More Random Graphs for Neural Architecture Search

Andrew Corum

May 4, 2021

1 Introduction

The application of deep neural networks to problems in computer vision is computationally expensive, due to the size of the input image space. A deep neural network for classifying $N \times N$ image requires $\mathcal{O}(N^2)$ weights just in the first hidden layer, which grows quickly for even relatively small images. To get around this, modern deep neural networks use convolution layers to reduce the size of the network. However, these convolution neural networks (CNNs) need to be carefully constructed to fit the desired application. Some CNN structures (such as AlexNet (figure 1) [1], VGGNet [2], ZFNet (figure 2) [3], and ResNet [4]) are already known to do well in many image classification settings. However, the performance of any given CNN is largely application dependent, so many times new CNNs must be carefully constructed for new projects and datasets.

The remainder of this paper is organized as follows. Section 2 briefly highlights other papers that seek to find optimal neural network architectures, as well as the main paper of interest. In section 3, the methods, design, and implementation of this project are detailed. Section 4 shows the results of experimentation of the models created in this paper. Finally, section 5 provides some discussion and interpretation of these results.

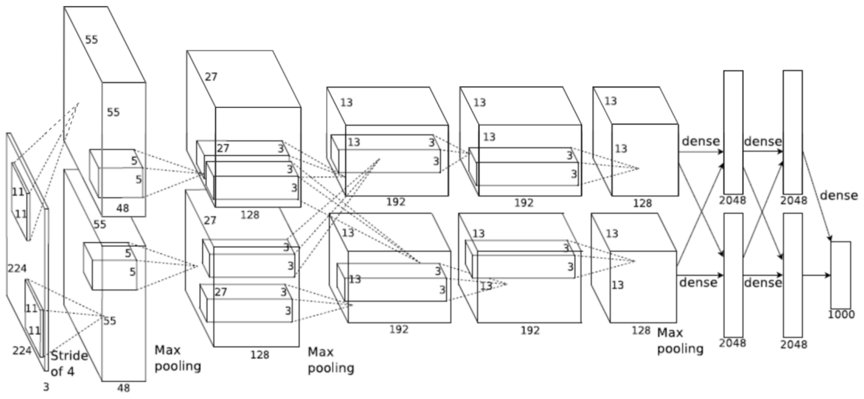


Figure 1: AlexNet architecture [1]

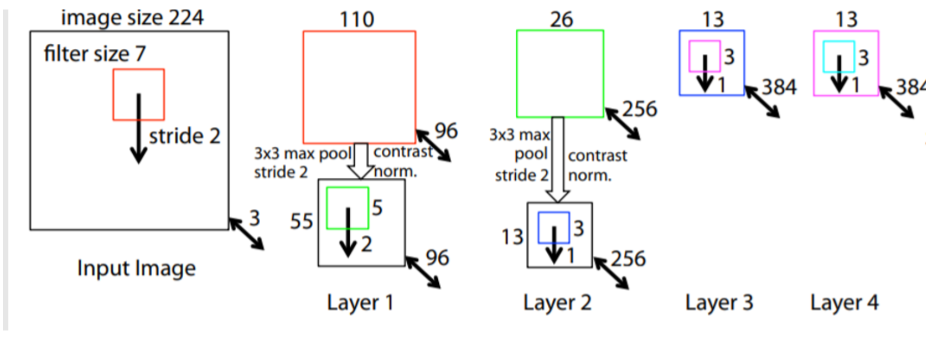


Figure 2: ZFNet architecture [3]

2 Related Work

One method to find promising CNN structures is called neural architecture search (NAS) [5, 6, 7]. NAS tries to systematically search for new layered architectures to optimizes the performance of the CNN. NAS-based models have recently been tested and performed very well, as seen in a Google paper showcasing NASNet [8]. Due to the large search space, NAS methods rely on heavy constraints to find graphs that will perform well in CNNs.

