Parallelizing Artificial Neural Network Training with the Multigrid Algorithm

Dillon Montag
Dr. Luca Formaggia

June 10, 2020

1 Introduction

This project aims to reproduce the numerical experiments from [2], a paper which involves training neural networks with the MGRIT algorithm. The C++ code implements both neural networks and MGRIT from scratch following the methods outlined in [2] in order to reproduce the results as closely as possible. While many of the results are similar to those found in [2], there are some differences that can likely be explained by both the randomization needed for data generation and neural network initialization, as well as some probable implementation differences between this project and [2]. However, many of the general conclusions from [2] can still be drawn from the results obtained in this project.

2 The MGRIT Algorithm

In this project, the MGRIT algorithm is used to explore the potential of parallelizing the training of artificial neural networks. Since the MGRIT algorithm is highly parallel, showing that the MGRIT algorithm can be used to train neural networks is equivalent to showing that the training of such networks can also be processed in parallel. This project aims to show that the MGRIT algorithm can be effectively used to train neural networks, although the non-linearity of the networks counteracts many of the nice properties the MGRIT algorithm normally has (note that an actual parallel implementation of the MGRIT algorithm is beyond the scope of both [2] and this project).

2.1 Outline of the MGRIT Algorithm

Let $\alpha^{(b)}$ be the learning rate of the neural network and N be the number of training steps used to train the network. The MGRIT algorithm aims to find

the weights of the network by solving the system:

$$A \equiv \begin{bmatrix} I & 0 & 0 & ... & \\ -\phi() & I & 0 & ... & \\ & & \ddots & \ddots & \\ & & & -\phi() & I \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{N^{(0)}} \end{bmatrix} = \begin{bmatrix} w_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \equiv g$$

where A is a $N \times N$ matrix, I is the identity matrix, ϕ is the operator that takes the weights of the neural network at step i and gives back the weights at step $i+1$ where $i \in \{0, 1, ..., N\}$, and w_i is the vector of weights at training step i.

MGRIT solves this system by breaking down the domain of training steps into a fine and coarse grid. The points on the fine grid will be called F-points and the points on the coarse grid will be called C-points. F-relaxation is defined as the process of updating all of the F-points by applying ϕ to the nearest C-point behind that F-point. C-relaxation is the opposite: all of the C-points are updated by applying ϕ to the F-point directly behind that C-point (see Figure 1 below).

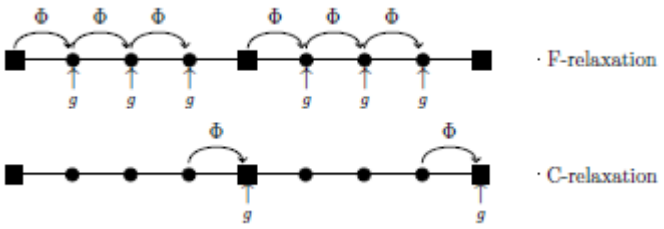

Figure 1: F and C-relaxation

The algorithm below shows the MGRIT implementation used in this project. Note that, in the algorithm, the superscript $^{(0)}$ denotes the fine grid, the superscript $^{(1)}$ denotes the coarse grid, m denotes the coarsening factor $m = \frac{N}{\text{number of nodes on the coarse grid}}$, and tol is a user-defined halting tolerance. The matrix B is defined as the square matrix

$$B = \begin{bmatrix} I & 0 & 0 & ... & \\ -\phi() & I & 0 & ... & \\ & & \ddots & \ddots & \\ & & & -\phi() & I \end{bmatrix}$$

where the size of B is the same as the number of nodes on the coarse grid.

Algorithm 1: MGRIT $(A^{(0)}, w^{(0)}g^{(0)}, tol)$

1 Apply FCF-relaxation to $A^{(0)}(w^{(0)}) = g^{(0)}$.
2 Inject the fine level approximation and its residual to the coarse level: $w^{(0)}_{mi} \rightarrow w^{(1)}_{i}$, $g^{(0)}_{mi} - (A^{(0)}(w^{(0)}))_{mi} \rightarrow r^{(1)}_{i}$
3 Solve $B^{(1)}(v^{(1)}) = B^{(1)}(w^{(1)}) + r^{(1)}$.
4 Compute the coarse level error approximation: $e^{(1)} = v^{(1)} - w^{(1)}$.
5 Correct $w^{(0)}$ at the C-points: $w^{(0)}_{mi} = w^{(0)}_{mi} + e^{(1)}_{i}$.
6 Update the F-points by applying F-relaxation to $A^{(0)}(w^{(0)}) = g^{(0)}$.
7 **if** $||g^{(0)} - (A^{(0)}(w^{(0)}))|| \leq tol$ **then**
8 | Halt.
9 **else**
10 | Go to step 1.
11 **end**

