Implementation of A Generalist Neural Algorithmic Learner

Duc Minh Nguyen
Boston University
nguymi01@bu.edu

Ashwin Daswani
Boston University
ashwind@bu.edu

Rohan Sawant
Boston University
rohan16@bu.edu

ABSTRACT

This report is inspired by the generalist neural algorithmic learner [4], which is a graph neural network capable of learning a wide range of algorithms and execute them for tasks that may required incorporating multiple different algorithms. We aim to create an agent that is able to solve multiple algorithmic tasks that generalize out of distribution. We plan to demonstrate a generalist neural algorithmic learner: a single GNN, with a single set of parameters, capable of learning to solve several classical algorithmic tasks simultaneously—to a level that matches relevant specialist models on average. This represents a significant milestone because it demonstrates how reasoning abilities may be effectively applied to situations with completely different control flows.

1 INTRODUCTION

The bedrock of neural algorithmic reasoning[16] is the ability to be able to solve algorithmic tasks in a way that generalises out of distribution. Deep neural network-based machine learning systems have made tremendous progress in recent years, particularly for tasks dominated by perception.

Prominent models in this space are typically required to generalize in-distribution, which means that their training and validation sets should be representative of the expected distribution of test inputs. To truly master reasoning-dominated tasks, a model must provide sensible outputs even when generalizing out-of-distribution (OOD).

Neural networks have succeeded in many reasoning tasks. Empirically, these tasks require specialized network structures. Graph Neural Networks (GNNs) perform well on many such tasks. Our goal is to assess the ability of existing Graph Neural Network architectures on their ability to solve algorithmic tasks, by learning to execute classical algorithms.

This is a good target for assessing reasoning abilities because classical algorithms are crucial "building blocks" for all theoretical computer science. Graph Neural Networks pre-trained on algorithmic tasks have been successfully utilized in implicit planning [2] and self-supervised learning. Prior breakthroughs in this area have been centered on developing specialized models, either focusing on a single algorithm or a collection of algorithms with identical control flows.

We aim to implement a generalist neural algorithmic learner that has a single set of weights and can solve several algorithmic tasks such as sorting, shortest-path-finding, and searching. The goal of the project plan to train a graph neural network through CLRS30 benchmark[14], a collection of thirty classical algorithmic tasks such that the model can perform all these individual tasks as well as a specialist model that was trained to perform a single algorithmic task.

First, we will train a single-task model for each algorithm and obtain their performance results. Then we will train one model that can handle multi-algorithmic tasks and compare the results with each individual single-task model. Our model learns to predict the execution trace of an algorithm; that is, given an intermediate state of an algorithm (such as the longest common sub-sequence, a common DP algorithm), it predicts the next state, after 1 step of algorithmic computation has been done.

2 RELATED WORK AND REFERENCES

The research on generalist neural algorithmic learner was constructed by multiple scholars from *Deep Mind* [4], training a graph neural network model on the CLRS30[14] data set, training for single algorithmic task. Then they train the model for all 30 algorithms and compare the results with the single task result when handling that algorithmic task. Prior to their work, the closest work related to this idea was NeuralExecutor++, a multi-task algorithmic reasoning model by Xhonneux[13]. NE++ focuses on a highly specialised setting where all of the algorithms to be learnt have an identical control flow backbone and the team was able to make crucial , helpful observations for *Deep Mind's* work, such as empirically showing the specific forms of multi-task learning that are necessary for generalising OOD.

The team at *Deep Mind's* [4] incorporated different methods on the dataset and during training to improve upon the base model of the CLRS Benchmark [14] such as: removing teacher forcing, augmenting the training data, soft hint propagation, static hint elimination, improving training stability with encoder initialisation and gradient clipping using the Xavier initialisation[9] for reducing the initial weights for scalar hints whose input dimensionality is just 1 and use the default LeCun initialisation [19] elsewhere. All of which contribute to the improvement of over 20% average across all 30 tasks in CLRS compare to the base model. The paper also goes in depth on tweaking encoders, decoders and the processor network, which is message parsing neural network (MPNN)[12].

The team also used different techniques, methods to avoid overfitting, one of which was randomised position scalar which we have implemented and will discuss in the later sections of the paper. Beside that, they also use permutation decoders and the Sinkhorn operator where they enforce a permutation inductive bias in the output decoder of sorting algorithms[14].

