Investigating Depth Properties of Deep Equilibrium Models

Sarah Fujimori
Stanford University
sfuji@stanford.edu

Ryan Lian
Stanford University
ryanlian@stanford.edu

Aditi Talati
Stanford University
atalati@stanford.edu

1 Introduction

Implicit models are a new way of thinking about neural networks where the data, rather than the programmer, dictates depth of the network. Traditional techniques in deep learning define the output of a layer as an explicit function of the input and stack these layers for depth. In contrast, implicit layers specify constraints that the output should satisfy, rather than providing an exact sequence of computation to compute the output.

Deep Equilibrium Models (DEQs), introduced in a paper by Bai, Kolter, and Koltun [1], are a specific class of implicit models that have been shown to use much less memory than classic models, as well as match or improve performance. Theoretically, any deep network can be represented as just a single layer or "cell" in a DEQ model with the same number of parameters, and a single DEQ layer can model any number of stacked DEQ layers. Thus, DEQs have much better memory-efficiency than traditional neural networks. Although there are still issues with the slower run-time, DEQ models present a unique approach to sequence modeling that enables for better representation and optimization.

This model has often been experimented and compared on medium to large scale tasks involving Penn Treebank and Wiki-Text103 corpus where its memory advantages shine, but not on the smaller sequential tasks that are within the computational limits of this project. As a result, we explore the performance of DEQs on a smaller task: learning nested arithmetic operations. We then compare the performance of a fully-connected neural network, GRU neural network, repeated GRU neural network, and a DEQ to predict the sign of output of an arithmetic expression.

Our goal is to understand how the DEQ compares with the other classical models when comes to this smaller but still complex sequential task, and we will conduct experiments to investigate interesting properties with the DEQ and whether its adaptive depth helps with the complex sequential hierarchies in the task.

2 Related Work

Deep Equilibrium Models were originally proposed by Bai et. al. [1] as a new approach to modeling sequential data, and was tested on language modeling tasks. The authors found that DEQs were able to achieve similar performance and computation requirements as state-of-the-art models on the same task, but vastly reduced memory usage, achieving up to 88% memory reduction. Bai et. al. [2] also introduced a new class of implicit networks called multiscale deep equilibrium models (MDEQs) for large-scale vision tasks including classification on the ImageNet dataset. Further work has been done to improve and optimize DEQs. For example, a paper by Bai et. al. [3] proposed the use of Jacobian regularization to stabilize the learning of DEQs while adding minimal computational cost, and a paper by Geng et. al. [4] introduces a way to use a gradient estimate for implicit models, called a phantom gradient, which accelerates the backward pass by a factor of 1.7 and maintains similar or better performance compared to state-of-the-art models.

Past studies have used both recurrent and non-recurrent models for learning hierarchical structures to evaluate nested arithmetic expressions. Hupkes et. al. [5] investigates the use of TreeRNNs, an artificial neural network which processes data recursively given a syntactic structure, and RNNs to compute the result of nested arithmetic expressions. The authors found RNNs can learn to predict the outcomes of these expressions, although the performance drops when increasing the length of the expressions. Alternately, Grashoff [6] uses the Transformer model for the same task, which, unlike an RNN, uses the mechanism of self-attention to process the data all at once rather than sequentially and recurrently. This study finds that a transformer can evaluate arithmetic expressions, and that the number of layers needs to increase as the depth of the nested expression increases. Lample and Charton [7] use a similar approach for symbolic expressions, even learning integrals and differential equations better than standard mathematical engines such as Matlab and Mathematica. Tran et. al. [8] compares the architectures of RNNs and Transformers to learn hierarchical structures and find that although both perform similarly, the RNN obtained slightly better accuracies and generalized better than the Transformer. They conclude that processing data sequentially and recurrently is essential for modeling hierarchical structure.

