Exploring the Forward-Forward Algorithm for Training Recurrent Neural Networks

Bachelor Semester Project S5 (Academic Year 2024/25), University of Luxembourg

Fedor Chikhachev
University of Luxembourg
Esch-sur-Alzette, Luxembourg
fedor.chikhachev.001@student.uni.lu

Supervisor: Prof. Luis A. Leiva
University of Luxembourg
Esch-sur-Alzette, Luxembourg
luis.leiva@uni.lu

Abstract

This report presents an investigation into applying Geoffrey Hinton's Forward-Forward (FF) algorithm to train Recurrent Neural Network (RNN) architectures for sequential data tasks. Traditional RNN training methods, such as Backpropagation, often struggle with issues of parallelization and computational efficiency due to their sequential nature. In contrast, the FF algorithm, which utilizes dual forward passes with positive and negative data samples to optimize network layers without explicit back error propagation, offers a potentially more efficient and biologically plausible alternative.

In this project, the FF algorithm is adapted to RNNs, enabling layer-wise learning by maximizing a local goodness score for positive sequences and minimizing it for negative ones. Experiments are conducted on sequential supervised learning tasks, in particular – Real vs. Synthetic data discrimination. The performance, accuracy, and efficiency of FF-trained RNNs are compared with those trained using traditional BPTT and show comparable and promising results.

1 Introduction

Artificial Intelligence (AI) has become a transformative force in the modern world, enabling breakthroughs in diverse domains such as natural language processing, healthcare, autonomous systems, and financial forecasting. At the heart of most AI systems lies the process of training deep neural networks, typically accomplished using the Backpropagation algorithm. Backpropagation, a gradient-based optimization method, has been a cornerstone of neural network training since its popularization in the 1980s, enabling the rapid advancements seen in machine learning today [12]. Nevertheless, while it has been successful, Backpropagation does have its constraints.

Recurrent Neural Networks (RNNs) are a key class of models used for processing sequential data due to their ability to capture temporal dependencies. The training of RNNs traditionally relies on Backpropagation Through Time (BPTT), a variant of Backpropagation designed to handle the unrolling of sequences [15]. However, BPTT is inherently sequential, requiring computation for each time step and each layer to be completed before moving to the next, making parallelization challenging. Additionally, Backpropagation relies on analytically differentiable activation functions, which restricts its applicability to architectures utilizing non-differentiable or complex functions (e.g. any kind of black box) in the underline.

In 2022, Geoffrey Hinton proposed the Forward-Forward (FF) algorithm as a biologically inspired alternative to Backpropagation [5]. The FF algorithm skips the need for a backward pass and instead uses two forward passes: one with positive data samples and one with negative samples. During these passes, a layer-wise "goodness" score is computed, and the network's weights are updated to maximize this score for positive data while minimizing it for negative data. This approach simplifies the training process and introduces the potential for parallel computation.

Thus far, the FF algorithm has primarily been demonstrated on Multi-Layer Perceptrons (MLPs) and evaluated on non-sequential datasets. Its applicability to sequential data and recurrent architectures, which have unique characteristics such as temporal dependencies, remains majorly unexplored. This study builds on Hinton's initial work by adapting the FF algorithm for RNNs, aiming to address the challenges posed by sequential dependencies while leveraging the benefits of FF.

The primary goals of this work are as follows:

- To adapt the Forward-Forward algorithm for use with RNN architectures, addressing the unique requirements of sequential data.
- To evaluate the effectiveness of FF-trained RNNs in supervised tasks, comparing their performance to RNNs trained via BPTT.
- To test these models on time-series tasks such as pointer movement prediction, characterized by spatial coordinates and timestamps.

The methodology includes implementing an RNN trained using both the BPTT and the adapted FF algorithm. The models are assessed on time-series data using metrics like

accuracy and AUC-ROC, with the results analysed to evaluate the feasibility of using the FF algorithm as a training method for sequential tasks.

2 Motivation & Related Work

2.1 The Forward-Forward Algorithm

The Forward-Forward (FF) algorithm, proposed by Geoffrey Hinton in 2022 [5], offers a biologically inspired alternative to Backpropagation. A *goodness function* is computed locally at each layer to evaluate how well the data aligns with the network's learned representations. For an input sample x_i, the goodness score for layer l is defined as the sum of the squared activations (outputs) for each layer:

$$G_i^{(l)} = \sum_j \left(y_{i,j}^{(l)}\right)^2,$$

where j represents an index of output neuron. Accordingly, based on the value of this function, a probability that an input vector is positive (i. e. real) is given by applying the logistic function, the goodness, minus some threshold, θ:

$$p(positive)^{(l)} = \sigma(G_i^{(l)} - \theta)$$

During training, weights in each layer are updated to maximize goodness for positive samples (e.g., correctly labeled or desirable inputs) and minimize goodness for negative samples (e.g., randomly perturbed or contrasted inputs). It is important to mention that the update is held locally, that is, the weights are updated according to the goodness of activations in one layer. This mechanism eliminates the need for backward gradient computation, simplifying the training pipeline and enabling layer-wise optimization. In classification or discrimination setting, future predictions are made by comparing the goodness scores for new inputs against thresholds for each class.

2.2 Applications and Variants of the Forward-Forward Algorithm

Hinton's initial work demonstrated the FF algorithm primarily on static datasets, such as CIFAR-10 [6] and MNIST [7]. For multi-label classification tasks, special data encoding was required to store label information during positive and negative passes, implying that positive data contains encoded correct label and negative shows an incorrect label (cf. Figure 1). This approach was also extended to an RNN architecture applied to MNIST, but only for static data, leaving its applicability to true sequential data untested.

