Network Compression Via Neural Network Memorization

Leonard Tang
Department of Mathematics
Harvard University
Cambridge, MA 02138
leonardtang@college.harvard.edu

Steve Li
School of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138
steveli@college.harvard.edu

Abstract

Neural networks have recently emerged as space-efficient implicit representations of 3D objects and scenes (Sitzmann et al., 2020). Dupont et al. (2021) further extended this notion of implicit neural representation, and *completely* overfit neural networks on 2D images as a form of compression. Here, the network itself is an extremely space-efficient encoding of an input image; such an encoding is obtained by feeding in (x, y) coordinates of an image and training a model to predict the corresponding (r, g, b) value of that pixel. Contrary to machine learning intuition, here overfitting is desired: the trained neural network should exactly *memorize* its given input. Contemporaneously, network sizes are steadily increasing over time, motivating novel compression approaches to reduce I/O times and storage costs. To that end, we propose to apply the implicit neural representation framework to the setting of network compression, thus performing network compression via network memorization. Critically, the approach we consider has the potential to significantly increase compression rates while being relatively simple to encode and decode.

1 Introduction

Lossless graph compression mainly consists of two methods: structural augmentations and succinct graph encoding. The former utilizes graph properties such as cliques, triangles, and degrees to apply reversible transformations, or uses domain-specific features such as text to group similar nodes together. The latter usually means providing encodings that make the nodes more amenable to traditional compression algorithms, such as the Burrows-Wheeler transform on genome data (Baier et al., 2015). More recently, there has been an emergence in neural network approaches to compress graphs. For example, Bouritsas et al. (2021) describe learning a probabilty distrubtion on isomorphic subgraphs, and Ge et al. (2021) propose a graph autoencoder for graph compression and representation learning.

Instead of explicitly defining a compression scheme, machine learning methods have recently been used to "implicitly" represent signal data such as sounds and images, maintaining the signal's spatial and temporal features. Here, we investigate a similar approach to compress networks: we ask if it is possible to encode a graph by overfitting a small MLP on node IDs to edge indicator values. Ultimately, we show that using MLPs with sine activation functions, also known as SIRENs, we are able to overfit graphs of size up to 1000 nodes using only a small number of parameters (< 800 in total). The main benefit of our approach is that it is completely *generalizable* to any graph, requiring no domain-specific knowledge to construct a graph encoding scheme.

We evaluate our model on a wide set of generated networks of different sizes and densities, including Erdos-Renyi and Small-World graphs. These graphs are all stored as adjacency matrices as `.mtx`

Preprint. Under review.

files. Although we do not achieve state-of-the-art compression with respect to the number of nodes and edges, we show that our model sizes are considerably smaller than stored adjacency matrix files on disk, reaching ratios up to 100x less. As graph restructuring is a common technique in network compression, it is likely that combining our approach with modern transformation techniques would lead to more promising results with regard to bits per edge.

2 Related Work

2.1 Domain Specific Approaches

A large portion of graph compression schemes mainly center around compressing graphs using domain-specific properties, such as Web-Graphs, social networks, and biological networks. A common technique amongst all of these domains is vertex relabeling, which augments the initial IDs of vertices to ones that are more easily compressible with modern compression software. For example, Boldi et al. (2009) proposes permutations on the Gray encoding, permuting adjacency matrix rows such that adjacent rows change according to the Gray code.

Other methods use latent properties in the dataset itself. For example, de Brujin graphs are directed graphs that represent overlaps between symbol sequences, and are typically used in *de novo* genome assembly (de Ng Dick Bruijn, 1946). Li et al. (2009) were able to reduce construction and navigation complexity of De Brujin graphs for human genome assembly by omitting read locations and paired-end information. The Burrows-Wheeler transform has also been utilized for a more compact representation of de Brujin graphs for genome assembly Baier et al. (2015).

Since we aim to provide a completely generalizable approach to graph compression, these approaches are less relevant to our work.

2.2 Structural Approaches

Several compression methods perform reversible transformations on the underlying graph structure to achieve a better representation. In particular, the transformations are usually stored using a dictionary or grammar that describes rules on how to reverse the changes.

A common theme throughout most structural methods is that achieving low compression ratios come at a cost of inefficient decoding or querying methods. Buehrer and Chellapilla (2008), for example, compress by replacing bi-cliques with virtual nodes. Their method reaches 1.95 bpe on the UK2002 graph, a webcrawl of the `.uk` domain in 2002 consisting of 18 million nodes and 2989 million edges. Querying access times, however, are not described, and they only use heuristics to find bi-cliques. On the other hand, Asano et al. (2009) use frequent patterns in the adjacency matrix, achieving 1.7 to 2.7 bpe on graphs of at most 19 million edges. Retrieval of these edges, unfortunately, is also not efficient, according to Claude and Navarro (2010).

2.3 Data Representation Approaches

In particular, a large body of work consists of the compression of web graphs, which utilize properties specific to web graphs such as locality and similarity. Some of these techniques involve sorting the node order lexicographically, as described in the original WebGraph framework by Boldi and Vigna (2004), in which they also perform compression of the adjacency list. Alberto and Drovandi (2009) provide a different encoding by first ordering the nodes by BFS-traversal and storing the adjacency list using an entropy-based encoding. Other methods involve performing compression on the adjacency matrix instead of the adjacency list. Brisaboa et al. (2009) use k^2-trees to encode adjacency matrices by splitting the matrix into rectangles. Rectangles that only include 0 values are then represented by a leaf with a value of 0. With this method, they were able to achieve 4.2 bpe for the UK2002 graph, with slightly slower access times than the WebGraph framework.

2.4 RDF Compression

RDF graphs are a fairly recent contribution to data formats, often being used to represent semantic information. In its purest form, an RDF graph consists of a set of triples of subject, predicate, and

object. The predicate, or property, acts as a directed edge from the subject to the object (w3c). All three of these values, therefore, can be represented as strings.

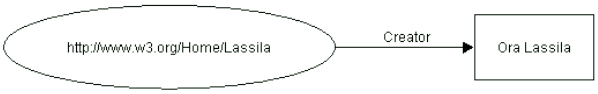


Figure 1: An example a simple RDF graph, taken from (w3c). The oval represents the subject, the edge represents the predicate, and the rectangle represents the object.

The bottleneck lies in storing the string triples, therefore it is common practice to encode the strings and maintain a dictionary mapping the output to its original value. A majority of the work on RDF graphs, therefore, tackle compressing either the dictionary or the underlying graph structure. For example, Jiang et al. (2013) compress the underlying graph structure by combining nodes that have the same neighbors and relations into a single node. On the other hand, Urbani et al. (2013) approach dictionary compression by using a parallelized MapReduce method.

