DREAM Final Report

Kate Raitz

Fall 2023 - Spring 2024

Jacobian-Free Backpropagation in Deterministic Optimization

Introduction: Modern Implicit Learning

Deep learning has cemented itself as an effective tool within the realm of machine learning by providing the ability to navigate the complexities of high-dimensional datasets. The burgeoning of access to Big Data and advancements in hardware capable of sustaining deep learning (DL) network training has led to a growing pursuit of the development of architectures that adeptly navigate the trade-off between minimizing computational resources and maintaining model performance. The contribution of implicit neural networks has emerged as a promising direction, challenging the conventions of traditional neural network architectures, and has led to the advent of Deep Equilibrium Models (DEQs).

The seminal work of Bai et al. (2019) introduced DEQs, which use methods to compute inferences by leveraging fixed points in lieu of explicit functions. These models harness the principles of Fixed Point Theorem (Banach, 1922) to construct networks that iteratively converge to a stable equilibrium point, thereby embodying the notion of 'infinite-depth' within a feasible computational framework. The prospect of training these models with fixed memory costs is attractive, but even with a constant memory quasi-Newton approximation approach, the hurdle of computing a Jacobian-based linear system during backpropagation remains (Bai et al., 2019).

Traditional feedforward networks are defined by their static structures, whereas implicit networks offer versatility through their ability to solve fixed point equations for their inferences (El Ghaoui et al., 2019). Yet at the same time implicit DL models are characterized by their depth-dependent memory cost via their required dynamic computational paths. The revolutionary Jacobian-Free Backpropagation (JFB) method (Wu Fung et al., 2021) lends way to a promising new direction in research by providing computational efficiency by circumventing the Jacobian-based equations that often hinder the training of implicit models. In this project, we seek to determine whether JFB, theoretically adept in deterministic environments, can maintain the comparable proficiency it has shown in stochastic processing under deterministic training regimes.

Contraction Mappings

L-Lipschitz continuity provides critical theoretical insight for understanding the behavior of functions and systems within the framework of DEQs. Lipschitz continuity in this context is a property applied to the function or transformation defined by the neural network model itself, and not the data it is trained on. Contraction mappings offer a more specialized perspective than other less restrictive L-Lipschitz conditions for gauging how functions can distort distances between points (Banach, 1922). A contraction mapping involves a function $f : X \mapsto X$ defined on a complete

metric space X that brings points progressively closer to a unique fixed point. This mapping to itself ensures that starting from any initial point x_0 in X, the iterative process $x_{n+1} = f(x_n)$ will converge to this fixed point.

Contraction mappings impose a global control over the behavior of a function by inherently restricting and bringing the distance between points closer together and ensuring it does not stretch beyond a certain limit, described by the Lipschitz constant L (Banach, 1922). For a mapping to be a contraction, it requires that this constant L be strictly less than 1, which strengthens the guarantee of convergence and stability of the system. This mathematical constraint can be applied to the derivative as a measure of controlling how sensitive the output of a function is to changes in its input. Using this relationship, the contraction principle is applied to DEQs in order to constrain the changes in weight updates in the iterative equilibrium output (Bai et al., 2019), keeping the function stable under small perturbations to the input (Ling et al., 2023). This provides an upper bound and proportionally smaller changes in the output, lending to linear convergence for the model in the finding of a fixed point (El Ghaoui et al., 2019). This stable response to inputs guarantees reliable training and inference in neural networks, especially useful when using techniques like implicit differentiation because of its simplification of the differentiation process during backpropagation (Geng et al., 2021). This characteristic of predictable control over how a function responds to changes in inputs underpins the effectiveness of DEQs.

