP achieved higher peak accuracy, FF demonstrated more stable and consistent training behavior for certain activation functions.
- **Custom Implementation of Layer-Wise Training:** Custom layers were implemented to adhere to the FF algorithm's philosophy of independent layer training. Our implementation introduced novel design choices, such as peer normalization and custom activations, to stabilize training dynamics and improve generalization.

Challenges and Unresolved Aspects

- **Performance Gap with BP:** While the FF algorithm showed stable training dynamics, its overall classification accuracy on CIFAR-10 lagged behind that of BP. Addressing this gap remains an open challenge, as the layer-wise nature of FF makes it difficult to coordinate learning across layers.

- **Scalability to Complex Datasets:** Although we successfully adapted the FF algorithm to handle CIFAR-10, its performance was limited by the inability to effectively model the intricate feature hierarchies of the dataset. This indicates a need for further exploration of feature extraction methods in FF-based models.

- **Optimization of Activation Functions:** While novel activations such as T-Distribution Activation were tested, further fine-tuning and experimentation are required to fully understand their role in the FF algorithm. Custom activations may have the potential to bridge the performance gap between FF and BP.

Anticipated Benefits

- **Gradient-Free Training:** By avoiding the need for gradient backpropagation, the FF algorithm opens new possibilities for training neural networks with non-differentiable components. This could pave the way for new architectures that are currently infeasible under gradient-based constraints.

- **Modular Layer-Wise Training:** Since each layer is trained independently, the FF algorithm introduces the potential for parallel training of layers, significantly reducing training times in large-scale networks.

- **New Perspectives on Activation Functions:** The introduction and analysis of novel activations such as ReLU Full Grad and T-Distribution Activation expand the design space for neural networks, offering new directions for research in network training.

Summary

To summarize, this work presented a comprehensive study of the Forward-Forward (FF) algorithm, extending its capabilities to handle more complex datasets like CIFAR-10. Through

systematic analysis of activation functions and a quantitative comparison with Backpropagation (BP), we highlighted the strengths, limitations, and potential future avenues for FF-based training paradigms. While BP remains the standard for accuracy, FF's unique approach to gradient-free, layer-wise training opens doors to alternative learning paradigms with novel activation functions and enhanced stability. Our findings provide a strong foundation for future research on FF algorithm optimization, hybrid learning methods, and alternative approaches to neural network training.

Chapter 6

Future Work

his study on the Forward-Forward (FF) algorithm highlights several promising directions for future research. Key areas for further exploration include:

- **Hybrid Architectures:** Investigate hybrid models that combine the strengths of FF for early feature extraction and BP for final classification layers to achieve better performance on complex datasets like CIFAR-10.
- **Advanced Activation Functions:** Explore more sophisticated activation functions, beyond T-Distribution and ReLU Full Grad, to improve goodness score optimization and facilitate better generalization.
- **Scalability to Larger Datasets:** Extend the FF algorithm to larger and more diverse datasets, such as ImageNet, to assess its robustness and adaptability in large-scale, real-world scenarios.
- **Layer-Wise Parallelization:** Leverage the independent layer training property of FF to develop parallelized training approaches, potentially reducing overall training time.
- **Theoretical Analysis and Convergence:** Conduct a deeper theoretical investigation of FF's convergence properties and loss landscape to better understand its optimization process and potential stability issues.

These future directions aim to bridge the performance gap between FF and BP, enhance the efficiency of FF-based models, and open new possibilities for neural network training paradigms.

Bibliography

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pp. 1139–1147, 2013.

[5] G. E. Hinton, "The forward-forward algorithm: Some preliminary investigations," *arXiv preprint arXiv:2212.13345*, 2023.

[6] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," *arXiv preprint arXiv:2002.05709*, 2020.

[7] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F.-J. Huang, "A tutorial on energy-based learning," *Predicting structured data*, vol. 1, pp. 1–23, 2006.

[8] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 770–778, 2016.

