Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Learning Graphons Using Neural Networks

Bc. Huy Hoang Vu

Supervisor: Ing. Ondřej Kuželka, Ph.D.
Field of study: Open Informatics
Subfield: Artifical Intelligence
August 2020

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Vu Huy Hoang**	Personal ID number:	**457850**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence**		

II. Master’s thesis details

Master’s thesis title in English:

Learning graphons using neural networks

Master’s thesis title in Czech:

Učení grafonů pomocí neuronových sítí

Guidelines:

The goal of this thesis is to explore if one can efficiently learn grahpons [1], which are representations of limits of dense graphs, using deep neural networks.
1. Get familiar with the basics of the theory of graphons [1].
2. Implement functions that will allow you to represent graphons (a graphon is just a symmetric function on the square [0;1]^2) using neural networks, to compute homomorphism densities of small graphs and to sample graphs from the graphon (a graphon is at the same time a probabilistic model from which one can sample).
3. Implement functions for estimating homomorphism densities from graphs (this will be used for training).
4. With the help of the supervisor, design and implement a method for learning graphons represented as neural networks by minimizing the discrepancy of homomorphism densities of small graphs in the training data and in the graphon.
5. (Optional) Design and implement an approximate method for learning graphons from data using maximum-likelihood estimation.

Bibliography / sources:

[1] Lovász, L. (2012). Large networks and graph limits (Vol. 60). American Mathematical Soc..
[2] Eldridge, J., Belkin, M., &amp; Wang, Y. (2016). Graphons, mergeons, and so on!. In Advances in Neural Information Processing Systems (pp. 2307-2315).
[3] Airoldi, E. M., Costa, T. B., &amp; Chan, S. H. (2013). Stochastic blockmodel approximation of a graphon: Theory and consistent estimation. In Advances in Neural Information Processing Systems (pp. 692-700).
[4] Goodfellow, I., Bengio, Y., &amp; Courville, A. (2016). Deep learning. MIT press.

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Name and workplace of master’s thesis supervisor:

Ing. Ondřej Kuželka, Ph.D., Intelligent Data Analysis, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **04.02.2020** Deadline for master's thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

Ing. Ondřej Kuželka, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt	Student’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I'd like to thank my supervisor for introducing me to this interesting area of research and for taking the time and guiding me through the experiments. I'd also like to thank my friends and family for their support. Thanks to Michal Cvach for being a constructive critic and a supportive friend in difficult times of rigorous studies at FEE CTU.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

Prague, 14. August 2020

Abstract

In this thesis, we explore the idea of modeling graphons with neural networks. Graphons are functions representing the structure of a large graph, and thus we try to approximate them with neural networks. To that end, we developed a gradient-based learning algorithm which we test on synthetic data. Lastly, we analyze the convergence of learning processes of our algorithm and the resulting graphons they produce.

Keywords: graphon, neural network, large, graph, network

Supervisor: Ing. Ondřej Kuželka, Ph.D.

Abstrakt

V této práci se zabýváme reprezentací grafonů pomocí neuronových sítí. Grafony jsou funkce zachycující strukturu velkých grafů. Neuronové sítě jsou v některých případech dobrými aproximacemi funkcí, proto se pokoušíme je aplikovat na aproximaci grafonů. Za tímto cílem jsme v rámci práce vyvinuli učící algoritmus založený na spádových metodách, které následně testujeme na uměle vytvořených datech. Nakonec analyzujeme konvergenci našeho algoritmu a grafony, které algoritmus produkuje.

Klíčová slova: grafon, neuronová síť, velký, graf, síť

Překlad názvu: Učení grafonů pomocí neuronových sítí

Contents

Figures

Tables

Chapter 1

Introduction

Network science is a field of study that analyzes networks such as social networks, traffic networks, biological networks, and computer networks to name a few. To study these networks, network science is in some sense a combination of multiple fields such as mathematics (graph theory), computer science (graph algorithms), data mining/statistics (inferential modeling), sociology (social structure), biology (biological systems).

The underlying theory for all network sciences is graph theory, in which mathematical advancements have been made recently in the form of developed theories namely graphons defined in [Lov12]. Graphons are mathematical objects representing the structure of enormous graphs, for example an atomic grid of a crystal. Modeling large graphs is challenging due to their sheer size. Sometimes we might not want to model on a vertex or edge level but rather we would like to capture higher-level yet intricate structural patterns. To that end, using graphons is an interesting direction to take.

Mathematically, graphons are just functions. In recent years, neural networks have proven to be powerful function approximators in some use cases so it is natural for us to explore if we can represent graphons with them. This is the aim of this thesis - to model graphons using neural networks.

Chapter 2
Graph(on) Theory

This chapter describes graph theory and graphon theory that is needed in order to get a grasp of graphons - the mathematical object representing the structure of a large graph. Here, we will also define homomorphism densities that are the backbone quantities with which we will train our neural network to represent a graphon. We suggest the reader to have a look at Lovász's book [Lov12], from which the theory in this chapter was drawn, for a more in-depth treatment of the study of graphons.

2.1 Graph Theory

This section does not introduce any complicated efficient algorithms on graphs or state theorems about them but rather defines some basic terms and graph properties so that the reader understands them when they appear in later sections.

2.1.1 Basic Terminology

Definition 2.1.1. A *simple graph* G of size n is a pair (V, E), where V is a finite set of n vertices (also nodes) and $E \subseteq \{\{a, b\} | a \in V, b \in V, a \neq b\}$ is a set of edges (also links).

We will use V_X and E_X to denote the set of vertices and the set of edges of a graph X respectively. Unless stated otherwise, we will assume that the vertices of a graph X are labeled by integers $\{1, 2, \dots, |V_X|\}$.

Notice that in the definition of edges, an edge is a two-element set of vertices which are unordered and have no direction. We call this graph an *undirected graph.*

Since two-element sets with the same elements are treated the same, multiple edges between the same two nodes are considered one and the same edge.

In this definition of a simple graph, a loop (an edge on the same vertex) is not allowed. However in our application and usage of graphs, loops do not really matter.

We will refer to simple graphs as finite graphs or just as graphs and primarily only work with them unless stated otherwise from now on.

Definition 2.1.2. An *induced subgraph* $H = (V_H, E_H)$ of a graph $G = (V_G, E_G)$ is a graph such that $V_H \subseteq V_G$ and $E_H = \{\{a,b\} | a \in V_H, b \in V_H, \{a,b\} \in E_G\}$.

Definition 2.1.3. A *complete graph* $K = (V_K, E_K)$ is a graph such that $E_K = \{\{a,b\} | a \in V_K, b \in V_K, a \neq b\}$, i.e. it contains all possible edges.

Definition 2.1.4. An *independent set* $I = (V_I, E_I)$ is a graph such that $E_I = \emptyset$.

Definition 2.1.5. An *edge weighted graph* $G = (V_G, E_G, w_G)$ has an additional weight function $w_G : E_G \to \mathbb{R}$ that maps a real value to each edge.

Definition 2.1.6. In a *directed graph* $G = (V_G, E_G)$ the edges are ordered pairs of vertices $E_G \subseteq \{(a,b) | a \in V_G, b \in V_G\}$ thus the edges have a direction.

Definition 2.1.7. A *clique* C of a graph G is its induced subgraph such that C is a complete graph.

Definition 2.1.8. A *degree* of a vertex v denoted by $\deg(v)$ in a simple graph G is the number of its direct neighbors, or more formally $\deg(v) = |\{e | v \in e, e \in E_G\}|$.

2.1.2 Properties of a Graph

Definition 2.1.9. A graph $H = (V_H, E_H)$ is said to be *isomorphic* with a graph $G = (V_G, E_G)$, if there exists a bijective mapping $i : V_H \to V_G$ such that $(\forall a, b \in V_H)(\{a,b\} \in E_H \iff \{i(a), i(b)\} \in E_G)$. We call such a mapping an *isomorphism*.

In other words, the graphs are the same up to a relabeling of the vertices. A non-edge has to map to a non-edge and an edge has to map to an edge.

Note that because i is a bijection, the isomorphism relation between graphs is symmetric.

As seen in figure 2.1 below, graphs H and G are the same up to a relabeling of vertices. An example of an isomorphism for those graphs is

$$i(v) = \begin{cases} 1, & v = A \\ 2, & v = B \\ 3, & v = C \\ 4, & v = D \\ 5, & v = E \end{cases}$$

Definition 2.1.10. A graph H is *homomorphic* with a graph G, if there exists a mapping $h : V_H \to V_G$ such that $(\forall a, b \in V_H)(\{a,b\} \in E_H \implies \{h(a), h(b)\} \in E_G)$. We call such a mapping a *homomorphism*.

Let us denote $H \longrightarrow G$ if a homomorphism of H into G exists.

- In contrast to an isomoprhism, a homomorphism can map a non-edge to an edge, a non-edge, or even a single vertex. An edge must still map to an edge though.

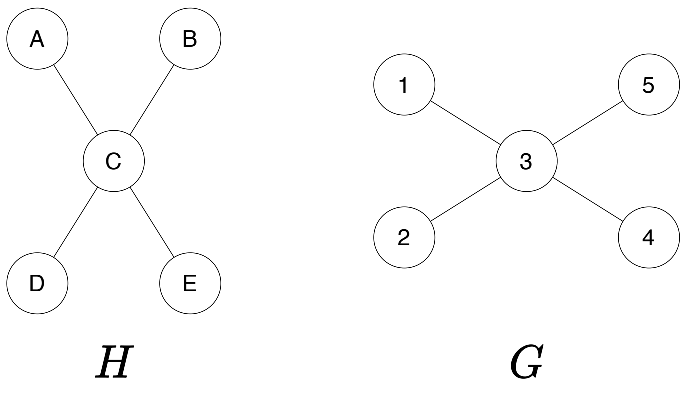


Figure 2.1: An example of isomorphic graphs

- An isomorphism is edge and non-edge preserving, while a homomorphism only preserves edges.
- Homoprhisms between graphs are not necessarily symmetric relations.

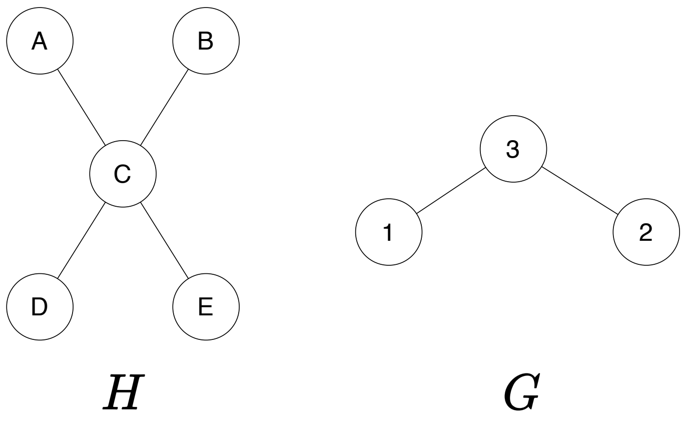


Figure 2.2: An example of H being homomorphic with G

An example of a homomorphism of H into G for the example 2.2 above would be

$$h(v) = \begin{cases} 1, & v = A \\ 1, & v = B \\ 3, & v = C \\ 2, & v = D \\ 2, & v = E \end{cases} \tag{2.1}$$

The existence of a homomorphism can encode surprisingly non-trivial information. For example, $K_n \longrightarrow G$ means that G contains a clique of size n because K_n doesn't have non-edges thus the mapping is definitely injective and since edges have to map to edges then there must be an edge between every pair of the image nodes in G. $G \longrightarrow K_n$ means that G is n-colorable because if two vertices in H are adjacent, then they must map to two different vertices in G, so we can use the vertex images in G as colors of vertices in H.

Homomorphisms are important because we will build on top of them by defining homomorphism densities in section 2.2.

Definition 2.1.11. We call a sequence of graphs G_n (growing with the number of vertices n) *dense*, if the number of edges grows asymptotically quadratically with the number of vertices $\mathcal{O}(n^2)$, or informally that the number of edges stays proportionate to the maximum number of all possible edges. A sequence of graphs that is not dense is called *sparse*.

From now on, we will only work with dense graphs unless stated otherwise. The motivation behind this is that we model large graphs by sampling induced subgraphs from them and if the graphs were sparse, then we would almost surely get an independent set. Many real world networks are sparse graphs, for example social networks. As of today (the time of writing this thesis), modeling sparse graphs is still an active research topic and the theory is not yet as well developed as the theory for dense graphs. However some real world networks are dense, for example protein-protein interaction graphs.