ACM Conference, , July 2017, Washington, DC, USA.

The team at *Deep Mind* also work on modifying the processor from the benchmark[4], specifically modyfing the MPNN.

3 METHOD

We plan to implement the improvements that the team from *Deep Mind* [4] has accomplished in multi algorithmic task training. We plan to use the dataset given by CLRS30 benchmark[14], that provides input/output pairs for the 30 algorithms, and trajectories. A trajectory is formed of inputs, the corresponding outputs and optional intermediary targets—of those 30 algorithms.

We use graph neural networks(GNNs) to train the dataset. A GNN is an optimizable transformation on all attributes of the graph (nodes, edges, global-context) that preserves graph symmetries (permutation invariances). GNNs adopt a "graph-in, graph-out" architecture meaning that these model types accept a graph as input, with information loaded into its nodes, edges and global-context, and progressively transform these embeddings, without changing the connectivity of the input graph. The model architecture we will use is Message Passing Neural Network [8]. Message passing works in three steps:

(1) For each node in the graph, gather all the neighboring node embeddings (or messages)
(2) Aggregate all messages via an aggregate function (like sum).
(3) All pooled messages are passed through an update function, usually a learned neural network.

We plan to implement the encode-process-decoder paradigm similar to CLRS benchmark [14]. The encoders and decoders will be task specific whereas the processor will be common across all the tasks. The goal is to use a single set of weights for the graph neural network model and the modle will be capable of solving several algorithmic tasks in a shared latent space with simple encoders/decoders.

Each algorithm in the CLRS benchmark[14] is specified by a number of inputs, hints and outputs. In a given sample, hints are time-series of intermediate states of the algorithm and the inputs, outputs are fixed. The size of each sample for a particular task corresponded to the number of nodes in the GNN, n, that will execute the algorithm.

A sample of every algorithm is represented as a graph, with each input, output and hint located in either the nodes, the edges, or the graph itself. The CLRS benchmark [14] defines five types of features: scalar, categorical, mask, maskone and pointer, with their own encoding and decoding strategies and loss functions such as a scalar type will be encoded and decoded directly by a single linear layer, and optimised using mean squared error.

At each time step t of a particular task τ, the task based encoder will embed inputs and hints as high dimensional vectors. These embeddings of inputs and hints located in the nodes all have the same dimension and are added together; the same happens with hints and inputs located in edges, and in the graph. In our experiments we use the same dimension, h = 128, for node, edge and graph embeddings. The embeddings are fed into the processor P, that performs one step of computation. The processor converts the input node, graph and edge embeddings into processed node embeddings and it uses the processed node embedding from previous time step as inputs.

We use Message Passing Neural Networks [12], using the max aggregation and passing messages over a fully-connected graph as our base model. We use the fully connected graph—letting the neighbours j range over all nodes (1 <= j <= n)— in order to allow the model to overcome situations where the input graph structure may be suboptimal. The processed embeddings are finally decoded with a task-based decoder to predict the hints for the next step, and the outputs at the final step. Akin to the encoder, the task-based decoder relies mainly on a linear decoder for each hint and output, along with a mechanism to compute pairwise node similarities when appropriate.

3.1 Single-task Experiment

At the current stage, we are following the architecture of the team at *Deep Mind* [4] using the baseline graph neural network model from the CLRS Benchmark [14] with embedding size h = 128, we train for 10,000 cycles in batches of size 32 using an Adam optimizer with learning rate 0.003 using message-passing neural network (MPNN) processor[12]. At the end of the encoding step for a time-step t of the algorithm, we have a single set of embeddings

$$\{x_i^{(t)}, e_{ij}^{(t)}, g^{(t)}\}$$

shapes n × h, n × n × h, and h, in the nodes, edges and graph, respectively.

To set up how we are gonna evaluate the model, we follow the CLRS Benchmark [14] with a method to evaluates a model on feedback. In this method, we will first get the prediction of the model on the feedback (samples), then call the evaluate method from the CLRS benchmark [14] and modify the resulted dictionary to suit the model parameters. Then, we evaluates predictions against feedback similarly.

