

FINAL REPORT

Kevin Jones

Purdue ID: 034244978

jone2342@purdue.edu

1 INTRODUCTION

The grokking effect was first noted by Power et al. (2022) when they observed that allowing a model to train far beyond the point where it has reached 100% training accuracy would result in the model suddenly generalizing perfectly to the test dataset. Later work by Nanda et al. (2023) applies a mechanistic interpretability lens to the grokking effect in an effort to understand exactly what general rules the model learns and mathematically how the neural network learns to solve the modulo arithmetic problem. This is all summarized by Labs (2025) which served as a foundation for our knowledge on this problem.

Our goals for this project were to replicate the grokking effect using the setup described by Nanda et al. (2023), to successfully grok a model using an optimization method learned in class (LBFGS), to engineer a way to grok a model using a large modulo number, and to investigate what kind of regularization is required to grok a model.

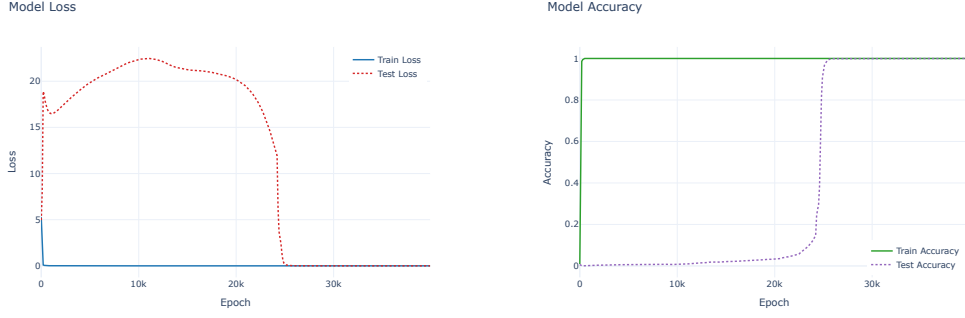
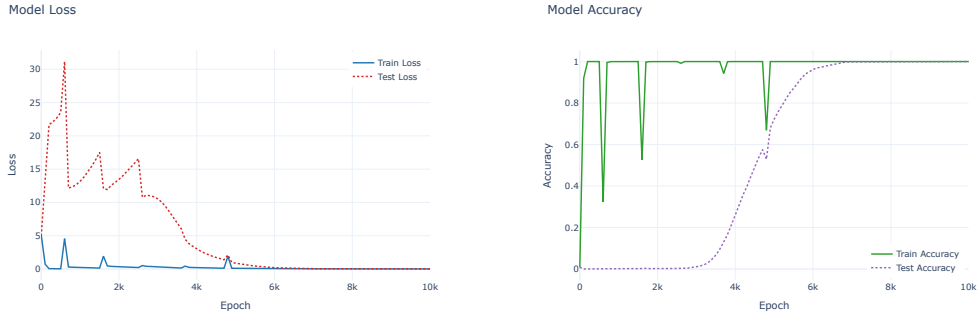
2 GROKING EXPERIMENTS

2.1 DATASET

We began by writing the code to generate the modulo arithmetic dataset. The dataset consists of $\{(x_i, y_i)\}_i^{N^2}$ pairs which will be used to train the model. Each x_i represents the input to the model and each y_i represents the ground truth label for the modulo arithmetic. For a given modulo number N , the modulo arithmetic problem can be described as

$$(a + b) \mod N = c \quad a, b, c < N$$

There are N^2 possible pairs of numbers that can act as inputs, each ranging from 0 to $N - 1$. Now we get to a design decision: how do we represent our inputs? The original grokking paper by OpenAI represents the input as $[a, \circ, b, =]$ Power et al. (2022). In the video, the author states that the input is represented as $[a, b, =]$ Labs (2025). We couldn't find anywhere in the Nanda et al. (2023) paper that mentioned their input representation. We first tried representing our input as $[a, b, =]$. When running the grokking experiment, it took $\sim 30,000$ epochs for the model to grok (Figure 1). Logically, we figured representing the input as $[a, b]$ should also work. A sequence prediction model will try to predict the next tokens for each position in the input, so the output $[x, y]$ will be a prediction of what follows the sequence $[a]$ (which is the prediction x) and a prediction of what follows $[a, b]$ (which is the prediction y). We don't need an '=' token to get a sequence prediction involving a and b . We therefore tried representing our input as just $[a, b]$. When

Figure 1: Grokking plots with input representation as $[a, b, =]$ Figure 2: Grokking plots with input representation as $[a, b]$

running the experiment with this input representation, we found that the model grokked much faster, after just $\sim 8,000$ epochs (Figure 2). So interestingly, the more simplistic input representation allows the model to grok faster. Therefore, we keep our input representation as $[a, b]$. The output is a single scalar value c . We wrap our dataset in a `torch.Dataset` object so that we can later use it with a `torch.DataLoader` object.

For training, we use 113 for our modulo number and use 30% of the dataset as the training dataset and the remaining 70% as the test dataset. This replicates the decisions in Nanda et al. (2023).

2.2 MODEL ARCHITECTURE

Next, we wrote the code to implement a single layer Transformer architecture and tried to match the architecture used in Nanda et al. (2023) as closely as possible.

The architecture begins with an embedding layer of `(modulo_number, d_embed)` so each possible input number gets an embedding of size `d_embed`. There is another embedding layer of `(2, d_embed)`. This is for learned positional embeddings. Because our input representation is 2-dimensional, we learn two embeddings. As an aside, because addition is commutative, it seems to us that this wouldn't be necessary; however, the Nanda et al. (2023) paper includes this so we do too. The next layer is a `MultiHeadAttention` layer which uses `d_embed`

as the feature size for the Q,K,V vectors and has 4 attention heads. The second part of the Transformer layer is the MLP, which we define as $\text{Linear}(\text{d_embed}, \text{d_ff}) \rightarrow \text{ReLU} \rightarrow \text{Linear}(\text{d_ff}, \text{d_embed})$. The final layer is the unembedding layer which is a linear layer of $(\text{d_embed}, \text{modulo_number})$. This gives our output logits.