[10] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[11] P. Jyoti Dabass, "Visualization of the vanishing gradient problem," Medium Article, accessed: 2024-12-03. [Online]. Available: https://miro.medium.com/v2/resize:fit:1400/1*rPafmg_KaY4PzPiufg-qnw.png

[12] F. Bre, "Artificial neural network architecture," Research Gate, accessed: 2024-12-03. [Online]. Available: https://www.researchgate.net/profile/Facundo-Bre/publication/321259051/figure/fig1/AS:614329250496529@1523478915726/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o.png

[13] C. C. A. . International, "An overview of backpropagation," Research Gate, accessed: 2024-12-03. [Online]. Available: https://www.researchgate.net/publication/356390636/figure/fig2/AS:1094521567875074@1637965685089/An-overview-of-backpropagation-a-Role-of-backpropagation-in-a-neural-network-b.ppm

[14] G. Hinton, "Forward forward algorithm," Research Gate, accessed: 2024-12-04. [Online]. Available: https://raw.githubusercontent.com/loewex/forward-forward/master/images/ForwardForward.jpeg

[15] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[16] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms," 2017. [Online]. Available: https://github.com/zalandoresearch/fashion-mnist

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison,

A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019. [Online]. Available: https://pytorch.org/

[19] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, sep 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[20] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. [Online]. Available: https://matplotlib.org/

.1 Code Appendix

https://github.com/venkateshtantravahi/ForwardForwardAlgorithm.git

.2 TensorFlow Model Summary

Layer (type)	Output Shape	Param #
layer_32 (Layer)	(1, 3, 128, 64)	73,792
re_lu_36 (ReLU)	(1, 3, 128, 64)	0
layer_33 (Layer)	(1, 3, 128, 64)	36,928
re_lu_37 (ReLU)	(1, 3, 128, 64)	0
max_pooling2d_10 (MaxPooling2D)	(1, 1, 64, 64)	0
layer_34 (Layer)	(1, 1, 64, 128)	73,856
re_lu_38 (ReLU)	(1, 1, 64, 128)	0
layer_35 (Layer)	(1, 1, 64, 128)	147,584
re_lu_39 (ReLU)	(1, 1, 64, 128)	0
max_pooling2d_11 (MaxPooling2D)	(1, 0, 32, 128)	0
layer_36 (Layer)	(1, 0, 32, 256)	295,168
re_lu_40 (ReLU)	(1, 0, 32, 256)	0
layer_37 (Layer)	(1, 0, 32, 256)	590,080
re_lu_41 (ReLU)	(1, 0, 32, 256)	0
layer_38 (Layer)	(1, 0, 32, 256)	590,080
re_lu_42 (ReLU)	(1, 0, 32, 256)	0
layer_39 (Layer)	(1, 0, 32, 256)	590,080
re_lu_43 (ReLU)	(1, 0, 32, 256)	0
max_pooling2d_12 (MaxPooling2D)	(1, 0, 16, 256)	0
layer_40 (Layer)	(1, 0, 16, 512)	1,180,160
re_lu_44 (ReLU)	(1, 0, 16, 512)	0
layer_41 (Layer)	(1, 0, 16, 512)	2,359,808
re_lu_45 (ReLU)	(1, 0, 16, 512)	0
layer_42 (Layer)	(1, 0, 16, 512)	2,359,808
re_lu_46 (ReLU)	(1, 0, 16, 512)	0
layer_43 (Layer)	(1, 0, 16, 512)	2,359,808
re_lu_47 (ReLU)	(1, 0, 16, 512)	0
max_pooling2d_13 (MaxPooling2D)	(1, 0, 8, 512)	0
layer_44 (Layer)	(1, 0, 8, 512)	2,359,808
re_lu_48 (ReLU)	(1, 0, 8, 512)	0
layer_45 (Layer)	(1, 0, 8, 512)	2,359,808
re_lu_49 (ReLU)	(1, 0, 8, 512)	0
layer_46 (Layer)	(1, 0, 8, 512)	2,359,808
re_lu_50 (ReLU)	(1, 0, 8, 512)	0
layer_47 (Layer)	(1, 0, 8, 512)	2,359,808

Layer (type)	Output Shape	Param #
re_lu_51 (ReLU)	(1, 0, 8, 512)	0
max_pooling2d_14 (MaxPooling2D)	(1, 0, 4, 512)	0
flatten_2 (Flatten)	(1, 0)	0
fully_connected_layer_6 (FullyConnectedLayer)	(1, 4096)	0
re_lu_52 (ReLU)	(1, 4096)	0
fully_connected_layer_7 (FullyConnectedLayer)	(1, 4096)	0
re_lu_53 (ReLU)	(1, 4096)	0
fully_connected_layer_8 (FullyConnectedLayer)	(1, 10)	0