Subsequent research has explored adapting the FF algorithm for more complex scenarios. Chen et al. [4] introduced Self-Contrastive Forward-Forward learning, applying FF using a contrastive learning framework. Positive and negative data were created by initial data augmentation, where a positive sample consists of two different concatenated views of the same data sample, and negative pairs are constructed from different samples. This method was demonstrated in action on visual data, using CNN architecture, and sequential data, such as audio data, using RNN architecture.

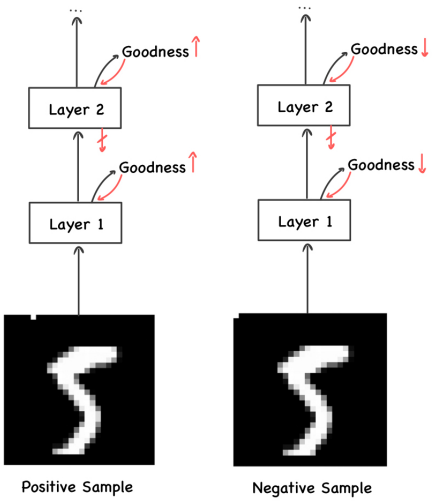

Figure 1. Example of label encoding for MNIST sample from original work of G. Hinton. Source: https://github.com/loeweX/Forward-Forward

This project focuses on the application of the Forward-Forward (FF) algorithm to time-series sequential data. While it is acknowledged that outperforming Backpropagation-based algorithms, as demonstrated in the aforementioned studies, is unlikely, this work seeks to achieve comparable performance levels. At the same time, it investigates potential advantages of the FF algorithm over Backpropagation Through Time (BPTT), such as computational efficiency. By evaluating FF-trained RNNs on sequential tasks, this study aims to provide insights into the feasibility of FF as a viable alternative for training sequential models.

3 Methodology

The primary task in this study is the detection of synthesized data versus human-recorded data. Given a dataset of sequential pointer movements, the goal is to classify whether a given sequence originates from natural human behaviour or has been artificially generated.

3.1 Datasets

The experiments are conducted on several time-series datasets representing sequential pointer movements. Each dataset consists of:

- **Natural Data:** Real user data collected from cursor movements or handwriting tasks.
- **Synthetic Data:** Artificially generated trajectories based on real data using the Sigma-Lognormal ($\Sigma\Lambda$) model, grounded in the Kinematic Theory [11]. This model assumes that human movements are composed

of elementary units or primitives, superimposed to form the resulting trajectory. Synthetic data introduces variations in timing, curvature, and stroke dynamics, creating negative samples for training.

Dataset Sources and Statistics:

- $1 Dataset [16]: Contains 5,200 samples of one-stroke drawn figures and symbols across 16 classes, with an average sequence length of 70.
- MCYT-Signatures [10]: A subset of the MCYT bimodal database, containing 2500 signature samples from 10 individuals with an average sequence length of 351.
- MobileTouchDB [14]: A large database of over 15,600 handwritten symbols, including numbers and letters, collected from more than 200 individuals with an average sequence length of 40.
- Mouse-Operation [13]: A dataset of 1000 samples, each consisting of 8 consecutive mouse movements with varying distances (short, medium, long) and directions spanning eight 45-degree intervals. The average sequence length is 1000.

3.2 Initial Data Normalization

To ensure consistency across datasets, we focus on x, y, t tuples, ignoring other columns in the dataset. The trajectories are normalized over time by calculating the difference between consecutive coordinates and dividing it by the corresponding timestamp difference, effectively transforming the data into approximated momentary velocity. This approach, proven effective in prior web-based mouse movement studies [3], enables the model to generalize across datasets with varying temporal and spatial properties. Furthermore, it ensures that all features are treated with equal importance and belong to the same information domain.

In addition to normalization, all sequences are truncated or padded to a uniform length to ensure compatibility with the RNN architecture. Uniform sequence lengths are essential for batch processing. The truncation length is chosen based on the average sequence length for each dataset. For instance, the MCYT and Mouse-Operation datasets contain very long sequences, which are truncated to a maximum length of 100 to retain sufficient temporal information while maintaining computational efficiency. Shorter sequences are padded with zeros.

3.3 Data Splits

All datasets are randomly split into 3 partitions: train (70%), validation (10%) and test (20%). We use the validation partition to prevent models from overfitting. The data in the datasets is equally balanced among classes. Each experiment is conducted 5 times to exclude random factors from the data split. The average metrics from these runs are then calculated and reported as the final results.

3.4 Evaluation Metrics

Model performance is evaluated using the following metrics:

- **Accuracy:** The proportion of correctly classified sequences, reflecting the overall effectiveness of the model.
- **AUC-ROC:** The area under the Receiver Operating Characteristic curve, which measures the model's ability to discriminate between classes across varying thresholds.
- **Training Time per Epoch:** The average time required for one training epoch. This metric highlights the computational efficiency differences between BPTT and FF algorithms.
- **Number of Epochs to Convergence:** The number of epochs required to reach convergence, determined using the EarlyStopping criterion (cf. Convergence criteria 3.6.5).

3.5 RNN Architecture

The network consists of 3 recurrent layers with basic RNN cells. The recurrent layers use the ReLU activation function, as required by the FF algorithm's original goodness function, which relies on values exceeding the $[0, 1]$ range to define split thresholds.

Key hyperparameters include:

- **Number of Hidden Units:** 96
- **Batch Size:** 600
- **Max Epochs:** 200
- **Activation function:** ReLU

3.6 Training Algorithms

Two models are trained with identical architectures but different training algorithms.