DEQs

Unlike traditional feedforward neural networks, DEQs are a class of implicit networks that define their outputs through an equilibrium state or fixed point rather than through sequential layer outputs (Bai et al., 2019). This equilibrium is not derived in a stepwise, linear progression, but instead is the solution to an equation involving the network itself. The iterative process adds a dynamic inner loop to the network's operation (Huang et al., 2021), where a function is applied repeatedly to the input until the network's output stabilizes, signifying that the equilibrium state has been reached. During the inference phase, or what is traditionally known as the forward pass, DEQs determine their outputs by finding this fixed point for any given input (Ling et al., 2023), hence assuming potentially 'infinite depth' through the repeated application of the transformation (Bai et al., 2019).

DEQ training still requires backpropagation to compute gradients with respect to the loss function for updating the model's parameters. However, instead of backpropagating through layers, DEQs backpropagate through the equilibrium-finding process itself (Geng et al., 2021). In fact, this is where the distinction between DEQs and traditional models becomes more pronounced: because the equilibrium state is defined implicitly (by the fixed point output with respect to the model parameters), traditional backpropagation does not apply directly. Instead, DEQs utilize implicit differentiation, calculating the gradients without explicitly unfolding the iterative process used during the forward pass (El Ghaoui et al., 2019). Implicit networks based on Jacobian matrices approach this by solving a linear system where the derivatives of both sides with respect to the parameters are considered, making it an implicitly defined function. This approach is feasible only because of the existence of a fixed point, which guarantees the differentiability with respect to the same parameters across the iterative applications of the transformation function (Geng et al., 2021) and for the norms of the Jacobian matrix to be bounded above by 1 (Banach, 1922). This is what allows the model to compute gradients and update parameters efficiently, even for 'infinitely deep' models represented by the equilibrium state.

A distinctive feature of DEQs is how they are parameterized, or how the network computes its outputs based on its inputs, with their use of weight-tied, input-injected layers (Bai et al., 2019). In this paradigm, weights are reused across different iterations, mimicking the effect of an infinitely deep

network, and the original input is reintroduced at each iteration, ensuring that the transformation $T(u; d)$ considers fresh input data alongside the evolving outputs.

Traditionally with DEQs, backpropagation involves intermediate hidden units from each iteration throughout the root-finding inference process: the process of implicit differentiation requires the computation of an inverted Jacobian matrix, resulting in a computationally intensive process to achieve equilibrium. Each iteration's evaluations are stored in a computational graph, allowing for backpropagation through what effectively becomes a deeply nested function (Huang et al., 2021). However, this can lead to high memory requirements, particularly for networks that represent deeper datasets.

JFB

Jacobian-Free Backpropagation (JFB) (Wu Fung et al., 2021) represents a significant advancement in the training of implicit neural networks. The core innovation of JFB lies in its approach to handling the Jacobian matrix, as traditionally its computation is demanding in high-dimensional spaces (Bai et al., 2019). JFB addresses these challenges by removing the need for the matrix-vector product computation of the Jacobian term at each step, and exchanging it for an identity operation. This effectively applies a preconditioner, or a more accelerated, straightforward gradient descent (GD) process, all while maintaining or even improving accuracy measures. By using a higher order Neumann approximation and avoiding the direct computation of the Jacobian matrix, JFB increases computational efficiency for what was already an architecture providing training at a fixed memory cost, making it more feasible for training a wider range of architectures and models (Wu Fung et al., 2021).

JFB's installation to DEQs represents an exciting confluence: while DEQs backpropagate on only the final layer of iteration (representing the converged-upon fixed point), JFB fundamentally changes that gradient. When the backpropagation step successfully performs implicit differentiation, it does so in a mysterious way by collecting a gradient based on the trivial identity operator. By fundamentally intervening with the backpropagation process responsible for the training overhead and transformation instability in DEQs, JFB accelerates training but also improves the stability and scalability of models (Wu Fung et al., 2021). All together, due to the nature of implicit differentiation, the substitute of the Identity, and the tracking of only the final layer or the last iteration during backpropagation, JFB reduces the memory and computational demands typically associated with training deep networks. It remains to be fully investigated to what extent this approximation is limited, or how it impacts the training environment.