As described in Nanda et al. (2023) we use $\text{d_embed}=128$ and $\text{d_ff}=512$.

2.3 GRADIENT DESCENT GROKING

Our training procedure also follows Nanda et al. (2023) as closely as possible. We train using full-batch gradient descent with a learning rate of 0.001. We use the AdamW optimizer with weight decay $\lambda = 1$ and train for 40,000 epochs. We use cross-entropy for the loss function.

Figure 1 shows the accuracy and loss plots that demonstrate our model grokking with the input representation $[a, b]$. Figure 2 shows the accuracy and loss plots that demonstrate our model grokking with the input representation $[a, b, =]$. As seen in the plots, when changing the input representation the model groks much faster.

2.4 LBFGS CASE STUDY

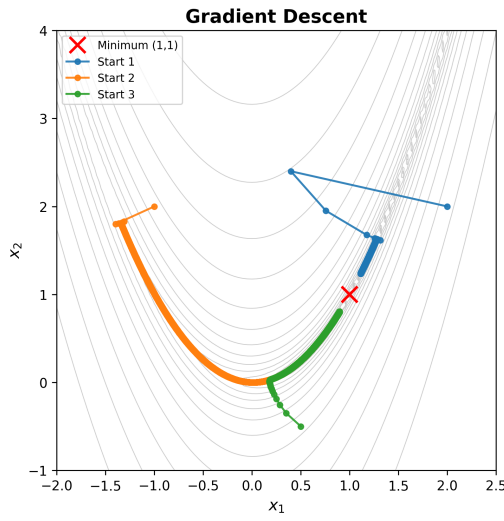
One of the overarching goals of our project was to achieve grokking using second-order optimization. In order to do this, we first learned more about exactly how the `torch.LBFGS` optimizer actually works. The core functionality of the optimizer is similar to the BFGS algorithm discussed in class with a couple of extra details. Firstly, the ‘L’ stands for limited-memory and is what allows us to perform second-order optimization for large, memory intensive models such as our transformer which has 227,057 parameters. LBFGS avoids storing the entire matrix T by using a limited history of position and gradient updates (s_k, y_k) from only the most recent iterations. This reduces the memory requirement from $\mathcal{O}(n^2)$ to $\mathcal{O}(mn)$ where m is the history size. Anytime the search direction p is needed, the LBFGS algorithm computes an approximation of $p = -Tg$ without constructing the full T matrix. It does this via a two loop recursive method. The algorithm looks like

```

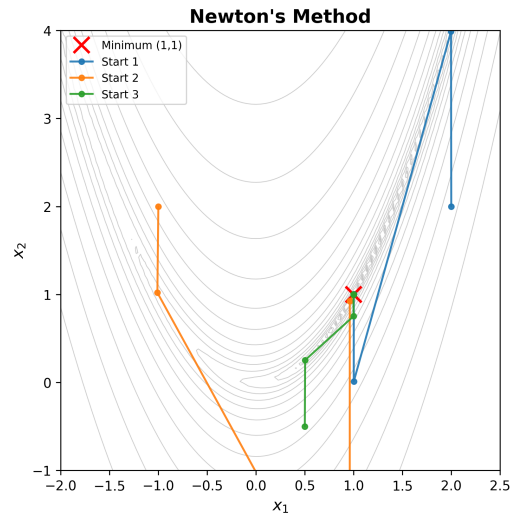
q = g_k
for i = k - 1 → k - m :
    α_i = ρ_i s_i^T q
    q = q - α_i y_i
r = H_k^0 q
for i = k - m → k - 1 :
    β = ρ_i y_i^T r
    r = r + s_i(α_i - β)

```

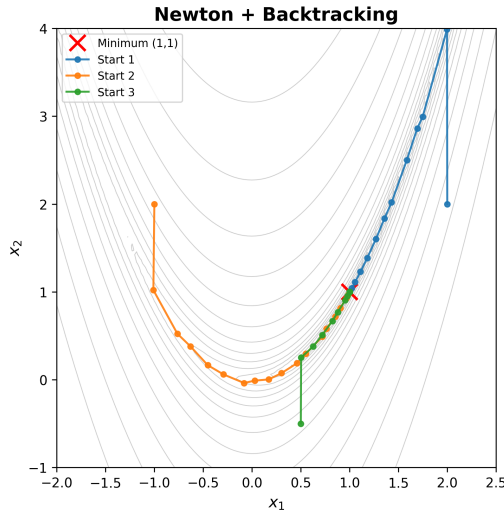
Above, ρ is defined the same way it was in class. In the middle step an initial Hessian approximation is formed. It is defined as $\frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} I$. Because I is diagonal, we only need to store n terms and never have to form a full $n \times n$ matrix. The