2.2 Graphon Theory

This section provides the definition of graphons, the object we want to model, as well as homomorphism densities, the quantities that in fact differentiate graphons from each other.

The main idea of using graphons is that when we sample induced subgraphs from a large network that we want to model, the structure of the subgraphs becomes increasingly similar to the structure of the large network itself as we increase the size of the sample. The structure then converges in some sense to a limit structure and that limit is the graphon. The convergence is defined via the convergence of homomorphism densities calculated on the subgraph samples.

2.2.1 Homomorphism Numbers and Densities

Definition 2.2.1. Let $\hom(H, G)$ denote the set of homomorphisms of graph H into graph G and let $|\hom(H, G)|$ be the *homomorphism number* of H into G (the cardinality of the set).

Homomorphism numbers depend on the sizes of the input graphs so we will have to define normalized versions of them:

Definition 2.2.2. Let us define the *homomorphism density* between graphs H and G as

$$t(H, G) = \frac{|\hom(H, G)|}{|V_G|^{|V_H|}}.$$

One can think of this number as the probability that a random mapping of vertices from H to G is a homomorphism.

Homomorphism densities would not make much sense if G was a sparse graph or a bounded degree graph since the number of all possible mappings between the vertices would quickly outgrow the number homomorphisms and $t(H, G)$ would tend to 0.

To that end, Lovász [Lov12] defined a different way of normalizing homomoprhism numbers:

Definition 2.2.3. A *homomorphism frequency* between graph H and a sparse graph G is

$$t(H, G) = \frac{|\hom(H, G)|}{|V_G|}.$$

Working with sparse graphs and representing them with graphons would be out of the intended scope of this thesis, especially when trying to represent them with neural networks and coming up with an algorithm that does this. So for the sake of simplicity, let us assume that we are working with dense graphs, but of course many real life graphs/networks do not fit the dense setting.

Definition 2.2.4. We further define *injective homomorphism density* and *induced homomorphism density* as

$$t_{inj}(H, G) = \frac{|\mathrm{inj}(H, G)|}{\prod_{i=|V_G|-|V_H|+1}^{|V_G|} i}$$

and

$$t_{ind}(H, G) = \frac{|\mathrm{ind}(H, G)|}{\prod_{i=|V_G|-|V_H|+1}^{|V_G|} i}$$

respectively, where $\mathrm{inj}(H, G)$ is the set of injective homomorphisms of H into G and $\mathrm{ind}(H, G)$ is the set of injective homomorphisms of H into G that are also non-edge-preserving.

2.2.2 Modeling and Sampling Large Graphs Versus Homomorphism Densities

As noted by Lovász, studying large networks that cannot be even stored in working memory, can be done by sampling smaller subgraphs. This kind of statistical inference on objects (such as graphs) other than numbers was first established by Goldreich et al. in 1998 [GGR98].

One could sample a large graph by uniformly sampling vertices of that graph and by determining the edges of the vertex sample, get an induced subgraph. Note that this method would not make sense for sparse graphs as one would almost surely get an independent set. This sample contains enough information to infer a lot of properties about the large graph with an error caused by the random sampling that can be pushed arbitrarily low with the growing size of the sample (in number of vertices). Also, sampling many smaller subgraphs yields less information than sampling a larger subgraph once.

It turns out that sampling (ordered, without repetition) of induced subgraphs from large graphs carries the same information as homomorphism densities. When sampling an induced subgraph of size n from a large graph G, the graph G creates a probability distribution over all graphs H of size n: $\sigma_{G,n}(H)$. In the dense case, this probability distribution exactly matches $t_{ind}(H, G)$, which is the probability that a random injective map from H to G is edge and non-edge preserving, or in other words that the random map corresponds to an induced subgraph of G that happens to be H.

Now that it is established that there is a one to one correspondence of subgraph sampling with t_{ind}, [Lov12] also introduces a relationship between t_{ind} and t_{inj}:

$$t_{ind}(H, G) = \sum_{H' \supseteq H} (-1)^{|E_{H'}|-|E_H|} \cdot t_{inj}(H', G), \tag{2.2}$$

where H' ranges over all graphs of the same size as H that are obtained by adding edges to H. In short, the induced homomorphism density (or subgraph density) can be expressed as a linear combination of injective homomorphism densities.

If we fix H, the distance between $t(H, G)$ and $t_{inj}(H, G)$ tends to 0 as the size of G grows [Lov12]:

$$|t(H, G) - t_{inj}(H, G)| \leq \frac{1}{|V_G|}\binom{|V_H|}{2} \tag{2.3}$$

Ultimately, the relationship between t, t_{ind} and t_{inj} is tight for large graphs and the error between them becomes negligible and so t carries the same information as subgraph sampling.

2.2.3 Graphons

In this subsection, we finally describe how Lovász [Lov12] defined graphons, the central objects in graphon theory, and explain how they relate to graphs.

Definition 2.2.5. A *graphon* (short for graph function) is a symmetric measurable function $W : \langle 0, 1\rangle^2 \to \langle 0, 1\rangle$.

A graphon can be interpreted in multiple different ways that will be described in more detail in the following subsections:

- A continuous analogue to a weighted adjacency matrix.
- A random graph model, or a distribution of random graphs of any size.
- A limit object for a sequence of graphs that have "the same structure".

Graphons as "continuous adjacency matrices"

A graphon W can be viewed as an infinite weighted adjacency matrix, where each real number in the unit interval $\langle 0, 1\rangle$ represents a vertex and the graphon value $W(x, y)$ of a particular pair of vertices $x, y \in \langle 0, 1\rangle$ represents the normalized edge weight or edge probability between them.

To show the relationship between a graphon and an unweighted simple graph, we can construct a graphon W_G from a simple graph G by equidistantly partitioning $\langle 0, 1\rangle$ into $|V_G|$ intervals of length $\frac{1}{|V_G|}$, each corresponding to a vertex of G and finally defining the values of W_G according to the presence of edges in G:

$$W_G(x, y) = \begin{cases} 1, & x \in \langle \frac{i-1}{|V_G|}, \frac{i}{|V_G|}\rangle,\ y \in \langle \frac{j-1}{|V_G|}, \frac{j}{|V_G|}\rangle,\ \{i, j\} \in E_G \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

as illustrated in figure 2.3.

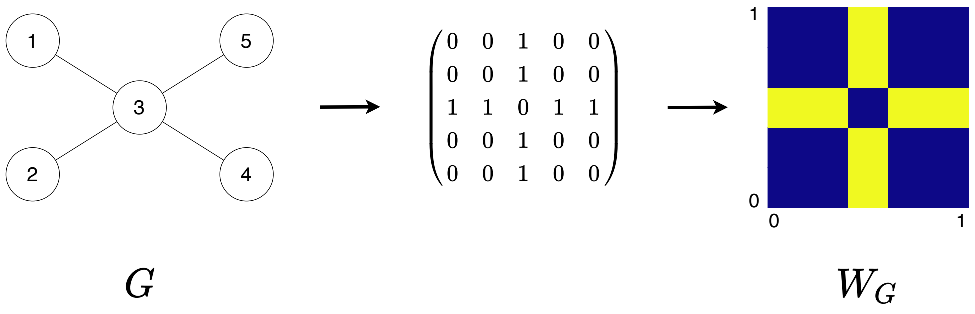


Figure 2.3: Construction of a graphon from a simple graph

Graphons as random graph distributions

A graphon can also be viewed as a distribution of random graphs from which a graph of any size can be sampled. See figure 2.4.

Let's say that we want to sample a simple graph G of size n from graphon W. We can do that by uniformly sampling n numbers S_n from $\langle 0, 1\rangle$ that will represent the vertices of G, then determine the edges by including an edge $\{x, y\}$ into E_G with probability $W(x, y)$, where $x \in S_n, y \in S_n$. Then, the numbers S_n, present in V_G and E_G, are mapped by an arbitrary bijection to integers $\{1, ..., n\}$. Let us denote such a graph G of size n sampled from a graphon W as a *W-random graph* $G_{n,W}$ and the distribution of such graphs as $\mathbb{G}_{n,W}$.

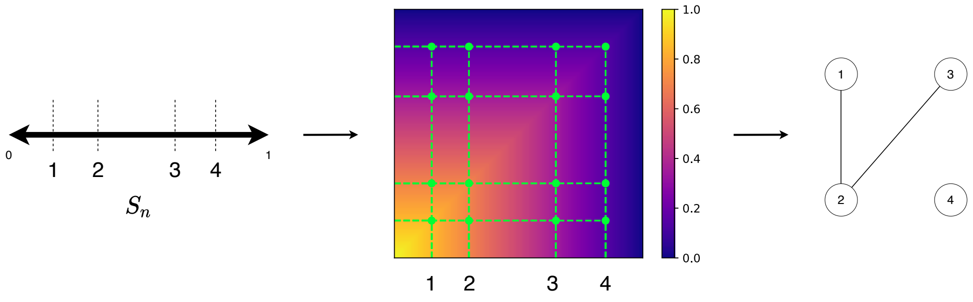


Figure 2.4: Sampling a simple graph of size 4 from a graphon

Graphons as limit objects for random subgraph sequences

The interpretation of graphons as limits of graph sequences will be more thoroughly explored in section 2.2.7 but below are two examples that bring a little bit of intuition to random graph sequences, their structural difference and their relationship to graphons.

Assume a setting where we want to model a large graph and our only only option is to sample induced subgraphs from it. As we increase the size of the sampled subgraphs, their structure becomes increasingly similar to the actual large graph and as we will see later, the object that the subgraphs will "converge to", will be a graphon. In figures 2.5 and 2.6, you can see graph sequences sampled from large graphs and then converted to graphons via equation 2.4. The large graphs' structures, from which the sequences were created, correspond to the constant graphon $W(x, y) = \frac{1}{2}$ and a so called *growing uniform attachment* graphon $U(x, y) = 1 - \max(x, y)$ respectively [Gla16]. A sequence of random graphs for W is defined trivially, just create n vertices and randomly add edges between pairs of vertices with probability $\frac{1}{2}$. A sequence of random graphs for U is defined inductively, the first graph is just a vertex and graphs of size $n \geq 2$ are created by adding a vertex to the previous graph of size $n - 1$ and connecting the new vertex with the other ones with probability $\frac{1}{n}$.

Graphons constructed from a simple graph with 2.4 are step functions, but a sequence of such graphons may converge to a graphon that is not a step function but a continuous function on $\langle 0, 1\rangle^2$. We can view it as a local averaging process that aggregates 0s and 1s. The sequence limit object does not have to belong to the same class of objects as the sequence elements. An example for comparison would be that a limit of a sequence of rational numbers might not be a rational number.

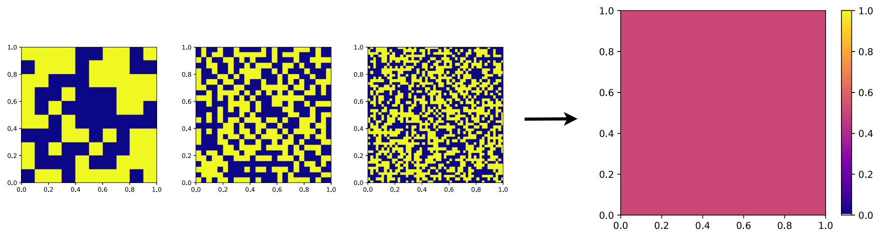


Figure 2.5: Sequence of random graphs with edge probability $\frac{1}{2}$

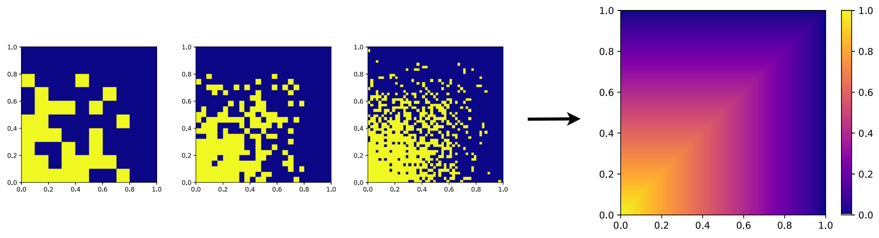


Figure 2.6: Sequence of growing uniform attachment graphs

2.2.4 Homomorphism Densities Generalization

We will later define convergence of graph sequences and how they tend to graphons but first, we will need to extend homomorphism densities, another one of the key ideas in the theory of graphons.