Afterwards, we collect batches of output and hint predictions and evaluate them using the above methods. This is accomplished by iterating through the samples, retrieve the feedback outputs and hints predictions, predictions, hints and append that all to an array for each then call the evaluate predictions againts feedback method above.

We follow the structure of the CLRS [4] baseline model. The loss is computed through the decoded hints and outputs according to their type during training. The hint prediction losses are averaged across hints and time for each sample in a batch, and the output loss is averaged across outputs (most algorithms have a single output, though some have two outputs) [14]. Then, the hint loss and output loss are added together to get the loss. Besides, the hint predictions at each time step are fed back as inputs for the next step, except possibly at train time if teacher forcing is used.

The evaluation metric for the perfomance of the model is F_1 score and evaluation used the fixed parameters established in the CLRS[14]

benchmark. At this stage, we are following the *Deep Mind* team improvement idea that is removing ground-truth hints, meaning the model has no access to the step-by-step hints in the dataset, and has to rely on its own hint predictions. Without this, losses tended to grow unbounded along a trajectory when scalar hints were present, and destabilised the training so we are still incorporating different significant stabilising changes to compensate for this and to avoid overfitting. (Work in progress).

During training, before implementing some of the methods we will discuss later, we obtained high or well above average scores for some tasks such as BFS, Task Scheduling, DAG Shortest Path and Activity Selector which is similar to the result from the paper [4]. However, some models that are trained for other algorithmic tasks does not perform as well as the result shown by the team at *Deep Mind* so we still need to improve on that.

The structure of the training process is set up as follows: first we generate samples for training, validating and testing, then created a CLRS model [14] with parameters stated above. Then, during the training loop we will evaluate the model periodically and create a check point when the validating score is higher compare to the previous validation score. At the end of the training loop, we will load the check point to retrieve the parameters for our model that is shown to preform the best (has the best validation scores).

We have saved the parameters (set of weights) of 23 models that was trained for a each individual algorithmic task. In the next section, We will show the improvement after implementing different techniques to train graph neural networks for the rest of the algorithms. We plot the models scores through batches of output and hint predictions of the resulted parameters evaluate on the test data set (this takes long computational time). This is the result of the old models we got when evaluating on test data set shown in Figure 1.

3.2 Multi-tasks Experiment

In the multi-task setting, we train a single processor across all CLRS-30 tasks. We keep encoders and decoders separate for each task. To perform the update, one might accumulate gradients from all the tasks before stepping the optimizer, or step independently after each batch from each algorithm. Batch size and learning rate are the same as in single-task experiments.

The trained model is evaluated periodically during training on samples of size n= 16 (n = 20 for string matching algorithms), and the best-performing model seen so far is evaluated on OOD samples, size n = 64 (n = 80 for string matching). Only OOD performance is reported in this paper. The OOD data used for evaluation is sampled on-the-fly, drawn randomly at each evaluation, the number of samples being the same as in the CLRS benchmark[11]

4 MODEL IMPROVEMENTS

We tried to implement techniques that were talked about or mentioned in the *Deep Mind* paper [4], which improved greatly on some

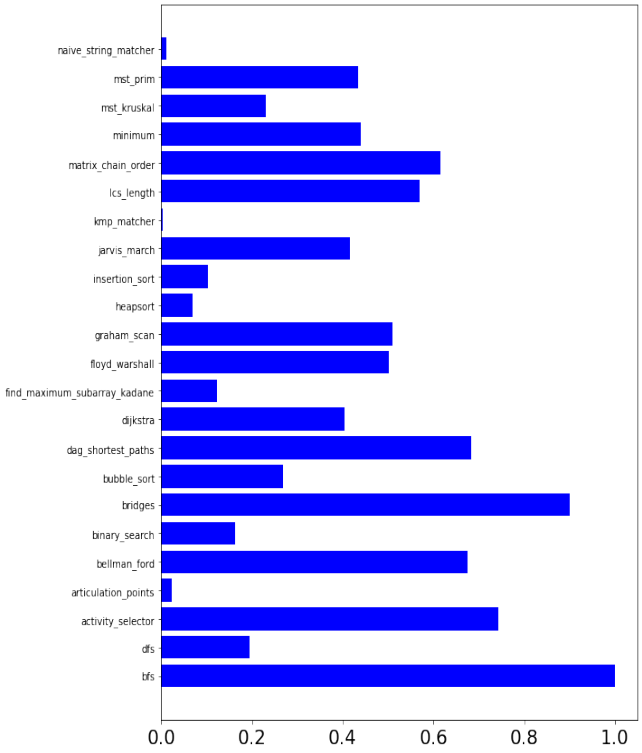

Figure 1: Single Task Experiment

our previous results, while some stay the same or get slightly lower score.