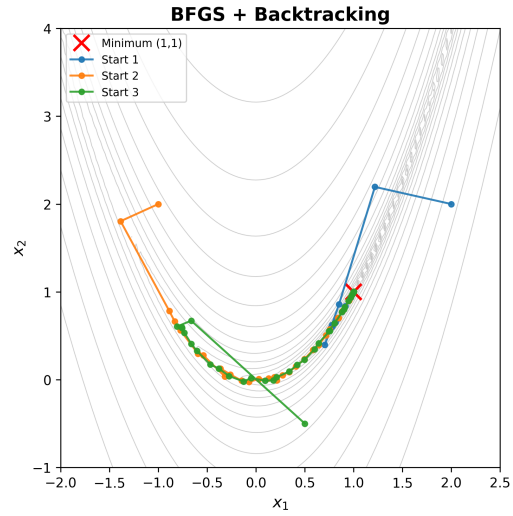
(a) Gradient Descent



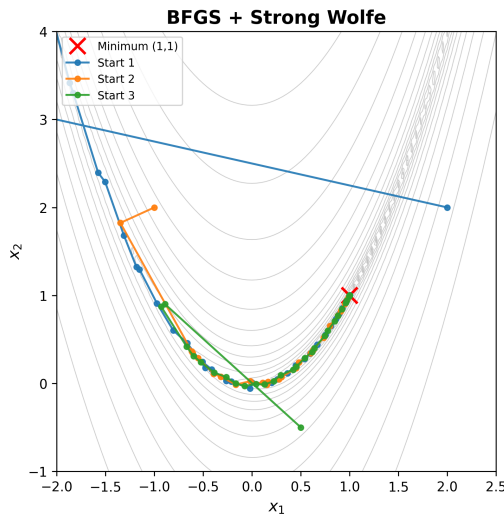
(b) Newton's Method



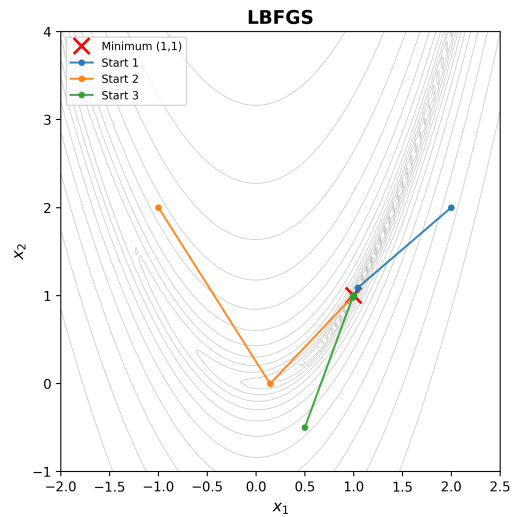
(c) Newton's Method with Backtracking LS



(d) BFGS



(e) BFGS with Torch's Line Search



(f) LBFGS

Figure 3: Comparison of optimization techniques

algorithm essentially builds H_k via the BFGS update rule from the last m steps but never actually stores the matrix Wang (2022). The other thing `torch.LBFGS` introduces is a line search algorithm to find a step length satisfying the strong Wolfe conditions. The line search uses a bracket and zoom strategy. The algorithm first finds (brackets) an interval containing a point satisfying the strong Wolfe criteria. Then the algorithm refines the bracket until an exact point satisfying the criteria is found. They use cubic interpolation during the zoom phase to find better step lengths.

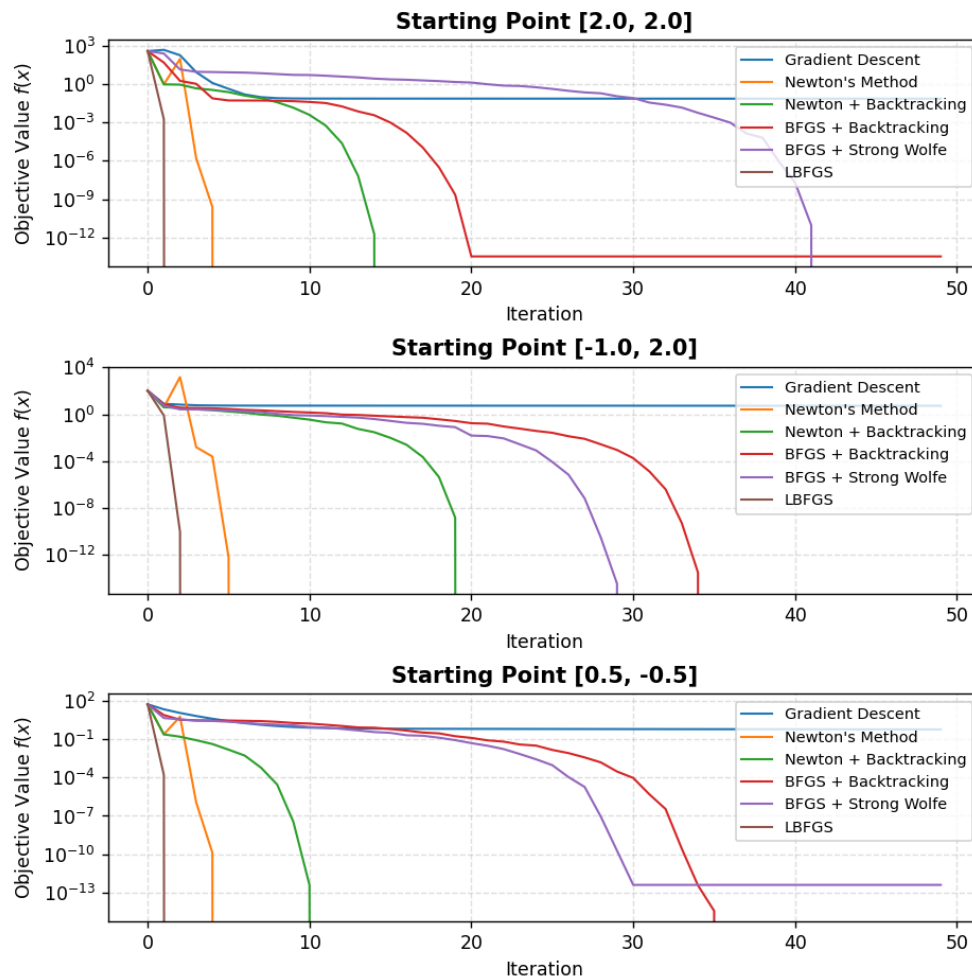


Figure 4: Convergence Rate Comparison

To ensure the robustness of the `torch.LBFGS` optimizer we compared it to a plethora of other optimization techniques learned in class on the Rosenbrock objective function starting at three different starting points: $[2.0, 2.0]$, $[-1.0, 2.0]$, and $[0.5, -0.5]$. The optimization techniques we used were:

- Standard Gradient Descent: implemented manually in torch without a `torch.optim` optimizer.