Definition 2.2.6. The *homomoprhism density* of a graph F into a graphon W is defined as

$$t(F,W) = \int_{\langle 0,1\rangle^{|V_F|}} \prod_{\{i,j\}\in E_F} W(x_i,x_j) \prod_{k\in V_F} dx_k$$

This definition of a probability that a mapping of vertices of the graph F into a graphon W is a homomorphism may be a little bit intimidating. Let us have look at a particular example in an analogous discrete setting, where we set F to be a cycle graph C_4 and assume a large graph G instead of W just to get an intuition. One of the possible ways of calculating the density is:

$$t(C_4,G) = \frac{1}{|V_G|^4} \sum_{v_1=1}^{|V_G|} \sum_{v_2=1}^{|V_G|} \sum_{v_3=1}^{|V_G|} \sum_{v_4=1}^{|V_G|} A^G_{v_1,v_2} \cdot A^G_{v_2,v_3} \cdot A^G_{v_3,v_4} \cdot A^G_{v_4,v_1},$$

where A^G is the binary adjacency matrix of G. An intuitive way of interpreting this is that each of the sums represent a `for` loop for each of the vertices in C_4 and these iterate over all possible mappings of the four vertices in C_4 to vertices in G. The body of the sums, $A^G_{v_1,v_2} \cdot A^G_{v_2,v_3} \cdot A^G_{v_3,v_4} \cdot A^G_{v_4,v_1}$, represent the condition that all of the edges $E_{C_4} = \{\{1,2\},\{2,3\},\{3,4\},\{4,1\}\}$ map to edges $\{\{v_1,v_2\},\{v_2,v_3\},\{v_3,v_4\},\{v_4,v_1\}\} \subseteq E_G$, or in other words that the mapping is a homomorphism. That condition is satisfied if and only if $A^G_{v_1,v_2} \cdot A^G_{v_2,v_3} \cdot A^G_{v_3,v_4} \cdot A^G_{v_4,v_1} = 1$. The normalization constant $\frac{1}{|V_G|^4}$ is there to turn the homomorphism mappings count (homomorphism number) to a density.

Coming back to the continuous version, we get

$$t(C_4,W) = \int_0^1\int_0^1\int_0^1\int_0^1 W(v_1,v_2)\cdot W(v_2,v_3)\cdot W(v_3,v_4)\cdot W(v_4,v_1)\cdot dv_1 dv_2 dv_3 dv_4,$$

which is, in a sense, a continuous analogy to the discrete case above. The four integrals "iterate" or integrate over the variables v_1, v_2, v_3, v_4 respectively, which represent the mapping of the four vertices of C_4 into four real numbers (vertices) from $\langle 0,1\rangle$. The product in the body then represents the homomorphism condition on a weighted graph because the graphon values don't have to necessarily be step function values of 0 or 1, but also anything in between.

We can also extend the notion of subgraph inducing density:

Definition 2.2.7. The *induced homomorphism density* in the graphon case is defined as

$$t_{\text{ind}}(F,W) = \int\limits_{\langle 0,1\rangle^{|V_F|}} \prod_{\{i,j\}\in E_F} W(x_i,x_j) \prod_{\{i,j\}\notin E_F, i,j\in V_F} (1-W(x_i,x_j)) \prod_{k\in V_F} dx_k.$$

We do not have to define the injective density because a random mapping of vertices of F into $\langle 0,1\rangle$ is injective with probability 1, therefore

$$P(t_{\text{inj}}(F,W) = t(F,W)) = 1, \ (\forall F)(\forall W).$$

There is a linear relationship between injective and subgraph inducing densities for the continuous case similarly to 2.2 [Lov12]:

$$t_{ind}(F,W) = \sum_{F'\supseteq F} (-1)^{|E_{F'}|-|E_F|} \cdot t(F',W). \tag{2.5}$$

2.2.5 Weak Isomorphism

Using graphons comes with a complication - they're distinguishable up to an equivalence class.

Definition 2.2.8. We call two graphons U, W *weakly isomorphic* if there exist measure preserving maps $\varphi, \psi : \langle 0,1\rangle \to \langle 0,1\rangle$ such that $U(\varphi(x),\varphi(y)) = W(\psi(x),\psi(y))$ almost everywhere.

This equivalence of graphons can be thought of as isomorphism between finite graphs or the relabeling/reshuffling of vertices. Take for example the graph in figure 2.7.

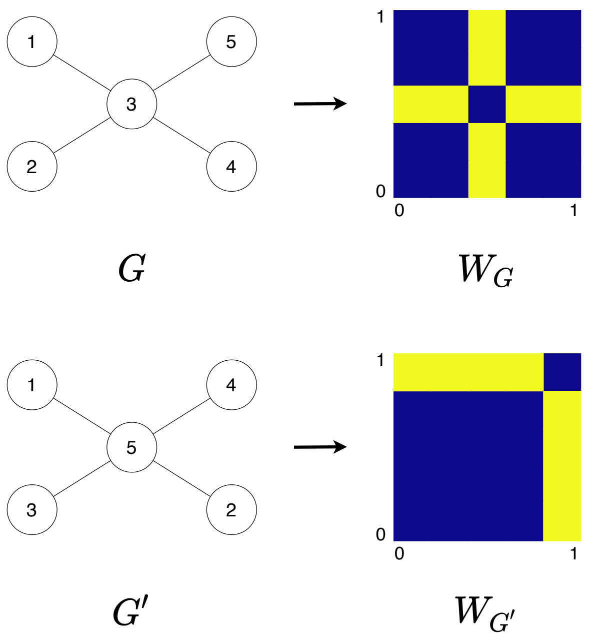


Figure 2.7: Two isomorphic graphs corresponding to two very different graphons

The two graphs G, G' are essentially the same but they have completely different corresponding graphons. Notice that this does not affect the homomorphism densities. Weakly isomorphic graphons produce the same densities for all finite graphs F and that is how Lovász first defined weak isomorphism - two graphons are weakly isomorphic if their homomorphism densities are equal for all finite graphs F.

2.2.6 Distance Between two Graphons

When we model large graphs with graphons, we would like to express and measure the similarity between two graphons based on properties that matter to us. For example, we would want two graphons U and W to be "close" to each other in a distance metric or even equivalent if the structure of graphs sampled from them is very similar.

Due to weak isomorphism, we can't simply measure the distance between graphons U and W with, for example, the L1 norm

$$||U - W||_1 = \int_{\langle 0,1 \rangle^2} |U(x,y) - W(x,y)| dxdy$$

since the distance between graphons W_G and $W_{G'}$ in figure 2.7 would be non-zero.

One of the distance metrics that is invariant to weak isomorphisms defined in [Lov12] is the sampling distance.

Definition 2.2.9. The *variation distance* d_{var} between two distributions α, β defined on the same set X is defined as

$$d_{var}(\alpha, \beta) = \sup_{X'} |\alpha(X') - \beta(X')|,$$

where X' goes over measurable subsets of X.

Definition 2.2.10. The *sampling distance* δ_{samp} between graphons U, W is defined as

$$\delta_{samp}(U, W) = \sum_{n=1}^{\infty} \frac{1}{2^n} d_{var}(\mathbb{G}_{n,U}, \mathbb{G}_{n,W}),$$

where $\mathbb{G}_{n,W}$ is the distribution of size n graphs sampled from graphon W defined in 2.2.3.

Lovász notes that the coefficient $\frac{1}{2^n}$ is an arbitrary choice that just ensures convergence of the sum but it also puts more weight on the distribution similarity of smaller graphs. This becomes important for our purposes later when we model graphons with neural networks just based on homomorphism densities. It turns out that the densities of smaller graphs F are indeed the more important ones (at least from a theoretical upper bound standpoint) as it is stated by 2.2.1.

The sampling distance can also be expressed as

$$\delta_{samp}(U, W) = \sum_{F} \frac{1}{2^{|V_F|+1}} |t_{ind}(F, U) - t_{ind}(F, W)|. \tag{2.6}$$

The second distance metric, invariant to weak isomorphisms, is the cut distance.

Definition 2.2.11. The *cut distance* $\delta_\square$ between graphons U, W is defined as

$$\delta_\square(U,W) = \inf_{\varphi,\psi} \sup_{S,T} \left| \int_{S\times T} U(\varphi(x),\varphi(y)) - W(\psi(x),\psi(y))\ dxdy \right|,$$

where φ, ψ are measure preserving mappings and S, T are measurable subsets of $\langle 0,1\rangle$.

We can understand the distance as calculating a difference graphon $U - W$ and taking subsets $S, T \subseteq \langle 0,1\rangle$ (which are not necessarily disjoint) to find the maximum cut of the edge discrepancy graphon between the vertex sets S and T by integrating over "boxes" $S \times T$. The infimum minimizes this maximum cut value over relabelings φ and ψ to handle weak isomorphism.

The cut metric can be defined for finite graphs F, G via the construction of their respective graphons W_F, W_G by using 2.4. That is

$$\delta_\square(F,G) = \delta_\square(W_F, W_G).$$

2.2.7 Convergence of Dense Graph Sequences

In this section, we finally define what it means for a sequence of dense graphs to converge and that the object they converge to is a graphon.

Definition 2.2.12. A sequence of dense graphs $(G_n)_{n=1}^\infty$ *converges* if the induced subgraph densities $t_{\text{ind}}(F, G_n)$ converge as $n \to \infty$ for every finite graph F.

Notice that this definition of convergence fits well with our framework of sampling increasingly larger induced subgraphs from a large network. The densities $t_{\text{ind}}(F, G_n)$ are equivalent to distributions over finite graphs sampled from G_n. When these densities converge, they directly reflect convergence in the structure of the sampled graphs G_n.

It was previously established that induced subgraph densities can be expressed as linear combinations of homomorphism densities in from 2.2, thus t_{ind} converges if and only if t_{inj} does. From 2.3 we get that t_{inj} converges if and only if t converges as $n \to \infty$, therefore the following statements about $(G_n)_{n=1}^\infty$ are equivalent:

- $t_{\text{ind}}(F, G_n)$ converges for all finite graphs F,
- $t_{\text{inj}}(F, G_n)$ converges for all finite graphs F,
- $t(F, G_n)$ converges for all finite graphs F.

Theorem 2.2.1 (Theorem 11.21. in [Lov12])**.** For every convergent sequence $(G_n)_{n=1}^{\infty}$ of dense graphs there exists a graphon W such that for every finite graph F

$$\lim_{n\to\infty} t(F, G_n) = t(F, W).$$

Let us define the limit of a convergent graph sequence $(G_n)_{n=1}^{\infty}$ as the graphon W and write that as $(G_n)_{n=1}^{\infty} \to W$.

Theorem 2.2.2 (11.22. in [Lov12])**.** For a graph sequence $(G_n)_{n=1}^{\infty}$ and a graphon W, $(G_n)_{n=1}^{\infty} \to W$ if and only if $\lim_{n\to\infty} \delta_{\square}(W_{G_n}, W) = 0$, where W_{G_n} is a graphon constructed from G_n via 2.4.

Theorem 2.2.3 (11.3. in [Lov12])**.** A graph sequence $(G_n)_{n=1}^{\infty}$ converges if and only if it forms a *Cauchy sequence* in the metric $\delta_{\square}$.

Given the cut metric space on graphs $(G, \delta_{\square})$, a *Cauchy sequence* of graphs is a sequence $(G_1, G_2, G_3, \dots)$ such that

$$(\forall \varepsilon \in \mathbb{R}, \varepsilon > 0)(\exists k \in \mathbb{N})(\forall i, j \in \mathbb{N})(i, j >= k \implies \delta_{\square}(G_i, G_j) < \varepsilon).$$

In other words, for any small number ε, all but a finite number of graphs from the sequence are no more far away from each other than ε in the cut metric.

Proposition 2.2.1. From 2.6 together with the definition of convergence and 2.2.7, we get that if two graph sequences converge to the same densities then the sampling distance δ_{samp} between the graphons they converge to goes to 0.

