Comparative Analysis of Feed Forward Algorithms Against Backpropagation

Team members: Jesse Inouye, Shashvat Shah, Flavjo Xhelollari

[1]New York University Tandon School Of Engineering
{*jai9962, sss9772, fx2078*}*@nyu.edu*

Abstract

This project aims to implement and evaluate the Forward-Forward (FF) algorithm proposed by G. Hinton (Hinton 2022), a novel learning procedure for neural networks. The FF algorithm replaces the forward and backward passes of backpropagation with two forward passes using positive and negative data, offering potential simplification in learning and video processing without the need for activity storage or derivative propagation. The focus of this project is on the simple feed-forward supervised method of the FF algorithm, implemented using PyTorch. The primary objective is to achieve error rates similar to the original paper (around 1.36% after 60 epochs for regular MNIST images) and compare its performance with backpropagation-based models of similar size across varying epochs and training times. Additionally, we will explore various implementations of the FF algorithm on the MNIST dataset, including different architectures with varying numbers of layers and learning rates. It is important to note that due to the limited availability of previous works and resources on the FF algorithm, this project aims to fully understand the algorithm's inner workings and provide a robust implementation with additional functionalities. To achieve this, we will present various architectures for the MNIST dataset as benchmarks for the project and also explore the application of the FF algorithm on the CIFAR-10 dataset.

The primary objective of this project is to assess the effectiveness of the FF algorithm as a viable alternative to backpropagation for training neural networks. By conducting a comprehensive evaluation and comparison with backpropagation on the MNIST dataset, this research aims to provide valuable insights into the performance and potential advantages of the FF algorithm. The code for this paper's implementation can be found in this repository https://github.com/shashvatshah9/FFPytorch

Introduction

The paper by G.Hinton (Hinton 2022) presents a new learning procedure for neural networks called the Forward-Forward (FF) algorithm. Although backpropagation is nearly ubiquitous in modern machine learning and deep learning models, there is little evidence that the brain learns in such a manner (Y. Song and Bogacz 2020). In an attempt to more closely mimic biological brain function, the FF algorithm replaces the forward and backward passes of backpropagation with two forward passes, one with positive data and the other with negative data. The algorithm has shown promising results on small problems and has the potential to simplify learning and enable video processing without storing activities or stopping to propagate derivatives.

In their initial research, Hinton uses three main overarching methods to evaluate the FF algorithm with the MNIST dataset - a simple feed-forward unsupervised method, a simple feed-forward supervised method, and a multi-layer recurrent neural network. In all methods they use a model with roughly four hidden layers each containing 2000 ReLUs, with slight variations to test the impact of different learning techniques. In the unsupervised method, they test with both fully connected layers and local receptive fields. In the supervised method, they test with different forms of classification - either feeding the network a "neutral label" composed of ten equal entries of 0.1 to represent the ten digits, or feeding the network each of the ten labels in separate runs and choosing the label with the best accumulated goodness. In the multi-layer recurrent method, the activity at any given layer is determined by the activity of the layers on either side.

In this project, we discuss the from-scratch implementation of the supervised version of this algorithm in PyTorch and perform a comparison with vanilla backpropagation by evaluating their performance on the MNIST dataset.

Implementation

The Forward Forward Algorithm

The Forward-Forward algorithm is a learning approach that draws inspiration from Boltzmann machines and Noise Contrastive Estimation. It introduces a unique twist to the traditional backpropagation method by employing two forward passes with opposing goals. In this algorithm, the positive pass adjusts the weights to enhance the goodness in each hidden layer, while the negative pass adjusts the weights to decrease the goodness. The goodness is evaluated using various metrics, such as the sum of squared neural activities or the negative sum of squared activities. The ultimate objective of the learning process is to accurately classify input vectors as positive or negative data. This classification

is achieved by applying a logistic function to the goodness. The algorithm aims to ensure that the goodness surpasses a predetermined threshold for real data and falls significantly below that threshold for negative data. The negative data can be either predicted by the neural network through top-down connections or supplied externally.

$$p(positive) = \sigma\left(\sum_j y_j^2 - \theta\right) \quad (1)$$

The FF algorithm implementation on the MNIST dataset involves a key aspect: generating negative data. In the original paper by Hinton, a method is hardcoded to combine two real images with two randomly generated noisy images. This combination ensures that the resulting "output" inherits features from both sources, effectively creating negative data that is related to real data. The paper argues that working with this negative data, aiming to make it more confusing, can enhance the learning capability of the neural network when using the FF algorithm. Furthermore, one might be interested in knowing how the model learns via the FF algorithm. Let's consider a supervised situation, which is the one we will be setting up the implementation later on. We get the input data, in our case MNIST images, and we make sure that the positive data includes labels, and for negative data, we use incorrect labels. At this point, what we want the network to do, is for it to be able to learn the difference between images with the correct label, and the ones with the wrong label. For more details on the way the network learns, one can refer to the Top-down Effect section in (Hinton 2022). Note that the situation is a bit more complicated when dealing with CIFAR-10, as this data set contains three color channels that must be stacked into one.