3.6.1 Backpropagation Through Time (BPTT). The baseline model is trained using BPTT. The network is unrolled over time steps, and gradients are backpropagated through the unrolled network. Binary cross-entropy loss is used for this experiment, and Adam optimizer is used for weight updates.

3.6.2 Forward-Forward (FF) Algorithm. The FF algorithm trains each layer independently using positive and negative samples. Here is a short breakdown of important steps for this algorithm.

- The model's input includes both positive and negative samples. To maintain balanced learning, each batch is composed of an equal number of positive and negative samples. During propagation, the process mimics supervised learning, where positive and negative samples are identified using masks of zeros and ones.
- Layer's output (set of hidden states) is a tensor of dimensions `[batch_size, seq_len, hidden_size]`. The original goodness function, defined as the sum

of squared activations, operates on this tensor. However, the temporal dimension (`seq_len`) remains unresolved. To simplify this, we compute the average goodness score over time:

$$G_i^{(l)} = \frac{1}{T} \sum_{t=1}^{T} G_{i,t}^{(l)}$$

- The layer-wise loss function for FF training is:

$$\mathcal{L} = -\mathbb{E}_{\text{pos}} \left[\frac{1}{T} \sum_{t=1}^{T} \log \sigma\big(G_{i,t,\text{pos}}^{(l)} - \theta^{(l)}\big) \right]$$

$$-\mathbb{E}_{\text{neg}} \left[\frac{1}{T} \sum_{t=1}^{T} \log \sigma\big(\theta^{(l)} - G_{j,t,\text{neg}}^{(l)}\big) \right],$$

  where $\theta^{(l)}$ is the layer-specific threshold. It was inspired by Self-Contrastive Forward-Forward approach [4]. The first term increases the goodness score for positive samples beyond the threshold, while the second term reduces the goodness score for negative samples below the threshold. This dual objective ensures effective discrimination between positive and negative data. In context of this work, we consider threshold the same for each layer in order to provide a proof-of-concept solution.
- Each layer's loss function is optimized independently. However, the overall objective for the model is defined as the sum of the layer-wise losses, which enables joint optimization across all layers while preserving their independence. Adam optimizer is used for weight updates.
- It is important to note that if the activities from the first hidden layer are directly passed as input to the second hidden layer, positive and negative data can be easily separated just by looking at the length of the activity vector in the first hidden layer. To avoid this issue and ensure meaningful processing in subsequent layers, there exist two options:
  1. **Standardization** scales data to have a mean of 0 and a standard deviation of 1, ensuring consistency and improving model performance by centering and normalizing feature distributions:

$$z = \frac{x - \mu}{\sigma};$$

  2. **Layer normalization [8]** is a simplified normalization method that scales activations based on their root mean square (RMS) value along the last dimension, without centering them:

$$z = \frac{x}{\sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2 + \epsilon}},$$

  where n is a dimension of vector $\mathbf{x}$.

  Standardization works best for data that follows a normal distribution, while layer normalization is better at managing relative relationships in the data. We apply and compare both methods to evaluate their performance.

3.6.3 Threshold. The threshold is a hyperparameter in the Forward-Forward algorithm, which defines the boundary between positive and negative samples during training. This value directly influences the layer-wise loss function value and, consequently, the model's ability to discriminate between classes and predict the class of sample on layer level.

Since the threshold is a hyperparameter, there is no precise way to determine its value before conducting the experiment. As a result, we test a range of thresholds, starting from small values near zero to larger ones. Preliminary experiments indicated that selecting a threshold within the range of 2 to 10 has little impact on the model's final prediction accuracy. For quantitative analysis of model performance, we use $\theta = 7$ across all layers, for both positive and negative samples.

3.6.4 Learning Rate. For this study, a learning rate value of 10^{-3} was set, as it balances training stability and convergence speed. To avoid plateaus during later epochs of training, an adaptive learning rate strategy is employed. If no improvement in training loss is observed for 5 consecutive epochs, the learning rate is halved.

3.6.5 Convergence Criteria. Both algorithms utilize the EarlyStopping technique to prevent overfitting and ensure a consistent comparison of training efficiency. This criterion monitors the RNN's performance on the validation partition based on the sum of per-layer losses. Training is halted if no improvement is observed in validation loss for 10 consecutive epochs, and the weights of the best-performing model are saved.

The EarlyStopping approach ensures that the final model represents its optimal state, as determined by the validation set. Before testing, the saved weights are reloaded to guarantee that the evaluation reflects the model's peak performance.

3.7 Implementation Details

The models are implemented in PyTorch and trained on a MacBook Pro equipped with an M2 Pro chip and 16GB of RAM. To ensure consistency and comparability, all experiments are performed using a single CPU core, avoiding hardware-based parallelization optimizations. Training time and resource utilization are recorded for both algorithms to evaluate their computational efficiency.

To enforce the locality of loss function computations in the Forward-Forward algorithm, the PyTorch `detach()` function is employed. This function creates a copy of the current tensor, breaking the computational graph and preventing

gradients from propagating to previous layers. After each layer's forward pass, the tensor is detached, ensuring that subsequent layers only operate on the current state of activations, without knowledge of previous transformations.

This approach aligns with the layer-wise optimization principle of the Forward-Forward algorithm, allowing the use of PyTorch's built-in optimizers. These optimizers, which rely on computational graph analysis for gradient descent, treat each layer independently. Figure 13 in the Appendix illustrates the computation tree for the Forward-Forward algorithm, where each layer is represented as an independent branch, highlighting the localized loss computation. In contrast, Figure 14 in the Appendix shows the computational tree for Backpropagation, where gradients are propagated through all layers, forming a single connected structure.