Stochastic and Deterministic Processing

In DL, training methodologies can be broadly classified into stochastic and deterministic approaches. While the use of mini-batches or stochasticity in training methodologies is a hallmark of modern DL practices, there are notable benefits to both kinds of processing (Hoffer et al., 2018; Keskar et al., 2016). These choices in processing techniques represent the trade-off in training between robustness, accuracy, and generalization of the model, with the latter often being the most important net goal.

Stochastic optimization is commonly implemented through mini-batch training by dividing the dataset into small batches. Each training iteration updates the model parameters based on the gradient computed from just that randomly selected mini-batch. This approach is repeated for a designated number of iterations, conventionally sampling from the entire dataset over the course of repeated random mini-batches. Stochastic Gradient Descent (SGD) is the default optimization choice in real world applications because it is often more effective than deterministic GD at avoiding

local minima or saddle points due to the noise introduced by the random selection of mini-batches (Hoffer et al., 2018). This noise can help a function with a complex landscape reach robust equilibrium states as well as enhance the model's ability to generalize to unseen data. Additionally, by processing fewer data points at each step, SGD optimizes more quickly compared to full-batch training, and so this subset processing can speed up training for large datasets.

Deterministic optimization, on the other hand, involves training models using the entire dataset at once. This method, often referred to as full-batch training, computes the precise global gradient of the loss function across all data points before updating the model parameters. Full-batch GD provides noise-free gradient estimates and bodes poorly for generalization because of its large updates that converge to sharp minimizers (Keskar et al., 2016). However, deterministic processing on high parallel computing hardware will have shorter epoch times because of amortization capabilities, resulting in better performance when fully used with large amounts of data (You et al., 2019).

As shown in Keskar et al.'s work (2016), it's often the case in traditional DL architectures that training methods that make more fine-grained updates make achieving global convergence easier, but it has yet to be shown that this is the case with JFB architectures which fundamentally change the Jacobian term defining the gradient of the cost. By not computing the Jacobian matrix directly, models relying on the output of JFB approximations may forego the large parameter updates that typically disrupt the training process in deterministic settings (Lee et al., 2023). Without fully understanding the limitations of this new approach, it cannot be assumed that gradients computed from the entire dataset will lead to overshooting or poor generalization.

The primary goal of this project is to determine the impact of deterministic processing within the framework of the JFB architecture in DEQs. By exploring how deterministic training influences the stability, convergence rate, and generalization of DEQs, we aim to identify the feasible and optimal training strategies in the context of JFB-based implicit networks. This exploration of different training paradigms is crucial for advancing our understanding of the performance of innovative neural network architectures.

Experiments

The experiments performed on a CPU using PyTorch are separated into analyses of mini-batch and full-batch training under fixed architectural parameters to methodically evaluate the performance of JFB in the opposing training settings. Side-by-side statistical analyses are gathered to draw comparisons in order to reach our conclusion. Code can be found at:
`https://github.com/kraitz/kraitz.github.io/files/deterministic_JFB.py`.

Architecture Design

Dataset

This empirical examination uses an abstract dataset with an input in $\mathbb{R}^3$ emulating a heat transfer equation for the implicit neural network to approximate its output with a scalar. The network is a regression model predicting the temperature values within a room of typical dimension (20, 13, 8: length, width, height) from an input dataset of size 3,000.

Training and validation datasets were split 80/20 and seeded to keep the randomly split data points consistent across batch size processing. Model width was limited to a latent space dimension of 3 to transform the input data. The target data was intended to have a complex optimization landscape which might contain numerous local minima, saddle points, or flat regions, making it challenging for the optimization algorithm to find a global minimum. The heat transfer function

was:

$$tanh\left(\frac{x}{1+e^{-y}}+\frac{y}{1+x^2+z^2}\right)$$

Network

The fixed point iteration employed our transformation to update the latent variable, and is mathematically represented as $T(u;d)$. This transformation involves the input data d getting mapped to the latent space $\mathcal{U}$ (call this mapping Q_Θ), then iteratively performing a self-map until convergence (call this self-map looping operation R_Θ), all while still injecting the original input d. ReLU was chosen as an activation function.