Overview

Our implementation focuses on the simple feed-forward supervised method of the FF algorithm, built using PyTorch and tested on the MNIST dataset. We have built a small, generic neural network with fully connected layers containing Gaussian Error Linear Unit (GELU) activations. The layers can be dynamically generated upon model initialization by passing input and output dimensions to the model instance. For the purposes of this project, we used models of varying layer sizes from 300 to 2000, to compare with models of similar sizes using backpropagation.

Similar to the original implementation of the FF supervised network by Hinton, we use MNIST data with a one-hot encoding embedded in each image. Using the first ten pixels, we can assign a label by zeroing out every pixel except the labeled pixel in a serial manner - i.e. the first pixel represents the label for class 1, the second pixel for class 2, etc. We can then generate positive and negative data by embedding a label that is either true to the image classification or false.

On top of the original implementation, we also trained the model on the CIFAR10 dataset. The additional challenge solved here is that the CIFAR10 dataset contains 3 channel images, instead of the grayscale images of the MNIST dataset. And to handle this additional dimensionality in data, we encoded the label values to all three channels of the image, and then stack them horizontally. After that, we just use the image, a 1-dimensional tensor and generate the positive and negative data just as mentioned for the MNIST dataset implementation.

The original paper by (Hinton 2022) doesn't mention explicitly the loss function that they have used for their experiments, yet in (Ororbia and Mali 2023) the loss function used for the experiments they run on MNIST with an adaptation of FF, called 'Predictive Forward Forward', is defined as :

$$\frac{1}{2}\left(\frac{1}{N}\sum_n^N log(1+e^{g_{pos}+\theta}) + \frac{1}{N}\sum_n^N log(1+e^{g_{neg}-\theta})\right) \quad (2)$$

Where g_{pos} is the output from all neurons in a layer with positive data, g_{neg} is the output from all neurons in a layer with negative data, θ is a predetermined threshold that the positive and negative data is pushed away from, and N is the number of neurons in each layer.

Technical Details

During initial testing, our implementation consisted of two layers with dimensions $784 \rightarrow 500 \rightarrow 500$, and trained each layer on all MNIST training images at once. Because we loaded the entire model and all training data onto cuda, the training time was very fast, completing roughly 100 epochs in 35 seconds. However, this method did not mimic the training of normal backpropagation methods - instead of running training data through the entire model in each epoch, it trained each layer for 100 epochs, then moved on to the next layer to repeat the process. It also did not allow for data batching, as the entire dataset was used for training in one very large batch. In an effort to more closely match the methods used in backpropagation, we restructured the code to accept multiple batches and train the entire model batch by batch before completing one epoch.

One unique aspect of the FF algorithm is the ability to train each layer individually. Because the FF algorithm uses the output goodness of each layer to update the weights of that layer only, we are able to run the entire set of training data through one layer at a time, either in batches or one large set. This allows for some unconventional training techniques - namely, we can load one layer of the network at a time onto cuda, significantly reducing the GPU RAM footprint. By combining this with batching, we were able to reduce GPU memory usage from 13.1 GB to only 2.0 GB during training. This also accommodates larger networks on hardware with less memory, with the obvious trade-off of training speed, as loading data on and off of cuda has a significant time cost.

Training in this manner requires a few code changes that deviate from regular backpropagation methods using PyTorch. First, the training data tensors must be "detached" from their computational graphs after each layer to prevent backpropagation to previous layers. We accomplish this by calling 'x.detach()', where x is the training data, after each layer is trained. In doing so, calling 'loss.backward()' does

not perform true backpropagation because it is only updating the weights of the previous layer.

Another advantage of the FF method is that the output layer does not have to match the number of possible classifications, but can instead have any size. This difference can be seen in Figures 1 and 2, where the first displays a simple, downsized representation of the FF network with a large output layer, and the second displays a simple, downsized representation of a similar network using backpropagations with an output layer of size 3 for 3 classes. Following the example of MNIST data, we can use the model to determine the likelihood that an image belongs to one of the ten classes by running the input image through the network ten times, each time with a different classification encoded in the image using the previously described encoding method. For each label, we accumulate the average goodness over every layer, and choose the label with the highest accumulated goodness. However, this requires ten passes through the network for inference on every input image, which significantly slows the evaluation throughput.