The algorithm above can be used for more than 1 coarse level by making it recursive at step 4. This project uses both V and F-cycles (as shown in Figure 2 below) to implement the recursion. As the authors of [2] do not specify an implementation for V and F-cycles, an implementation for this project is chosen based off of one of the common practices in the current literature (see [1] for the pseudocode). Also, as the authors of [2] do not specify the number of levels used for V and F cycles; the level 10 is chosen for this project based off of experimental results which provide similar outcomes. However, despite these potential implementation discrepancies with [2], many of the results obtained in this project are comparable to those obtained in [2].

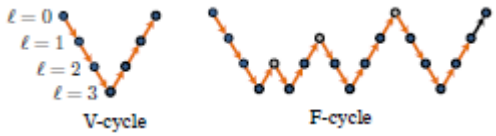

Figure 2: V and F-cycles

2.2 Brief Description of the C++ MGRIT Implementation

The C++ code is divided into the source code and unit tests. The headers for the source code are found in the *include* directory, the implementation of the headers are found in the *src* directory. The unit tests are found in the *tests* directory.

2.2.1 Source Code

The source code is designed to create maximum efficiency while balancing user flexibility. For this reason, all parameters to functions are passed as constant references to save memory, except when the function modifies a particular parameter, in which case that parameter is passed by value. A vector of matrices is chosen as the data structure to store the weights of the neural network. This data structure is chosen (as opposed to a structure like an array) primarily to simplify the code while still maintaining an acceptable level of efficiency. The source code is divided into two primary parts: the MGRIT implementation found in the *mgrit* directory and the neural network implementation found in the *neural_network* directory.

The MGRIT implementation consists of a main class, *MGRITSolver*, along with 3 helper classes. The *MGRITSolver* class is responsible for holding data about the MGRIT algorithm - such as the coarsening factor, the list of phi functions to use on each grid, and the maximum level the algorithm should recurse to - in order to be able to give the user flexibility to fine-tune the parameters of the algorithm. The class provides one public method to the user - *run* - which solves the linear system $Aw = g$ described above. The user provides w and g as the first two arguments of the method, along with a desired convergence tolerance as the third argument. The fourth argument - a boolean - determines whether the MGRIT algorithm runs V-cycles or F-cycles. If set to false, the algorithm runs V-cycles, otherwise it runs F-cycles. The list of phi functions is actually a list of lists. The outer list determines what set of phi functions to use on each grid, whereas the inner list determines what phi functions to use on certain nodes in a particular grid. As an example, $phis[1][2]$ uses the third user-specified phi function on the first coarse grid. This design is chosen primarily to support serialized training, which requires many different phi functions to be used on each grid.

The three helper classes are designed to handle the implementation of specific, independent functional pieces of the MGRIT algorithm. The *MGRITHelper* class implement functions that add/subtract two lists of weights, calculates the Euclidean norm of a list of weights, solves the system $Aw = g$ using a forward solve, calculates the product Aw, and finds the residual $g - Aw$, respectively. The *MoveGrids* class is designed to move a given set of objects from a fine to a coarse grid and vice versa. The *Relax* class is responsible for the relaxation, or smoothing, part of the MGRIT algorithm and implements F and C relaxation as described above.

The neural network implementation consists of one class, *NeuralNetwork*, which is responsible for implementing user-designed neural networks. The *NeuralNetwork* class stores a learning rate α and a set of weights which represent the values on the edges of the network. The class provides the user with the ability to train the network with the *train* method, which takes a set of inputs (the

training data) and a set of target values (the expected output) and performs one training step of the network. This training step involves feeding the input through the neural network and then backpropagating the error through the network using stochastic gradient decent. Note that the user is responsible for ensuring the dimensions of the matrices that represent the weights, as well as the input and target values, all have the correct dimension.