3 Method

Our methods were centered around constructing a small neural network to *entirely* overfit a given network as a means of compression. We refer to the overfitted neural network as the neural encoding of the input network. Once we obtain this neural encoding, we can recover the original network by running inference on all possible pairs of nodes to see if the neural encoding predicts an edge.

3.1 Encoding a Network in an Neural Network

Neural Compression Framework: Consider an arbitrary graph $G = (V, E)$ that we wish to encode. For $u, v \in V$, we have $E(u, v) \in \{0, 1\}$. Naturally, we can model the network compression problem as binary classification of whether or not an edge exists. With N data points, $p_{u,v}$ representing the predicted probability of (u, v) being an edge, and using Cross-Entropy Loss, we have the following optimization problem:

$$\min_{\theta} \left(-\frac{1}{N} \sum_{u,v} \left[E(u,v) \log(p_{u,v}) \ + (1 - E(u,v)) \log(1 - p_{u,v}) \right] \right)$$

The problem can also be formulated using Mean Squared Error Loss by assuming the model output to be continuous across 0 and 1, therefore giving us an additional minimization:

$$\min_{\theta} \frac{1}{N} \sum_{N} (y_i - \hat{y_i})^2$$

Model Architecture: Choosing an optimal model architecture to encode our graph is a critical challenge of this work. Given a sufficiently large number of parameters, it is of course possible overfit our network. However, as we are interested in maximal compression, our goal is to overfit our network by using the smallest number of parameters as possible.

In the case of a simple multi-layer perceptron (MLP), this means experimenting with the number of hidden layers and the number of nodes per layer, both of which are subject to change based off of the graph size. Furthermore, weight quantization from 32-bit to 16-bit precision reduced our model size below our source files. An explanation of our model architectures is described in section 5.

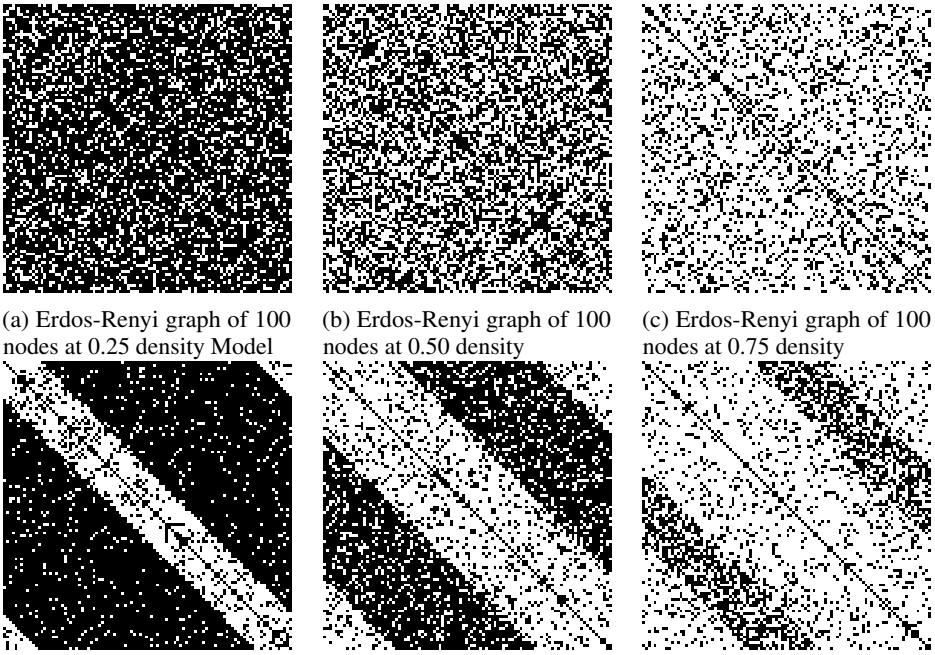
3.2 Decoding a Network from a Neural Network

Decoding the original network from our learned neural network representation simply consists of running all possible pairs of nodes (u, v) through our network and observing the output. The

advantage of this decoding method compared to other modern approaches is that our method is completely *generalizable* to any network, being agnostic to any graph properties.

4 Data

Using the NetworkX library, we evaluated our models on Erdos-Renyi graphs of 25%, 50%, and 75% densities at 100, 500, and 1000 nodes each. However, we noted that Erdos-Renyi graphs are not particularly amenable to compression due to their random generation process (analogous to trying to compress random pixels in the image domain, or random characters in the language domain). Thus, in order to model more realistic network datasets and compression use cases, we generated small-world graphs using the Newmann-Watts–Strogatz model. We implicitly control density by varying the number of neighbors each node contains in its corresponding connected ring topology.



(a) Erdos-Renyi graph of 100 nodes at 0.25 density Model

(b) Erdos-Renyi graph of 100 nodes at 0.50 density

(c) Erdos-Renyi graph of 100 nodes at 0.75 density

(d) Small-World graph of 100 nodes at 0.25 density Model

(e) Small-World graph of 100 nodes at 0.50 density

(f) Small-World graph of 100 nodes at 0.75 density

Figure 2: Image representations of adjacency matrices for Erdos-Renyi and Small-World graphs. Here, a black pixel represents an absence of an edge and a white pixel represents the presence of an edge.

The Newmann-Watts–Strogatz model creates $\frac{NK}{2}$ edges from a parameter mean degree K, usually an even number. We ensured model density by constructing K such that:

$$K = \left\lceil d \cdot (N-1) \right\rceil$$

for a given density d, derived from the fact that:

$$d = \frac{NK}{2} \cdot \frac{1}{\binom{N}{2}}$$

Additionally, we attempted our methods on a Harvard Facebook Group social network consisting of 15-thousand nodes and 824-thousand edges, having a density of approximately 0.007. However, we encountered several memory issues and GPU compute limitations, most of which will be explored in our discussion section.

In terms of infrastructure, we used the Weights and Biases framework to keep track of our experiments and training runs. We implemented our models in PyTorch and perform all experiments on a single RTX2080Ti GPU with 11GB of memory.