The implementation of the algorithm was influenced by the original Matlab source code by G. Hinton, and its vision in Pytorch framework by Sindy Löwe [9]. The complete code for all the experiments will be made available on GitHub shortly.

4 Results

4.1 Quantitative Analysis

Table 1 highlights the difference in performance of different normalization techniques for FF algorithm.

Table 2 summarizes the performance metrics for both the BPTT-trained and FF-trained RNN models across the various tasks.

Table 1. Accuracy and AUC Scores for Forward-Forward Algorithm Based on In-Layer Data Transformation Choices

Dataset	Layer Norm.		Standardization	
	Accuracy	AUC	Accuracy	AUC
$1	0.8344	0.9245	**0.8552**	**0.9320**
MobileTouch	0.8221	0.8906	**0.8315**	**0.9063**
MCYT	**0.9046**	0.9528	0.8876	**0.9602**
Mouse-Ops.	0.8927	0.9527	**0.9182**	**0.9750**

Table 2. Results for Forward-Forward (Standardization technique) and Backpropagation Through Time (BPTT), accuracy and AUC scores

Dataset	FF		BPTT	
	Accuracy	AUC	Accuracy	AUC
$1	0.8552	0.9320	0.9636	0.9928
MobileTouch	0.8315	0.9063	0.8689	0.9498
MCYT	0.8876	0.9602	0.9804	0.9971
Mouse-Ops.	0.9182	0.9750	0.9792	0.9945

In general, both approaches for data transformation between layers in the Forward-Forward algorithm show close results, with Standardization being a leader for most of the measurements. We consider this version while comparing results with Backpropagation algorithm.

Accuracy: The Forward-Forward algorithm achieves lower accuracy compared to Backpropagation, which was expected due to its layer-wise optimization. However, accuracy is sensitive to the threshold value used to classify predictions, making it less robust for evaluating the model's discriminative power. Moreover, while preserving the locality, a Forward-Forward model can also be augmented with a downstream linear classifier to improve the final prediction mechanism and raise accuracy score.

AUC-ROC:. The AUC-ROC results for Forward-Forward are closer to those of Backpropagation, indicating strong discriminative capabilities. This suggests that the FF algorithm is effective in separating positive and negative samples, despite its simpler training process. Fine-tuning the thresholds may further reduce the accuracy gap accordingly.

4.2 Computational Efficiency

Table 3. Performance comparison for two versions of Forward-Forward algorithm with Layer Normalization (FF-Norm) and Standardization (FF-Std) transformations and Backpropagation Throught Time (BPTT)

FF-Norm	Epochs	Time/Epoch (s)
$1	160	1.36
MobileTouch	198	2.19
MCYT	69	1.48
Mouse-Ops.	30	0.31
FF-Std	**Epochs**	**Time/Epoch (s)**
$1	107	3.00
MobileTouch	123	5.03
MCYT	73	3.15
Mouse-Ops.	37	0.55
BPTT	**Epochs**	**Time/Epoch (s)**
$1	50	1.36
MobileTouch	84	1.55
MCYT	51	1.49
Mouse-Ops.	53	0.20

Table 3 presents a comparison of time per epoch and the number of epochs required to reach convergence across the three training approaches evaluated in our experiments: Forward-Forward with Layer Normalization, Forward-Forward with Standardization, and Backpropagation.

In terms of convergence, Backpropagation requires fewer epochs than both versions of the Forward-Forward algorithm. This can be attributed to the absence of global error propagation in FF, which limits the availability of information for optimizing weights across layers.

When analysing time per epoch, the Forward-Forward variant with Layer Normalization demonstrates equal or slightly lower performance with Backpropagation. However, the Standardization-based FF approach is much slower in this metric. Its slower performance is likely due to its higher computational complexity and memory usage. Specifically, the process involves calculating both the mean and variance for each sample, followed by *broadcasting* (alignment of array shapes for element-wise operations) these values across larger tensor dimensions for further operations. This additional overhead increases the overall computation time per epoch.

Additionally, our implementation of the Forward-Forward algorithm may lead to lower performance because we did not prioritize memory efficiency. As mentioned earlier, we use PyTorch's `detach()` function to ensure the method remains local. Each time this function is used, it creates a new copy of the tensor in memory, and allocating new memory is a very time-consuming process.

4.3 Qualitative Observations

4.3.1 Per-layer Behaviour in Forward-Forward Algorithm. The Forward-Forward algorithm's layer-wise optimization makes each layer act as an independent classifier. At each forward propagation step, the goodness score for every sample is computed, allowing predictions to be made based on the performance of individual layers.

Initial investigations by G. Hinton on MLP architectures suggested that the first hidden layer often performs worse than subsequent layers [5]. However, in our experiments with RNNs, the performance of the first layer is comparable to that of later layers. Figures 2 and 3 illustrate the per-layer accuracy for layer normalization and standardization methods, respectively, highlighting their relative performance. This result can be explained by the sequential nature of RNNs, where even the initial layer processes temporal dependencies within the data, unlike MLPs, which operate on static inputs. This behavior is consistent across both normalization and standardization techniques.

4.3.2 Discriminative Separation.

Standardization: To assess the discriminative capabilities of the Forward-Forward algorithm, we analyse the distribution of goodness scores across layers for the MCYT dataset. Figures 4, 5 and 6 in the Appendix section corresponding to the standardization case demonstrate that positive samples tend to follow a Gaussian-like distribution, while negative samples exhibit an exponential-like behaviour under the default threshold values.