2.2.2 Unit Tests

Each class has its own unit test associated to it that tests the public interface of that class. The *tests* directory is structured exactly like the *include* and *src* directories to make finding the corresponding unit tests easy. When the project is built, the user can run the unit tests at any time (as described in the instructions in the README file).

2.3 Normal Workflow

When using the codebase, the normal workflow is as follows:

1. Construct the neural network you want to use by creating a set of initial weights and instantiating a NeuralNetwork object.
2. Initialize the weights and construct the right-hand side to use in the MGRIT algorithm (the initial w and g in the algorithm above).
3. Build the phi functions you want to use on each grid.
4. Use the above objects to instantiate a MGRITSolver object and use that object to call the run function, which returns the final weight vector.

To see examples of this workflow, view the *three_layer_problem.cpp* and *four_layer_problem.cpp* files under the *src* directory. Note that, should you want to experiment with different parameters for the MGRIT solvers using these neural networks, all you need to do is change the lines of code shown below.

```
const int N = 100;
const int m = 2;
const int max_level = 10;
const float alpha_b = 0.1;
const float alpha_max = 30.0;
const bool serialized_training = true;
const bool f_cycles = true;
const bool display_output = true;
```

3 Results

All of the results in this section are aimed at reproducing the results from [2]. The coarsening factor is set to $m = 2$. Following the notation of [2], in the following tables $N^{(0)}$ is the number of training steps (which corresponds to the number of fine nodes in the MGRIT algorithm), Iters is the number of iterations the algorithm takes to converge, and ρ is the average convergence rate computed using the geometric average $\rho = (\frac{||r_k||}{||r_0||})^{\frac{1}{k}}$ where r_k is the residual after k iterations and $||\cdot||$ is the Euclidean norm. The halting tolerance is $10^{-9}\sqrt{N^{(0)}}$.

3.1 Brief Definitions of the Solvers

The Naive Solver uses the two-level implementation of the MGRIT algorithm, a learning rate of $\alpha^{(b)} = 1.0$ on both levels to train the neural network, and utilizes batch training. Solver 1 is the same as the Naive Solver except that it uses more than one coarse grid to train the network and increases $\alpha^{(b)}$ by a factor of 2 on each successive coarse grid. Solver 2 is the same as Solver 1 except that it utilizes a serialized method instead of a batch method for training the neural network.

3.2 Three-Layer Problem Results

For the results in this section, the same three-layer problem as found in [2] is considered. This neural network is a three-layer network with three input nodes, four hidden nodes, and one output node (and so sixteen weights) that is trained to learn an exclusive OR (XOR) operation. The standard sigmoid function is used as the threshold function and the network is trained with backpropogation. Many of the results are comparable to the results found in [2], although the convergence of the MGRIT algorithm appears to be faster in this project when using F-Cycles. All solvers for the three-layer problem can be constructed by changing the parameters mentioned above in the *three_layer_problem.cpp* file in the *src* directory.

3.2.1 Naive Solver

For the Naive Solver, the following results are obtained:

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800	25,600
Iters	13	14	21	27	34	41	46	50	54
ρ	.23	.24	.40	.51	.58	.64	.68	.71	.73

Table 1: Naive Solver - Two-Level Setting, $\alpha^{(b)} = 1.0$, $\alpha^{(max)} = 1.0$

Note that, while MGRIT converges slightly faster in this case than in [2], the same pattern in the convergence rate can be observed.

3.2.2 Solver 1

For Solver 1, the following results are obtained:

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800	25,600
Iters	7	7	6	6	6	6	6	6	6
ρ	.05	.05	.05	.05	.05	.05	.05	.05	.05

Table 2: Solver 1 - Two-Level Setting, $\alpha^{(b)} = 1.0$, $\alpha^{(max)} = 2.0$

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800	25,600
Iters	4	5	8	15	13	12	13	32	50+
ρ	.007	.01	.10	.29	.25	.23	.25	.58	N/A

Table 3: Solver 1 - F-Cycles, Ten-Level Setting, $\alpha^{(b)} = 1.0$, $\alpha^{(max)} = 8.0$

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800	25,600
Iters	4	4	4	6	6	6	7	15	50+
ρ	.004	.004	.005	.04	.05	.05	.08	.32	N/A

Table 4: Solver 1 - F-Cycles, Ten-Level Setting, $\alpha^{(b)} = 0.5$, $\alpha^{(max)} = 8.0$

Note that, while once again the MGRIT algorithm converges faster for the problem in this project as compared to [2], the same pattern in the convergence rate is present. However, the algorithm has a hard time converging for training instances over 12,800, which may indicate that the α value of 8.0 is unsteady for this particular problem. This seems further supported by the fact that the two-level solver is able to converge in the same number of iterations for all problems sizes, which may indicate that $\alpha = 2.0$ is a more stable value.