3 Dataset and Features

Our dataset included 50000 training, 5000 testing, and 5000 validation examples of nested arithmetic expressions of length 31. Each symbol in these expressions is either one of the digits $1, \ldots, 9$ (where 0 is excluded to prevent invalid expressions dividing by 0), an open or close parenthesis, or one of the arithmetic operations $+$, $-$, $*$, and $/$. The task is a binary classification task to predict whether the result of evaluating the arithmetic expression is positive or negative. The motivation behind choosing this task is that nested arithmetic expressions have hierarchical structure, so we can test whether DEQs can learn this structure and generalize well with its "infinite depth".

We generated the data by randomly forming combinations of digits, operators, and parentheses, eliminating any invalid expressions, and computing the sign of the result of each expression using the numerical expression evaluator `numexpr`. Some examples of the generated data are shown in Figure 1. We note that the data is approximately 66% positive examples. We then used a one-hot embedding for each symbol. For different experiments, we also generated longer expressions and biased expressions, where the $-$ symbol is more likely to appear than the other operators.

Expression	Sign
$(6 + (7 + 4 - 9)/6 * 1 + 4 - 9/6/3 * 3 + 6/4 * 3)$	$+$
$(((((5 - 6 + (1 + 1/5))))))/((((3 * 9))))$	$+$
$(7 - 1 * 8 * 9 * 2 * 1 - 5 + 5 - 7 - (((2 - 7 - 7))))$	$-$
$(7 + 3 - 7/7) - (((9 - 4/9 * 8))) * ((8 * 4))$	$-$
$((7 * (6 - 6 * ((3 + 3 * 5 + 9)))))) + 1 + 9/7/3$	$-$

Figure 1: Examples of generated expressions.

We also ran PCA for dimension reduction to 2 and 3 dimensions. The results are shown in Figures 2 and 3 respectively, where the different colors represent the labels of the examples. We can observe that there is a lot of noise and no separation between positive and negative examples, which shows that the data is quite complex and cannot be predicted by simple GLMs.

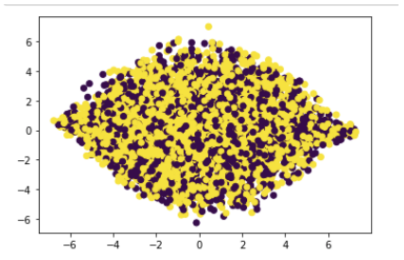

Figure 2: PCA reduction to 2 dimensions

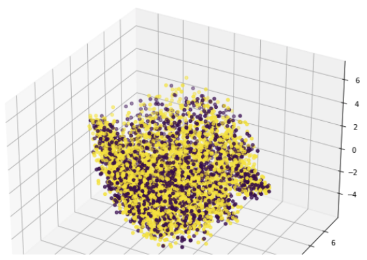

Figure 3: PCA reduction to 3 dimensions

4 Methods

4.1 Evaluation

Since this is a binary classification problem, the task is to minimize the standard **Binary Cross-Entropy loss function**

$$L_{BCE}(\hat{y}) = \frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i),$$

where y is the vector of ground-truth labels, $\hat{y}$ is the vector of predictions, and N is the number of examples.

We applied the binary cross-entropy loss function with logits, which meant that our loss function applied the sigmoid layer to our final output before computing the loss; this helped avoid a gradient explosion that would otherwise occur when the log of our values got too small. However, this meant that the output of our model was a raw output, that happened before applying a sigmoid layer; thus, we manually applied a sigmoid to our predictions when computing accuracies on our test set.

To evaluate the models, we will use three metrics: **accuracy**, **precision**, and **recall**. To calculate this, we will calculate the true positives (TP), false positives (FP), true negatives (TN), and false negatives(FN) in a confusion matrix. Then, the metrics are defined by the equations

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP},$$

$$precision = \frac{TP}{TP + FP},$$

$$recall = \frac{TP}{TP + FN}.$$

4.2 Baseline (Fully Connected Network)

As seen from the PCA, the data is too non-linear and complex for traditional classifiers such as logistic regression and SVMs. As a result, the first model we will be using is a standard fully-connected neural network. This network is composed with first a flattening layer (flattening each example from sequence of vectors into one long vector), then 3 linear layers with a tanh activation in-between, and apply tanh again to the output layer. Since we suspect this model will under-fit the data, we did not add any regularization.