Lemma 2.2.1 (**Inverse Counting Lemma** from [Lov12] based on [BCL+08])**.** Let k be a natural number, and let U, W be graphons. If for every simple graph F of size k

$$|t(F, U) - t(F, W)| \leq \frac{1}{2^{k^2}}$$

then

$$\delta_{\square}(U, W) \leq \frac{50}{\sqrt{\log k}}.$$

This lemma states that there is a theoretical upper bound on the cut distance that decreases with diminishing returns as we match the homomorphism densities between graphons U and W with and increasing size k of the graphs F. In other words, the upper bound on similarity of two graphons is most influenced by matching homomorphism densities of smaller graphs F.

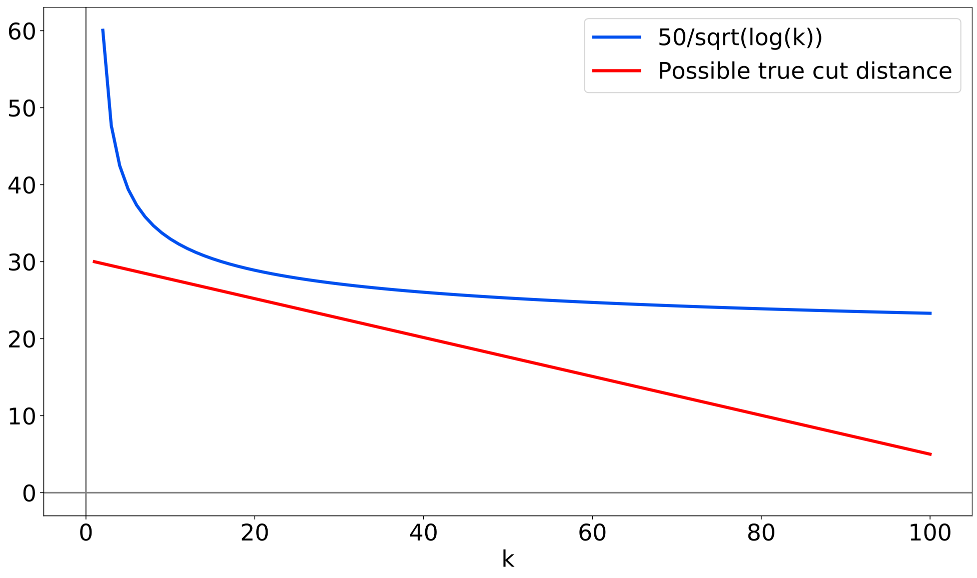


Figure 2.8: Inverse Counting Lemma upper bound on cut distance $\delta_\square(U, W)$ as $k \to \infty$

Unfortunately, the lemma does not directly say that homomorphism densities for smaller graphs F are more important for two graphons to be close to each other in the cut distance because it is an upper bound that decreases very slowly for high k's which does not entail that the true convergence of $\delta_\square$ is quickest for small k's, see figure 2.8 for a counter-example for that - the upper bound still holds, but the cut distance still decreases linearly as $k \to \infty$.

Chapter 3

Neural Networks

This chapter focuses on the introduction of neural networks and the description of some of the areas of this study that are key to this thesis. The research that has been done in this field is vast and some of the areas are not really relevant to what we are trying to achieve in this thesis (for example recurrent network architectures for sequence modeling), so we will go more in-depth only in the neural network architectures that we will use. Additionally, we are going to list some of the community-verified techniques that aim to mitigate some of the problems with deep learning which we unsurprisingly encounter when modeling graphons.

3.1 Introduction

Artificial neural networks (ANNs) were originally intended to be computational models of biological learning - to mimic intelligent behavior of a real neural network, the brain. Today, ANNs are viewed rather as biology-inspired computational models that follow a more general principle of learning - composing simpler concepts into more complex ones [GBC16]. ANNs essentially map a set of inputs to a set of outputs which is exactly what we need from a graphon and that is to map the probability of an edge occurrence to a set of two vertices represented by two real numbers.

3.2 Deep Feed-Forward Neural Networks

The architectures that we are going to use in our modeling are deep feed-forward neural networks (DFFNNs), also called multi-layer perceptrons (MLPs), or just feed-forward networks. As opposed to recurrent neural networks, these networks do not use their outputs as their inputs (a feedback loop).

A DFFNN can be represented as a directed acyclic graph (DAG), where each vertex represents a function applied on the output of its preceding vertices, see figure 3.1. These functions are also referred to as *units* or *neurons*. Vertices with no predecessor are called sources and in that case they are the input that we pass into the network. If a unit has no successor then it is called a sink and it produces the output of the network. The output of a DFFNN is essentially a composition of simpler functions and the largest number of chained function applications, or the longest path between a source and a sink determines its *depth*, hence the word deep in DFFNN. Units are organized in *layers* based on their distance to a source. Sources are in the *input layer*, sinks are in the *output layer* and units in-between are considered to be in *hidden layers*.

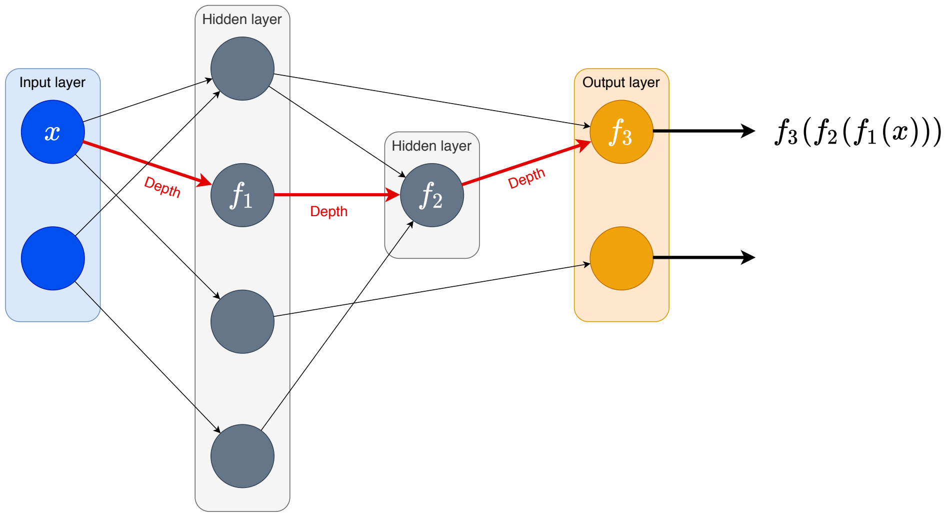


Figure 3.1: A deep feed-forward neural network as a directed acyclic graph

The complexity of a DFFNN, or its capacity to represent complex functions depends on several things: the complexity of the units themselves, e.g. whether they use linear or non-linear functions, the depth of the model, the width of the model (number of units in a layer), and the connections between units.

The motivation behind using hidden layers is that they perform a mapping of the input space into a representation space where the problem at hand is easier to solve, linearly separable for example. We can view classical machine learning algorithms like linear regression, decision trees, k-nearest-neighbours as neural networks that do not have hidden layers and instead they perform the entire classification or regression task in the output layer only while accepting raw inputs from the input layer. Another advantage of hidden layers in a DFFNN is that the mapping from the input space into the representation space is learned automatically with little to no help from humans by assuming priors about the data. See [GBC16] for a more involved example of a so called feature map of the input space on the XOR problem.

3.2.1 Functions of MLP Units or Neurons

As we've seen in section 3.2, the units, also called neurons, are the basic building blocks of an MLP. The first architecture of a neural network unit, as we know it today, was the Perceptron which was introduced by Frank Rosenblatt in 1960 [Ros60]. The perceptron performs a linear combination on the input values, adds a bias and thresholds the output to 0 or 1 as seen in figure 3.2. The dot operator between $\mathbf{w}$ and $\mathbf{x}$ is a dot product. The definition of the perceptron loosely mimics the biological neuron which sends (activates, hence activation function) electrical signals down its axon (output) if enough signals were received on dendrites (input).

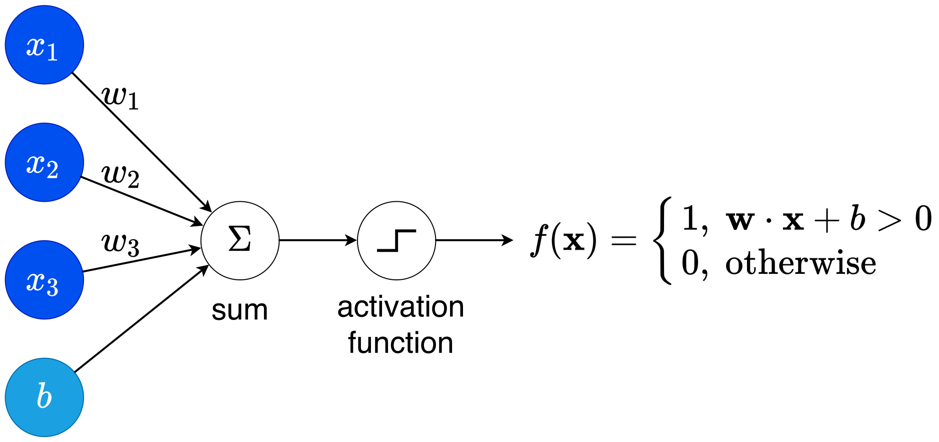

x_1
x_2
x_3
b
w_1
w_2
w_3
Σ
sum
activation function
$f(\mathbf{x}) = \begin{cases} 1, & \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0, & \text{otherwise} \end{cases}$

Figure 3.2: Rosenblatt's perceptron

The general function of a neuron in an MLP can be seen below in equation 3.1.

$$f(\mathbf{x}) = a(\mathbf{w} \cdot \mathbf{x} + b) = a(\sum_{i=1}^{D} w_i \cdot x_i + b), \tag{3.1}$$

where D is the number elements in the input vector $\mathbf{x}$, a is an *activation function*, and $\mathbf{w}$ are the *weights* or *parameters* of the neuron. Without loss of generality, we add an extra dimension $x_{D+1} = 1$ to the input vector so that we can add the *bias* b into the weights vector, i.e. $w_{D+1} = b$, so that we can write $\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{D+1} w_i \cdot x_i$ to be more succinct.

This function forms the basis for almost all of the neurons that we are going to use and the only part that we are going to change is in fact the activation function. Bellow, we are going to list some of the activation functions that are or have been widely used at some point in time along with their advantages and disadvantages.

- No activation function $f(x) = x$, figure 3.3

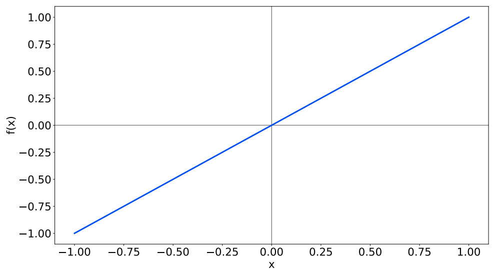


Figure 3.3: Identity activation function

Sometimes it might be useful to use and identity activation function in the output layer, for example when the output is an unbounded real vector. Using this function however does not bring any non-linearity to our neuron which makes it essentially a linear regressor.

- Sigmoid activation function $f(x) = \frac{1}{1+e^{-x}}$, figure 3.4

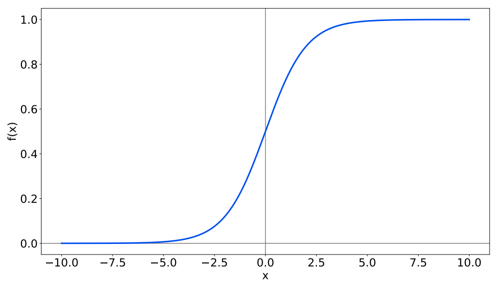


Figure 3.4: Sigmoid activation function

The sigmoid function is useful when we want to ensure that the output of our neuron stays between 0 and 1 which can represent some probability for example. We will use this activation in the output layer of our models since we require the values of a graphon to be between 0 and 1. The sigmoid also behaves near linearly with input values near zero, however when input values are very positive or negative then the sigmoid quickly approaches 0 or 1 respectively. This is called saturation and contributes to the problem of a so called vanishing gradient that we will return to in section 3.7.

- Hyperbolic tangent activation function $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$, figure 3.5

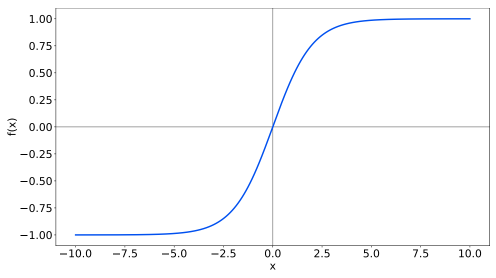


Figure 3.5: Tanh activation function

The hyperbolic tangent has similar properties to the sigmoid function but instead it ranges from -1 to 1.