In an ICCV 2019 paper, Xie et al attempted to circumvent the constraints put on current NAS methods by generating randomly wired neural networks, creating a CNN called RandWire [9]. Surprisingly, their results showed that certain randomly generated networks were competitive with even some of the best modern CNNs. This discovery implies that there may be many other powerful CNN architectures that are overlooked due to over-constrained NAS techniques. The CNN development process could benefit greatly from further exploration into the use of these random/diverse architectures.

3 Methods

The primary goal of this project is to reproduce the results of Xie et. al. [9]. It is fascinating that random wirings within CNNs could perform on-par with other modern architectures, and there is strong motivation to explore this idea further. In order to reproduce the results of Xie et. al., we need to construct the random graph generators used in their paper, the build RandWire CNN from these graphs. We also should construct some of the classic CNN architectures to compare performance with RandWire.

Instrumental in my aiding construction of RandWire was a PyTorch implementation found on GitHub [10]. Since I was able to find a PyTorch RandWire network, I thought it would be interesting to re-create it in TensorFlow instead. Additionally, there seems to be a couple discrepancies between the PyTorch implementation and what is described in [9], so my results and construction differ from what is found in the PyTorch repository [10].

3.1 Random Graphs

First the random graphs are constructed, applying the graph-theoretic methods outlined in RandWire [9]. The three methods used are Erdős-Rényi [11], Barbási-Albert [12], Watts-Strogatz [13]. In this project, $N = 32$ nodes was used for each graph, and the nodes are numbered $0..(N-1)$.

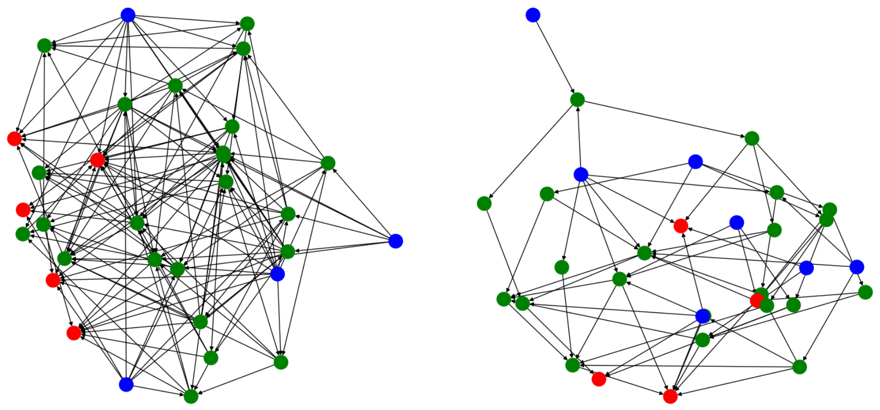


Figure 3: ER DAGs generated using N=32. Left uses P=0.3, and right P=0.15

The Erdős-Rényi (ER) method takes a parameter P, which is the probability that any potential edge is added to the graph. The algorithm works by simply iterating over all possible edges of the form $e = (n_i, n_j)$ (where $i < j$, to ensure the directed graph is acyclic), and adding the edge to the graph if $P > P_i \sim U_{0..1}$ (ie, P_i is sampled from the uniform distribution between 0 and 1). Figure 3 shows two ER graphs generated by my project. The blue nodes represent the input nodes, green are hidden/internal nodes, and red are output nodes.

Barbási-Albert's (BA) method of graph generation takes a parameter M. $BA(M)$ starts with M nodes, then adds a new node that is fully connected to the previous M nodes. For the remaining $N - M - 1$ nodes, when node n_j is added, M new edges are created. An edge $e = (n_i, n_j)$ is chosen with probability proportional to $\frac{D(n_i)}{\sum_{k=0}^{j-1} D(n_k)}$, where $D(n_k)$ is the degree of a node n_k. Figure 4 shows two BA graphs. Again, these will have a directed acyclic structure, which is needed for easier CNN construction and backpropagation.