The model depth defining the latent space simply consisted of 3 linear layers: a latent layer (encapsulated initially by the latent variable guess of u, then being represented as u^k until convergence at u^*), an input-injected layer (encapsulated by feeding $Q_\Theta(d)$ into all calls of R_Θ effectively except the first initial guess), and an output layer (represented by the inference of y). Put together, the compositional transformation defining the forward pass of this model is

$$u^{k+1} = R_\Theta(u^k; Q_\Theta(d))$$

The stopping criterion for this fixed point iterative process used the infinity norm $\|\cdot\|_\infty$, representing the maximum change in any component of the tensor when measuring the difference between successive iterations, determining convergence once this difference fell below a threshold of 0.001. Because fixed parameters were tested between both deterministic and stochastic training, the infinity norm is appropriate here because it doesn't shift or grow with changing dimension or batch size, which would help keep it consistent across configurations. The maximum number of iterations was set to 1,000. Since Banach's Contraction condition states that any initial guess is guaranteed to reach the fixed point (Ling et al., 2023), the latent variable u was initialized with the zero vector of the same dimension as the input data so that the matrix-vector product would be possible during the transformation.

The forward pass is called implicitly due to the redirection of PyTorch's `nn.Module`, and was internally called with the given arguments when the network function instance was called in the training loop. A crucial aspect of the forward pass' definition was delivered in the use of PyTorch's `torch.no_grad()` function, which is critical for use with JFB in particular because of its fixed point iteration without gradient tracking. The main contribution of JFB is that there is no need to add each iterate to the computational graph: it's necessary to keep track of the outputs from the latent space, but not to track the latent variable as it is refined towards its fixed point. This PyTorch function prevents backpropagation, so once training converged to the fixed point, the transformation was applied one final time outside of this function to be able to store the final gradient estimate. This is the only result that gets saved to the computational graph and the final step in the forward pass is to transform it to match the desired output dimension. Then, the training loop backpropagated just through that final layer for 3,000 epochs. The `Adam` algorithm was implemented via PyTorch with a step size of 0.001. The fixed hyperparameters between training processes of varied batch sizes were: maximum epochs, maximum iterations, tolerance for convergence to fixed point, dataset size, latent space dimension and latent variable initial guess, layer configuration, activation function, transformation definition, device, and applied optimization algorithm.

Results

After processing a full pass over the training dataset, loss and accuracy calculations were recorded to serve as indicators of model performance, with a focus on the loss characterizing the optimization process. The results from the predictions after every epoch as well as epoch times and training times

in seconds were recorded for visualization purposes, to see how the model's predictions evolved during training and to compare the different time complexities involved (see **Figures** section). Hyperparameters were kept fixed between training sessions so that any variation in performance could more confidently be attributed to the changes in the configuration rather than those variations.

The cumulative training loss was tracked over batches for both training and validation datasets. Each batch's loss was multiplied by its respective batch size to account for any varying number of samples in each batch, weighting the loss appropriately and ensuring that batches with more data had a proportionally greater impact on the overall loss. This led to an average training loss per epoch to track loss across training. Validation losses were computed in the same way by iterating over the validation dataset in evaluation mode to ensure consistent behavior across the different runs.

Accuracy was monitored as a metric by turning it into a regression metric: first the absolute error between the predicted values and the actual target values was found and checking if it was within a predetermined threshold. If the absolute difference was less than 0.1 it was within the threshold, meaning the prediction was considered accurate for that particular data point. These boolean values of 1.0 and 0.0 were averaged over the batch to proportionally represent number of correct predictions. Similar to the total loss calculations, the average training and validation accuracy for the epoch was calculated by taking the mean of all batch accuracies.