- Rectifier or hinge activation function $f(x) = \max(0, x)$, figure 3.6

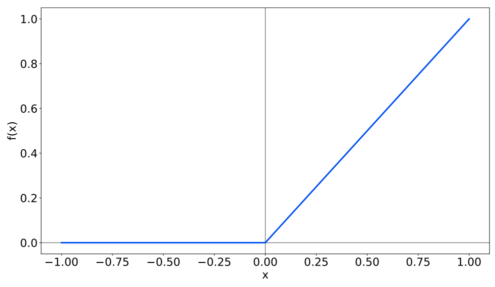


Figure 3.6: Rectifier activation function

The rectifier function is faster to compute compared to the sigmoid and hyperbolic tangent functions. The gradient is 1 for $x > 0$ which helps with the vanishing gradient problem, however it comes with another numerical instability problem because it is unbounded.

Another one of the main features of the rectifier is that it collapses all negative input values to a 0 output. This causes interesting effects that can be seen as positive or negative. For one, it leads to so called sparse network representations, $x < 0$ turns the neuron completely off since it outputs only 0 and the gradient w.r.t its parameters is also 0, thus its weights won't be changed by back-propagation. So an entire network might only end up with a few functioning neurons. This might be a good thing as Glorot et. al. suggest, because it separates (disantangles) factors that differentiate the data, it is more likely that the data are linearly separable in the representation space, different patterns in the data can be efficiently modeled by different pathways in the network, and a sparse network can be condensed or its outputs may be efficiently computed by just ignoring dead neurons while maintaining the same or nearly the same performance. See [GBB11] for the full comparison of activation functions.

The effect where neurons turn off (they die) may be undesirable when a large number of them dies and causes the network to lose the flexibility to learn complex functions. LeakyReLU [MHN13], a special case of PReLU [HZRS15], is a fix that somewhat retains the sparsity while preventing neurons from completely dying. LeakyReLU has a small non-zero slope for negative inputs so the gradient is never truly zero.

$$LeakyReLU(x) = \begin{cases} x, & x >= 0 \\ a \cdot x, & x < 0 \end{cases},$$

where a is a small constant.

3.3 Back-Propagation and Gradient-Based Learning

The goal of training a neural network $f(\mathbf{x};\theta)$ is to find its parameters or weights θ such that $f(x;\theta) = f^*(\mathbf{x})$, where f^* is the desired function that performs perfect inference in our problem. The way we evaluate the performance of our model f is via a *cost* or *loss* function $\mathcal{L}(f_\theta)$. Training a neural network becomes an optimization problem of finding θ^*, where we minimize the objective $\mathcal{L}(f_\theta)$, i.e. $\theta^* = \arg\min_\theta \mathcal{L}(f_\theta)$.

See table below for frequently used, maximum likelihood based losses for different tasks.

Task	Loss Function	Assumptions
Binary classification	$-\frac{1}{m}\sum_{i=1}^{m} y_i \log f(\mathbf{x}_i;\theta) + (1-y_i)\log(1-f(\mathbf{x}_i;\theta))$	$y \sim \text{Bernoulli}(y\|p)$
Multinomial classification	$-\frac{1}{m}\sum_{i=1}^{m}\sum_{c=1}^{K} y_{ic} \log f(\mathbf{x}_i;\theta)$	$y \sim \text{Multinoulli}(y\|\mathbf{p})$
Regression	$\frac{1}{m}\sum_{i=1}^{m}(y_i - f(\mathbf{x}_i;\theta))^2$	$y \sim N(f(\mathbf{x}_i;\theta),\sigma^2)$

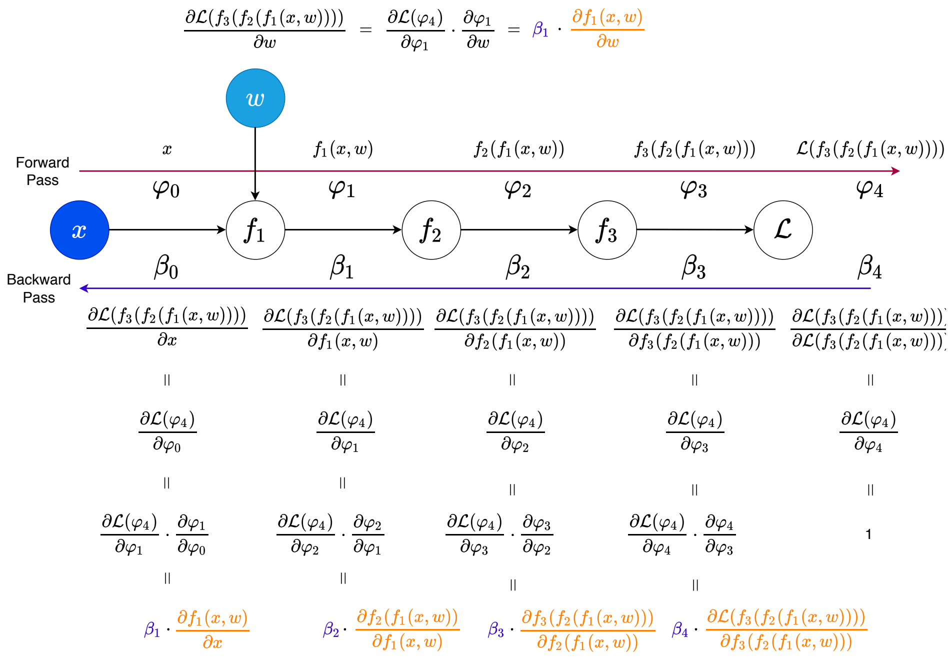
Table 3.1: Frequently used loss functions, taken from [Drc]

3.3.1 Back-Propagation

When a network is composed of many non-linear functions, the optimization objective is non-convex w.r.t. the parameters w. Gradient-based learning methods provide a computationally efficient way of optimizing an objective in place of efficient algorithms that rely on convexity. These methods iteratively move the parameters in the steepest descent direction to minimize the loss function. Today's gradient-based methods have a common way of actually computing the gradients of the loss function w.r.t. w and that is *back-propagation* or just *backprop* introduced by Rumelhart et.al. [RHW86].

To compute the gradients, modern libraries, that perform backprop, construct a *computational graph*, which is a DAG where each node represents a differentiable operation on a variable that is passed in from preceding nodes. Source nodes are, again, the input data nodes. Variables are usually tensors of various dimensions (scalars, vectors, matrices, D-dimensional tensors). Data flow from sources to sinks is called the *forward pass.* The forward pass produces the network's outputs which is then used to calculate the value of the loss function $\mathcal{L}(f_w)$. After the loss function has been computed, backprop then calculates the gradient of the loss function w.r.t. the inputs and parameters for each node starting from the sinks, usually the loss function, this is called the *backward pass.*

Please refer to figure 3.7 for a visual example of back-propagation. In the figure, a series of operations is performed on the input variable x and the graph ends with the value of the loss function $\mathcal{L}$. Intermediate outputs of the operations, or the forward messages, are denoted by φ_i. The gradients of $\mathcal{L}$ w.r.t. the output of each operation, or the backward messages, are denoted by β_i and are recursively computed from the sink and passed to preceding nodes. Notice that the chain rule of calculus is applied in each node in order to accumulate the gradients of succeeding nodes into the backward message β_i by multiplication, thus each node only has to compute the partial derivative of its output w.r.t. its input in order to produce β_i's and this is depicted by the orange color in the figure. These partial derivatives usually have closed form solutions for standard differentiable operations. The β_i's represent the sensitivity of the loss with respect to the operation outputs. To calculate the gradient of the loss function w.r.t. parameters w, the example shows that it is only necessary to calculate the partial derivative of the operation f_1 output w.r.t. w while utilizing the backward message passed from succeeding nodes. To see that it is just repeated application of the chain rule, it might be helpful for the reader to iteratively substitute β_i's from the higher indexes down, colored by the indigo color in the last row of the figure.



$\frac{\partial \mathcal{L}(f_3(f_2(f_1(x,w))))}{\partial w} = \frac{\partial \mathcal{L}(\varphi_4)}{\partial \varphi_1} \cdot \frac{\partial \varphi_1}{\partial w} = \beta_1 \cdot \frac{\partial f_1(x,w)}{\partial w}$
w
Forward Pass
x
$f_1(x,w)$
$f_2(f_1(x,w))$
$f_3(f_2(f_1(x,w)))$
$\mathcal{L}(f_3(f_2(f_1(x,w))))$
φ_0 φ_1 φ_2 φ_3 φ_4
x f_1 f_2 f_3 $\mathcal{L}$
β_0 β_1 β_2 β_3 β_4
Backward Pass
$\frac{\partial \mathcal{L}(f_3(f_2(f_1(x,w))))}{\partial x} = \frac{\partial \mathcal{L}(\varphi_4)}{\partial \varphi_0} = \frac{\partial \mathcal{L}(\varphi_4)}{\partial \varphi_1} \cdot \frac{\partial \varphi_1}{\partial \varphi_0} = \beta_1 \cdot \frac{\partial f_1(x,w)}{\partial x}$
$\frac{\partial \mathcal{L}(f_3(f_2(f_1(x,w))))}{\partial f_1(x,w)} = \frac{\partial \mathcal{L}(\varphi_4)}{\partial \varphi_1} = \frac{\partial \mathcal{L}(\varphi_4)}{\partial \varphi_2} \cdot \frac{\partial \varphi_2}{\partial \varphi_1} = \beta_2 \cdot \frac{\partial f_2(f_1(x,w))}{\partial f_1(x,w)}$
$\frac{\partial \mathcal{L}(f_3(f_2(f_1(x,w))))}{\partial f_2(f_1(x,w))} = \frac{\partial \mathcal{L}(\varphi_4)}{\partial \varphi_2} = \frac{\partial \mathcal{L}(\varphi_4)}{\partial \varphi_3} \cdot \frac{\partial \varphi_3}{\partial \varphi_2} = \beta_3 \cdot \frac{\partial f_3(f_2(f_1(x,w)))}{\partial f_2(f_1(x,w))}$
$\frac{\partial \mathcal{L}(f_3(f_2(f_1(x,w))))}{\partial f_3(f_2(f_1(x,w)))} = \frac{\partial \mathcal{L}(\varphi_4)}{\partial \varphi_3} = \frac{\partial \mathcal{L}(\varphi_4)}{\partial \varphi_4} \cdot \frac{\partial \varphi_4}{\partial \varphi_3} = \beta_4 \cdot \frac{\partial \mathcal{L}(f_3(f_2(f_1(x,w))))}{\partial f_3(f_2(f_1(x,w)))}$
$\frac{\partial \mathcal{L}(f_3(f_2(f_1(x,w)))}{\partial \mathcal{L}(f_3(f_2(f_1(x,w)))} = \frac{\partial \mathcal{L}(\varphi_4)}{\partial \varphi_4} = 1$

Figure 3.7: Backprop example

Now that the inner workings of backprop are clear, a problem may come to ones mind where if the longest path from a source to a sink in the computational DAG is very long and if the partial derivatives in each node are smaller than 1 then the backward messages become near-zero the closer they got to sources. This is caused by the recursive multiplication of the chain rule. A derivative of an operation may be small for example with the output of a sigmoid activation becomes saturated (approaching 0 or 1) because the function is very flat at those regions. This is called the *vanishing gradient problem*.

3.3.2 Gradient Descent

The simplest form of gradient descent accepts a training data set $\mathcal{T}^m = \{(\mathbf{x}_i, y_i) \mid i \in \{1, \dots, m\}\}$, where $\mathbf{x}_i$ is the i-th input data point (or *features*) and y_i is the desired output that we want our network to produce for the data $\mathbf{x}_i$) (or *label/target*). The algorithm moves a model f's parameters w in the direction of steepest descent in terms of the loss function value and it does so iteratively. In each iteration, it subtracts the gradient of the loss function w.r.t. the network's parameters $\nabla_w \mathcal{L}(f)$ from the parameters themselves with a learning rate α. The gradient to descend with, can and is usually computed by using back-propagation. The loss is a function of points from the training set - the data that we pass in. The algorithm stops when a stopping criterion is met which could be a low enough loss value, a certain number of iterations or *epochs*, or convergence of the loss value to name a few examples. See algorithm 1.