Some of the implementation we tried on dataset and training were removing teacher forcing[17], soft hint propagation, static hint elimination and randomised position scalar.

4.1 Randomised position scalar

We implemented a randomised position scalar method that is talked about in the paper by the team at *Deep Mind* [4]. There exists a position scalar input which uniquely indexes the nodes, with values linearly spaced between 0 and 1 along the node. By default, the scalar is uniformly spaced between 0 and 1 across the nodes.

What happened with our previous results was that some algorithm perform much better during training, getting high validation score but much lower test scores when perfroming on the test samples. This could be a reason of overfitting when training. We followed the paper and replace the uniformly spaced position scalar with an ordered sequence of random scalars.

The benefit of this change is notable in some algorithms we tested where it may have been easy to overfit to these positions, such as string matching. Namely, the model could learn to base all of its computations on the assumption that it will always be finding a certain character pattern with length k inside a string with n characters,

despite the fact that at test time, k and n will increase tremendously.

To do this, first we set up by calling a random generator with seed 42. Then iterate through the training samples and get the array of the position scalars, which has the name 'pos', from the inputs features. Then, for the current sample iteration, we went through one batch from that sample at a time, checked if there are splits in the 'pos' array, marked by 0s. Then we can perform randomizing the position scalar by sorting the random values in each split and concatenating them then save them in a new array after each batch. At the end of the iteration, replace the position scalar of the inputs features with the new one and we will get randomised position scalar.

After implementing randomised position scalar, the performance for some algorithmic tasks has significant improvements, as high as 10 times on the test samples, while some others stay relatively the same or a bit worse. Some notable algorithmic tasks that benefit from this approach are Articulation Points, Naive String Matcher[7], Binary Search[3], Dijkstra[6].

4.2 Static Hint Elimination

Eleven algorithms in CLRS [15] specify a fixed ordering of the nodes, common to every sample, via a node pointer hint that does not ever change along the trajectories.Prediction of this hint is trivial (identity function), but poses a potential problem for OOD generalization, since the model can overfit to the fixed training values. We therefore turned this fixed hint into an input for these 11 algorithms: Binary Search, Minimum, Max Subarray, Matrix Chain Order, LCS Length, Optimal BST, Activity Selector, Task Scheduling, Naïve String Matcher, Knuth-Morris-Pratt and Jarvis' March.

4.3 Improving training stability with encoder initialisation and gradient clipping

The scalar hints have unbounded values, in principle, and are optimised using mean-squared error, hence their gradients can quickly grow with increasing prediction error. Further, the predicted scalar hints then get re-encoded at every step, which can rapidly amplify errors throughout the trajectory, leading to exploding signals (and consequently gradients), even before any training takes place.

To rectify this issue, we use the Xavier initialisation, effectively reducing the initial weights for scalar hints whose input dimensionality is just 1. This combination of initializations should have proved important for the initial learning stability of our model over long trajectories.

We surprisingly didn't find any improvements by using different initialization and ended up reverting to the default LeCun initialization .

4.4 Soft Hint Propogation

When predicted hints are fed back as inputs during training, gradients may or may not be allowed to flow through them. In previous work, only hints of the scalar type allowed gradients through, as all categoricals were post-processed from logits into the ground-truth format via argmax or thresholding before being fed back. Instead, in this work we use softmax for categorical, maskone and pointer types, and the logistic sigmoid for mask types. Without these soft hints, performance in sorting algorithms degrades (similarly to the case of teacher forcing), as well as in Naïve String Matcher

5 RESULTS

By incorporating the changes described in the previous sections we arrived at a single model type, with a single set of hyper-parameters, that was trained on CLRS-30. After implementing all the above improvements, the performance for some single algorithmic tasks has significant greater score, as high as 10 times on the test samples, while some others stay relatively the same or a bit worse. Some notable algorithmic tasks that benefit from this approach are Articulation Points, Naive String Matcher[7], Binary Search[3], Dijkstra[6].