Normal distribution is approximated with its probability density function:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

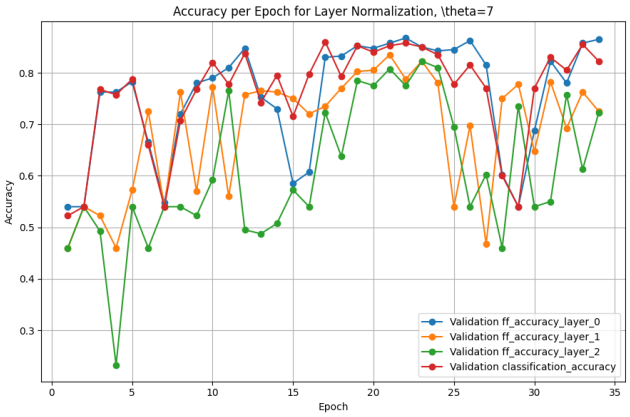

Figure 2. Per-layer accuracy score on validation set for each training epoch for layer normalization technique; MCYT dataset, $\theta = 7$

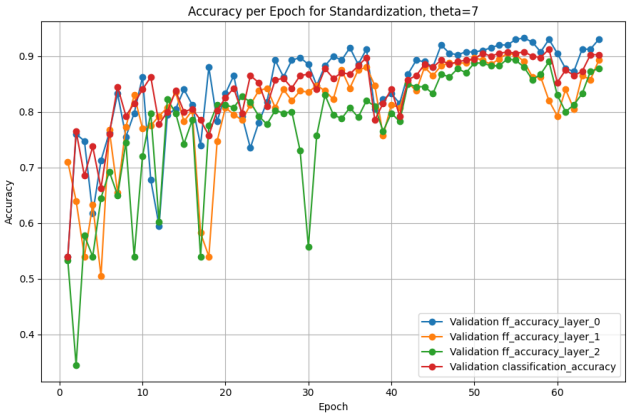

Figure 3. Per-layer accuracy score on validation set for each training epoch for standardization technique; MCYT dataset, $\theta = 7$

where μ is the mean value of goodness for positive or negative samples, and σ is the variance.

Exponential (λ) distribution is approximated with its rate parameter:

$$\lambda = \frac{1}{\mathbb{E}[x]}$$

and its probability density function:

$$f(x) = \lambda e^{-\lambda x}$$

This behaviour appears from the anti-symmetry of the loss function and the nature of the goodness function, which is constrained to non-negative values. Specifically, the following inequality illustrates how this limitation influences the distribution of goodness scores for negative samples:

$$G^{(l)}_{j,t,\text{neg}} \geq 0 \Rightarrow \theta^{(l)} - G^{(l)}_{j,t,\text{neg}} \leq \theta^{(l)} \Rightarrow$$

$$\Rightarrow \log \sigma\left(\theta^{(l)} - G^{(l)}_{j,t,\text{neg}}\right) \leq \log \sigma(\theta^{(l)})$$

Here, the threshold $\theta^{(l)}$ acts as a key parameter shaping the distribution. Small threshold values restrict the range of negative goodness scores, causing the scores to cluster near zero and form an exponential-like distribution.

However, increasing $\theta^{(l)}$ reduces this imbalance, as the logarithmic scaling of σ becomes less impactful when $\sigma(\theta^{(l)}) \approx 1$. This hypothesis is supported by the results shown in Figures 7, 8 and 9 in the Appendix, where setting $\theta = 7$ results in two similar Gaussian-like distributions for positive and negative samples across all three layers of the RNN for the MCYT dataset.

Layer Normalization: The behaviour changes notably when layer normalization is applied (except for the 1st layer). As shown in Figures 10, 11, and 12 in the Appendix, the samples no longer align with any specific standard distribution, unlike in the standardization case, while attracting data to some spike forms.

This can be attributed to the properties of the layer normalization function. Before each propagation step, the magnitude of the values along the last dimension is adjusted to achieve a root mean square of approximately 1. As a result, every sample—whether negative or positive—has a vector length close to 1 before passing through the layer, while preserving distinct orientations. This mechanism encourages the RNN to focus on learning relative dependencies rather than magnitude, which ultimately influences the overall goodness metric distribution.

5 Discussion & Future Work

The results of this study demonstrate that the Forward-Forward algorithm is a viable alternative to Backpropagation for training RNNs on sequential tasks. FF algorithm shows comparable AUC-ROC scores, indicating strong discriminative ability.

These comparable results actually give potential for the development of distributed versions of the FF algorithm. During forward passes, the optimization of local objective (loss) functions for each layer can be performed independently, enabling efficient parallelization. In rough estimations, the proposed architecture in this work can be parallelized and takes 3 times less time for training. Current investigations in this area have demonstrated a 3.75 times speed-up on the MNIST benchmark, highlighting the potential computational advantages of FF in distributed or hardware-accelerated training environments [1]. However, an industrial implementation of a framework for distributed computations requires a lot of work on other aspects as well, for example, data synchronization and volatility.

Additionally, the two data transformation techniques considered for effective FF training—standardization and layer normalization—present a trade-off between better predictive accuracy and faster processing per layer. However, given that both methods are parallelisable, we can prioritize the more precise standardization approach, which encourages the data to follow specific distributions, therefore potentially enhancing model performance.