Algorithm 1: Gradient Descent

```
Input: training data $\mathcal{T}^m = \{(\mathbf{x}_i, y_i) \mid i \in \{1, \dots, m\}\}$
Input: neural network $f_w$ with parameters $w$
Output: neural network $f_{\hat{w}}$ with updated parameters $\hat{w}$ with a low
        $\mathcal{L}(f_{\hat{w}})$
$\hat{w} \leftarrow w$
$k \leftarrow 1$
while stopping criterion not met do
    $\nabla_{\hat{w}} \mathcal{L}(f_{\hat{w}}) \leftarrow$ backprop on $\mathcal{L}(f_{\hat{w}}(\mathcal{T}^m))$
    $\hat{w} \leftarrow \hat{w} - \alpha \cdot \nabla_{\hat{w}} \mathcal{L}(f_{\hat{w}})$
    $k \leftarrow k + 1$
end
```

It is important to set the learning rate α right. If we set the learning rate too low, then the algorithm would take many iterations to converge into a local minimum, on the other hand if we set the learning rate too high, then the loss would skip over the local minimum. See figure 3.8 for a visualization of learning rate.

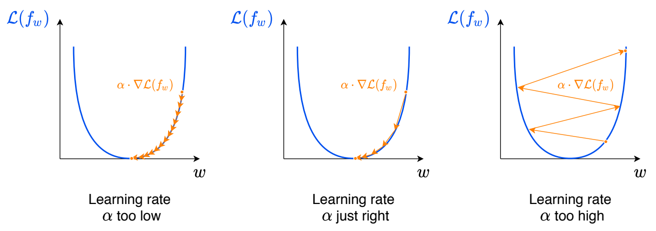

$\mathcal{L}(f_w)$
$\alpha \cdot \nabla\mathcal{L}(f_w)$
w
Learning rate α too low
$\mathcal{L}(f_w)$
$\alpha \cdot \nabla\mathcal{L}(f_w)$
w
Learning rate α just right
$\mathcal{L}(f_w)$
$\alpha \cdot \nabla\mathcal{L}(f_w)$
w
Learning rate α too high

Figure 3.8: Learning rate settings

3.3.3 Stochastic Gradient Descent

Since many loss functions are losses averaged over all training samples $\mathcal{T}^m$ and the gradient is a linear operation, the overall gradient of the loss function is the average gradient over all training samples, i.e.

$$\nabla_w \mathcal{L}(f) = \nabla_w \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(\mathbf{x}_i)) = \frac{1}{m} \sum_{i=1}^{m} \nabla_w \mathcal{L}(f(\mathbf{x}_i)).$$

We can treat the gradient computed on the entire training set as the true mean or expected value which we can approximate using an average of a smaller subset of $\mathcal{T}^m$ called a *minibatch*. This has several advantages and possible disadvantages:

- If we only use a subset of $\mathcal{T}^m$, then the computation of the gradient will be quicker, because we would be using less samples.

- The average gradient is still a good approximation of the true gradient with a relatively small number of samples. Remember from probability theory that the distribution of the average converges to a normal distribution, assuming the gradient has a finite expected value and finite variance. When constructing a confidence interval for the expected value, the probabilistic error of the average is a factor of $\frac{\sigma}{\sqrt{n}}$, where σ is the standard deviation of the gradient distribution and n is the number of samples used to calculate the average gradient. We can see that the error shrinks with diminishing returns as we increase the number of samples n (n increases linearly, the error shrinks with a rate of $\sqrt{n}$). [GBC16]

- The randomness of a minibatch also regularizes the network (makes it more accurate on unseen data, this is called *generalization*) [WM03] - random perturbations in the steps cause the learning algorithm to explore the parameter space more and is less likely to get stuck in local minima and saddle points. However, this leads to longer convergence times because SGD has to make more steps to converge. SGD never truly stabilizes due to the randomness, therefore it is important to tune the learning rate accordingly.

  In practice, it is common to linearly decay the learning rate with the number of iterations [GBC16] starting from a fixed learning rate α_{start} at iteration 1 and ending with a fixed learning rate α_{end} at iteration K, i.e.

$$\alpha_k = (1 - \min(1, \frac{k}{K})) \cdot \alpha_{\text{start}} + \min(1, \frac{k}{K}) \cdot \alpha_{\text{end}}.$$

Algorithm 2: Stochastic Gradient Descent

```
Input: training data $\mathcal{T}^m = \{(\mathbf{x}_i, y_i) \mid i \in \{1, \dots, m\}\}$
Input: neural network $f_w$ with parameters $w$
Input: learning rates $\alpha_k$
Output: neural network $f_{\hat{w}}$ with updated parameters $\hat{w}$ with a low
        $\mathcal{L}(f_{\hat{w}})$
$\hat{w} \leftarrow w$
$k \leftarrow 1$
while stopping criterion not met do
    for minibatch $p \in$ randomized near-equal-sized
     partitioning of $\mathcal{T}^m$ do
        $\nabla_{\hat{w}}\mathcal{L}(f_{\hat{w}}) \leftarrow$ backprop on $\mathcal{L}(f_{\hat{w}}(p))$
        $\hat{w} \leftarrow \hat{w} - \alpha_k \cdot \nabla_{\hat{w}}\mathcal{L}(f_{\hat{w}})$
    end
    $k \leftarrow k + 1$
end
return $f_{\hat{w}}$
```

In algorithm 2, we can see that SGD is quite similar to gradient descent with the difference that in each iteration, the true gradient is estimated with a minibatch of $\mathcal{T}^m$ and then the weights are updated with that estimated gradient. The minibatch is small random subset of $\mathcal{T}^m$ sampled without replacement, disjoint from other minibatches, or in other words a random partition of uniform size.

The size of the minibatches can range anywhere from 1 to $|\mathcal{T}^m|$. If it is 1, the algorithm is then called *Stochastic Gradient Descent*, if it is $|\mathcal{T}^m|$ then it is simply gradient descent or *Batch Gradient Descent*, if it is anything in-between then it is referred to as *Minibatch Gradient Descent*. The size of the minibatch is usually set to a power of 2 because today's tensor operations are done on GPUs which work well for power of 2 sized structures.

3.3.4 Momentum

The momentum method [Pol64], among other things, aims to reduce the number of steps needed to descend in the parameter space quickly in situations, where the gradient jumps from "cliff" to "cliff" in a "valley" configuration depicted in figure 3.9. The method simulates inertia of a moving object by accumulating velocity with an exponentially weighted average of previous gradients. The update portion of SGD with momentum has the form of

$$\mathbf{v} \leftarrow \mu\mathbf{v} + \alpha_k \cdot \nabla_{\hat{w}}\mathcal{L}(\hat{f}_{\hat{w}})$$

$$\hat{w} \leftarrow \hat{w} - \mathbf{v},$$

where $\mu \in \langle 0, 1\rangle$ is the exponentially weighted average or momentum parameter.

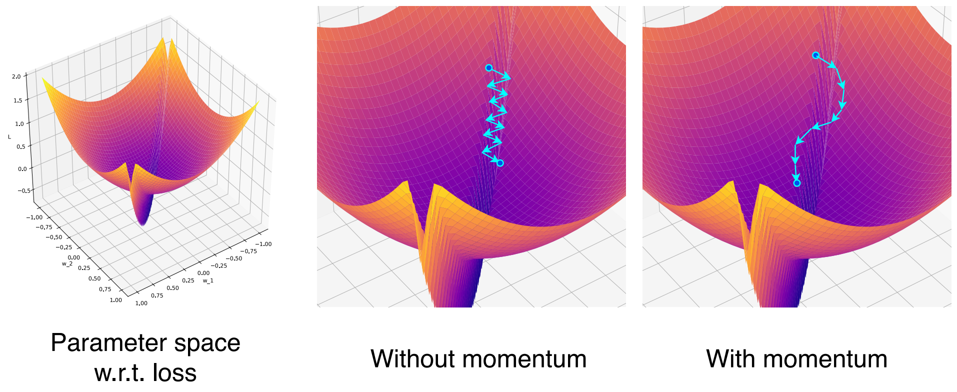


Figure 3.9: Valley situation and momentum

The momentum also somewhat mitigates the randomness of SGD by stabilizing the gradient with the exponentially averaged gradient.

3.3.5 Adam

Adam (adaptive moments) [KB14] is a widely used minibatch gradient descent method that re-scales (adapts) gradients during the course of training by dividing them by their second moment (element-wise squared values of the gradient). The motivation behind adaptive methods is that partial derivatives of different parameters may have different magnitudes so the re-scaling equalizes their importance. The magnitude of each element of a gradient is closely related to its learning rate, thus the learning rate of each parameter is automatically tuned in a sense during training. A parameter update in Adam is done like this

$$m_1 \leftarrow \rho_1 m_1 + (1-\rho_1)\mathcal{L}(\hat{f}_{\hat{w}})$$

$$m_2 \leftarrow \rho_2 m_2 + (1-\rho_2)\mathcal{L}(\hat{f}_{\hat{w}}) \cdot \mathcal{L}(\hat{f}_{\hat{w}})$$

$$m_1 \leftarrow \frac{m_1}{1-\rho_1^k}$$

$$m_2 \leftarrow \frac{m_2}{1-\rho_2^k}$$

$$\hat{w} \leftarrow \hat{w} - \alpha \frac{m_1}{\sqrt{m_2} + \text{small constant}},$$

where m_1 and m_2 are the first and second moment of the gradient respectively, ρ_1 and ρ_1 are their respective exponential moving average parameters, k is the iteration number, and the small constant is added for numerical stability. Notice that the moments are calculated using historical values so the momentum is built in.

Chapter 4

Modeling Graphons with Neural Networks Using Gradient-Based Learning

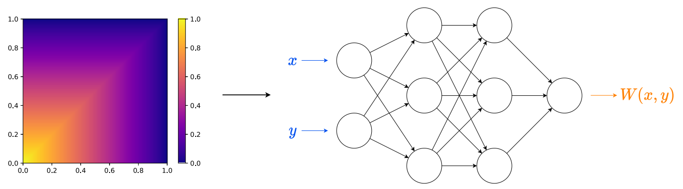


Figure 4.1: A neural network representing a graphon

In this chapter, we connect graphon theory with neural networks. Since graphons are just bounded functions on the unit square and neural networks are good function approximators, it is natural for us to try to represent graphons using neural networks.

The chapter begins with our proposed algorithm that learns the representation of a graphon using gradient descent, motivated by graphon theory. Then an analysis of the performance of our training algorithm and the neural network follows, that shows potential pitfalls of our approach. Lastly, several experiments, that aim improve our approach using various modifications and parameter tuning, are shown.

4.1 Gradient-Based Learning Algorithm Construction and Definition

This section introduces our learning algorithm for representing a graphon, the part original to this thesis.

4.1.1 Theoretical Motivation

Before we get into the algorithm itself, let us lay the theoretical motivations down first. In section 2.2.7, we've seen that homomorphism densities in fact identify graphons up to weak isomorphism. That is why we will require our neural network W to produce the same homomorphism densities as the homomorphism densities of the large graph G that we want to model, therefore we wish to minimize the discrepancy between ground truth homomorphism densities $t(F, G)$ and the densities produced by our network $t(F, W)$ for all finite graphs F. That gives rise to a loss function based on squared errors between the densities

$$\mathcal{L}(W) = \sum_F \frac{1}{2^{|V_F|}} (t(F,G) - t(F,W))^2.$$

The factor $\frac{1}{2^{|V_F|}}$ is there for convergence reasons to make the loss finite.

Of course, it is intractable to compute the densities for all finite graphs of which there is an infinite number. We will restrict ourselves to a few graphs of small size, say up to 6. A question whether that will compromise our learning somehow may arise. In the definition of sampling distance 2.2.10, the variation distances are weighted so that smaller graphs are more important which goes in our favor, however those weights are there just for convergence of the distance. Without computing all possible homomorphism densities, we cannot say that the resulting graphon that our network learns corresponds to the graphon representing the large graph G, but is it close enough? The Inverse Counting Lemma kind of hints at the importance of homomorphism densities for smaller graphs but not directly, see the counter example for that sort of use of the lemma 2.2.1. What it means for us is that we cannot rely solely on the density matching of smaller graphs. The question then remains open and it is to be answered by further experiments which utilize the graphon in various use cases like graph clustering, edge completion, graph anomalies etc. and whether the learned graphon actually provides useful and accurate information in those use cases.