Figures

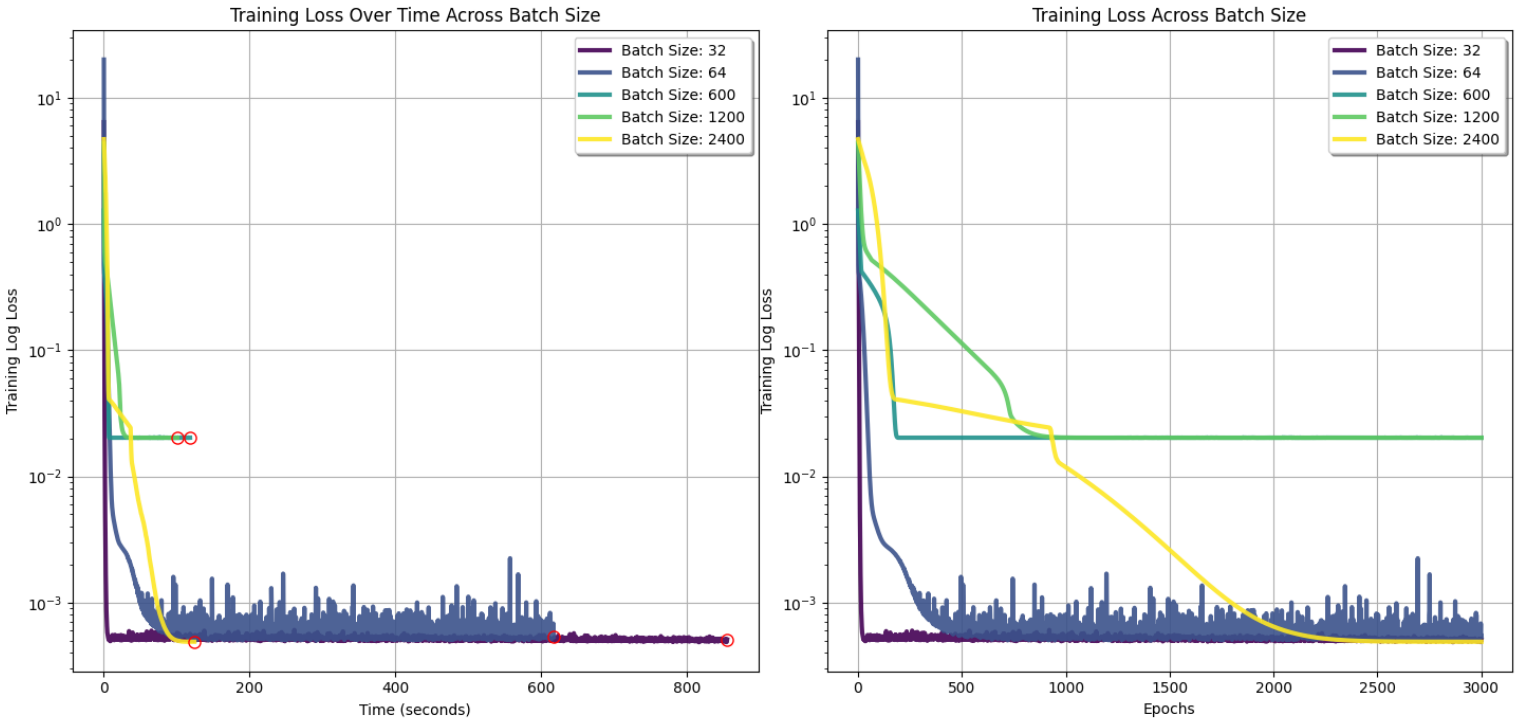

Figure 1: By observing Figure 1, we can compare impacts of batch size on different timing measures for cost function minimization for the training dataset. On the right figure, we see that our full-batch only starting reaching its lowest total error at around 2,500 epochs out of the 3,000 maximum epochs. It continued lowering its cost up until its last epoch. However, as we can see on the

left figure, those 2,500 epochs took significantly less time to run, as the full 3,000-epoch training was completed in 146.5 seconds. Meanwhile, our lowest mini-batch losses which were equivalent to the full-batch lowest loss came from the two smallest batch sizes of 32 and 64, which both took approximately 5x longer to complete training. Interestingly, the largest mini-batches of sizes 1,200 and 600 took comparable time to complete training as the full-batch (despite being 50% and 25% smaller, respectively, than the full batch size), but also had lowest loss values that were two orders of magnitude larger.