4.3 Gated Recurrent Unit (GRU) Neural Network

The GRU is a specific recurrent neural network (RNN) architecture designed to address the problematic short term memory of RNNs. According to the comparison in [9], it is similar to the LSTM network and comparable in performance, except it has a less complex structure since it has one less gate, and is thus more computationally efficient. To understand the structure of this network, we will briefly recall the mechanism of RNNs.

Let $x = (x_1, \dots, x_T)$ be a sequence of data (e.g. words in a sentence). Then a simple RNN updates its hidden state as follows:

$$h_t = g(Wh_{t-1} + Ux_t),$$

where g is usually some smooth function, W and U are weight matrices, h_{t-1} is the previous hidden state, and x_t is the current vector in the sequence. In the example where x represents a sentence, each hidden state would then represent the information from all of the previous words processed, and the next hidden state would be updated after seeing the next word. This continues until the end of the sequence.

The GRU is fundamentally the same, but instead of a simple function, it passes the hidden state and data through a specially designed gated unit that works to preserve long-term memory and has better performance. A diagram of the cell is shown in Figure 4.

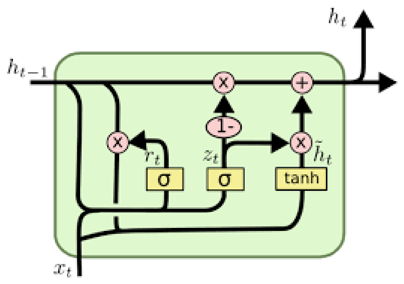

Figure 4: A single Gated Recurrent Unit cell.

With each example being a sequence, the neural network applies the GRU to each example, turning it into another sequence with the length of its hidden layer. Then, it flattens each example into regular vectors and apply a final linear layer as the prediction. A diagram of the model architecture is shown in Figure 5.

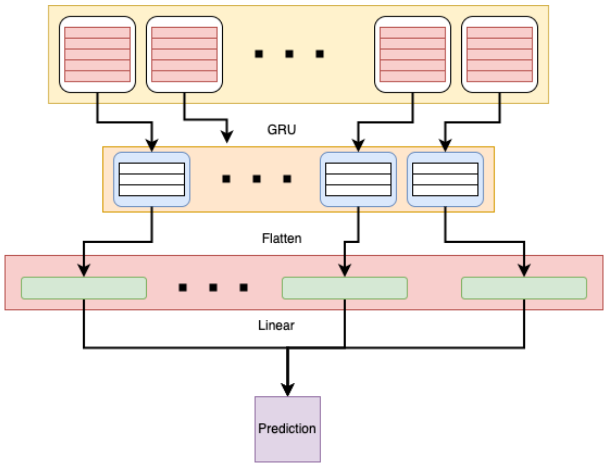

Figure 5: Model Architecture

4.4 Stacked GRU Neural Network

Since the differential equation solver, in essence, repeats our function call multiple times, we also wanted to see how the performance of the DEQ model compared to multiple layers of the GRU neural network. Thus, we included a stacked GRU neural network, where we called a 3-layer GRU. This passed the output of our first GRU into two more GRU cells before flattening the final output and adding a linear layer to get our final one-dimensional prediction.

4.5 DEQ with GRU Cell

The design of the DEQ model is inspired by the observation that the layers of a deep weight-tied sequential model converge to some fixed point [1]. Formally, for a deep weight-tied model, the forward operation can be written as

$$z_{1:T}^{[i+1]} = f(z_{1:T}^{[i]}, x_{1:T}),$$

where i represents the index of layers, $z_{1:T}^{[i]}$ is the hidden sequence of length T at layer i, $x_{1:T}$ is the input vector, and f is some function.

The DEQ then uses a root solver to solve for the fixed point $z^* = f(z^\star, x)$, which corresponds to the output of an "infinite-layer" network. Through implicit differentiation, DEQ can back-propagate through "infinite-layers" using constant memory.