- Newton’s Method: implemented manually in torch without a `torch.optim` optimizer. Used the torch hessian function to get the second derivatives and used a fixed step length.
- Newton’s Method with backtracking line search: same as above but using a backtracking line search to find a step length satisfying the first Wolfe condition.
- BFGS with backtracking line search: a custom BFGS implementation using the theory in class without a `torch.optim` optimizer.
- BFGS with torch’s line search: same as above but instead of using backtracking line search we use the same line search that `torch.optim.LBFGS` uses.
- Torch’s LBFGS: we use the `torch.optim.LBFGS` optimizer

All second-order optimization techniques performed very well and far outperformed gradient descent. Figure 3 shows plots comparing the routes taken by each algorithm. The gray contour lines in the background show the Rosenbrock function. Each line color (blue, orange, green) correspond to the starting points $[2.0, 2.0]$, $[-1.0, 2.0]$, and $[0.5, -0.5]$ respectively. As seen in the figure, LBFGS quickly finds the optimal solution for all solutions. We also compare the convergence speeds for each algorithm by plotting the current value of the function at each iteration on a logarithmic scale. Figure 4 shows these convergence plots. All second order methods converge super-linearly, and LBFGS converges the fastest for all starting points. The LBFGS optimizer quickly converges to the correct solution for different starting points. Figures 3 and 4 verify the torch implementation of the LBFGS optimizer and allow us to confidently use the `torch.optim.LBFGS` optimizer to grok our model.

2.5 LBFGS GROKING

Much of the code when switching optimizers stayed the same. The `torch.optim.LBFGS` optimizer requires a closure function which it can call to get a function evaluation which in our case is the loss. For our function evaluation, we use the loss over the entire training dataset. The biggest change we had to make was adding regularization to the loss. When using the `torch.AdamW` optimizer, there is a built in option for weight decay which doesn’t directly add an $L2$ loss to the total loss function, it instead applies weight decay directly to the parameter. Since we are using LBFGS, we have to now manually add the weight decay. We do this as

$$\text{weight_decay} = \sum_p^{\text{params}} p^2$$

The total loss then becomes

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{CE}} + \frac{\gamma}{2} \text{weight_decay}$$

Initially, we set $\gamma = 1$ just as we had done for our gradient descent optimization. When we ran the optimization, we found that the model didn’t grok and the

optimization got stuck very quickly. After significant experimentation with hyper-parameters, we ended up stumbling upon $\gamma = 0.001$. With this value for γ the model successfully grokked. We suspect that with a high γ value, the model first tried to drive down parameter magnitudes, and then solve the objective and never generalized. When we lowered the γ value, we suspect the model first tried to solve the objective, and then tried to drive down parameter magnitudes which yielded the grokking effect. Figure 5 shows the model grokking when optimized by LBFGS.

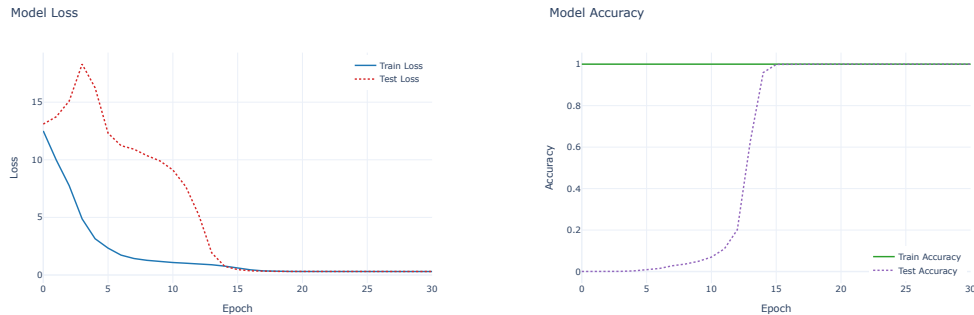


Figure 5: Grokking using LBFGS

As seen when comparing Figure 5 to Figure 2, the model optimized with LBFGS groks extremely quickly, requiring only approximately 20 epochs. To compare the optimization efficiency, we recorded how long it took and how many function calls were required before the test accuracy hit 100%. When optimizing with gradient descent (AdamW), the model took 256 seconds and 5520 function calls to grok. When optimizing with LBFGS, the model took 23 seconds and 383 function calls to grok. To our pleasant surprise, optimizing the model using LBFGS demonstrates a clear advantage over AdamW both in the amount of time taken and the total number of function calls.

2.6 LOSS SPIKES

One interesting observation is the presence of loss spikes during training. We only observe this phenomenon when using gradient descent. Throughout our project we ran many grokking experiments and observed these loss spikes on most runs when using gradient descent regardless of input representation (Figure 1 happens to be a case where we didn't observe loss spikes). Loss spikes like these are not unheard of when training models, especially with a fixed learning rate. These spikes occur when a gradient descent update pushes the model parameters into an unstable “sharp” region where the model performs very poorly. Usually, the model recovers in the next couple of training steps and training restabilizes. We never experience any loss spikes when training using LBFGS. This is because LBFGS uses a strong Wolfe line search in a direction based on second-order curvature information. This keeps model parameters in a stable region of the loss landscape.