3.2.3 Solver 2

For Solver 2, the following results are obtained:

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800	25,600
Iters	11	13	16	19	24	24	23	23	22
ρ	.15	.20	.28	.36	.42	.42	.44	.43	.42

Table 5: Solver 2 - Two-Level Setting, $\alpha^{(b)} = 1.0$, $\alpha^{(max)} = 2.0$

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800	25,600
Iters	6	8	10	14	23	27	31	36	50
ρ	.04	.08	.15	.26	.42	.49	.53	.57	.69

Table 6: Solver 2 - F-Cycles, Ten-Level Setting, $\alpha^{(b)} = 1.0$, $\alpha^{(max)} = 8.0$

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800	25,600
Iters	3	3	4	4	5	5	7	9	15
ρ	.0006	.001	.003	.005	.01	.03	.06	.14	.33

Table 7: Solver 2 - F-Cycles, Ten-Level Setting, $\alpha^{(b)} = 0.1$, $\alpha^{(max)} = 30.0$

Note that, while the results for Table 5 are almost identical to [2], the MGRIT algorithm converges much faster for Tables 6 and 7. There are several possible reasons for the discrepancies between the results. The random assignment of the initial weights, the number of maximum grid levels, and the implementation of the F-Cycle algorithm could all be different in this project than in [2], which could contribute to the faster convergence rate seen in this project. However, the same relative increase in the convergence rate as the one in [2] is present.

3.3 Four-Layer Problem Results

For the results in this section, the same four-layer problem as described in [2] is considered. Namely, a four-layer network with 12,032 weights is constructed. The network takes two random binary numbers (each 12 bits in length) and computers the 12 bit binary sum. The input layer of the network has 24 nodes, the first hidden layer has 128 nodes, the second hidden layer has 64 nodes, and the final output layer has 12 nodes. Once again, the sigmoid function and backpropogation are used to train the network. All solvers for the four-layer problem can be constructed by changing the parameters mentioned above in the *four_layer_problem.cpp* file in the *src* directory.

Some of the results in this section deviate from those found in [2], although there are several reasons which may explain these deviations and justify the results obtained in this project. For the following results, any run of the MGRIT algorithm which exceeded 50 iterations was automatically terminated. Results in which this happened are indicated with a "50+" symbol in the "Iters" row and "N/A" in the ρ row.

3.3.1 Naive Solver

For the Naive Solver, the following results are obtained when training over 500 training examples:

$N^{(0)}$	40	80	160	320	640	1280
Iters	8	12	15	15	15	15
ρ	.04	.15	.21	.21	.20	.20

Table 8: Naive Solver - Two-Level Setting, $\alpha^{(b)} = 0.1$, $\alpha^{(max)} = 0.1$

$N^{(0)}$	40	80	160	320	640	1280
Iters	10	21	43	50+	50+	50+
ρ	.06	.17	.56	N/A	N/A	N/A

Table 9: Naive Solver - Two-Level Setting, $\alpha^{(b)} = 0.025$, $\alpha^{(max)} = 0.025$

The patterns for the convergence rate of the Naive Solver are similar to the results found in [2] for Table 8, although the algorithm converges considerably faster in this project and the value of $\alpha^{(b)} = 0.1$ appears to be steady. Once again, both the initial randomization and the number of the weights could contribute to these discrepancies. Note that $\alpha^{(b)} = 0.025$ appears to be unsteady for the Naive Solver. This is different than the result found in [2], where the value of $\alpha^{(b)} = 0.025$ appears to be steady whereas $\alpha^{(b)} = 0.1$ appears to be unsteady. As there is not yet any theoretical underpinning to explain why different α values on different grid levels lead to different convergence rates, it is difficult to conjecture as to why this particular result is different. However, as the authors of [2] mention, the four-layer problem size and additional hidden layer seem to create difficulties for the MGRIT algorithm not present in the three-layer problem. It is possible that a stable α value lies inside of a certain region that is dependent on the initial state and configuration of the problem, both of which could be different between this project and the one reported by [2] due to randomization of both the training data and the initial weights.