To design a DEQ layer, we just need to find some f that mimics the layer structure we want. For our task, we wanted to mimic the structure of the GRU network, so we designed the DEQ layer

$$f(z^{[i]}, x) = \tanh(GRU(z^{[i]}) + x),$$

where $z^{[i]}$ is the output at i-th layer, x is the initial input, and $GRU(\cdot)$ is the GRU cell applied to every sequence in $z^{[i]}$. Intuitively, the DEQ should then be equivalent to stacking the GRU layers infinitely many times (with input injection and $\tanh$ for stability). This should handle more complex sequential structures than single layer GRU. We also initialized z to be $GRU(x)$ where x is the input sequence so that the GRU sequence lengths line up. The full model then does the following:

1. Initialize $z_0 = GRU(x)$
2. Pass z_0 to DEQ layer, which computes the "fixed point" with Anderson Acceleration
3. Flatten and pass to linear layer for output.

For the math to work, the DEQ depends on the existence of an stable fixed point, so f is usually a contraction. In this case, however, we decided it would be interesting to see if the DEQ would perform without there being a stable fixed point.

4.6 Repeated GRU Neural Network

We also wanted to understand what the DEQ function was doing when there were no fixed points to find, which we found was the case in our experiments. A simplified model of a differential equation solver (not the one we used, since our DEQ model used Anderson acceleration) repeatedly applies the function f until $||f(z) - z||$ goes below a certain value. Since our function does not reach a fixed point, this meant it was being called until the Anderson acceleration solver "gave up" which happened at a threshold of 50 iterations. So we decided that a good model to compare our DEQ model to would be one where we initially apply a GRU cell to our input x to get $z^{[0]}$, and then applying the function

$$f(z^{[i]}, x) = \tanh(GRU(z^{[i]}) + x)$$

50 times; that is, we passed our input through the *same* GRU cell 50 times. Afterwards, we flattened the output and applied

a linear layer to get our final one-dimensional prediction, as in the previous examples.

4.7 Anderson Acceleration

Implicit models rely heavily on methods to solve the fixed point problem: given a function f, find a solution to $f(x) = x$. Like Bai et. al. [1], we use Anderson acceleration as introduced by Walker and Ni [10], which, given an initial guess x_0 and a positive integer m, computes a sequence of iterates to find a fixed point, setting $x_1 = f(x_0)$ and

$$x_{k+1} = \sum_{i=0}^{m_k} (\alpha_k)_i f(x_{k-m_k+i})$$

where $m_k = \min\{m, k\}$, and α_k can be computed quickly from the residuals $g_k = f(x_k) - x_k$ of previous iterates. Unlike Newton's method, which requires a potentially costly exact computation of the derivative f' during each iteration, Anderson's method only requires one computation of f per iteration, and no evaluation of its derivative. The implementation we use sets $m = 6$, and stops at 50 iterates or when $\|x_{k+1} - x_k\| < 10^{-3}$.

5 Results and Discussion

Our implementation included code from the papers [1], [2], and [3]. For our hyper-parameters, we used batch size 64 and ran for 40 epochs. This seemed reasonable given the amount of training data, and the models' accuracies began to plateau around epoch 40. We did a grid search for the learning rate from $[0.1, 0.01, 0.001]$ and the hidden size from $[50, 100, 200]$, and the combination of learning rate being 0.1 and hidden size of 50 yielded the highest accuracies.

To prevent exploding gradients, which is often an issue in deep learning, we also use the method of gradient clipping, which scales the parameters of the model so that they are less than some constant c (we take $c = 1$ in this case). This method simply maps a parameter g to $\frac{g}{\|g\|} \cdot c$ if $\|g\| > c$.

After every epoch of training each model, we tested its accuracy on the validation data set, and we found that the accuracy per epoch was what we expected - that the DEQ does better than the one-layer and three-layer GRU at each iteration, because of this extended power of the implicit layer.