3 FURTHER DIMENSIONS OF GROKING

In addition to grokking our model using a method learned in class, LBFGS, we explored two other facets of the problem. We investigated large input spaces by increasing the modulo number n . We also studied the effects of different types of regularization, namely L1, L2, L3, and L4 regularization.

3.1 LARGE INPUT SPACES

There are several challenges when increasing the modulo number n . Firstly, the size of the dataset grows quadratically. This increases the training time because each epoch consists of much more data. Additionally, there is more the model has to learn so training can take longer and may require a larger model. We decided to continue to use LBFGS as our optimizer because our results from last week indicated far quicker training. This also adds a challenge. The LBFGS optimizer requires a `closure()` function which it can call to evaluate the objective. For small input spaces it is easy to evaluate the full objective function because the training dataset is small. However, for larger datasets this can take a long time. Initially, we investigated using a stochastic approach where at each function evaluation (or call to `closure()`), we sample without replacement a batch of data and use that to evaluate the function. We tested this initially with the same modulo number of 113. The size of the training dataset is 3830. We tried using a batch size of 256 within `closure` but with such a small batch size, the model was unable to even learn the training dataset. We next tried a batch size of 1000. Figure 6 shows the re-

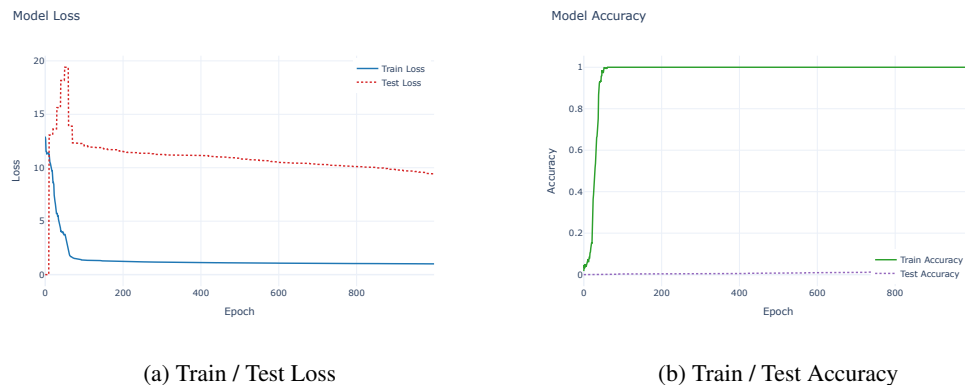


Figure 6: Results with Batch Size of 1000

sults. As can be seen from the loss and accuracy curves, the model is able to learn the training dataset but never groks. After these failed experiments, we realized that the closure function will need to calculate the objective value over the entire dataset. This means that training will slow quadratically as the modulo number increases. Additionally, for large datasets not all samples will fit in the GPU memory. To fix this we use the standard batching approach. We simply average the loss over all batches to get the value of the objective function over the entire dataset. And we still get the gradient over the entire dataset when we call `loss.backward()`. We then started training our model on larger modulo numbers. We kept all model

configuration parameters the same and used $n = 213$. Figure 7 shows the resulting

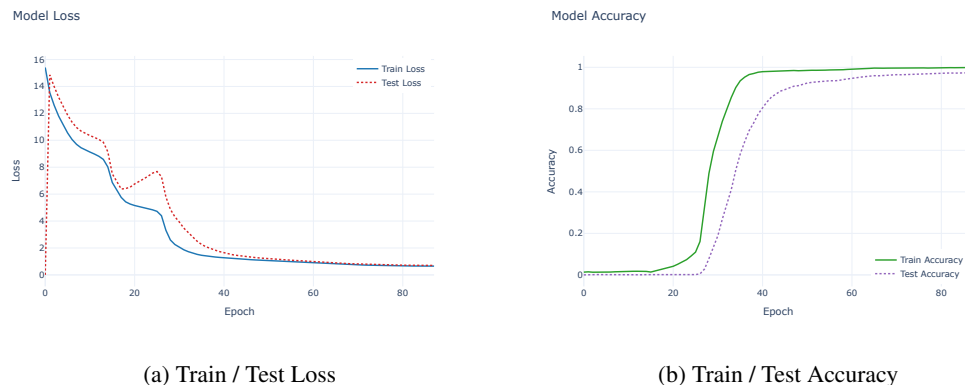


Figure 7: Results with $n = 213$

loss and accuracy curves. Interestingly, this model seemed to grok almost immediately. While we aren't sure exactly why, we think it may be because increasing the modulo number increases the number of parameters (due to the initial embedding layer) from 227,057 to 252,757. This means that the L_2 regularization penalty is slightly higher because there are more parameters. If the penalty is too high, the model will not learn the objective and just drive down the parameter norms. However, it may be that this small increase in the penalty was enough for the model to learn the objective and then quickly reduce the norm of the parameters to allow generalization. It could also just be due to the stochastic nature of training these models.