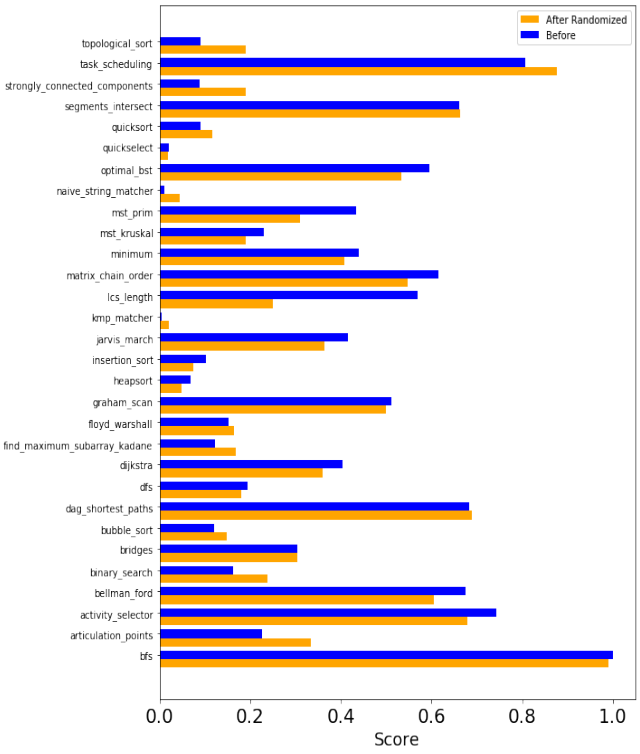

Figure 2: Single Task Experiment with Improvements

Figure 3 shows the comparison of multitask training with randomised position scalar method, removing static hints for 11 algorithms, soft hint propagation and gradient clipping with the result from the original single task experiments. We trained each algorithm for 300 cycles at a time and repeated the process for 5 iterations. Due to the compute constraints we weren't able to train them for more iterations. The results for many tasks are in the

same ballpark as the single task experiments with a few notable improvements on algorithms such as Strongly Connected Components, KMP matcher and Activity Selector. Multitask training however led to worse results for some tasks.

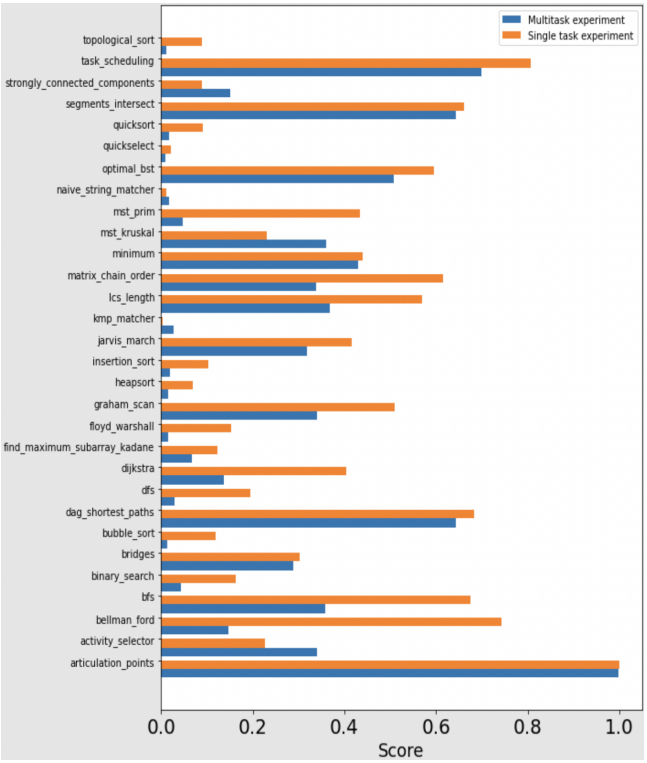

Figure 3: Multi Task Experiment

6 NEXT STEPS

We have trained the MPNNs[12] in both single task and multi task fashion. But some of the tasks don't generalise that well as the score of OOD test is much much lower than the score of the training and validation datasets. We plan to analyse which tasks are able to generalise well in all the numerical difficulties such as unstable gradients are only amplified while running the multi-task experiments. We are able yet to implement some the improvements suggested in the paper such as adding gating mechanisms to the processor MPNNs.