The stability of the lowest error value for the full-batch training indicates convergence was reached. This seems to be the case for all mini-batches as well except for the 2nd smallest size of 64, which has a continually oscillating cost function, indicating it did not reach a stable state. The smallest batch size, though showing smoothing towards the final epoch, also has some disturbances in its function rather than a smooth loss progression. These two smallest batch sizes are a significantly smaller proportion of the whole dataset, chosen in order to make direct comparison against. The noise introduced via the mini-batches appears to degrade the model performance once size becomes less than approximately 2% of the full dataset (as is the proportion of batch size 64 for a full-batch of 2,400). This could also suggest less robustness to initializations and a requirement for more fine-tuning of model parameters, such as making the tolerance for convergence higher. Overall, we can use these timing plots to understand total training time for different batch sizes and the batch size impact on number of epochs needed to minimize cost.

Table 1: Cost Minimization Across Batch Sizes

Batch Size	Avg time per epoch (s)	Lowest Loss Value	Epoch of Lowest Loss
2400	146.5	0.0005	3000
1200	100.5	0.0202	2561
600	114.2	0.0202	2775
64	597.0	0.0005	1410
32	750.8	0.0005	2863

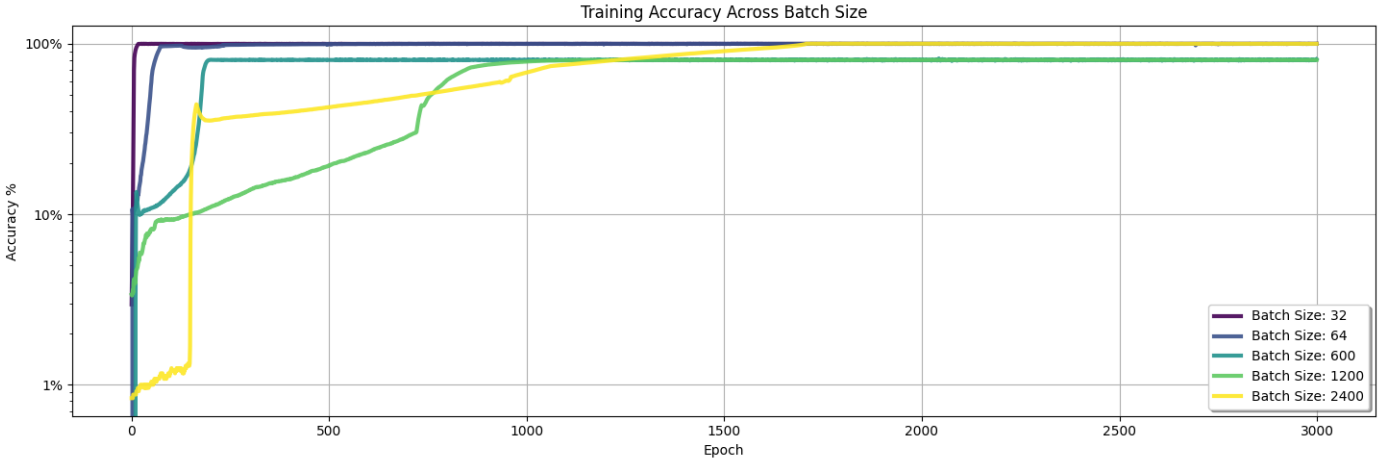

Figure 2

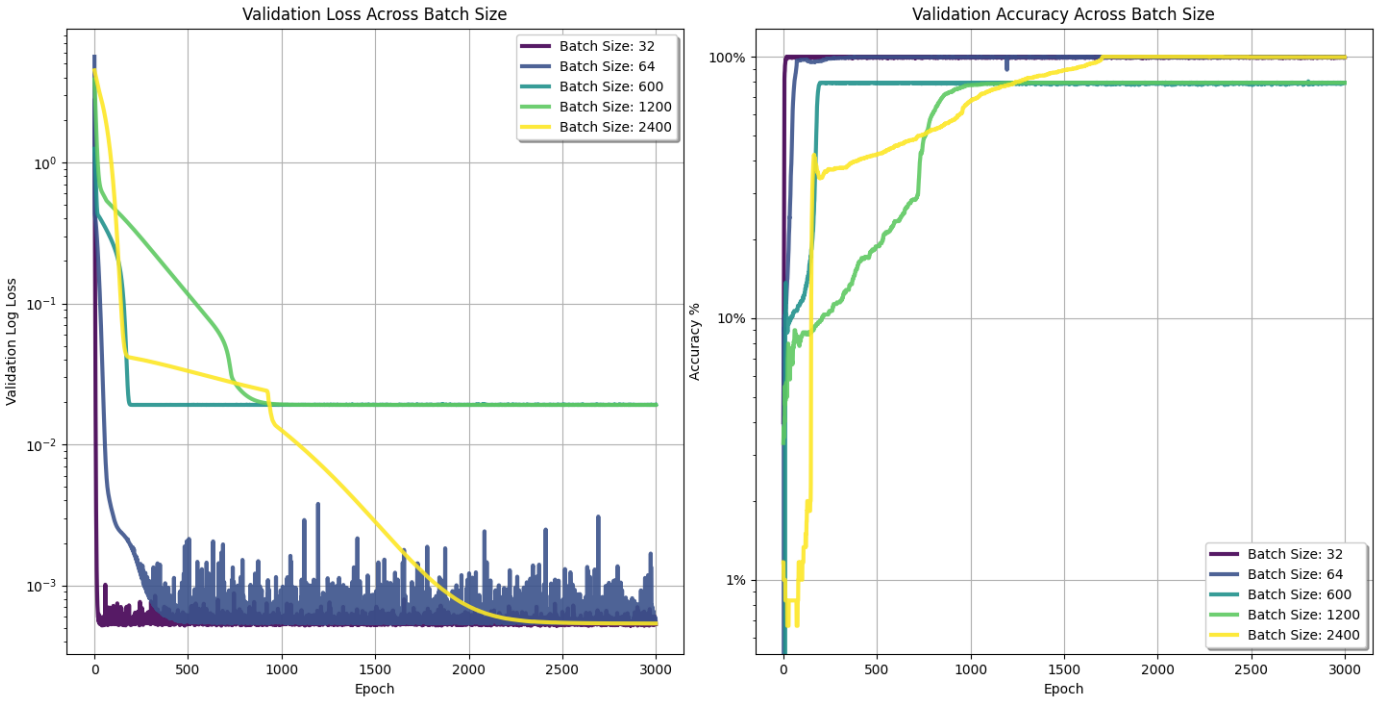

Figure 3