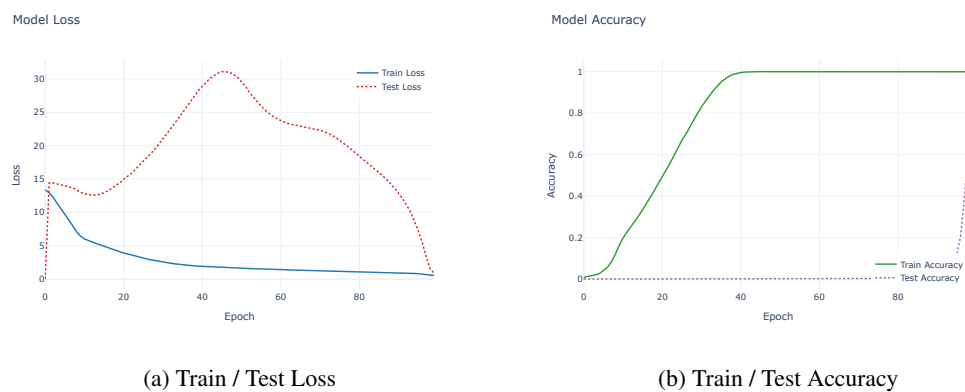
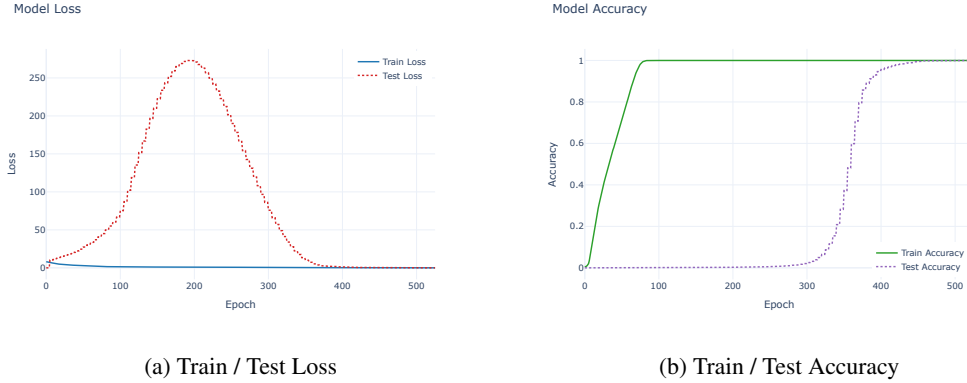


Figure 8: Results with $n = 419$

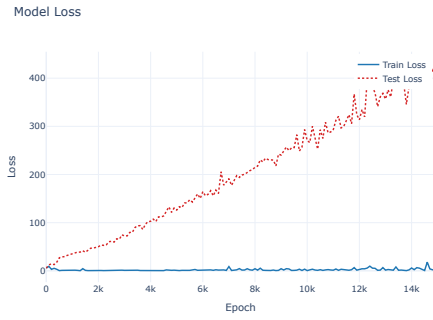
We next tried $n = 419$. Figure 8 shows the resulting loss and accuracy. Unfortunately, we only trained for 100 epochs and so the curves are cut off but one can clearly see that the model did grok. With the original hyperparameters, the model struggled to learn the dataset due to its increased size. To account for this we increased the `dembbed` parameter of the model from 128 to 256. Consequently, we reduced γ (the L_2 regularization parameter) from 0.001 to 0.00015. The next experiment was $n = 821$. Figure 9 shows the resulting loss and accuracy curves. To

Figure 9: Results with $n = 821$

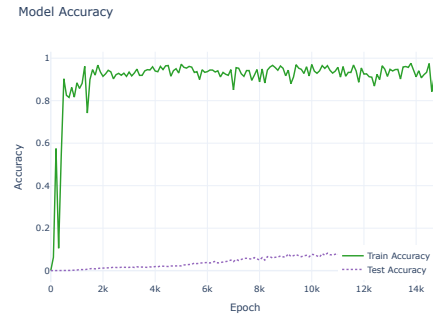
improve efficiency, we only calculate test loss and accuracy every 5 epochs which explains the step-like structure in the plots. We again increased number of parameters in the model because it is a more complex task. We used `dembd = 384` and `dff = 768`. `dff` is the size of the hidden layer in the final MLP. We adjusted γ to $1e^{-5}$. Additionally, to help with memory requirements, we used history size of 30 in the LBFGS optimizer as opposed to the default history size which is 100. We were happy to see that with these hyperparameters the model successfully grokked with a large modulo number. We repeated the experiment several times and each time the model successfully grokked. We then attempted to use $n = 1117$; however, we were unable to get the grokking effect with this modulo number due to our computing constraints. In order to create a model large enough to effectively learn, we had to increase `dembd` to 512 and `dff` to 1024. With this large model size, each epoch took a very long time, on the order of an hour, and so we were unable to grok for $n = 1117$. Nevertheless, we demonstrate a clear pattern that if the correct balance between the number of parameters and γ is found, any arbitrary modular arithmetic problem can be learned and grokked given enough compute.

3.2 REGULARIZATION STUDY

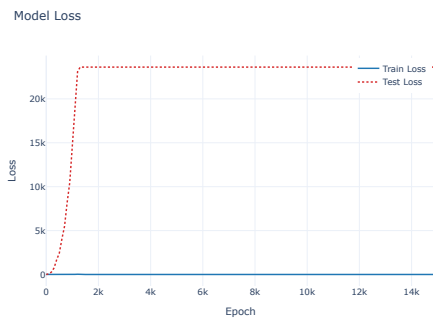
We studied the effects of implicit vs explicit regularization and different types of explicit regularization and observed their effects on the grokking behavior. Our goal was to gain a further understanding into what is required for a model to grok. We began by looking at implicit regularization, specifically, stochastic gradient descent (SGD). The idea is that by using small batches of data, the gradient signal is noisy which can help explore the loss landscape and can drive the model to more robust solutions that generalize well. To set up the experiment, we went back to our original gradient descent setup and used batch sizes of 16, 64, and 256. For each batch we perform a forward and backward pass to update the model parameters. Figure 10 shows the results. The model trained with SGD with a batch size of 16 does not grok in the sense of how we have observed it thus far, but the test accuracy does steadily increase and achieves 10% accuracy by the end of training. This indicates that the implicit regularization via SGD does help the model generalize to an extent. However, this behavior is somewhat random and lucky. When we ran the