To reproduce similar error or accuracy for CIFAR-10 dataset, we first reused the architecture similar to the training architecture proposed by (Hinton 2022). The model consisted of 2 layers with dimensions $3072 \rightarrow 500 \rightarrow 500$. The model didn't converge to the expected accuracy levels. So then we used another model with dimensions $3072 \rightarrow 1000 \rightarrow 1000, \rightarrow 500$. This model again converged to a modest error value of 0.55 on the test dataset. The train and test error for this model have been plotted in Figure 8.

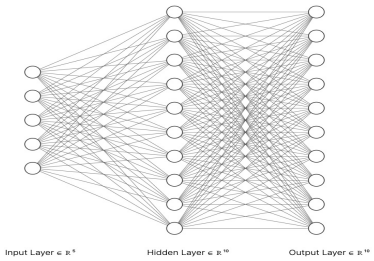

Figure 1: Simple, downsized representation of FF network

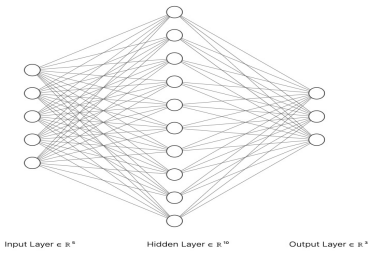

Figure 2: Simple, downsized representation of a similar network using backpropagation

Results

Our approach to testing the FF network involved changing the overall training method, network structure, hyperparameters, and batch size. We first tried two different training methods - one method that trains the entire dataset on each layer, one layer at a time, and another method that splits the dataset into batches and trains all layers on one batch before moving on to the next batch, which more closely matches typical training techniques with backpropagation models. We found that the first method ran much faster, but produced worse results with an error rate around 6.8% after 100 epochs, while the second method achieved an error rate around 3.49% after 100 epochs.

We then used the second method, which we will call "batched training", on three different network architectures with the following layer dimensions: architecture 1 - [784, 500, 500], architecture 2 - [784, 300, 300, 300], and architecture 3 - [784, 2000, 2000, 2000, 2000]. The errors for these architectures over ten epochs using a learning rate of 0.03 and batch size of 50 can be seen in Figure 3 below. We can see that architecture 1, the shortest of the three, performed the best. Architecture 3, the largest, performed the worst, with the test error exploding to 90%, far off the scale of the figure below. Using architecture 3 with a learning rate of 0.01 improved the results significantly, however after 40 epochs the error again exploded to 90%. We aren't entirely sure what causes this sudden change, but we theorize that is may be due to the gradient over shooting the local minimum.

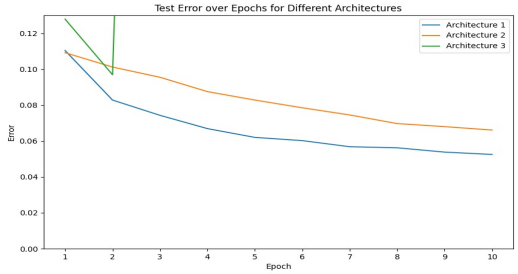

Figure 3: Test errors for architectures 1, 2, and 3, with layer sizes [784, 500, 500], [784, 300, 300, 300], and [784, 2000, 2000, 2000, 2000] respectively

We decided to continue testing with architecture 1, which has 643,000 trainable parameters, comparing the model performance on different batch sizes. With a batch size of 50, we reach an error rate of 3.35% after 50 epochs, as seen in Figure 4. After around 40 epochs, the test error levels out while the training error continues to decrease, which may lead to over-fitting. With a batch size of 10, we reach an error rate of 3.38%, after 50 epochs, as seen in 5. Though after around 40 epochs the model starts over-fitting.