Based on the results, this project highlights several opportunities for future research to address its limitations and build on its findings:

- **Alternative Discrimination Functions and Similarity Metrics:** Exploring alternative designs for the discrimination function or employing different similarity metrics within the FF framework could significantly enhance its performance. The current loss function, based on the original goodness metric, has certain limitations, as evident in the observed distribution differences. Improvements could include introducing separate thresholds for positive and negative data, or using layer-specific thresholds to better capture the nuances of each layer's activations.
  Moreover, alternative similarity metrics could further improve class separability. For instance, Kullback-Leibler (KL) divergence could be employed to refine score distributions and enforce specific behaviours, enhancing the model's discriminative power [17]. Additionally, incorporating weighted goodness functions or attention mechanisms could address the limitation of treating all input parameters equally.
- **Goodness Scores for Other Activation Functions and RNN Cells:** The original FF algorithm relies on ReLU activation due to its wide value range. However, modern RNN architectures, such as LSTMs and GRUs, constrain activation ranges, potentially limiting the effectiveness of the current goodness function. Future work could explore alternative goodness formulations tailored to the activation dynamics of these advanced architectures.
- **Lightweight Inference:**
  The layer-wise goodness metric allows layers to make independent predictions, each with its own confidence level. This opens the possibility of dynamically terminating processing at intermediate layers based on the "goodness" scores, reducing computational requirements for simpler tasks. Such a dynamic mechanism has been proposed and explored in lightweight FF architectures, further emphasizing its scalability for constrained and adaptive environments [2].

While the Forward-Forward algorithm remains in its early stages of development, this study highlights its potential as a lightweight and scalable alternative to traditional training methods. Its ability to train neural networks with reduced computational complexity makes it particularly suitable for deployment on resource-constrained devices. Future

advancements in discrimination functions, preprocessing techniques, and parallelization strategies could further unlock its potential for handling more complex tasks and larger datasets, paving the way for biologically inspired and efficient neural network training paradigms.

6 Acknowledgement

I would like to thank Prof. Luis A. Leiva for his assistance and feedback during the completion of this project, as well as BICS administration for the new format of the course.

References

[1] Ege Aktemur, Ege Zorlutuna, Kaan Bilgili, Tacettin Emre Bok, Berrin Yanikoglu, and Suha Orhun Mutluergil. 2024. Going Forward-Forward in Distributed Deep Learning. *arXiv preprint arXiv:2404.08573* (2024).

[2] Amin Aminifar, Baichuan Huang, Azra Abtahi Fahliani, and Amir Aminifar. 2024. LightFF: Lightweight inference for forward-forward algorithm. In *27th European Conference on Artificial Intelligence, ECAI-2024*, Vol. 392. IOS Press, 1728–1735.

[3] Lukas Brückner, Ioannis Arapakis, and Luis A Leiva. 2020. Query abandonment prediction with recurrent neural models of mouse cursor movements. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 1969–1972.

[4] Xing Chen, Dongshu Liu, Jeremie Laydevant, and Julie Grollier. 2024. Self-Contrastive Forward-Forward Algorithm. *arXiv preprint arXiv:2409.11593* (2024).

[5] Geoffrey Hinton. 2022. The forward-forward algorithm: Some preliminary investigations. *arXiv preprint arXiv:2212.13345* (2022).

[6] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[7] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[8] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *ArXiv e-prints* (2016), arXiv–1607.

[9] Sindy Loewe. 2023. Reimplementation of the Forward-Forward Algorithm. https://github.com/loeweX/Forward-Forward.

[10] Javier Ortega-Garcia, J Fierrez-Aguilar, D Simon, J Gonzalez, Marcos Faundez-Zanuy, V Espinosa, A Satue, I Hernaez, J-J Igarza, C Vivaracho, et al. 2003. MCYT baseline corpus: a bimodal biometric database. *IEE Proceedings-Vision, Image and Signal Processing* 150, 6 (2003), 395–401.

[11] Rejean Plamondon. 1998. A kinematic theory of rapid human movements: Part III. Kinetic outcomes. *Biological Cybernetics* 78, 2 (1998), 133–145.

[12] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.

[13] Chao Shen, Zhongmin Cai, Xiaohong Guan, and Roy Maxion. 2014. Performance evaluation of anomaly-detection algorithms for mouse dynamics. *computers & security* 45 (2014), 156–171.

[14] Ruben Tolosana, Javier Gismero-Trujillo, Ruben Vera-Rodriguez, Julian Fierrez, and Javier Ortega-Garcia. 2019. MobileTouchDB: Mobile touch character database in the wild and biometric benchmark. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 0–0.

[15] Paul J Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE* 78, 10 (1990), 1550–1560.

[16] Jacob O Wobbrock, Andrew D Wilson, and Yang Li. 2007. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. 159–168.

[17] Gongqing Wu, Huicheng Zhang, Ying He, Xianyu Bao, Lei Li, and Xuegang Hu. 2019. Learning Kullback-Leibler divergence-based gaussian model for multivariate time series classification. *IEEE Access* 7 (2019), 139580–139591.

7 Appendix

7.1 Goodness distribution, Standardization, low threshold

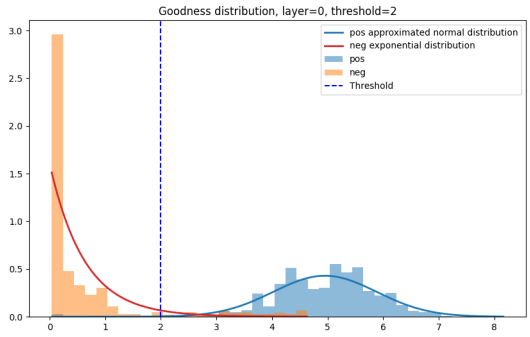

Figure 4. Goodness score distributions for layer 0 in the MCYT dataset, $\theta = 2$. The blue and red lines represent approximate normal and exponential distributions, respectively.