Figures 2 and 3 show us further performance metrics helping to evaluate the model. Training and validation statistics mirrored the other, indicating model generalization across all batch sizes. By applying this model to the different configurations and exposure of subsets of data, we can better assess the stability of JFB in different training environments.

Conclusion

The conclusions drawn from our experimental observations aim to shed light on the effectiveness of JFB in deterministic processing environments. Our goal was not to use or analyze the predictions given directly from the model, but rather to evaluate the model's performance on a validation set or during training. Our results numerically reinforce the robustness of JFB: with the right balance of hyperparameters, deterministic JFB performs comparably or better than stochastic JFB applications. Not only does a deterministic model paired with JFB converge to a global minimum, but it does so using significantly less training time than other batches landing on the same minimum. JFB also reaches comparable accuracy as other mini-batches with use of JFB, and it does so about halfway through it's training epochs - representing approximately under a minute. Only the smallest mini-batches representing less than 2% of the dataset size also reach full accuracy, and while they reach this level of accuracy faster than the full-batch (immediately: after approximately 1 second), this would also come at the cost of much longer training time to reach epoch of lowest loss as aforementioned (see Table 1). Our results align with deterministic training providing a consistent estimation of the gradient, ensuring the direction of the updates is always aligned with the global gradient, even when the Jacobian term is effectively removed from the equation of the gradient.