4.1.2 Algorithm Construction

There are several problems that need to be solved in order to model graphons with a neural network:

1. A graphon is a real function on the unit square and our neural network has to conform to that format which means that the network has to accept two real inputs from $\langle 0,1\rangle^2$ and produce a single output in the range $\langle 0,1\rangle$. However, we are trying to match homomorphism densities which are not a direct output of the network, so it is not the classical setting of matching network output with ground truth via a training set of features and corresponding targets.
2. A graphon for an undirected large graph is supposed to be a symmetric function. How do we make the output of our network symmetric?
3. How do we obtain ground truth homomorphism densities?
4. How do we obtain homomorphism densities of our network?

Problem 2 is the easiest to solve. We can always sort the input pair of numbers (x, y) in ascending order, i.e. $x \leq y$, and model only on the upper triangle of the unit square $\langle 0,1\rangle$.

Let us address problems 3 and 4 of calculating homomorphism densities. To obtain ground truth homomorphism densities, we can for example compute them from an input induced subgraph G_{train} of the large graph G that we want to model if we get one, or we can compute them from the graph G itself if we are allowed to sample small induced subgraphs from it at least. There exist polynomial algorithms that count homomorphisms from graph F to a graph G for special types of graphs, for example we can just count the number of edges if $F = K_2$, however there is no polynomial algorithm for a general simple graph F to our knowledge [HN90], unless $P = NP$. To make the computation fast for general graphs, we relax on the exactness of the density value and approximate it via Monte Carlo Methods.

Homomorphism Density Approximation with Monte Carlo Methods

A homomorphism density $t(F, G)$ for finite graphs F and G can be expressed as the sum

$$t(F, G) = \frac{1}{|V_G|^{|V_F|}} \sum_{v_1=1}^{|V_G|} \sum_{v_2=1}^{|V_G|} \cdots \sum_{v_{|V_F|}=1}^{|V_G|} \prod_{\{i,j\} \in E_F} I(\{v_i, v_j\} \in E_G),$$

where $I(\text{condition})$ is an indicator function returning 1 if the condition in its argument is met, otherwise it returns 0. See 2.2.4 for the intuition. The indicator function I can be replaced by an adjacency matrix $A^G_{v_i,v_j}$ of graph G. We can view this sum as an expected value of a discrete distribution of a random variable X which is a function h of random variables $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{|V_F|} \in V_G = \{1, 2, ..., |V_G|\}$, that is

$$X = h(\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{|V_F|}) = \prod_{\{i,j\} \in E_F} I(\{\mathcal{V}_i, \mathcal{V}_j\} \in E_G).$$

The general form of the expected value of the variable X is

$$\mathbb{E}(X) = \sum_x x \cdot P(X = x)$$

and we can match that to the homomorphism density

$$\mathbb{E}(X) = \sum_x x \cdot P(X = x)$$

$$\downarrow$$

$$\sum_{v_1=1}^{|V_G|} \sum_{v_2=1}^{|V_G|} \cdots \sum_{v_{|V_F|}=1}^{|V_G|} h(v_1, v_2, \ldots, v_{|V_F|}) \cdot P(X = h(v_1, v_2, \ldots, v_{|V_F|}))$$

$$\downarrow$$

$$\sum_{v_1=1}^{|V_G|} \sum_{v_2=1}^{|V_G|} \cdots \sum_{v_{|V_F|}=1}^{|V_G|} \prod_{\{i,j\} \in E_F} I(\{v_i, v_j\} \in E_G) \cdot P(\mathcal{V}_1 = v_1, \mathcal{V}_2 = v_2, \ldots, \mathcal{V}_{|V_F|} = v_{|V_F|}).$$

By assuming that the probability $P(\mathcal{V}_1 = v_1, \mathcal{V}_2 = v_2, \dots, \mathcal{V}_{|V_F|} = v_{|V_F|})$ is uniform, which essentially means that all mappings are equally likely, we can set $P(\mathcal{V}_1 = v_1, \mathcal{V}_2 = v_2, \dots, \mathcal{V}_{|V_F|} = v_{|V_F|}) = \frac{1}{|V_G|^{|V_F|}}$, thus we get

$$\mathbb{E}_{\text{unif}}(X) = \frac{1}{|V_G|^{|V_F|}} \sum_{v_1=1}^{|V_G|} \sum_{v_2=1}^{|V_G|} \cdots \sum_{v_{|V_F|}=1}^{|V_G|} \prod_{\{i,j\} \in E_F} I(\{v_i, v_j\} \in E_G).$$

Now that we know that $t(F,G)$ is an expected value, we can approximate it with an unbiased and consistent estimator - the average of the summand with repeatedly sampling a realization $(v_1, v_2, \dots, v_{|V_F|})$ of the random vector $(\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_{|V_F|})$, where $\mathcal{V}_k \sim \text{unif}(1, |V_G|)$ for all $k \in V_F$, thus creating a set S_n of n random vector realizations:

$$\hat{t}(F,G) = \frac{1}{n} \sum_{(v_1, v_2, \dots, v_{|V_F|}) \in S_n} \prod_{\{i,j\} \in E_F} I(\{v_i, v_j\} \in E_G).$$

Each of the n samples is a random assignment of vertex numbers of G to variables $\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_{|V_F|}$, or in other words a random mapping of vertices from F to G. This probabilistic approximation method is called *Monte Carlo Integration*. See [Kat09] for more advanced approximation techniques. This way of estimating the homomorphism density might have been intuitive all along, because the product in it is just a check whether the mapping is a homomorphism from F to G with a binary outcome. In short, the homomorphism density is the probability of a random mapping of vertices from F to G being a homomorphism and we can estimate that probability by simply sampling a random mapping, checking if it is a homomorphism or not and average the results. This gives us a solution to obtaining ground truth homomorphism densities with two possible options:

- Either we get a subgraph G_{train} of G and approximate $t(F, G_{\text{train}})$ for a number finite graphs F of small size,
- or approximate $t(F,G)$ directly by randomly mapping vertices of F to random vertices of G, checking if it is a homomorphism, repeat average.

Regarding the approximation of the continuous version of the densities w.r.t. a graphon or neural network W, the procedure is very much analogous to the discrete one. The densities take the form of

$$t(F,W) = \int_{\langle 0,1 \rangle^{|V_F|}} \prod_{\{i,j\} \in E_F} W(x_i, x_j) \prod_{k \in V_F} dx_k.$$

Now we factorize the integrand into a function of a random vector

$$h(x_1, x_2, \dots, x_{|V_F|}) = \prod_{\{i,j\} \in E_F} W(x_i, x_j),$$

and its probability density function (PDF)

$$P(h(x_1, x_2, \dots, x_{|V_F|})) = p(x_1, x_2, \dots, x_{|V_F|}) = 1,$$

thus getting an expectation integral

$$\mathbb{E}_p(x_1, x_2, \ldots, x_{|V_F|}) = \int_{\langle 0,1\rangle^{|V_F|}} h(x_1, x_2, \ldots, x_{|V_F|}) \cdot p(x_1, x_2, \ldots, x_{|V_F|}) \prod_{k \in V_F} dx_k,$$

Again, assuming that we sample from the unit hypercube $\langle 0, 1\rangle^{|V_F|}$ uniformly, we can set the PDF to $p(x_1, x_2, \ldots, x_{|V_F|}) = 1$. Note that this uniform sampling is also in line with the way we sample graphs from a graphon. Altogether we get

$$\mathbb{E}_p(x_1, x_2, \ldots, x_{|V_F|}) = \int_{\langle 0,1\rangle^{|V_F|}} 1 \cdot \prod_{\{i,j\} \in E_F} W(x_i, x_j) \prod_{k \in V_F} dx_k$$

which can be approximated by the average

$$\hat{t}(F, W) = \frac{1}{n} \sum_{(x_1, x_2, \ldots, x_{|V_F|}) \in S_n} \prod_{\{i,j\} \in E_F} W(x_i, x_j),$$

where S_n is a set of n realizations of $(x_1, x_2, \ldots, x_{|V_F|})$ distributed accoring to p.

Accuracy of Homomorphism Density Approximations

We would like to put at least a probabilistic bound on the error of our approximations given the number samples. From the consistency of averages, we know that they converge to the true expected value and from the central limit theorem, we know that their distributions converge to a normal distribution, therefore we could construct a confidence interval for our density approximations, however that requires the knowledge of the variance of the summands or integrands of $\hat{t}$ which we don't have.

We can utilize Hoeffding's inequality which does not require the variance.

Theorem 4.1.1 (Hoeffding [Hoe14])**.** If $X_1, X_2, \ldots, X_n$ are independent and have the same expected value μ and $0 \le X_i \le 1$ for $i = 0, \ldots, n$ then for $0 < \varepsilon < 1 - \mu$

$$P(|\frac{1}{n} \sum_{i=1}^{n} X_i - \mu| \ge \varepsilon) \le e^{-2n\varepsilon^2}.$$

Given an error bound ε, we can express the probability that our approximation will stay within a margin of ε from the true densities

$$P(|\hat{t}_n - t| < \varepsilon)$$

and using Hoeffding's inequality, we can put a lower bound to this probability

$$P(|\hat{t}_n - t| < \varepsilon) = 1 - P(|\hat{t}_n - t| \ge \varepsilon) \ge 1 - e^{-2n\varepsilon^2}.$$

Additionally given a particular lower bound $c = 1 - e^{-2n\varepsilon^2}$ and solving for n, we get a minimum number of samples needed for our approximation to be within the error margin ε around the true value with probability at least c

$$n \ge \lceil -\frac{\ln(1-c)}{2\varepsilon^2} \rceil.$$

Density Discrepancy Back-Propagation

The Monte Carlo sampling approximation of homomorphism densities in 4.1.2 gives us a hint at how to deal with problem 1.

As shown in figure 4.2, to compute $\hat{t}(F, W)$, where F is a small graph and W is our neural network, we can generate n random assignments s_i of vertices V_F to $\langle 0, 1\rangle$ and create inputs for our neural network by listing edges E_F as pairs of real numbers $\{(s_i(u), s_i(v)) \mid \{u, v\} \in E_F\}$ while sorting $s_i(u)$ and $s_i(v)$ to only train on the upper triangle of the graphon making it symmetric. After computing $\hat{t}(F, W)$ and evaluating the loss function with it, we can back-propagate the gradient of the loss function w.r.t. to the approximated density which was in turn produced by the network W and some differentiable operations. Ultimately we can calculate the gradient w.r.t. the network's output which allows us to follow up with backprop as in the classical setting for each input sample, therefore we can compute the gradient of the loss w.r.t. W's parameters and update them with gradient descent.

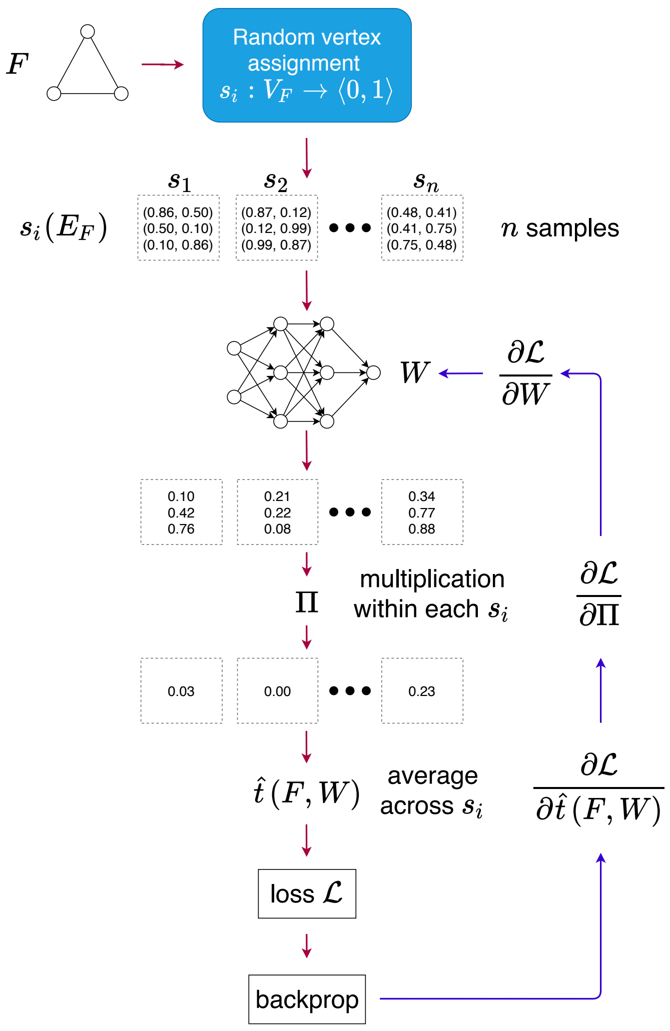


Figure 4.2: Back-propagation of homomorphism density discrepancies

Algorithm Outline

See algorithm 3, for the pseudo code that follows from the algorithm construction.