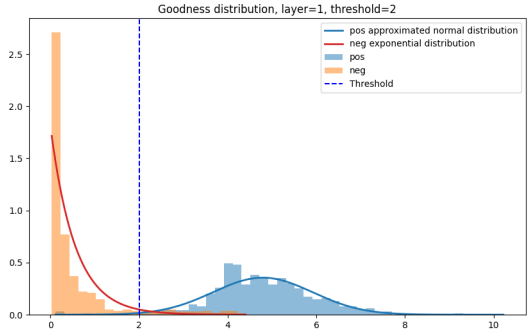

Figure 5. Goodness score distributions for layer 1 in the MCYT dataset, $\theta = 2$. The blue and red lines represent approximate normal and exponential distributions, respectively.

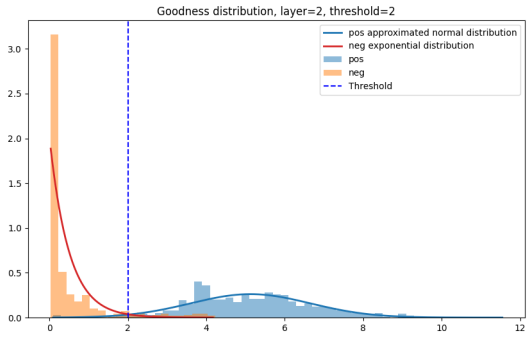

Figure 6. Goodness score distributions for layer 2 in the MCYT dataset, $\theta = 2$. The blue and red lines represent approximate normal and exponential distributions, respectively.

7.2 Goodness distribution, Standardization, high threshold

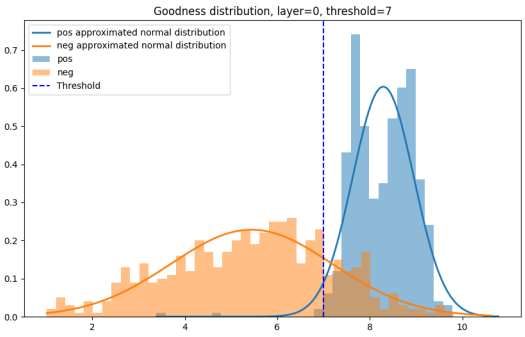

Figure 7. Goodness score distributions for layer 0 in the MCYT dataset, $\theta = 7$. The blue and orange lines represent approximate normal distributions for positive and negative samples, respectively.

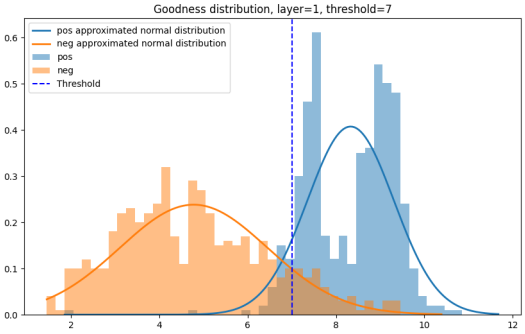

Figure 8. Goodness score distributions for layer 1 in the MCYT dataset, $\theta = 7$. The blue and orange lines represent approximate normal distributions for positive and negative samples, respectively.

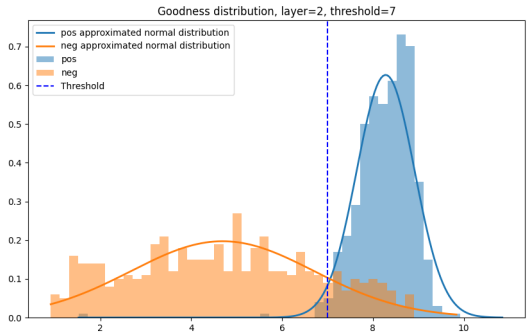

Figure 9. Goodness score distributions for layer 2 in the MCYT dataset, $\theta = 7$. The blue and orange lines represent approximate normal distributions for positive and negative samples, respectively.

7.3 Goodness distribution, Layer Normalization

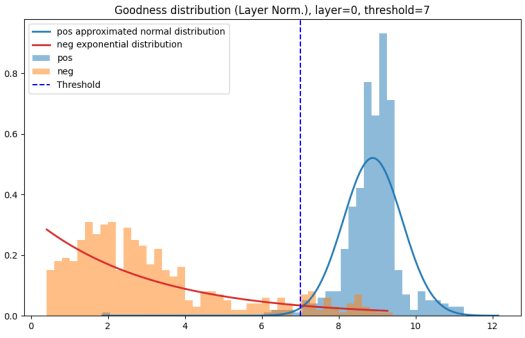

Figure 10. Goodness score distributions for layer 0 in the MCYT dataset, $\theta = 7$. The blue and orange lines represent approximate normal distributions for positive and negative samples, respectively.

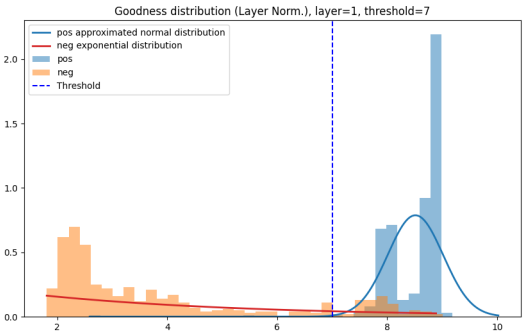

Figure 11. Goodness score distributions for layer 1 in the MCYT dataset, $\theta = 7$. The blue and orange lines represent approximate normal distributions for positive and negative samples, respectively.