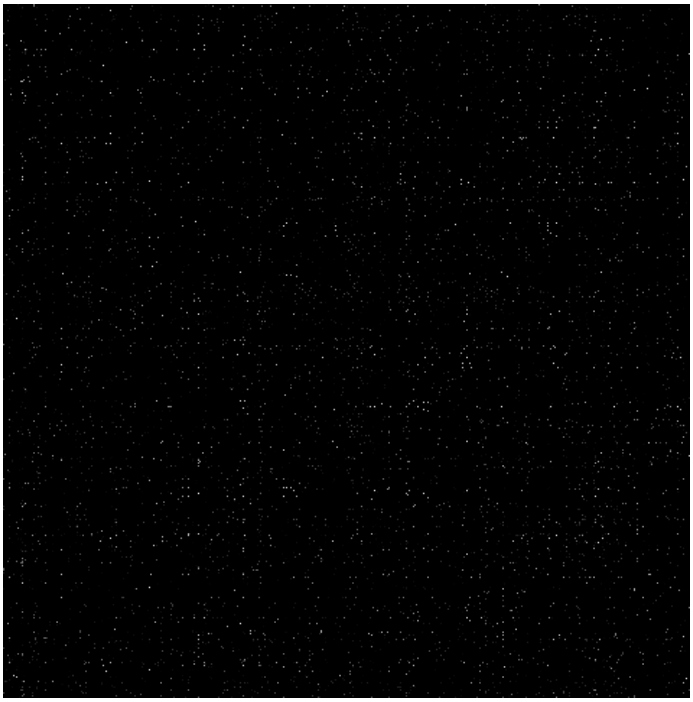


Figure 3: A sample section of a social network graph representing a Harvard Facebook Group with a density of approximately 0.007. Loading this graph proved to be unfeasible given memory and speed requirements for training.

5 Failed Approaches and Learnings

5.1 Support Vector Machines

In the small network regime (graphs with less than 100 nodes), we found that a Support Vector Machine (SVM) classifier with a radial basis function kernel were able to completely overfit the data. However, in their most common implementation, SVMs require storing a $N \times N$ kernel matrix where N is the number of datapoints. Clearly, the kernel size is quadratic with respect to N. Thus, training a SVM on a large dataset incurs a prohibitively large memory requirement; a dataset of 700000 samples, for example, would require approximately 250 GB of RAM to store. Alternatively, if we were to bypass storing this explicit kernel matrix in memory by directly computing the kernel function between pairs of data points, we would experience wasteful and time-consuming training due to redundant computation of this kernel function.

Indeed, in our experiments, increasing N from 100 to 1000 resulted in a practically infeasible training time and memory requirement, even when using parallelization and GPUs.

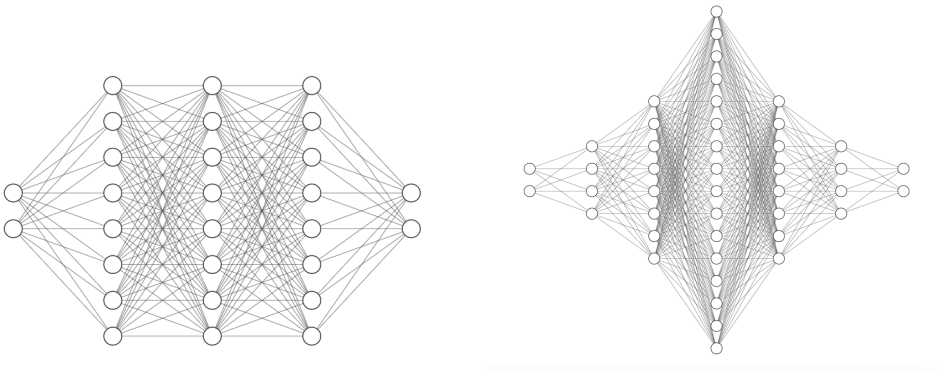
Additionally, choosing the most suitable kernel function for a SVM is a difficult endeavour, particularly if we lack expert knowledge pertaining to a given dataset and some notion of similarity required to separate the data. Moreover, given that our objective is to create a generalizable compression scheme, deciding on an ad-hoc kernel function for a specific instance of a dataset seems inelegant at best, and infeasible at worst.

5.2 XGBoost

The motivation for using gradient boosted trees came from the iterative refinement nature of the underlying algorithm. That is, XGBoost trains models in succession, with each new model being trained to correct errors made by previous ones. Given that our objective is complete memorization of a dataset, we believed that continuously stacking models on top of each other would be sufficient to eliminate all incurred mistakes during training. Indeed, an XGBoost classifier with a maximum

tree depth of 20 and a total of 500 estimators was capable of perfectly overfitting smaller networks of 100 nodes.

Our intuition was correct in the sense that building a model with a large number of trees, each of which are deep, leads to perfect memorization of the network input. However, the required amount of parameters to build a tree with sufficient memorization capacity was far too large to be of practical use. For example, storing the weights of the classifier above actually resulted in data *decompression*, requiring over 40 times the storage space as the input graph. Moreover, when scaling this approach up to larger input networks, XGBoost was incapable of overfitting, regardless of how large our estimator count or maximum tree depth was.



(a) Block Model (b) UnSqueeze Model

Figure 4: Initial neural network architectures that we tested. The Block Model (a) serves as a simple baseline, and the UnSqueeze model (b) aims to decompress inputs using a layer progression that is the opposite of a Variational Autoencoder.

5.3 Initial Neural Network Architectures

We first considered the following neural network structures. The input to each of these models are a pair of node IDs, and the output is a 2-dimensional vector representing binary output probabilities. Our initial architectures used Rectified Linear Units (ReLU) at each hidden layer and were supervised with Cross-Entropy loss.

Block Model The Block model is a simple Mutlilayer Perceptron (MLP) that is composed of identical fully connected layers. In particular, each layer has the same width, hence the "Block" structure of the network.

UnSqueeze Model Inspired by Variational Autoencoders (VAEs), we construct the UnSqueeze model. The number of nodes per layer is increased by a factor of 2 until a maximum throughput threshold is reached, upon which the number of nodes in each layer is decreased by a factor of 2 until the desired number of output classes is reached. This model was designed to decompress our inputs – with the midpoint layer enabling maximum throughput – thus serving the opposite purpose of a VAE, which is to learn a space-efficient latent space by *reducing* input dimension by progressive reduction in layer size.

WideNet We also consider an extremely simple MLP that consists of only a single hidden layer. The width of this hidden layer is exactly equal to the total number of nodes in the graph. This approach was primarily inspired by the recent study of Neural Tangent Kernels (NTK) (Jacot et al., 2018) and universal function approximation capabilities of infinite-width neural networks (Savarese et al., 2019). One may view the WideNet architecture as a limiting instance of the UnSqueeze model, where there is only one hidden layer and the maximum throughput is equal to the number of nodes in the graph.

Figure 1 visualizes the above architectures. Each of these models was trained using the Adam Optimizer with a learning rate of 0.001 at 1000 epochs each. With the Block and UnSqueeze model,

we performed a hyperparameter sweep over the total number of layers as well as the number of nodes per hidden layer. When trained on Erdos-Renyi graphs, we found that the best achieved accuracy of these architectures only matched, at best, the maximum of the graph's density or complement density. For instance, given a graph with 50% edge density, the UnSqueeze model was only capable of achieving 53.25% percent accuracy at the end of training. Table 1 displays results of the UnSqueeze model on Erdos-Renyi grpahs of 100 nodes at various densities. These early results were extremely disheartening, given that similar, and oftentimes better, accuracy could be obtained simply by predicting the majority class.