To reduce the memory footprint of multi-task training we can implement a chunked training mode, where trajectories are split along the time axis for gradient computation and, when they are shorter than the chunk length, are concatenated with the following trajectory so as to avoid the need of padding, Thus, while a standard-training batch consists of full trajectories, padded to the length of the longest one, a chunked-training batch has a fixed time length (16 steps in our experiments) and consists of segments of trajectories. Immediately after the end of one trajectory the beginning of another one follows, so there is no padding. Losses are computed independently for each chunked batch, and gradients cannot flow between chunks. Since the output loss is computed only on the final sample of each trajectory, a chunk may give rise to no output loss, if it contains no end-of-trajectory segments. Chunking, therefore, changes the balance between hint and output losses depending on the length of trajectories.

The range of improvements to the dataset, optimisation and architectures for neural algorithmic reasoning, led to over 20% absolute improvements over the prior best known result[18]. We also plan to explore different graph neural network architectures and specialized multi-task optimizers.

7 GITHUB REPOSITORY

The projects can be accessed at

https://github.com/ashwindaswani/generalist-neural-algorithmic-learner

The dependencies are added in requirements.txt in the github. These libraries such as jax[5] (which is not supported for windows), numpy[10], tensorflow[1], and mainly clrs from deepmind [15] will need to be installed

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Ognjen Milinkovic Pierre-Luc Bacon Jian Tang Mladen Nikolic ´ Andreea Deac, Petar Velickovi ˇ c. 2021. *Neural Algorithmic Reasoners are Implicit Planners.*

[3] Jon Bentley. 1977. Programming pearls: algorithm design techniques. , 865-873 pages.

[4] George Papamakarios Kyriacos Nikiforou Mehdi Bennani Róbert Csordás Andrew Dudzik Matko Bošnjak Alex Vitvitskyi Yulia Rubanova Andreea Deac Beatrice Bevilacqua Yaroslav Ganin Charles Blundell Petar Velickovi Borja Ibarz, Vitaly Kurin. 2022. *A Generalist Neural Algorithmic Learner.*

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[6] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. , 269–271 pages.

[7] Jr Donald E Knuth, James H Morris and Vaughan R Pratt. 1984. Fast pattern matching in strings. SIAM journal on computing. , 323–350 pages.

[8] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. [n.d.]. Neural Message Passing for Quantum Chemistry. https://doi.org/10.48550/ARXIV.1704.01212

[9] Xavier Glorot and Yoshua Bengio. 2010. Proceedings of the thirteenth international conference on artificial intelligence and statistics.

[10] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[11] Shengding Hu Zhengyan Zhang Cheng Yang Zhiyuan Liu Lifeng Wang Changcheng Li Maosong Sun Jie Zhou, Ganqu Cui. 2020. *Graph neural networks: A review of methods and applications.*

[12] Patrick F Riley Oriol Vinyals Justin Gilmer, Samuel S Schoenholz and George E Dahl. 2017. *International Conference on Machine Learning.*

[13] Petar Velickovic Louis-Pascal Xhonneux, Andreea-Ioana Deac and Jian Tang. 2021. How to transfer algorithmic reasoning knowledge to learn new algorithms? Advances in Neural Information Processing Systems.

[14] David Budden Razvan Pascanu Andrea Banino Misha Dashevskiy Raia Hadsell Charles Blundell Petar Velickovi, Adria Puigdomenech Badia. 2022. *The CLRS Algorithmic Reasoning Benchmark*.

[15] Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. 2022. The CLRS Algorithmic Reasoning Benchmark. https://github.com/deepmind/clrs

[16] Petar Veličković and Charles Blundell. 2021. *Neural algorithmic reasoning*. Vol. 2. 100273 pages. https://doi.org/10.1016/j.patter.2021.100273

[17] Ronald J Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. , 270-280 pages.

[18] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2019. What Can Neural Networks Reason About? https://doi.org/10.48550/ARXIV.1905.13211

[19] Yoshua Bengio Yann LeCun, Léon Bottou and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE.