Forward-Forward: Is it time to bid adieu to backprop?

Sai Anuroop Kesanapalli
kesanapa@usc.edu

Shashank Rangarajan
sr87317@usc.edu

Anukaran Jain
anukaran@usc.edu

Avtaran Jain
avtaranj@usc.edu

Abstract

The aim of this work is to delve deeper into the Forward-Forward (FF) algorithm (Hinton, 2022) by characterizing and analyzing its performance in comparison to the traditional backpropagation (backprop) approach. The FF algorithm, by eliminating the need for chaining gradients and storing activations, allows a neural network to be trained with limited memory. The FF algorithm performs on par with backprop on benchmark datasets, but showcases deteriorating performance as the complexity of the dataset increases. Despite this, FF demonstrates its potential advantages over backprop due to its significantly lower run times for the datasets in consideration. This opens up an opportunity for further exploration into both system analysis and performance of hybrid models, which we aim to focus in the next phase.

1 Methodology

(Hinton, 2022) explores the feasibility of the FF approach when compared to backprop and shows that FF performs nearly as well as backprop when dealing with simple feed-forward multi-layer neural networks. We were able to reproduce the results demonstrated in the paper (Table 1). Additionally, we experimented with the algorithm's performance on various datasets and gauged the system metrics for both FF and backprop (Sections 2 and 3).
We explore three different set of models in our work:

1. **`baseline_ff_model`:** We built on top of the code available on GitHub (Pezeshki). The model contains 2 fully connected layers, each of size 500. We also added relevant `Dataloader` for different datasets 2.1. Furthermore, we adapted the `overlay` method that produces positive and negative samples shown in Figure 1 for the CIFAR100 dataset. Next, we implemented our logging harness as described in Section 2.

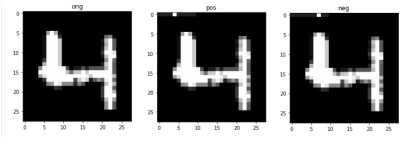


Figure 1: An example of positive and negative samples in FF

2. **`baseline_CNN_model`:** We adapted the same code (Pezeshki) to create a custom `Conv2D` layer that has a goodness objective. This initial implementation of ours did not yield promising results and hence was excluded out of the scope of this project.

3. **`baseline_backprop_model`:** We implemented the same architecture as that employed in Point 1 but with slight modifications as described in Section 2, and trained using backprop.

2 Experimental Setup

We performed experiments to compare the E2E and GPU Compute times of FF and backprop. The best mode of operation for both the approaches were noted down, while keeping the neural network architectures common across implementations.

2.1 Datasets

For our experiments, the following 5 vision datasets were used – MNIST (LeCun et al., 2010), FashionMNIST (Xiao et al., 2017), CIFAR10, CIFAR100 (Krizhevsky, 2009) and SVHN (Netzer et al., 2011).

2.2 Forward Forward Models

We used the `baseline_ff_model` described in [1], and ran the experiments with 1000 epochs per layer. Here, an epoch means that the FF algorithm has *seen* the entire train dataset once. We used a single batch with the entire datasets for both train and

test. The training happens layer-wise – the first layer gets trained for 1000 epochs followed by the second layer. Adam optimizer was used with a learning rate of 0.03 and the *sum of squared activites* as the goodness function with a threshold value of 2.0, as described in (Hinton, 2022).

2.3 Back-propagation Models

The `baseline_backprop_model` [3] is used for backprop, with the addition of a final classification layer to match the output class dimensions. We trained the model for 20 epochs and used a batch size of 256 across the datasets. We used *cross entropy* loss and Adam optimizer with a learning rate of 0.001.

2.4 Metrics Logged

Three main metrics, namely – `E2E_time`, `GPU_compute_time`, and `accuracy` were analysed. `E2E_time` measures end-to-end time for training the model, and `GPU_compute_time` measures the time during which training occurs on GPU. For logging purposes, we used `time.time()` for capturing `E2E_time` and `torch.cuda.Event` for `GPU_compute_time`. We only observed a $\sim$ 20 miliseconds difference between the two utilites. The experiments were carried out on Google Colab environment. We propose to replicate the same on CARC resources in the next phase of our project for inclusion of those results in the final report.

3 Results

3.1 `E2E_time` comparision

Figure 2 shows E2E times of of FF and backprop for different datasets. We made the following observations:

- E2E times for FF remain below 100 seconds while exceeding 250 seconds for backprop. This highlights that FF is $\sim 3.7\times$ faster than backprop.
- Figure 2b shows the per layer E2E times of FF, and we note that the difference is attributed to the warmup overheads (dataset download, image preprocessing).
- In Figure 2c we note an anomaly of the magnitude $\sim$ 280 seconds in SVHN, which we again attribute to the large download size.

We will address these artifacts and make our logging more fine-grained for the final-report. The key takeaway is that FF is far more sensitive to minor overheads than backprop, due to its E2E time being much lower in magnitude than the latter.

3.2 `GPU_compute_time` comparison

Figures 3 and 4 show the per epoch GPU Compute times for FF and backprop for different datasets. We make the following observations:

- We note that the size of the dataset plays a crucial role in determining the GPU compute time per epoch in FF. MNIST (Figure 3a) and FashionMNIST (3b) have similar sizes and hence similar compute times. The same is observed for CIFAR10 (3c), and CIFAR100 (3d). Whereas, it remains somewhere in between for SVHN (3e). Overall we observe fluctuations due to the millisecond scale.
- From Figure 4 we note that the scale is in the order of seconds. This can be explained by the higher time requirement for forward and backward passes in backprop.
- We note that the GPU compute time for backprop remains almost constant $\sim$ 15 seconds for all the datasets except SVHN, where it is $\sim$ 22 seconds.

3.3 Accuracy comparison

From Table 1 we make the following observations:

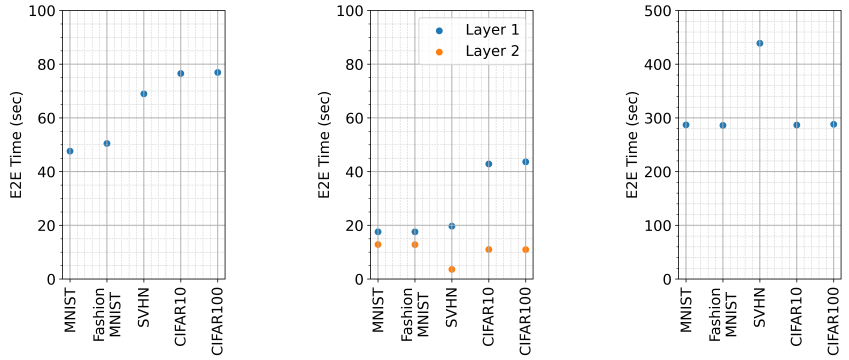
- FF performs comparable to backprop for MNIST, FashionMNIST, and SVHN, while performing poorly for CIFAR10, and CIFAR100.
- We attribute the performance gap to factors such as – batch size, simplistic architecture, and the need for threshold tuning.
- Overall FF still offers an advantage compared to backprop in terms of the E2E times. We believe this will hold true for memory requirements as well (which we aim to study in the next phase).

4 Next Phase

1. Include CPU/GPU memory utilization, power and energy consumption in our logging harness. However, its feasibility depends on availability of `tegrastats` and level of access to the compute nodes.
2. Design a hybrid FF + backprop approach and perform similar system analysis with our (extended) logging.
3. Explore implementing Attention mechanism with FF.

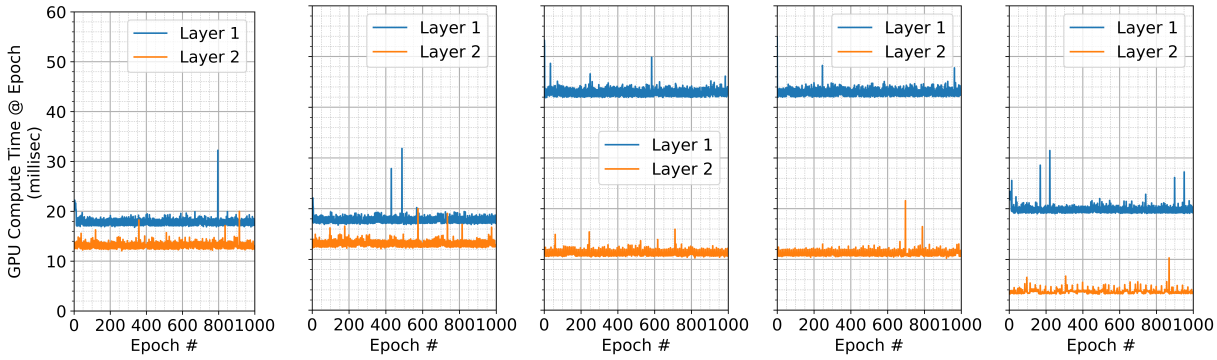
	MNIST	FashionMNIST	CIFAR10	CIFAR100	SVHN
FF	0.932/0.931	0.83/0.83	0.49/0.46	0.13/0.12	0.63/0.60
backprop	0.99/0.97	0.95/0.89	0.93/0.53	0.77/0.23	0.88/0.81