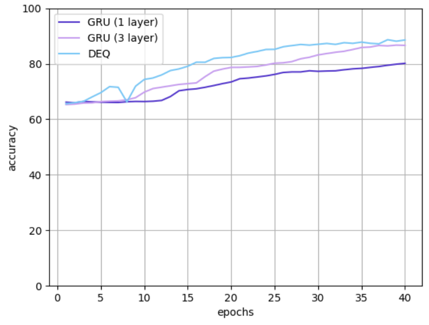

Figure 6: Accuracy of GRU, stacked GRU, and DEQ over 40 epochs

Y However, none of these models seem to be able to reach perfect accuracy - if we train them for long enough, all three models reach around 90% accuracy. We can see that, moreover, the baseline neural net is unable to get more than 66% accuracy - because it doesn't have enough computing power to actually parse and understand the equations, it ends up learning that guessing that every equation has a positive output gives it 66% accuracy, and it isn't able to do much better than that.

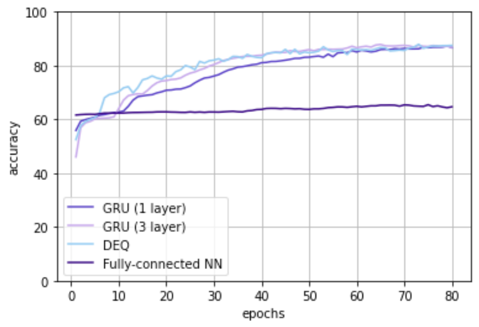

Figure 7: Accuracy of GRU, stacked GRU, DEQ, and FC NN over 80 epochs.

We also evaluated the performance of the single-layer GRU and the DEQ on simplified sequences with only the operations $+$, and $-$. The test results are shown in Figure 8, with the confusion matrices shown in Figure 5. We note that around half (43/80 and 47/89 for the base model and DEQ, respectively) of the false positives evaluated to 0, which may have been caused by the task being binary classification rather than regression (so it is likely harder for the model to classify examples with results that are close to 0).

We can see that the performance between the two is comparable, with the DEQ performing slightly better. We also test the performance of these models on biased data where the $-$ operator is three times more likely to appear than the $+$ operator, and find that the DEQ generalized slightly better than the base model.

Model	Test data accuracy	Biased data accuracy
One-layer GRU	96.0%	94.2%
DEQ	96.3%	94.7%

Figure 8: Results of testing models on test and biased data with two operations.

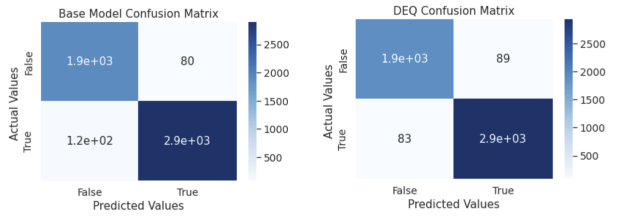

Figure 9: Confusion matrices for two-operation data shows comparable performance between the base model (GRU) and the DEQ.

Model	Accuracy	Precision	Recall
FC NN	64.58 %	77.73 %	58.57 %
One-layer GRU	80.73 %	91.05 %	75.78 %
Stacked GRU	85.0 %	92.5 %	81.97 %
DEQ	87.62 %	92.72%	86.43 %
Repeated GRU	87.41 %	91.86 %	87.03 %

Figure 10: Models Metrics on Test Data

Model	Accuracy	Precision	Recall
FC NN	62.13 %	70.26 %	16.64 %
One-layer GRU	75.00 %	83.67 %	56.63 %
Stacked GRU	82.50 %	85.02 %	72.49 %
DEQ	85.9 %	88.38 %	71.27 %
Repeated GRU	84.9 %	84.10 %	74.36 %

Figure 11: Results of testing models on test and biased data.

We then trained and ran tests on the four operation data, with the results in Table 10. The interesting observation is that the precision are similar among models, but recall has a more significant difference. This means that there are a lot of false negatives, and might be related to the two operation trend earlier. Not surprisingly, the DEQ again out performs the rest of the models (with its mimic repeated GRU closely behind).

In general, it seems like the DEQ model is able to generalize better to new datasets. When testing on a biased dataset, where the likelihood of $-$ signs is more than in the original test and training dataset (which in turn would increase the likelihood of a negative output), we see that the DEQ model trained for 40 epochs on the unbiased dataset is able to get up to 85.9% accuracy on the biased dataset, while the one-layer GRU only gets 75.9% accuracy and the three-layer GRU gets 82.5% accuracy. These results are shown in Figure 11.