3.3.2 Solver 1

For Solver 1, the following results are obtained when training over 500 training examples:

$N^{(0)}$	40	80	160	320	640	1280
Iters	5	8	10	10	10	10
ρ	.01	.04	.09	.09	.09	.09

Table 10: Solver 1 - Two-Level Setting, $\alpha^{(b)} = 0.1$, $\alpha^{(max)} = 0.2$

$N^{(0)}$	40	80	160	320	640	1280
Iters	10	21	50+	50+	50+	50+
ρ	.04	.32	N/A	N/A	N/A	N/A

Table 11: Solver 1 - Two-Level Setting, $\alpha^{(b)} = 0.025$, $\alpha^{(max)} = 0.05$

Note that the results are inverted with respect to the results found in [2]: the value $\alpha^{(b)} = 0.1$ appears to produce results which are both convergent and scalable, while $\alpha^{(b)} = 0.025$ produces results that appear to indicate stability issues on the coarse grid. This is consistent with the results found for the Naive Solver above.

3.3.3 Solver 2

For Solver 2, the following results are obtained:

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800
Iters	11	13	15	18	21	26	34	45
ρ	.13	.17	.23	.29	.36	.43	.54	.64

Table 12: Solver 2 - Two-Level Setting, $\alpha^{(b)} = 0.025$, $\alpha^{(max)} = 0.05$

The table above shows that MGRIT can converge for this particular problem, although the iteration counts are not scalable. The following tables show that the algorithm can converge a bit faster when increasing the number of grids, although the iteration count is still increasing. There does, however, appear to be a convergence problem when the algorithm has too many training steps to deal with, as shown by the last few entries in the tables.

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800
Iters	6	8	9	11	14	18	24	50+
ρ	.02	.05	.08	.13	.20	.28	.41	N/A

Table 13: Solver 2 - F Cycles, Ten-Level Setting, $\alpha^{(b)} = 0.025$, $\alpha^{(max)} = 0.2$

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800
Iters	12	16	20	25	32	41	50+	50+
ρ	.15	.23	.33	.42	.51	.59	N/A	N/A

Table 14: Solver 2 - V Cycles, Ten-Level Setting, $\alpha^{(b)} = 0.025$, $\alpha^{(max)} = 0.2$

To see if it is possible to get MGRIT to converge for large training steps, α is changed to grow at a rate of 25 percent (instead of doubling) on each level. As shown below, this actually increases the iteration count for F-cycles and worsens the ρ value, which suggests that these alpha values are somehow unstable on the coarse grids.

$N^{(0)}$	100	200	400	800	1,600	3,200	6,400	12,800
Iters	6	7	10	13	20	34	50+	50+
ρ	.01	.03	.09	.19	.34	.54	N/A	N/A

Table 15: Solver 2 - F Cycles, Ten-Level Setting, $\alpha^{(b)} = 0.025$, $\alpha^{(max)} = 0.2$, α increases by 25% on each level

These various tests all indicate that the value of α on the coarse grids has a large effect on both whether the MGRIT algorithm converges and how fast it converges. However, the precise nature of how these different α values are effecting the algorithm still remains a mystery.

4 Conclusion

While some discrepancies between [2] and this project exist, many of the results reproduced demonstrate similar trends and patterns. This is not only a good indication that the code is working as intended, but it also provides some interesting insight into how nonlinear problems interact with algorithms intended to solve linear problems. Of particular interest is the relationship between the learning rate of the neural network α on coarse grids and the convergence of the MGRIT algorithm, a relationship which appears to be crucial but still evasive. Also of interest is the potential speedup of the algorithm which demonstrates the potential utility of using MGRIT for neural networks in real life applications. However, further computing power, experimentation, and theoretical understanding is needed to verify the accuracy of these conclusions and guide further research into the connection between neural network training and the MGRIT algorithm.

References

[1] Multigrid method, Apr 2020.

[2] Jacob B. Schroder. Parallelizing over artificial neural network training runs with multigrid. *CoRR*, abs/1708.02276, 2017.