Table 1: Train / Test accuracies of FF and backprop



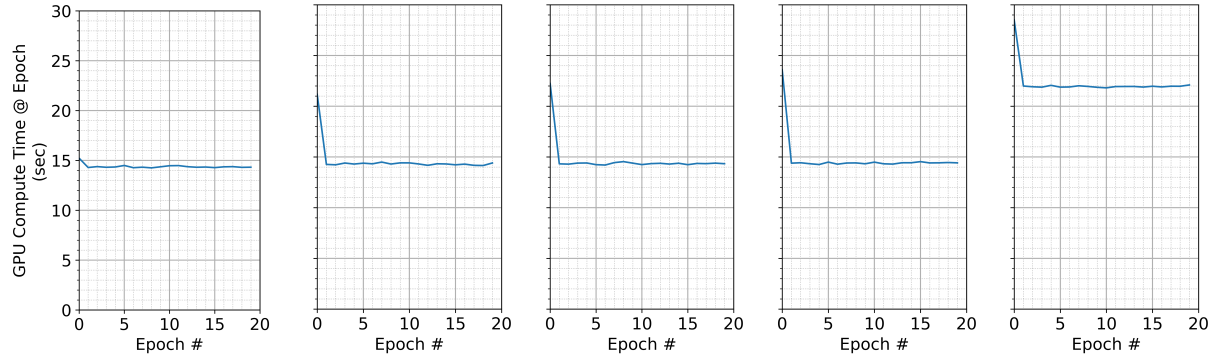
(a) FF | E2E time (b) FF | E2E time per Layer (c) backprop | E2E time

Figure 2: E2E time vs Dataset



(a) MNIST (b) FashionMNIST (c) CIFAR10 (d) CIFAR100 (e) SVHN

Figure 3: FF | GPU Compute time



(a) MNIST (b) FashionMNIST (c) CIFAR10 (d) CIFAR100 (e) SVHN

Figure 4: backprop | GPU Compute time

References

Geoffrey Hinton. 2022. The forward-forward algorithm: Some preliminary investigations.

Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. Technical report.

Yann LeCun, Corinna Cortes, and CJ Burges. 2010. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. 2011. Reading digits in natural images with unsupervised feature learning.

Mohammad Pezeshki. Mohammadpz/pytorch_forward_forward: Implementation of hinton's forward-forward (ff) algorithm - an alternative to back-propagation.

Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

APPENDIX

A Back-propagation

Back-propagation, also known as backprop, is a widely-used algorithm in training neural networks. Its breakthrough lies in its ability to enable the optimization of models with many layers, by computing the gradients of each parameter with respect to the loss function. This is done by propagating the errors from the output layer back to the input layer, hence the name *back-propagation*. The key idea is to apply the chain-rule in calculus to compute the gradients at each layer, by multiplying the gradients of the layer ahead with the local gradients of the current layer. These gradients are then used to update the model parameters using gradient descent. Backprop is efficient because it stores the gradients and activations, and only needs to compute the gradients once, which can then be reused during the optimization process.

B Forward-Forward

The core idea behind FF algorithm is to use two forward passes, one with positive (real) data and the other with negative (generated) data. Instead of computing the gradients with respect to the global loss, every layer has its own objective function called the *goodness* metric. This allows for the training of every layer in a neural network to be done separately. During training, the weights of each layer are adjusted to increase the *goodness* value above a specified threshold in the positive pass, while the opposite is done in the negative pass. In our experiments throughout, the sum of squared activities is used as the *goodness* function.

B.1 *Goodness* metric

Suppose that the *goodness* function for a layer is the sum of the squares of the activities of the rectified linear neurons in that layer. *Goodness* should be well above some threshold value θ for real data and well below that value for negative data. The aim is to correctly classify input vectors as positive data or negative data depending on the following *logistic* function:

$$p(positive) = \sigma\Big(\sum_j y_j^2 - \theta\Big)$$

$$p(negative) = \sigma\Big(\sum_j y_j^2 + \theta\Big)$$

where y_j is the activity of hidden unit j before layer normalization. This threshold θ becomes a hyper-parameter that can be tuned for better performance.

B.2 Method to produce positive and negative samples

```python
def overlay_y_on_x(x, y, num_labels):
    """Replace the first num_labels pixels of data [x] with one-hot-encoded label [y]
    """
    x_ = x.clone()
    x_[:, :num_labels] *= 0.0
    x_[range(x.shape[0]), y] = x.max()
    return x_
```

Figure 5: `overlay` method snippet

In FF, a positive datum belongs to the training dataset whereas a negative datum is adversarially generated to ensure that the model learns contrastively. An easy way to generate the negative data is to mislabel the training samples. We do this using the `overlay` method in Figure 5.

C Links to scripts

Access to these drive folders will be given on-request.
Backprop implementation:
https://shorturl.at/lxCW9
Backprop plotting scripts:
https://shorturl.at/syM06
FF implementation:
https://shorturl.at/jxEFV
FF plotting scripts:
https://shorturl.at/pJOQV

Individual Contributions

Mid-term report contributions:

- SAK - Sections 2.4, 3.1, 3.2
- SR - Sections 1, 3.3, A
- KJ - Sections 2, 4, B.1
- AJ - Abstract, Sections 1, 2.3, B

Code contributions:

- SR, SAK - Implementing baseline FF
- SR, AJ - Implementing baseline CNN
- SAK, KJ - Implementing logging harness, plotting
- KJ, AJ - Implementing backprop
- SAK, SR, AJ, KJ - Experimenting with datasets

Code, sections and parts of the report not listed above were equally contributed by all the members.

Name Legend:
SAK - Sai Anuroop Kesanapalli,
SR - Shashank Rangarajan,
KJ - Anukaran Jain,
AJ - Avtaran Jain