Total params: 20pt 20,096,384 (76.66 MB)
Trainable params: 5pt 20,096,384 (76.66 MB)
Non-trainable params: 0 (0.00 B)

.3 PyTorch Model Summary

The PyTorch model architecture summary is shown below:

```
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
SequentialDeepNetwork
Sequential: 1-1                          [32, 10]                  --
    LocalReceptiveFieldLayer: 2-1        [32, 64, 32, 32]          --
        Conv2d: 3-1                      [32, 64, 32, 32]          4,864
        BatchNorm2d: 3-2                 [32, 64, 32, 32]          128
        ReLU: 3-3                        [32, 64, 32, 32]          --
        AdaptiveAvgPool2d: 2-2           [32, 64, 16, 16]          --
    LocalReceptiveFieldLayer: 2-3        [32, 64, 16, 16]          --
        Conv2d: 3-4                      [32, 64, 16, 16]          102,464
        BatchNorm2d: 3-5                 [32, 64, 16, 16]          128
        ReLU: 3-6                        [32, 64, 16, 16]          --
        AdaptiveAvgPool2d: 2-4           [32, 64, 8, 8]            --
    LocalReceptiveFieldLayer: 2-5        [32, 64, 8, 8]            --
        Conv2d: 3-7                      [32, 64, 8, 8]            102,464
        BatchNorm2d: 3-8                 [32, 64, 8, 8]            128
        ReLU: 3-9                        [32, 64, 8, 8]            --
        AdaptiveAvgPool2d: 2-6           [32, 64, 8, 8]            --
    LocalReceptiveFieldLayer: 2-7        [32, 64, 8, 8]            --
        Conv2d: 3-10                     [32, 64, 8, 8]            102,464
        BatchNorm2d: 3-11                [32, 64, 8, 8]            128
        ReLU: 3-12                       [32, 64, 8, 8]            --
        AdaptiveAvgPool2d: 2-8           [32, 64, 8, 8]            --
    LocalReceptiveFieldLayer: 2-9        [32, 64, 8, 8]            --
        Conv2d: 3-13                     [32, 64, 8, 8]            102,464
        BatchNorm2d: 3-14                [32, 64, 8, 8]            128
        ReLU: 3-15                       [32, 64, 8, 8]            --
        AdaptiveAvgPool2d: 2-10          [32, 64, 8, 8]            --
    LocalReceptiveFieldLayer: 2-11       [32, 64, 8, 8]            --
        Conv2d: 3-16                     [32, 64, 8, 8]            102,464
        BatchNorm2d: 3-17                [32, 64, 8, 8]            128
        ReLU: 3-18                       [32, 64, 8, 8]            --
```

```
        AdaptiveAvgPool2d: 2-12     [32, 64, 8, 8]           --
    LocalReceptiveFieldLayer: 2-13   [32, 64, 8, 8]           --
        Conv2d: 3-19                [32, 64, 8, 8]           102,464
        BatchNorm2d: 3-20           [32, 64, 8, 8]           128
        ReLU: 3-21                  [32, 64, 8, 8]           --
        AdaptiveAvgPool2d: 2-14     [32, 64, 8, 8]           --
    LocalReceptiveFieldLayer: 2-15   [32, 64, 8, 8]           --
        Conv2d: 3-22                [32, 64, 8, 8]           102,464
        BatchNorm2d: 3-23           [32, 64, 8, 8]           128
        ReLU: 3-24                  [32, 64, 8, 8]           --
        AdaptiveAvgPool2d: 2-16     [32, 64, 4, 4]           --
    LocalReceptiveFieldLayer: 2-17   [32, 64, 8, 8]           --
        Conv2d: 3-25                [32, 64, 8, 8]           102,464
        BatchNorm2d: 3-26           [32, 64, 8, 8]           128
        ReLU: 3-27                  [32, 64, 8, 8]           --
        AdaptiveAvgPool2d: 2-18     [32, 64, 4, 4]           --
Sequential: 1-2                      [32, 10]                  --
    SimpleLayerWithPeerNormalization
         Linear                     [32, 64]                  65,600
         ReLU                       [32, 64]                  --
         PeerNormalization          [32, 64]                  --
         Dropout                    [32, 32]                  --
    SimpleLayerWithPeerNormalization
         Linear                     [32, 32]                  2,080
         ReLU                       [32, 32]                  --
         PeerNormalization          [32, 32]                  --
         Dropout                    [32, 32]                  --
         Linear                     [32, 10]                  330
==========================================================================
Total params: 893,738
Trainable params: 893,738
Non-trainable params: 0
Total mult-adds (G): 2.47
==========================================================================
Input size (MB): 0.39
Forward/backward pass size (MB): 56.65
Params size (MB): 3.57
Estimated Total Size (MB): 60.62
==========================================================================
```