After testing different variations of the FF network, we tested a similar simple neural network with backpropagation. We used two different layer sizes: one with sizes [782, 500, 10], and another with sizes [784, 2000, 2000, 2000, 2000, 10]. Both networks performed significantly better than

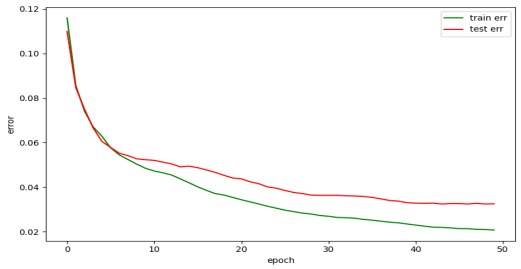

Figure 4: Test error over epochs of the FF network with batch size 50

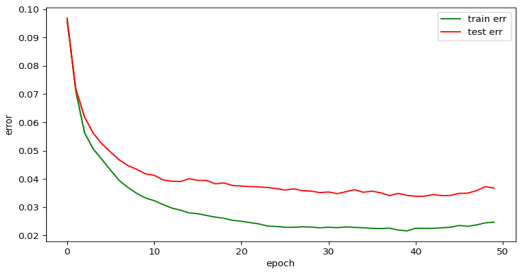

Figure 5: Test error over epochs of the FF network with batch size 50

the FF network after only 20 epochs, as seen in Figure 6 and Figure 7. The smaller network reaches a test error around 2.23% and the larger network reaches a test error around 1.88%.

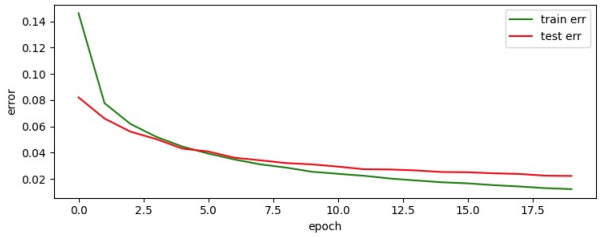

Figure 6: Test error over epochs of the backpropagation model with batch size 50, layer sizes [782, 500, 10]

In the end, we are also showing the error convergence for the CIFAR10 model in Figure 8, where the model architecture was not so competent, but the model did show learning capability just like a normal image classification model.

Conclusion

After conducting multiple experiments involving various model architectures and fine-tuning hyperparameters, our findings demonstrate that fully connected layers have the capability to approximate complex tasks, such as image classification, which conventionally rely on Convolutional Neural Networks (CNNs). However, as the dimensionality of the data increases, there is a need for more efficient data encoding techniques within the fully connected network. Currently, we employ a simplistic approach of encoding the labels as one-hot values, which proves to be impractical for larger datasets like Imagenet (ima 2009). It is also clear that backpropagation on networks of similar sizes is considerably more efficient, producing lower error rates in fewer epochs.

Moreover, we observed that increasing the number of layers in the network did not yield improvements in the classification results. This observation suggests that shallow models perform on par with deep CNNs. In fact, introducing additional layers to the fully connected architecture led to similar challenges as encountered in deep CNNs, such as the issue of vanishing gradients. Consequently, the advancements made in scaling the Forward Forward network for deep networks can be repurposed to enhance the Forward Forward architecture, making it more applicable to refined and expansive datasets.

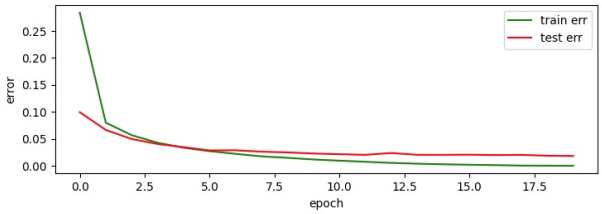

Figure 7: Test error over epochs of the backpropagation model with batch size 50, layer sizes [784, 2000, 2000, 2000, 2000, 10]

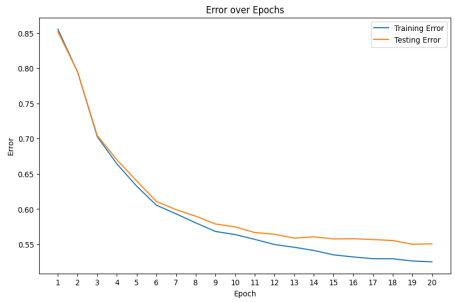

Figure 8: Error over epochs for CIFAR10 dataset

References

2009. ImageNet: A large-scale hierarchical image database.

Hinton, G. 2022. The forward-forward algorithm: Some preliminary investigations. *arXiv preprint arXiv:2212.13345*.

Ororbia, A.; and Mali, A. 2023. The Predictive Forward-Forward Algorithm. *arXiv preprint arXiv:2301.01452*.

Y. Song, Z. X., T. Lukasiewicz; and Bogacz, R. 2020. Can the Brain Do Backpropagation? —Exact Implementation of Backpropagation in Predictive Coding Networks. *Advances in neural information processing systems*.