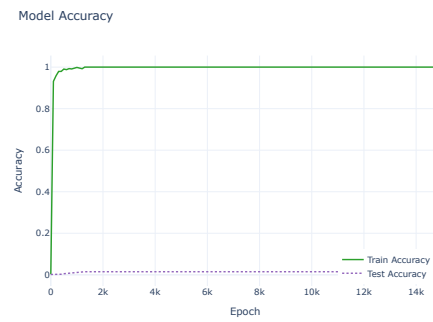
(a) Batch Size 16 Loss Curve



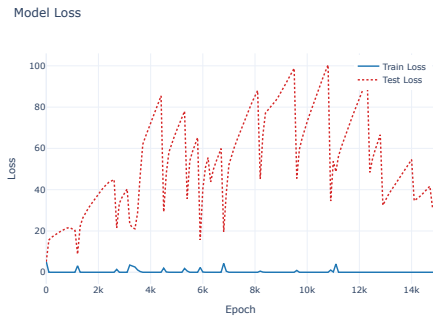
(b) Batch Size 16 Accuracy Curve



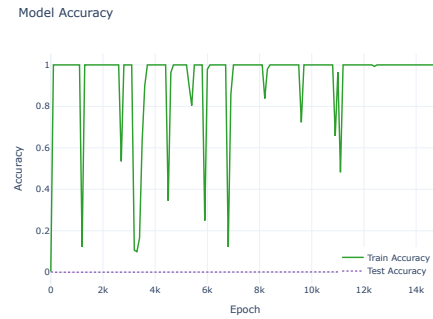
(c) Batch Size 64 Loss Curve



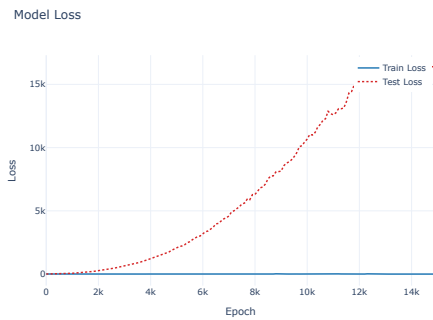
(d) Batch Size 64 Accuracy Curve



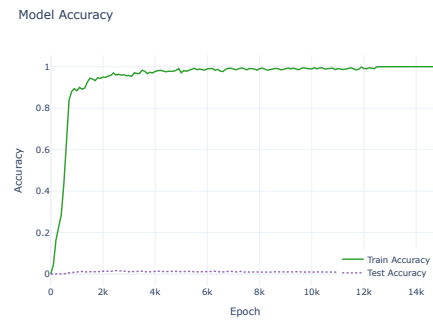
(e) Batch Size 256 Loss Curve



(f) Batch Size 256 Accuracy Curve



(g) Overparameterized Loss Curve



(h) Overparameterized Accuracy Curve

Figure 10: Implicit Regularization Experiments

exact same training code again, the test accuracy only got to around 3.05% and then flat-lined. For larger batch sizes of 64 and 256 the test accuracies remained around 0% throughout the duration of the training.

We tried one additional implicit regularization experiment where we drastically increased the model size. We used `demb ed = 512` and `dff = 2048`. Theoretical work has shown that models with far more parameters than the problem requires can lead to solutions that generalize better. For the experiment we again used a batch size of 16. Figure 10 shows the results. The implicit regularization attempt was unsuccessful and the model was unable to grok. These experiments demonstrate that generally, models are unable to grok without explicit regularization.

We also experimented with different types of explicit regularization. Previously, all of the models that successfully grokked used L_2 regularization or weight decay. L_2 regularization is defined as

$$\sum_p^{\text{params}} p^2$$

We don't take the square root of all the sum because it is computationally expensive and mathematically unnecessary. We also experimented with L_1 regularization, defined as

$$\sum_p^{\text{params}} |p|$$

For our experiment setup, we used the LBFGS optimizer and the same hyperparameters as in previous experiments. Most parameter values are < 1 so the L_2 penalty will be smaller than the L_1 penalty. To account for this, we reduced γ by a factor of 10 from 0.001 to 0.0001.

Figure 11 shows the training and test curves. Even with the explicit L_1 regularization, the model is unable to grok. The test accuracy very slowly increases to around 10% before training stops.

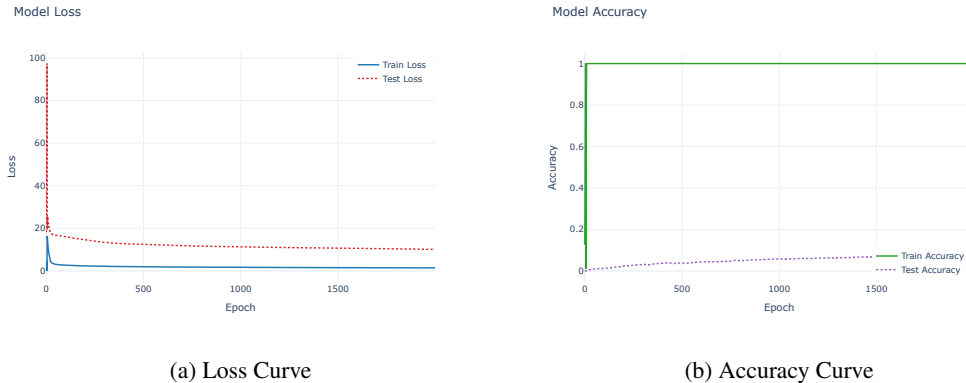


Figure 11: L1 Regularization

We next experimented with L3 regularization. Just as L2 regularization, we do not take the cube root of the total sum. Therefore, our L3 penalty is

$$\sum_p^{\text{params}} |p|^3$$

Because most model parameters are < 1 , the L3 penalty will be smaller than the L2 penalty. To compensate for this, we increase γ by a factor of 10 to 0.01.