Density	0.05	0.1	0.25	0.5	0.75
Reconstruction Accuracy	54.30	15.35	44.95	53.25	74.45

Table 1: Reconstruction accuracy (measured in percentages) of our UnSqueeze model. Notice that in many instances the model accuracy does not even surpass a simple majority-class predictor. We do not report the Block model results here, as they are even worse from the outset of training.

Results on Small-World graphs were slightly better, as shown in Table 2. Though these models clearly do not overfit the data, they were capable of at least outperforming a simple majority class predictor.

5.4 Oversampling

Closely observing the data in Table 2, we were particularly interested in the discrepancy in results between the 0.25 and 0.75 density graphs. We had initially hypothesized that our models would be equally capable of predicting the presence of vs. lack of an edge between nodes, though these results would suggest otherwise. Motivated by this, we oversampled our data such that the ratio of 1-labels to 0-labels was 1:1. For example, given a graph with 20% edge density, we would duplicate data points with label 1 until we had an equal number of data points with labels 0 and 1. Using this approach, we obtain very slightly better results in Table 3 using the UnSqueeze model.

Density	**Block Model Accuracy**	**UnSqueeze Model Accuracy**
0.25	93.69	94.37
0.5	77.31	89.46
0.75	83.92	79.82

Table 2: Reconstruction accuracy (measured in percentages) of our Block and UnSqueeze models. The accuracy of these models are perfect, though they are noticeably better than in Table 1.

Density	0.25	0.5	0.75
Reconstruction Accuracy	90.95	88.15	84.25

Table 3: Reconstruction accuracy (measured in percentages) of our UnSqueeze model with oversampling of minority class data. The results are largely comparable to those in Table 2, and certainly not good enough to be considered lossless.

Another architecture we considered required transforming our input data into a one-hot vector encoding of an edge. Concretely, instead of inputting a node ID pair (i, j), we input two vectors of length N with a 1 at position i in the first vector and a 1 at position j in the second vector. These two one-hot vectors were concatenated together to thus form a vector of length $2N$. We used a similar structure, dubbed OneHotNet, to the Block model using GeLU activation functions instead of ReLU and adjusting for larger input data dimension. Ultimately, OneHotNet yielded similarly poor results as our existing models.

To get a better sense of what our models were memorizing and not memorizing, we visualized their predictions by plotting an colored indicator variable for model output during inference. Figure 5 shows an example of the predicted values of the Block model after being fitted to a 100 node Small-World graph with 40% edge density. Similar to this example, in each of our initial experiments

our models were only capable of correctly memorizing the middle "band" of edges, but not any of the edges away from this band. In other words, our models were capable of memorizing edges that together formed a sequential neighborhood in a ring topology (e.g. during the initialization of a Small-World graph), but not more sparse edges situated away from a these neighborhoods.

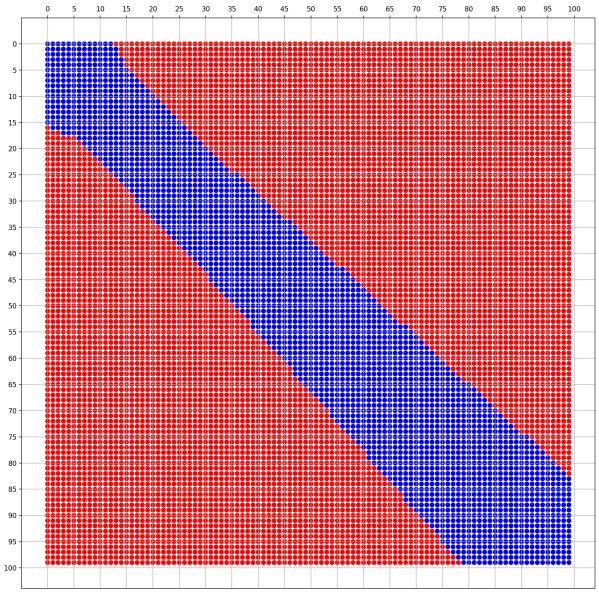


Figure 5: Block model inference plot on a 100 node Small-World graph with 40% edge density. A red pixel indicates that the model predicted the absence of an edge, and a blue pixel indicates that the model predicted the presence of an edge.

5.5 Node Relabeling

Randomizing IDs We sought to eliminate any sort of arbitrary spatial association our models may have made with respect to the numerical ordering of node IDs. Unlike in images, the "coordinates" of a node in a graph – with respect to an adjacency matrix definition – are not particularly informative; in particular, they do not exhibit any sort of local correlation as is the case with image coordinates. To that end, we randomly permuted the node labels prior to training.

Graph Traversals Taking the above approach one step further, we experimented with graph traversals in an attempt to manifest label locality within node neighborhoods. We relabeled nodes in increasing order based on the visited nodes in BFS as well as DFS. We hoped that these traversal-based node orderings would lend themselves to be more spatially correlated. The intuition for this is that the traversal process would at least force adjacently numbered nodes to have an edge.

Strongly Connected Components We also attempted to leverage strongly connected components. In particular, we relabeled our node IDs based on their location in strongly connected components of the graph. For example, for some gap number L, we labeled all k_1 nodes in the first strongly connected component from 1 to k_1, all k_2 nodes in the second strongly connected component from $k_1 + L$ to $k_1 + L + k_2 - 1$, and so on.

Ordering by Degree In a similar vein, after sorting nodes by their degree, we relabeled nodes sequentially at each degree value, spacing the node IDs by a fixed gap as above in an attempt to separate the individual clusters. For example, for a gap number L and a graph with 2 nodes of degree

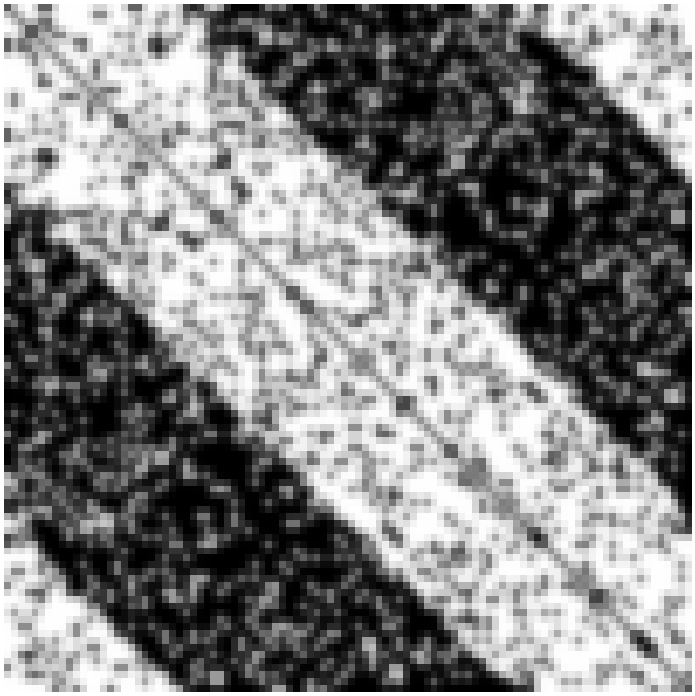


Figure 6: Example of a 50% dense Small-World graph viewed as a blurred image arising naturally from the adjacency matrix representation. Here, a black pixel represents an absence of an edge and a white pixel represents the presence of an edge. Gray pixels arise from Gaussian blur applied on top of the image.

4, 3 nodes of degree 5, and 1 node of degree 6, we would relabel our nodes as $1, 2, 2+L, 3+L, 4+L$, and $4+2L$.

However, our relabeling methods did not ultimately improve our results, only managing to match the reconstruction accuracies of approaches that did not use relabeling.

5.6 Cross-Entropy Supervision

Examining the outputs of our model during training revealed that Cross Entropy Loss pushed our models to predict the majority-class label very early on in training. Therefore, we sought to explore other loss functions, as detailed in Section 6.

5.7 Documenting Experiments

In spite of these failures, we thoroughly documented our experiments using Weights and Biases (Biewald, 2020). This enabled us to look back on previous experiments to reason carefully about what combinations of architectures, training parameters, training procedures, node labellings, and dataset combinations had worked best. This turned out to be very fruitful, allowing us to ultimately discover a more promising approach.

6 Towards Successful Compression

Though we encountered multiple failed approaches and underwhelming results, we eventually formulated a lossless compression approach that is reasonably successful for graphs of small sizes. We hope that these early results pave the way for future research and work in this direction.

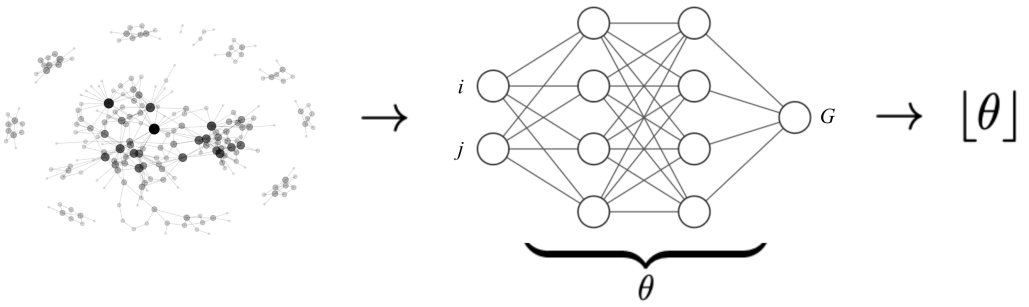


Figure 7: Encoding a network within a neural network. We overfit a network with a neural network mapping pairs of nodes (i, j) to a *continuous* grayscale value between 0 and 1 representing the existence of an edge, under the graph-image interpretation. Our neural network is supervised by MSE. The learned network weights are then quantized to a lower bit-width and transmitted.

6.1 Interpreting Graphs as Images

The key framework shift that allowed our approach to come to fruition was to interpret graphs not as pairs of nodes matched with edge labels, but rather as an image naturally arising from the graph's adjacency matrix. We reasoned that the local neighborhoods of node pairs within a graph lacked spatial correlation, a key distinction from local neighborhoods of coordinates in the image domain. For example, in the graph setting, the existence of an edge between node k and node $k + 1$ is largely independent of an edge's existence between node $k - 1$ and $k + 1$. However, one might easily imagine an image, say a landscape picture containing a vast blue sky, where the pixel at coordinate (x, y) being blue strongly increases the likelihood of the pixel at coordinate $(x + 1, y)$ being blue. Analogous to the image compression setting, we refer to node-pairs (k_1, k_2) as *coordinates*.

Motivated by this observation, we set out to manifest local spatial correlations in our graphs by interpreting it *as an image* and then blurring it. In particular, a blur allowed for smooth transitions of pixel values within any neighborhood of the image, building in much stronger spatial correlation than sharp cutoffs between 0's and 1's. For example, applying a Gaussian blur with $\sigma = 0.7$, we produce the image in Figure 6. Given the newly continuous nature of the output pixels within the blurred image, we supervised our models with Mean Squared Error (MSE) loss instead of Cross-Entropy. Indeed, on this blurred image, our UnSqueeze model was able to faithfully reconstruct the image with a MSE of 0.0004.

However, we soon realized that memorizing a blurred version of an input graph image was largely useless, given that there was no straightforward approach to map a blurred pixel back to its original binary value. One might imagine defining a quantization map based on some threshold pixel value, but this reasoning quickly fails, in particular because a blurred pixel value is sensitive to how many of its surrounding neighbors original values are 0/1. Since these neighborhood properties are highly variant within an image, there is no universal threshold that could easily discriminate a blurred pixel from being 0 or 1. This issue arises in particular because Gaussian blur is not an invertible transformation. One could try and learn an invertible blur transformation, but that itself is a completely separate research problem and endeavour.

6.2 Exact Graph-Image Reconstruction with SIRENs

Instead of searching for an invertible blur function, we first wanted to test if it was possible to exactly memorize the image-graph without blurring at all. We sought inspiration from Sitzmann et al. (2020) and Dupont et al. (2021), which uses MLPs endowed with periodic activation functions to capture continuous, differential signal representations, such as images, with fine-grained detail. To that end,

we set up a sinusoidal representation network (SIREN) with 10 layers of 28 nodes each to directly reconstruct graph-images without blurring.

In particular, we define our model Φ as:

$$\Phi(\mathbf{x}) = \mathbf{W}_n(\phi_{n-1} \circ \phi_{n-2} \circ \cdots \circ \phi_0)(\mathbf{x}) + \mathbf{b}_n,$$

$$\mathbf{x}_i \mapsto \phi_i(\mathbf{x}_i) = \sin(\mathbf{W}_i\mathbf{x}_i + \mathbf{b}_i)$$

Here, $\phi_i : \mathbb{R}^{M_i} \to \mathbb{R}^{N_i}$ is the i^{th} layer of the network. $\mathbf{W}_i \in \mathbb{R}^{N_i \times M_i}$ and $\mathbf{b}_i \in \mathbb{R}^{N_i}$ canonically represent the relevant weight and biases, which are passed through sine nonlinearities. This formulation allows for much faster memorization of an input image when compared to traditional nonlinearities such as ReLU, Tanh, Sigmoid, GeLU, and so on.

Given that the purpose of a SIREN is to represent smooth signals, we choose to supervise our models with MSE instead of Cross-Entropy, despite our ground truth pixel values only being 0 or 1. Under this framework, we are able to losslessly compress graphs containing 100, 500, and 1000 nodes. Please refer to Table 4 in Section 7 for final compression results.

Despite being especially capable of representing data with high-frequency signals such as images, SIRENs are notoriously difficult to train. As such, we resorted to brute-force architecture search to find optimal models for each type of network data we encounter. Figure 7 provides an overview of our full encoding procedure.

6.3 Coordinate Normalization

A critical procedure we perform prior to model training is the coordinate normalization. That is, we normalize every node value in a coordinate pair to be within -1 and 1. This simple step brought our approach to lossless compression within a reasonable training time. We hypothesize that this normalization may be particularly useful for the SIREN model given the narrow domain of the sine activation function.

One practical issue with normalization is the fact that as the number of nodes in a network increases, there will be redundant scalings of nodes at a finite precision. That is, for a large enough network, two adjacently numbered nodes may map to the same floating point number under a -1 to 1 scaling. For our experiments, this is not an issue, but we highlight this as a potential future concern. As a starting point, we've shown that scaling coordinates to -5 to 5 or -10 to 10 is also beneficial for learning.

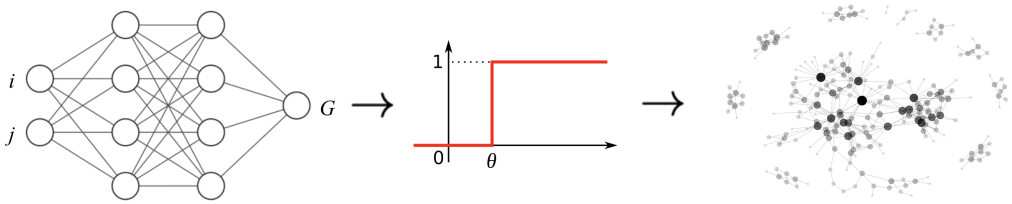


Figure 8: Decoding a network from a neural network. Given a learned neural network representation of the network data, we perform inference on the coordinates to obtain a scalar value that is thresholded by θ, the mean of the prediction values, to be either 0 or 1. After this thresholding, we recover the original network.

6.4 Aggressive Weight Quantization

One unique benefit of our task setup and training paradigm of reconstructing pixel values is that our models' predicted regression values need not be completely perfect. During decoding, we pass the graph-image coordinates through our learned model to obtain initial predictions and post-process them to either 0 or 1. In particular, we round the initial output based on whether it is above (map to 1)

or below (map to 0) the mean predicted value. Given this cleanup procedure, our models can sustain a healthy amount of error before causing noticeable issues to the ultimate post-processed output.

This allows us to take advantage of weight quantization post-training. Even after quantizing model weights from 32-bit to 16-bit precision, regression predictions are not altered significantly enough to change the post-processed output. Please refer to Table 4 in Section 7 for a full comparison of network sizes before and after quantization. Given the time constraints, we were unable to explore quantization-aware training methods, but that would certainly be of interest for future work.

6.5 Minimizing Training Time for Practical Encoding

Given that our models needed to completely overfit the data, training often required an extremely large number of epochs. For reference, Sitzmann et al. (2020) and Dupont et al. (2021) specify 50,000 training epochs as the default for their models. To enable any practical use case of our method, then, we sought out to minimize the time necessary to memorize any given graph.

While Sitzmann et al. (2020) and Dupont et al. (2021) are able to train their models via full-batch gradient descent by directly passing tensors stored in memory to their GPU, our network data is significantly larger and thus infeasible to apply full-batch gradient descent on. Instead, we needed to write batched data loaders to accommodate our limited GPU hardware. To that end, we noticed that the typical PyTorch DataSet and DataLoader classes were extremely slow and impractical for our purposes, in particular since we needed to train our models for thousands of epochs to convincingly memorize our data. Using various system utilization tools and profilers, we realized that our training time was heavily limited not by GPU computations, but rather by data loading.

The DataLoader Problem To alleviate this bottleneck, we first attempted to parallelize our data loading and increase the batch size. This required automatically and carefully finding an optimal combination of number of workers and batch size. However, even with this optimal search implemented, our data loading was still the dominating friction in training time. In a similar vein, we implemented a parallelized streaming DataLoader due to the large size of our datasets. Unfortunately, this also led to no noticeable speed up in data loading times.

We wondered if storing all the data in memory was causing the critical slowdown, and as such, we implemented a disk-reading DataLoader leveraging the HDF5 binary data format, known to be amenable for large-scale numerical data. However, this ultimately increased data loading times even further, due to the large number of individual disk reads the DataLoader needed to perform for each item.

Chunking Tensors in Memory Ultimately, we realized that the main bottleneck arising from any DataLoader implementation was the requirement that each data point (i.e. each pair of coordinates and labels) be loaded individually within the `__getitem__` function of our DataSets. In the DataLoader framework, we needed to repeatedly access a N-long coordinates list and labels list before constructing a data tensor. Noticing that the coordinates and labels list could already be stored in memory directly, we abandoned the DataLoader approach altogether and decided to first send the lists to the GPU, and interact with and manipulate the resulting tensors directly.

To maximize GPU usage and training throughput, we automatically calculate the maximum size K of a tensor that could fit onto our hardware, given our model definition. We then split our original data tensors of length N into $\lceil N/K \rceil$ equal sized tensors, barring the last tensor with size at most K. Conceptually, this is the exact same as implementing a DataLoader with batch size K. However, since we now operate on continuous chunks of tensors instead of building batches by accessing data points individually, our data loading time is significantly reduced, allowing for a more reasonable training time.

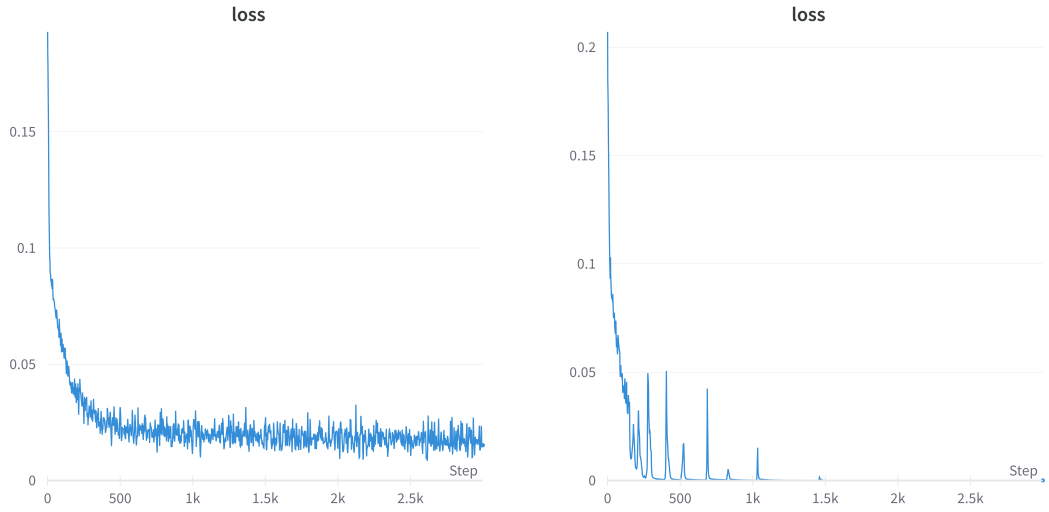
Two properties of our training task enable us to take advantage of tensor chunking. First, given that we are not aiming for any generalization capabilities whatsoever, we need not randomize the order of our data; that is, we do not need to spend extra time shuffling our data prior to training and thus are able to train immediately after chunking our tensors. Second, we benefit from the fact that our data is relatively small – we only need to deal with lists of 2-tuples (coordinates) and lists of integers (labels). Had we worked with image data, text, or any other more common and larger form of data, tensor chunking may not have been as practical as the traditional disk-read DataLoader approach.

Ultimately, while we are unable to pass our network data in entirety to our model, we are able to closely match training speed by splitting the dataset to large tensor chunks in memory.

Learning Rate Scheduling and Early Stopping After close observation of loss curves revealed inconsistent and oftentimes non-monotonic learning behavior during training, we were motivated to experiment with different learning rate schedulers, including cyclic learning rates (Smith, 2015) and other heuristics. In practice, we find that initializing learning rate to 2×10^{-4} and then multiplying by 0.75 after every 250 epochs yields much faster convergence compared to other schedulers and training without a scheduler. This heuristic arises from the intuition that our optimizer was making too large of updates towards the middle and end of training, and instead required more careful fine-tuning within the deeper valleys of the loss landscape. In spirit, this is very similar to the well-known Reduce Learning Rate On Plateau function in PyTorch, though more specific in implementation to accommodate the extremely inconsistent loss dynamics particular to the SIREN family of models. Figure 9 demonstrates the benefit of using our particular $\times 0.75$ scheduler.

To further reduce training time, we observe that an average training loss of less than 0.01 yields a sufficiently accurate model (refer to Section 6.4 for discussion on why we need not have a perfect loss of 0). In our implementation, we save every network that achieves a running minimum training loss, and break out of our training loop once the loss has broken the 0.01 barrier.

In aggregate, these techniques reduced the overfitting time for a 1000-node network from over 7 hours to less than 5 minutes. While this speedup allowed for much quicker experimentation on our end, it also yields a much more practical encoding algorithm.



(a) Training loss without learning rate scheduler

(b) Training loss with learning rate scheduler

Figure 9: Loss curves from model overfitting with and without using the $\times 0.75$ learning rate scheduler. In (a), the loss is extremely inconsistent and stochastic, never reaching 0. Meanwhile in (b), the loss reaches 0 in less than 500 epochs, due to the learning rate scheduler.

7 Results

Below, we report the key compression results of our SIREN-based model outlined in Section 6. We evaluate our compression method on Small-World graphs of size 100, 500, and 1000, each constructed with edge swapping probability $p = 0.5$ and no augmentation or relabelling. To encode 100-node graphs, an architecture search yielded a SIREN with 10 hidden layers with 28 nodes each. For 500-node graphs, we use a SIREN with 14 hidden layers with 84 nodes each. For 1000-node graphs, we use a SIREN with 18 hidden layers with 168 nodes each. Critically, each architecture entirely overfits the input network, thus performing lossless compression. Table 4 displays the compression ratios resulting from these architectures on each graph size. Looking at the trends in the **Compression**

Ratio column, we are optimistic that larger networks can experience even better compression gains using our method.

Graph Size	File Size (.mtx)	FP Model Size	HP Model Size	Compression Ratio
100	116.20 kB	29.7 kB	14.8 kB	7.85
500	2880.80 kB	372.6 kB	186.3 kB	15.46
1000	11,511.56 kB	1933.3 kB	966.7 kB	119.08

Table 4: Sizes of neural network weights that have fully memorized the input graph of given size across all densities. Both full-precision (FP) and half-precision (HP) model sizes are reported, alongside the compression ratio calculated with respect to .mtx file size and half-precision model size.

We also report the overfitting loss, bits-per-node (bpn), and bits-per-edge (bpe) for our half-precision models on varying densities of each graph size in Table 5.

Graph Size	Edge Density	Overfitting Loss	HP BPN	HP BPE
100	0.25	0.00060	1187.36	47.49
	0.5	0.00060	1187.36	23.75
	0.75	0.00016	1187.36	15.83
500	0.25	0.0003	2981.02	23.84
	0.5	0.00565	2981.02	11.92
	0.75	0.0073	2981.02	7.95
1000	0.25	0.00178	7733.39	30.93
	0.5	0.00347	7733.39	15.47
	0.75	0.00195	7733.39	10.31

Table 5: Overfitting loss, bits-per-node, and bits-per-edge results for half-precision SIREN models on graph sizes of 100, 500, and 1000 at 0.25, 0.5, and 0.75 edge density.

Note that since our architecture is fixed for any particular network size, our model size and corresponding bits-per-node is constant as well. This means that the denser the graph, the lower the bits-per-edge will be. As such, our compression method seems particularly well-suited for compressing dense graphs, and correspondingly less so for sparse graphs. This is clearly reflected within the **HP BPE** column of Table 5. Notice also that our overfitting loss is not perfectly 0. As mentioned in Section 6.4, our post-inference thresholding procedure allows for some amount of error in the direct output of our models.

Finally, we present our best bits-per-edge results alongside commonly used compression schemes in Table 6. We report the best BPE from each method as well as the best BPE of our method. While clearly not a perfect comparison given the vastly different nature of our graphs from a size, density, and topological perspective, we hope this comparison nontheless contextualizes our approach amongst more traditional encoding algorithms. Though our method is not state-of-the-art, we are certainly in reasonable striking distance.

Ours	WebGraph	BFS	Maserrat	RePair	Claude & Ladra	Deo & Litow
7.95	2.6	1.83	13.9	4.23	2.27	9.9

Table 6: Best bits-per-edge performance of our network memorization approach compared to the best bits-per-edge performance of traditional and commonly-used entropy-based encoding schemes.

8 Discussion and Conclusion

The question of how to compress large networks is at the core of many problems across science and engineering. Neural network memorization may provide a new tool for many of these by offering a generalized, domain-agnostic approach to network compression. Though many off-the-shelf machine learning methods fail at the network memorization task, we ultimately demonstrate that relatively small MLPs with sine activation functions are capable of exactly reconstructing graphs of somewhat nontrivial size. Our approach does not require access to unreasonable compute resources (needing only 1 modest GPU to run), is not prohibitively slow at encoding (only requiring a few minutes), and is extremely fast at decoding (finishing within seconds). Moreover, our results indicate favorable compression rates both with respect to raw input size, as well as existing entropy-based compression schemes. Trends in our compression results potentially suggest even more favorable compression rates at larger scales. While we realize our evaluations were carried out on relatively small networks, we are optimistic that our approach can scale to larger inputs.

9 Future Work

Compression Rate Lower Bound for Arbitrarily Large Networks In the worst case, our results indicate that we have already found a way to compress any arbitrary network with a 119.08 compression ratio. That is, given any network larger than 1000 nodes, we are able to split the graph-image into, say, k 1000×1000 sub-images. Applying our network memorization approach on each of the k sub-images and storing the resulting network weights, we would be able to achieve the same compression rate as in the 1000-node network case.

Scaling Up our Models To beat the lower bound detailed above, we would need to devise a method for scaling up our SIREN models while preserving overfitting capacity and stability. Though this seems feasible, as we have shown scaling up from graphs with 100 nodes to graphs with 1000 nodes, it is possible that the sine activation function may behave differently with respect to significantly deeper and wider networks. We are certainly interested in studying the phase transitions of SIREN performance as dataset size increases.

Towards More Elegant Memorization Though our method is performant in the most absolute sense, it seems like an inelegant way to approach the network memorization problem. That is, it seems unnatural to think of a graph memorization from the perspective of memorizing an image of continuous grayscale values, instead of a coordinate-wise classification problem. Thus, we are interested in revisiting our binary classification memorization paradigm. Inspired by a Zoom call with Dupont et al. (2021), we are eager to cast our discrete classification problem into a signal representation problem that is more amenable to Cross-Entropy supervision. In particular, we observed through our experiments that Cross-Entropy supervision often resulted in overly aggressive parameter updates, forcing predictions to be majority 1 or 0 for continuous chunks of input coordinates very early on in training. To mitigate this phenomenon, we are interested in representing our coordinates with (somewhat) continuous node embeddings. At the same time, we also hope to derive first-principle methods arising from the Graph Fourier Transform.

References

Resource description framework (rdf) model and syntax specification. URL https://www.w3.org/TR/PR-rdf-syntax/Overview.html.

Apostolico Alberto and Guido Drovandi. Graph compression by bfs. *Algorithms*, 2, 09 2009. doi: 10.3390/a2031031.

Yasuhito Asano, Yuya Miyawaki, and Takao Nishizeki. Efficient compression of web graphs. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, 92-A: 2454–2462, 10 2009.

Uwe Baier, Timo Beller, and Enno Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the Burrows–Wheeler transform. *Bioinformatics*, 32(4):497–504, 10 2015. ISSN 1367-4803. doi: 10.1093/bioinformatics/btv603. URL https://doi.org/10.1093/bioinformatics/btv603.

Lukas Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, page 595–602, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 158113844X. doi: 10.1145/988672.988752. URL https://doi.org/10.1145/988672.988752.

Paolo Boldi, Massimo Santini, and Sebastiano Vigna. Permuting web and social graphs. *Internet Mathematics*, 6:257–283, 01 2009. doi: 10.1080/15427951.2009.10390641.

Giorgos Bouritsas, Andreas Loukas, Nikolaos Karalias, and Michael M. Bronstein. Partition and code: learning how to compress graphs, 2021. URL https://arxiv.org/abs/2107.01952.

Nieves Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-trees for compact web graph representation. pages 18–30, 08 2009. ISBN 978-3-642-03783-2. doi: 10.1007/978-3-642-03784-9_3.

Gregory Buehrer and Kumar Chellapilla. A scalable pattern mining approach to web graph compression with communities. New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939272. doi: 10.1145/1341531.1341547. URL https://doi.org/10.1145/1341531.1341547.

Francisco Claude and Gonzalo Navarro. Fast and compact web graph representations. *ACM Trans. Web*, 4(4), sep 2010. ISSN 1559-1131. doi: 10.1145/1841909.1841913. URL https://doi.org/10.1145/1841909.1841913.

de Ng Dick Bruijn. A combinatorial problem. 1946.

Emilien Dupont, Adam Goliński, Milad Alizadeh, Yee Whye Teh, and Arnaud Doucet. Coin: Compression with implicit neural representations. In *Neural Compression Workshop @ ICLR 2021*, 2021.

Yunhao Ge, Yunkui Pang, Linwei Li, and Laurent Itti. Graph autoencoder for graph compression and representation learning. In *Neural Compression: From Information Theory to Applications – Workshop @ ICLR 2021*, 2021. URL https://openreview.net/forum?id=Bo2LZfaVHNi.

Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. 2018. doi: 10.48550/ARXIV.1806.07572. URL https://arxiv.org/abs/1806.07572.

Xiaowei Jiang, Xiang Zhang, Feifei Gao, Chunan Pu, and Peng Wang. Graph compression strategies for instance-focused semantic mining. In Guilin Qi, Jie Tang, Jianfeng Du, Jeff Z. Pan, and Yong Yu, editors, *Linked Data and Knowledge Graph*, pages 50–61, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-54025-7.

Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, S.Z. Li, Huanming Yang, Jian Wang, and Jun Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20:265–72, 12 2009. doi: 10.1101/gr.097261.109.

Pedro Savarese, Itay Evron, Daniel Soudry, and Nathan Srebro. How do infinite width bounded norm networks look in function space?, 2019. URL https://arxiv.org/abs/1902.05040.

Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *ArXiv*, abs/2006.09661, 2020.

Leslie N. Smith. Cyclical learning rates for training neural networks, 2015. URL `https://arxiv.org/abs/1506.01186`.

Jacopo Urbani, Jason Maassen, Niels Drost, Frank Seinstra, and Henri Bal. Scalable rdf data compression with mapreduce. *Concurrency and Computation: Practice and Experience*, 25(1): 24–39, 2013. doi: https://doi.org/10.1002/cpe.2840. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.2840`.