In conclusion, JFB does well without the noise introduced via mini-batch training in regards to timing and performance. Full-batch processing simplifies the evaluation of the loss gradient and produces a single, consistent gradient at each iteration, which directly points towards the steepest

descent direction for minimizing the loss across the entire dataset. This consistency reduces the variability seen in methods like SGD where gradients vary significantly between iterations. Most importantly, JFB reduces resource utilization, and by describing the operational boundaries within which JFB functions, we can proceed more confidently and efficiently with model development.

Further Work

This study set the stage for further exploration into the integration of deterministic principles in training JFB-based implicit models. Subsequent research will further delve into the limitations and conditions under which JFB maintains its computational efficiency and model performance by exploring its use in large-scale, real-world applications. Since this project utilized a synthetic dataset, the researcher added an error term to model to simulate real-world noise in data measurement, and althought it was set to 0 and not used in this experiment, it has been left in the code to effectively be used to measure the model's robustness to input perturbations. Exploring a more complex dataset can be paired with tuning the transformation function and consequently implementing different networks with more types of layers (e.g. convolutional layers, regularization layers). In this work, it can be dually explored if increasing the latent space dimension and giving the model more parameters to learn from, or adding more layers to increase its depth, may inherently further facilitate full-batch effectiveness. The potential for hybrid stochastic-deterministic training in JFB frameworks may also be explored. While not implemented in this study, experimenting with an adaptive learning rate and utilizing a scheduler may produce different results (You et al., 2019). Other PyTorch or custom optimization algorithms may also be used in accordance.

References

Bai, S., J. Z. Kolter, and V. Koltun (2019). Deep equilibrium models. *CoRR abs/1909.01377*.

Banach, S. (1922). Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. In *Fundamenta Mathematicae*, Volume 3, pp. 133–181.

El Ghaoui, L., F. Gu, B. Travacca, and A. Askari (2019). Implicit deep learning. *CoRR abs/1908.06315*.

Geng, Z., X. Zhang, S. Bai, Y. Wang, and Z. Lin (2021). On training implicit models. *CoRR abs/2111.05177*.

Hoffer, E., I. Hubara, and D. Soudry (2018). Train longer, generalize better: closing the generalization gap in large batch training of neural networks.

Huang, Z., S. Bai, and J. Z. Kolter (2021). Implicit layers for implicit representations. In *Advances in Neural Information Processing Systems*, Volume 34, pp. 9639–9650.

Keskar, N. S., D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang (2016). On large-batch training for deep learning: Generalization gap and sharp minima. *CoRR abs/1609.04836*.

Lee, S., J. Park, and J. Lee (2023). Implicit Jacobian regularization weighted with impurity of probability output. In *Proceedings of the 40th International Conference on Machine Learning*, Volume 202 of *Proceedings of Machine Learning Research*, pp. 19141–19184.

Ling, Z., X. Xie, Q. Wang, Z. Zhang, and Z. Lin (2023). Global convergence of over-parameterized deep equilibrium models. In *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, Volume 206 of *Proceedings of Machine Learning Research*, pp. 767–787.

Wu Fung, S., H. Heaton, Q. Li, D. McKenzie, S. J. Osher, and W. Yin (2021). Fixed point networks: Implicit depth models with jacobian-free backprop. *CoRR abs/2103.12803*.

You, Y., J. Li, J. Hseu, X. Song, J. Demmel, and C. Hsieh (2019). Large batch optimization for deep learning: Training bert in 76 minutes. *CoRR abs/1904.00962*.