What's interesting is that if we check the output z of our implicit layer in the DEQ, and then compute $||z - f(z)||$, we can check whether our implicit layer is actually reaching a fixed point. Plotting this norm after every 100 batches of training the DEQ gives us the graph in Figure 12.

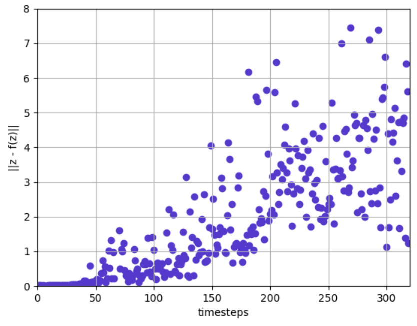

Figure 12: Graph of $\|z - f(z)\|$ over time

This makes it clear that we are not, in fact, reaching a fixed point. It is interesting that the DEQ still performs well when this is the case, since this means that when applying this new type of model, we are not necessarily limited to contractive maps that have some fixed point.

The question then becomes whether the additional power of our DEQ comes solely from the repetition of our function f within each iteration of the model. Table 10 seems to indicate that this is the case, because we can see that the repeated GRU is able to get similar results to the DEQ on our testing data.

6 Conclusion and Future Work

In conclusion, we saw that RNNs, specifically GRU networks, can learn hierarchical structures in nested arithmetic expressions quite well, especially when limiting these expressions to addition and subtraction instead of all four operators. We found that although the DEQ takes much more time to train, it consistently outperforms and seems to generalize better than a fully connected neural network, a single-layer GRU, a three-layer GRU, and a repeated GRU despite not finding a fixed point.

For future work, it is crucial to understand why the DEQ still performs well when there is no fixed point—this will open up exciting modeling potentials while opening up the black box. We could try implementing further improvements to the DEQ model to optimize time and memory usage, such as calculating the phantom gradient in [4] as a gradient estimate instead of the gradient estimate, and also apply Jacobian regularization as in [3]. The slow run time is definitely an issue for wide implementations.

Another interesting direction to explore would be to adapt the method of diagnostic classifiers used by Hupkes et. al. [5] to DEQs. The idea is that if a neural network is keeping track of or computing certain information at a given time, a simple linear model should be able to easily predict this information given the hidden state space of the model. The result can then give us a better idea of what the model is conceptually doing on the algorithmic level.

7 Acknowledgments

We are grateful to have been assisted by the mentorship of Michael Poli.

8 Contributions

We contributed the the same amount.

References

[1] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. Deep equilibrium models, 2019.

[2] Shaojie Bai, Vladlen Koltun, and J. Zico Kolter. Multiscale deep equilibrium models, 2020.

[3] Shaojie Bai, Vladlen Koltun, and J Zico Kolter. Stabilizing equilibrium models by jacobian regularization. *arXiv preprint arXiv:2106.14342*, 2021.

[4] Zhengyang Geng, Xin-Yu Zhang, Shaojie Bai, Yisen Wang, and Zhouchen Lin. On training implicit models. *Advances in Neural Information Processing Systems*, 34, 2021.

[5] Dieuwke Hupkes, Sara Veldhoen, and Willem Zuidema. Visualisation and'diagnostic classifiers' reveal how recurrent and recursive neural networks process hierarchical structure. *Journal of Artificial Intelligence Research*, 61:907–926, 2018.

[6] Daan Grashoff. On how transformers learn to understand and evaluate nested arithmetic expressions. 2022.

[7] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

[8] Ke Tran, Arianna Bisazza, and Christof Monz. The importance of being recurrent for modeling hierarchical structure. *arXiv preprint arXiv:1803.03585*, 2018.

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[10] Homer F Walker and Peng Ni. Anderson acceleration for fixed-point iterations. *SIAM Journal on Numerical Analysis*, 49(4):1715–1735, 2011.