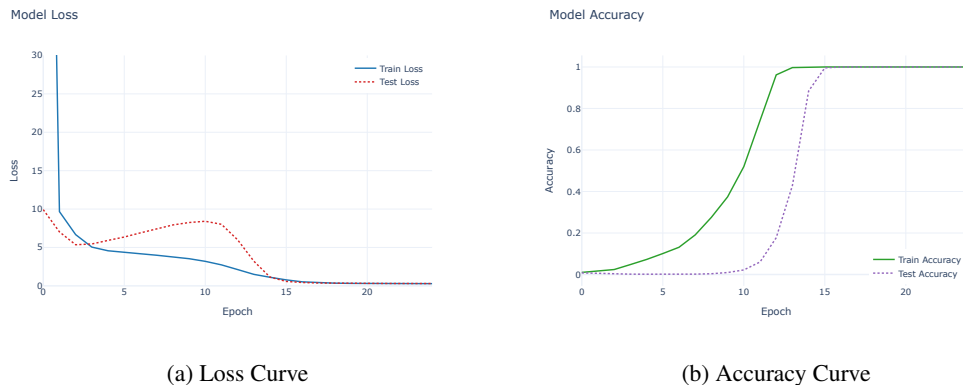


Figure 12: L3 Regularization

Figure 12 demonstrates the results. Using this form of regularization is successful and the model groks very quickly. We also experimented with L4 regularization defined as

$$\sum_p^{\text{params}} p^4$$

We further increased λ to 0.05.

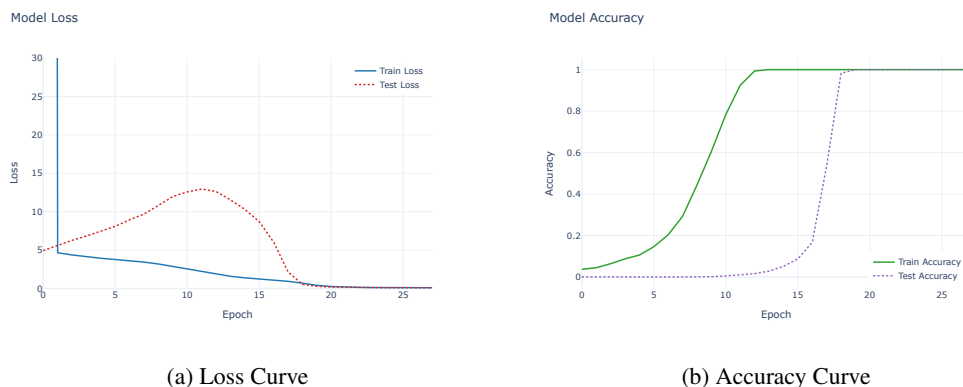


Figure 13: L4 Regularization

Figure 13 demonstrates the results. With L4 regularization the model also groks very quickly. Our experiments demonstrate convincing evidence that implicit regularization is not enough for the model to grok. Additionally, L1 explicit regularization is also not successful, but L2, L3, and L4 regularization are. We suspect this

is due to the derivatives of the regularization. The derivative of each regularization per parameter is

$$\frac{\partial L1}{\partial p} = \lambda \text{sign}(p) \quad \frac{\partial L2}{\partial p} = \lambda p \quad \frac{\partial L3}{\partial p} = \lambda |p|p \quad \frac{\partial L4}{\partial p} = \lambda p^3$$

For L1 regularization, the derivative is constant, and does not depend on the magnitude of the parameter at all. From our empirical results, it appears that to observe the grokking effect, the derivative of the regularization must be dependent on the magnitude of the parameter. Therefore, L2, L3, and L4 regularization all result in the model grokking, but L1 does not suffice because during each parameter update step, the parameters are being pushed smaller by only a constant value.

4 CONCLUSION

Our project provides extensive experiments surrounding the grokking effect observed on the modulo arithmetic problem. We successfully replicate the architecture and training procedure required to observe the grokking effect. Additionally, we discovered a simpler, more effective input representation, $[a, b]$ instead of $[a, b, =]$, which allows the model to grok quicker than previously observed in experiments by Nanda et al. (2023). We successfully applied a quasi-Newton optimization technique, LBFGS, to our training procedure and by manually adding a weight decay, the model was able to grok when trained with this technique. Furthermore, we show a substantial speed-up when using this technique, both in time required and in number of function calls. This speed up allowed us to perform more efficient future experiments. Next, we demonstrate that by scaling up model parameters and tuning λ accordingly, we can achieve the grokking affect for large modulo numbers. Finally, we show that implicit regularization is not sufficient for a model to grok. Additionally, we provide theoretical work showing that any explicit regularization whose derivative is a function of the model parameters themselves, combined with an appropriate λ value, will be sufficient for the model to grok. Overall, this project provided a very insightful look into the grokking phenomenon as well as how advanced optimization techniques learned in class can be used to improve machine learning training.

REFERENCES

- Welch Labs. The most complex model we actually understand. <https://www.youtube.com/watch?v=D8GOeCFFby4>, 2025.
- Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*, 2023. doi: 10.48550/arXiv.2301.05217. URL <https://arxiv.org/abs/2301.05217>.
- Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets. *arXiv preprint arXiv:2201.02177*, 2022. doi: 10.48550/arXiv.2201.02177. URL <https://arxiv.org/abs/2201.02177>.

Taorui Wang. L-bfgs two-loop recursion. *Medium*, April 2022.
URL [https://medium.com/@tru11631/
l-bfgs-two-loop-recursion-6976b6298f6c](https://medium.com/@tru11631/l-bfgs-two-loop-recursion-6976b6298f6c).