The Watts-Strogatz (WS) method of graph generation takes two parameters, K and P. $WS(K, P)$ starts with a ring of N nodes, each connected to its $K/2$ neighbors. Then for $K/2$ iterations, we randomly rewire M edges if $P > P_i \sim U_{0..1}$. Figure 5 shows two WS graphs created in this project. Notice, when P is low ($P = 0.1$), not many of the edges get rewired, so the graph still keeps a ring-like topology.

3.2 RandWire

The random DAGs constructed using the methods described above then need to be combined into a single CNN architecture. Xie et al explain how to do this in detail. First the random graph needs to be converted into a DAG layer. Directed edges mark the flow of data/tensors through the network. At each node, incoming tensors are aggregated using a weighted sum (with trainable weights). After aggregation, we perform a ReLU activation, 2D batch normalization, and finally a 3x3 kernel 2D Convolution. The result of this convolution is then sent along the outgoing edges.

The RandWire network uses three of these DAG layers in succession. Two convolution layers are used before the DAG layers, and one fully connected layer after the last DAG. Figure 6 from Xie et al gives a nice description of this CNN architecture/layout, and figure 7 shows how this architecture might look when using the DAG layer.

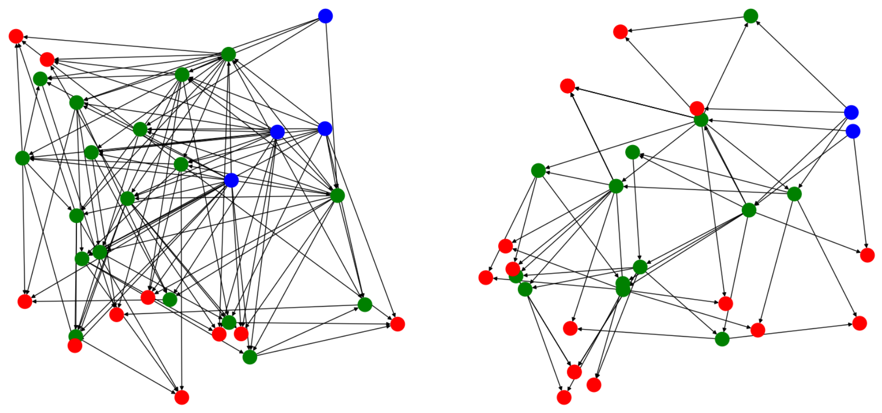


Figure 4: BA DAGs generated using N=32. Left uses M=4, and right M=2

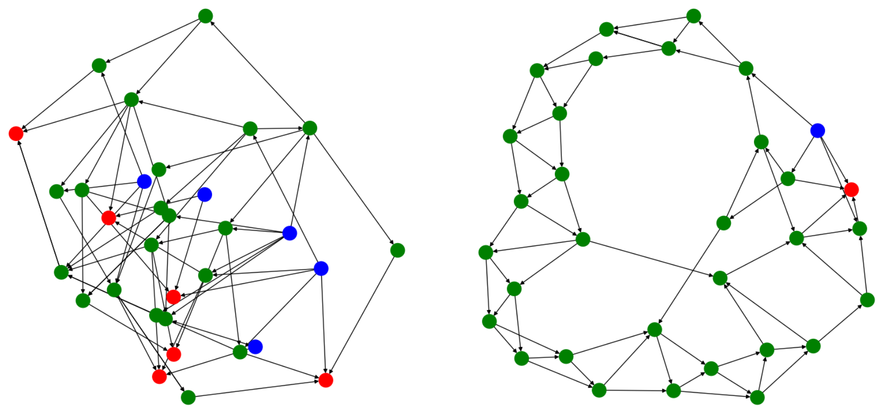
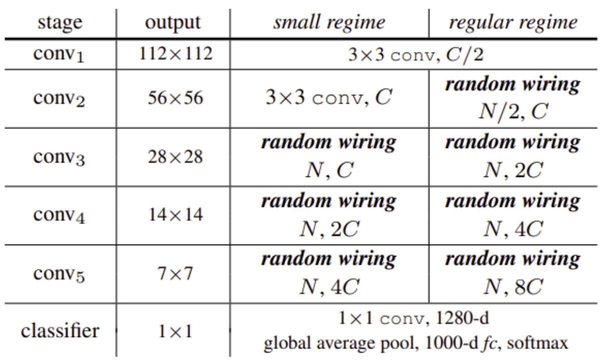


Figure 5: WS DAGs generated using N=32. Left uses K=4 P=0.75, and right K=4 P=0.1



stage	output	*small regime*	*regular regime*
conv$_1$	112×112	3×3 conv, $C/2$	
conv$_2$	56×56	3×3 conv, C	***random wiring*** $N/2, C$
conv$_3$	28×28	***random wiring*** N, C	***random wiring*** $N, 2C$
conv$_4$	14×14	***random wiring*** $N, 2C$	***random wiring*** $N, 4C$
conv$_5$	7×7	***random wiring*** $N, 4C$	***random wiring*** $N, 8C$
classifier	1×1	1×1 conv, 1280-d global average pool, 1000-d *fc*, softmax	

Figure 6: Sample RandWire architecture [9]

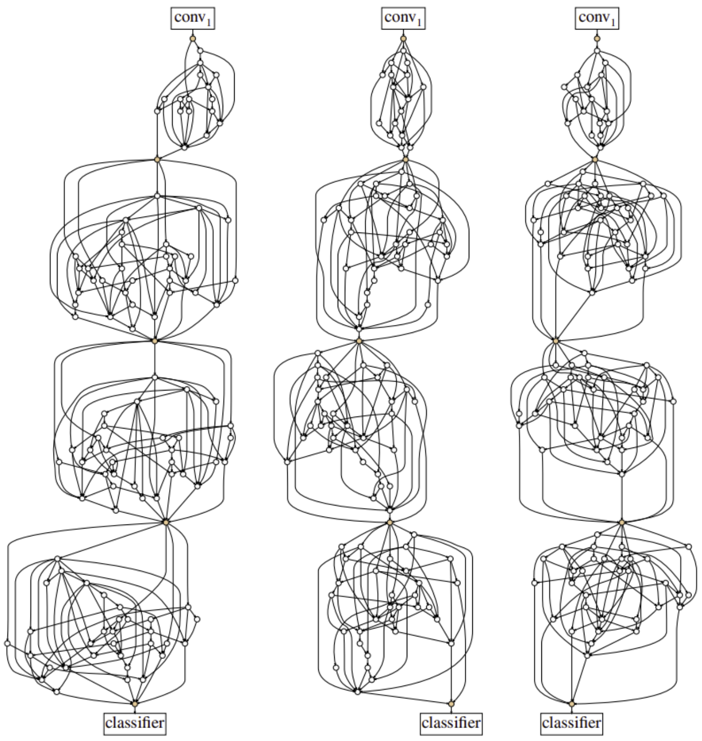


Figure 7: Sample RandWire architectures [9]

3.3 Other CNNs

After creating RandWire, this project constructed three other CNNs to help benchmark the performance of RandWire. The chose CNNs are AlexNet, a TinyCNN, and a HandmadeCNN. AlexNet [1] is a well known architecture for its great performance on the ImageNet dataset. This project used an implementation of AlexNet described by [14], but altered to work on the (28x28) MNIST dataset. Then TinyCNN contains just three Conv2D layers (using 3x3 kernels and output sizes of 32, 64, 64). These convolution layers are then followed by three fully connected layers of size 70, 30, and 10. The HandmadCNN has gone through many iterations to improve its performance on MNIST specifically. It uses five Conv2D layers, batch normalization, and dropout to prevent overfitting. The convolution layers are then followed by two fully connected layers of size 128 and 10.

4 Results

Three different RandWire models were tested. Each has three DAG layers of 32 nodes. The first RandWire model uses $ER(P = 0.2)$ for all the layers, the second $BA(M = 5)$, and the third model applies $WS(K = 4, P = 0.75)$. These particular parameters were chosen based on the highest-performing graphs demonstrated by Xie et al. All three Randwire models, AlexNet, TinyCNN, and HomemadeCNN were traind on the MNIST dataset. With the exception of AlexNet, the models were trained for 100 epochs on 60,000 images using batch size $2,048$, and tested on 10,000. Due to

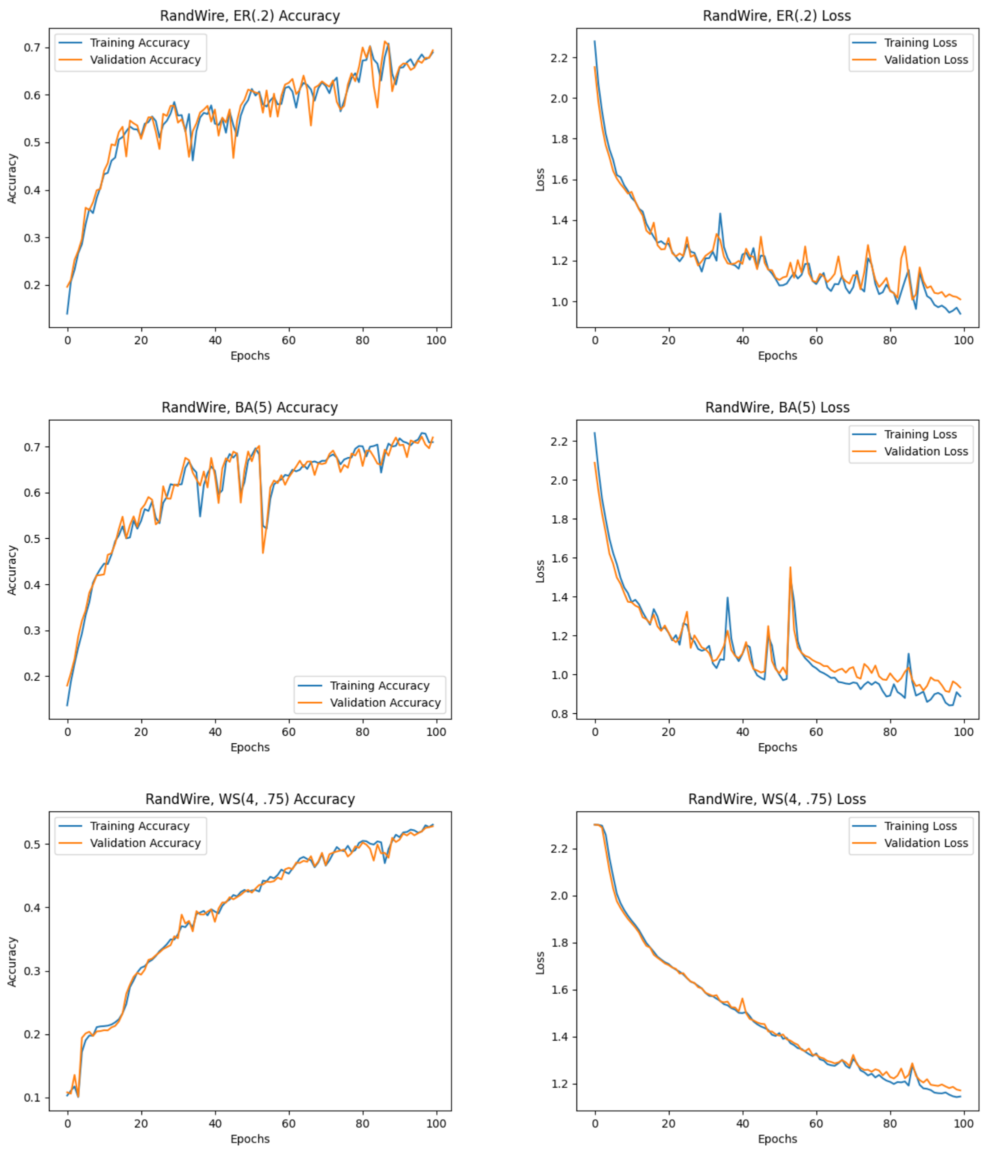


Figure 8: Training curves for RandWire CNNs on MNIST data

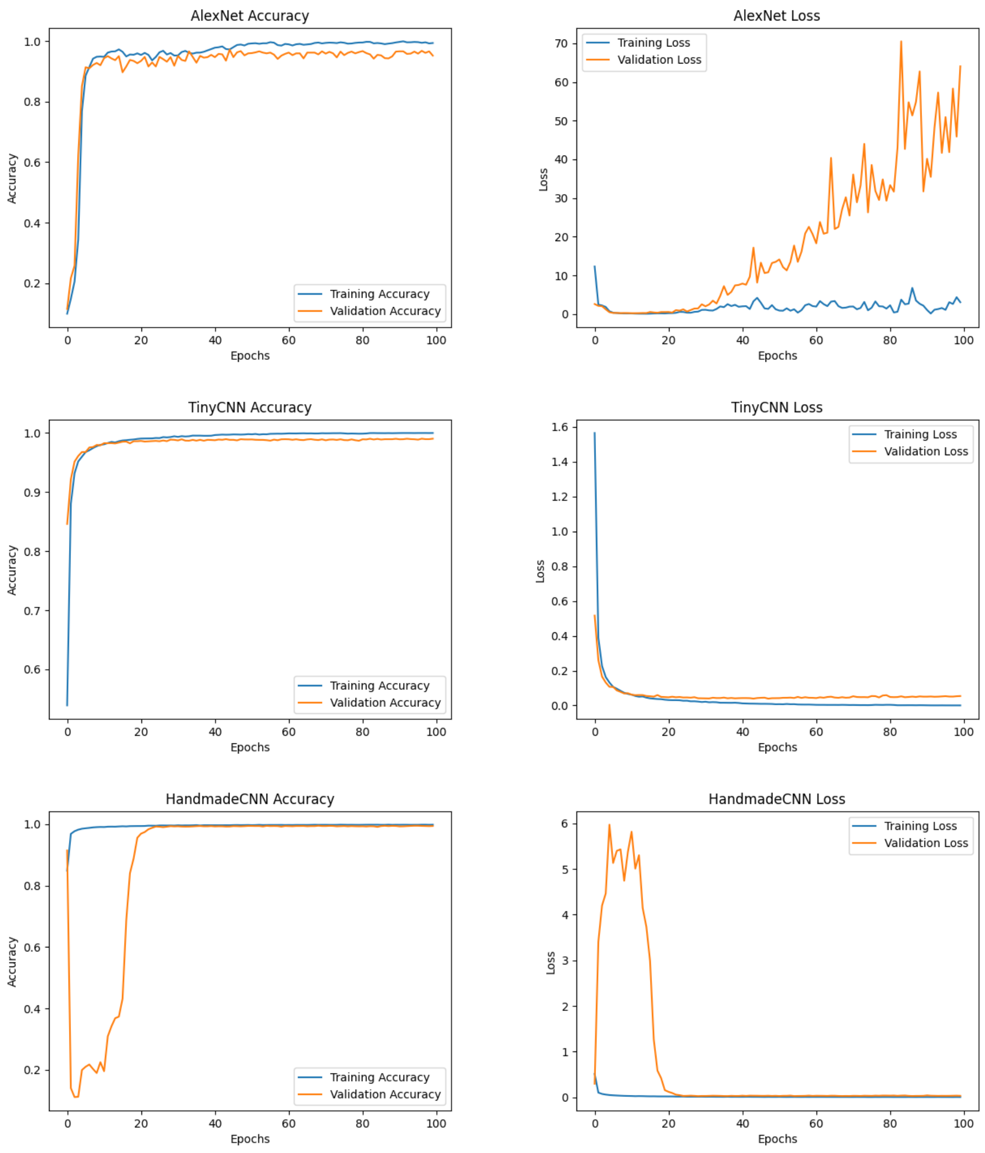


Figure 9: Training curves for classic CNNs on MNIST data

Model	**Test Accuracy**	**Test Loss**
RandWire, ER(.2)	69.49%	1.016
RandWire, BA(5)	71.67%	0.9562
RandWire, WS(4, .75)	54.31%	1.146
AlexNet	96.40%	57.07
TinyCNN	99.17%	0.04160
HandmadeCNN	99.46%	0.02386

Table 1: CNN results on test data.

limited computational/GPU resources, AlexNet could only be trained on a third of this data. The resulting training curves of these experiments are shown in figures 8 and 9. The results of these models on test data can be found in table 1.

5 Discussion

From the results described in section 4, it is clear that my implementation of RandWire does not perform at the same level as what Xie et al tout in their recent ICCV paper. Does this mean RandWire does not have the potential that is promised by Xie et al? Not necessarily. RandWire was built to perform on ImageNet, which contains images that are significantly larger than those in the MNIST dataset. As seen in the attempt to directly apply AlexNet to the MNIST dataset, larger models may need to be restructured to perform well on smaller data.

Additionally, given the RandWire training curves shown in figure 8, it seems that RandWire models could have improved performance with more learning time. These curves have not yet flattened out at 100 epochs. Perhaps also tweaking the learning rate could speed up RandWire's ability to learn the MNIST dataset. This project uses the Adam learning algorithm, with `learning_rate` = 0.001 and `epsilon` = 0.00001, but tweaking these parameters, or choosing a different optimizer, could improve the RandWire learning rate significantly.

6 Conclusion

In recent years, CNNs have become a realistic approach to solving the image classification problem. In an attempt to find good CNN architectures, Neural Architecture Search (NAS) arrives as a technique to systematically find and test new CNNs. RandWire [9] greatly expands the search space of traditional NAS algorithms, by using random graphs to wire the CNN. RandWire can take advantage of pre-designed graph-theoretic algorithms to create the internal directed acyclic graphs (DAGs). This project reconstructed RandWire using both their paper description [9] and an open PyTorch construction [10] as references. The reconstructed RandWire model did not perform as well as AlexNet or the other two CNNs created for reference (TinyCNN, HandmadeCNN). However, this discrepancy is likely due to the fact that this particular RandWire model designed to be used on larger data (from ImageNet) rather than the small data found in MNIST. Addtional tweaks to this project's RandWire learning algorithm could also increase performance, as the learning curves in figure 8 seem to indicate that the model has not yet reached minimum loss. Future work should seek to further investigate RandWire's performance by testing its performance on ImageNet and with different optimization algorithms.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[3] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[5] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[6] Juhong Min, Jongmin Lee, Jean Ponce, and Minsu Cho. Hyperpixel flow: Semantic correspondence with multi-layer neural features. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3395–3404, 2019.

[7] Jiemin Fang, Yuzhu Sun, Qian Zhang, Yuan Li, Wenyu Liu, and Xinggang Wang. Densely connected search space for more flexible neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10628–10637, 2020.

[8] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018.

[9] S. Xie, A. Kirillov, R. Girshick, and K. He. Exploring randomly wired neural networks for image recognition. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1284–1293, 2019.

[10] SeungWonPark. Randwirenn, 2019.
https://github.com/seungwonpark/RandWireNN/tree/0850008e9204cef5fcb1fe508d4c99576b37f995.

[11] Paul Erdos, Alfréd Rényi, et al. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[12] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[13] Duncan J Watts and Steven H Strogatz. Collective dynamics of ‘small-world’networks. *nature*, 393(6684):440–442, 1998.

[14] Abhishek Verma. Alexnet, an explanation of paper with code, 2020.
https://towardsdatascience.com/alexnet-8b05c5eb88d4.