.4 FF Model Detailed View

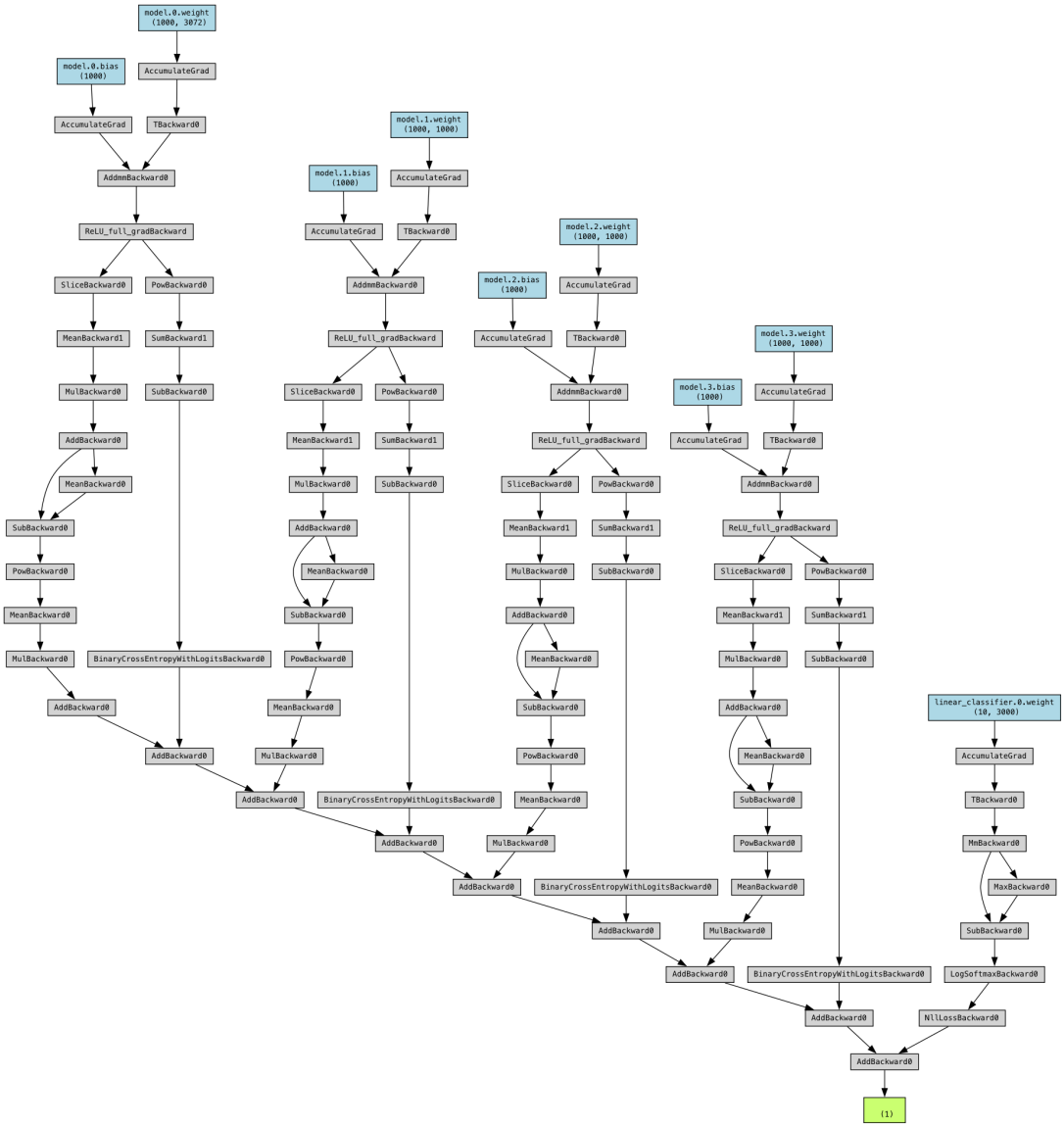

Figure 1: Detail View of FF