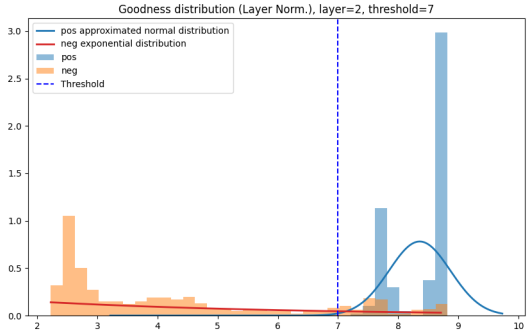

Figure 12. Goodness score distributions for layer 2 in the MCYT dataset, $\theta = 7$. The blue and orange lines represent approximate normal distributions for positive and negative samples, respectively.

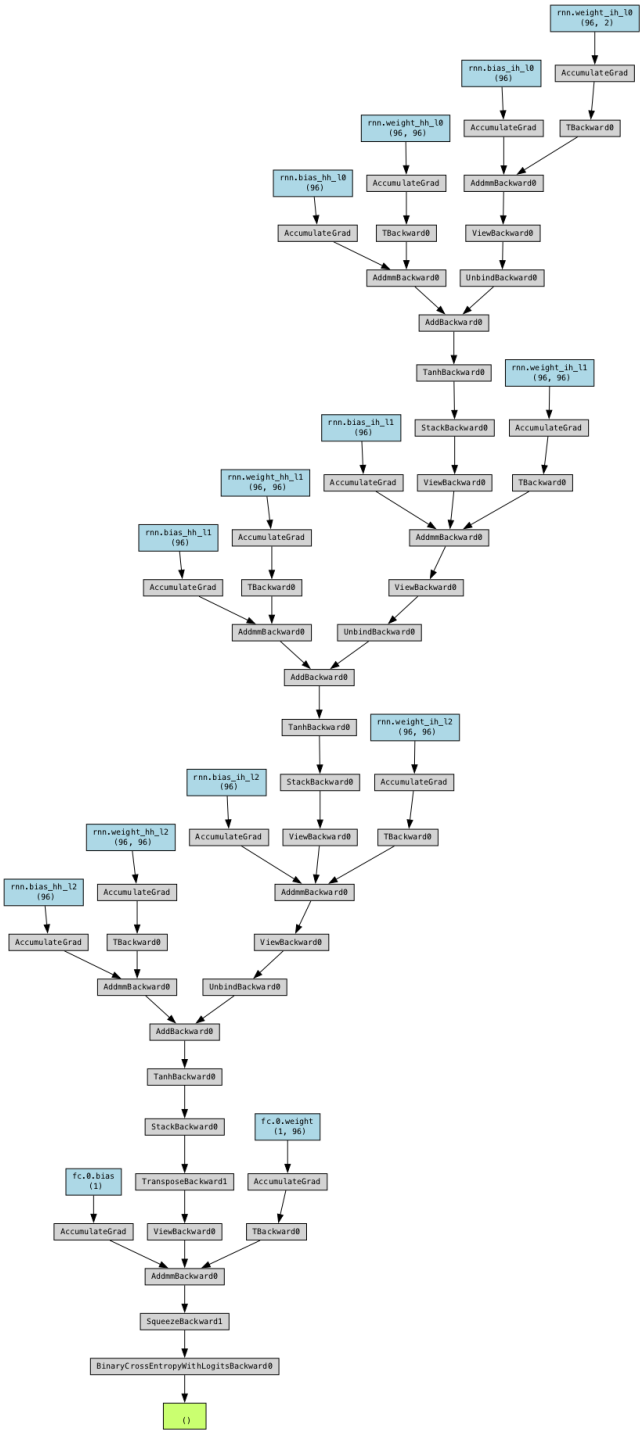

Figure 14. Computational graph for Backpropagation algorithm optimization problem. The gradient calculation is strictly sequential in this case, making it less suitable for parallelization compared to the Forward-Forward approach. Blue nodes correspond to trainable parameters of RNN: weights and biases.

7.4 Computational graphs

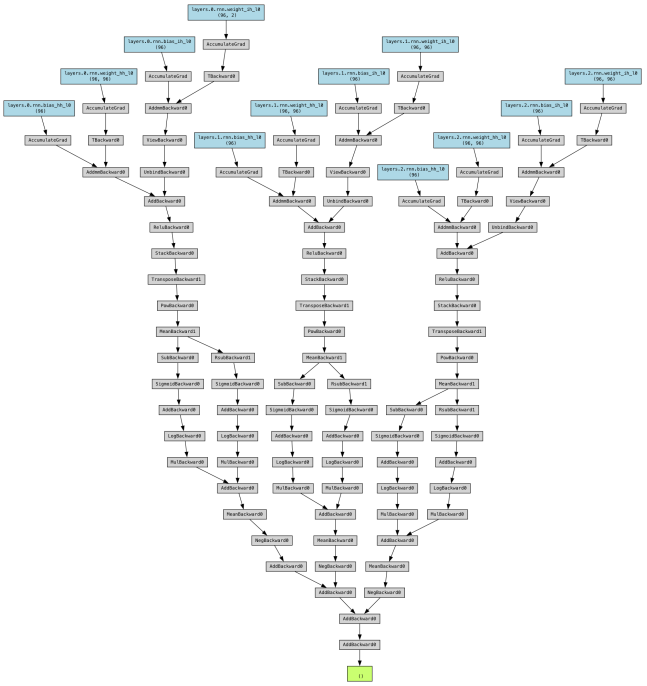

Figure 13. Computational graph for Forward-Forward algorithm optimization problem. An early separation at the root (green node) signifies the independence between loss functions of each individual layer. Blue nodes correspond to trainable parameters of RNN: weights and biases.