Algorithm 3: Gradient Descent for Graphons

```
Input: small finite graphs $\mathcal{F}$ to compute homomorphism densities with
Input: ground truth homomorphism densities $t(F,G)$, for each $F \in \mathcal{F}$
Input: $\hat{t}$ approximation error bound $\varepsilon$
Input: $\hat{t}$ approximation confidence probability $c$
Input: neural network $W_\theta$ with parameters $\theta$
Input: learning rates $\alpha_k$
Input: loss function $\mathcal{L}$
Input: stopping criterion $T$
Output: neural network $W_{\hat{\theta}}$ with updated parameters $\hat{\theta}$ with a low $\mathcal{L}$
1  $\hat{\theta} \leftarrow \theta$
2  $n \leftarrow \lceil -\frac{\ln(1-c)}{2\varepsilon^2} \rceil$
3  $k \leftarrow 1$
4  while $T$ not met do
5      for $F \in \mathcal{F}$ do
6          $S_n \leftarrow \emptyset$
7          for $i \in \{1,\dots,n\}$ do
8              $s_i \leftarrow$ generate random vertex mapping $V_F \to \langle 0,1 \rangle$
9              $S_n \leftarrow S_n \cup s_i$
           end
10         $\hat{t}(F, W_{\hat{\theta}}) \leftarrow 0$
11         for $s_i \in S_n$ do
12             $\hat{t}(F, W_{\hat{\theta}}) \leftarrow \hat{t}(F, W_{\hat{\theta}}) + \prod_{\{u,v\} \in E_F} W_{\hat{\theta}}(s_i(u), s_i(v))$
           end
13         $\hat{t}(F, W_{\hat{\theta}}) \leftarrow \frac{\hat{t}(F, W_{\hat{\theta}})}{n}$
       end
14     t $\leftarrow \{t(F,G) \mid F \in \mathcal{F}\}$
15     t$_W$ $\leftarrow \{\hat{t}(F, W_{\hat{\theta}}) \mid F \in \mathcal{F}\}$
16     loss $\leftarrow \mathcal{L}($t,t$_W)$
17     $\nabla_{\hat{\theta}}$loss $\leftarrow$ backprop on loss
18     $\hat{\theta} \leftarrow \hat{\theta} - \alpha_k \cdot \nabla_{\hat{\theta}}$loss
19     $k \leftarrow k + 1$
   end
20 return $W_{\hat{\theta}}$
```

The algorithm 3 above is technically SGD. Notice however that there are no minibatches. That is because the computation of $\hat{t}$ is already stochastic, thus we can control the "minibatch size" by different settings of n.

4.2 Implementation and Experiments Setup

This section serves as an overview of the specifics that went into the graphon modeling experiments which produced the results that we are going to discuss later in this chapter.

- **Hardware**

  The experiments were run on Google Colababoratory[1] GPU runtime in Jupyter[2] notebooks which provide 2 single-threaded 2.3 GHz CPU cores, 25 GB of RAM and an NVIDIA Tesla K80 GPU with 12 GB of VRAM. In our experiments, VRAM amount plays an important role because our experiments were GPU accelerated and GPU memory places direct constraints on n when approximating $\hat{t}$, see 4.1.2. It depends on neural network architecture and size but we set n to utilize the GPU memory to the maximum which was around $n = 10000$.

- **Software**

  The implementation was written in Python 3.6[3]. The optimized matrix manipulation NumPy[4] library was heavily used in conjunction with the PyTorch[5] library. The PyTorch library is perfect for our experiments because it manipulates tensors (basically multidimensional arrays) and tracks operations on them, thus building the computational graph of our neural network dynamically. With the computational graph built, we can simply call `loss.backward()` and the gradients w.r.t. our networks parameters are automatically computed based on the graph and stored. The Matplotlib[6] library used for visualizations. The NetworkX[7] library was used for auxiliary graph algorithms like isomorphism checking.

- **Source code** All of the source code can be found on GitHub[8].

- **Learning Details**

  We used synthetic data in our experiments, i.e. graphs sampled from graphons we defined ourselves. Examples of these graphons include complete bipartite graphon, constant graphon and growing uniform attachment graphon, see figure 4.3.

[1] `https://colab.research.google.com/`
[2] `https://jupyter.org//`
[3] `https://www.python.org/downloads/release/python-360/`
[4] `https://numpy.org/`
[5] `https://pytorch.org/`
[6] `https://matplotlib.org/`
[7] `https://networkx.github.io/`
[8] `https://github.com/honzahoang/nngraphons`

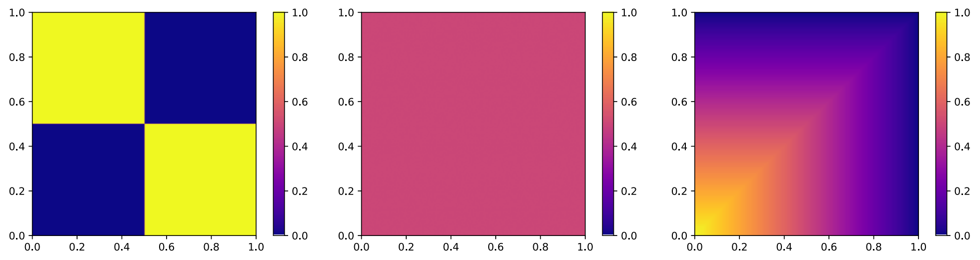


Figure 4.3: Complete bipartite graphon (left), constant graphon (middle), growing uniform attachment graphon (right)

We modeled each synthetic graphon separately by sampling a big training graph G_{train} of size $|V_{G_{\text{train}}}| = 10000$ from it. The sampling procedure is described in 2.2.3. Note that the vertices (and edge endpoints) of G_{train} were mapped randomly to integers so there is no way for the positional information of vertices to leak to the neural network. This was done to ensure that we are learning graphons only based on structure.

To calculate the ground truth homomorphism densities $\hat{t}(F, G_{\text{train}})$ and densities w.r.t. the network $\hat{t}(F, W)$, we approximated them with the method described in 4.1.2 for all possible isomorphism-unique finite graphs of size at most 6: $F \in \mathcal{F} = \{\text{simple graph } F \mid |V_F| \leq 6\}$. We used an error ε of 0.001 and confidence c of 0.95 for the approximation accuracy of the ground truth densities. Setting the value for ε is vague here without too much room for interpretation but generally the lower the better. We picked these values as a trade-off between accuracy and real time consumption. Tuning this parameter is advised for different use cases. We did not use Hoeffding's inequality to probabilistically assure an accuracy $\hat{t}(F, W)$ because our PyTorch implementation and hardware limited the number of samples n since it stores the computational graph on the GPU VRAM. However, beyond a certain threshold, n did not affect convergence of the loss function too much from our observations.

The neural network architectures used in the experiments will be mentioned and discussed in each experiment in section 4.3.

4.3 Results and Analysis

In this section, we go over the graphons that our neural networks managed to learn and analyze the learning convergence process.

4.3.1 Complete Bipartite Graphon

For this experiment we used a moderately deep network with 10 hidden layers with 64 neurons in each hidden layer. The input layer also consisted of 64 neurons accepting two real inputs. The output layer consisted of one neuron with a sigmoid activation function to ensure an output in $\langle 0, 1\rangle$. All of the neurons except the output one had a LeakyReLU activation function. All of the layers were fully connected. Weights of the network were initialized with Xavier-Glorot uniform initialization introduced in [GB10]. The mean squared error of homomorphism densities was used as the loss function -

$$\mathcal{L}_{\text{MSE}}(W) = \frac{1}{|\mathcal{F}|} \sum_{F \in \mathcal{F}} (\hat{t}(F, G_{\text{train}}) - \hat{t}(F, W))^2.$$

For the approximation of $\hat{t}(F, W)$, $n = 10000$ was used. The Adam optimizer from PyTorch was used as the gradient descent algorithm with a learning rate of 0.001. The stopping criterion was a loss value under 0.00001 that was chosen solely based on visual similarity to the ground truth graphon. The results of one particular run (Run 1) are visualized in figures 4.4, 4.5 and 4.6.

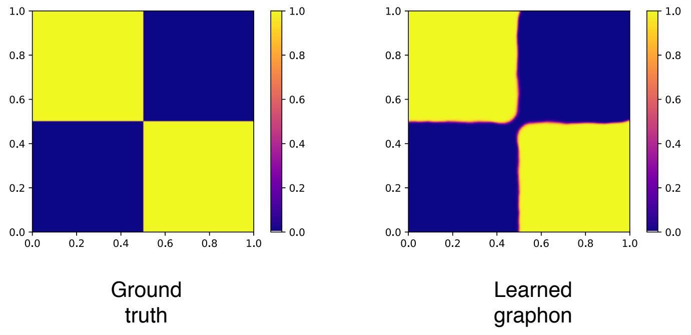


Figure 4.4: Run 1, Ground truth graphon and learned graphon

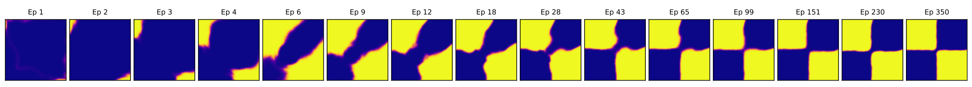


Figure 4.5: Run 1, Neural network development over the course of learning

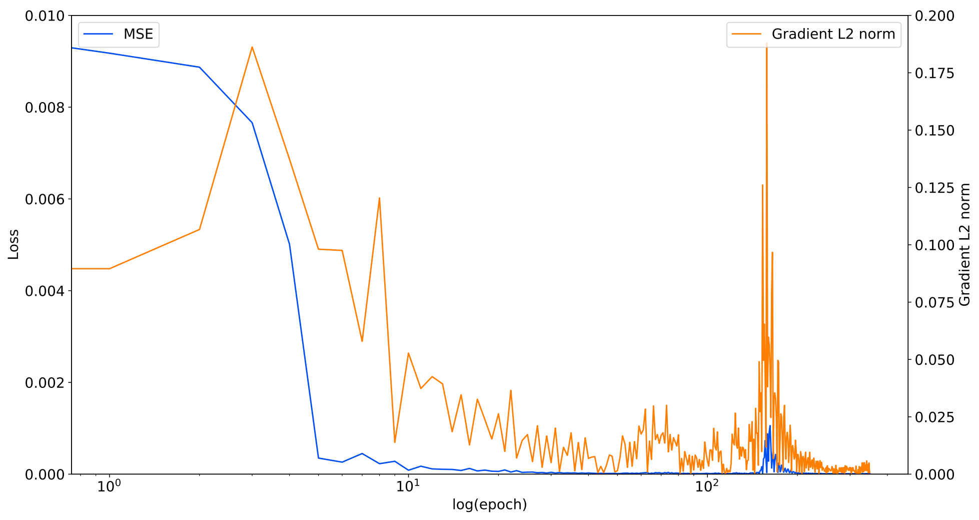


Figure 4.6: Run 1, Loss convergence with gradient norms

The network learned the complete bipartite graphon quite well and converged after around 350 epochs. The loss and gradient norm peaks shortly after the 100th epoch indicate a high learning rate problem, see figure 4.6. But the network converged nonetheless. We would like to note that this particular run was one of the lucky initialization ones, where the network converged quite quickly to demonstrate that it is capable of learning the graphon, however in some other runs the network encountered plateaus frequently or did not converge at all. See figure 4.25 to get an idea of the loss convergence distribution across multiple runs.

Let us have a look at squared errors of individual graphs F in figure 4.7 from the first run.

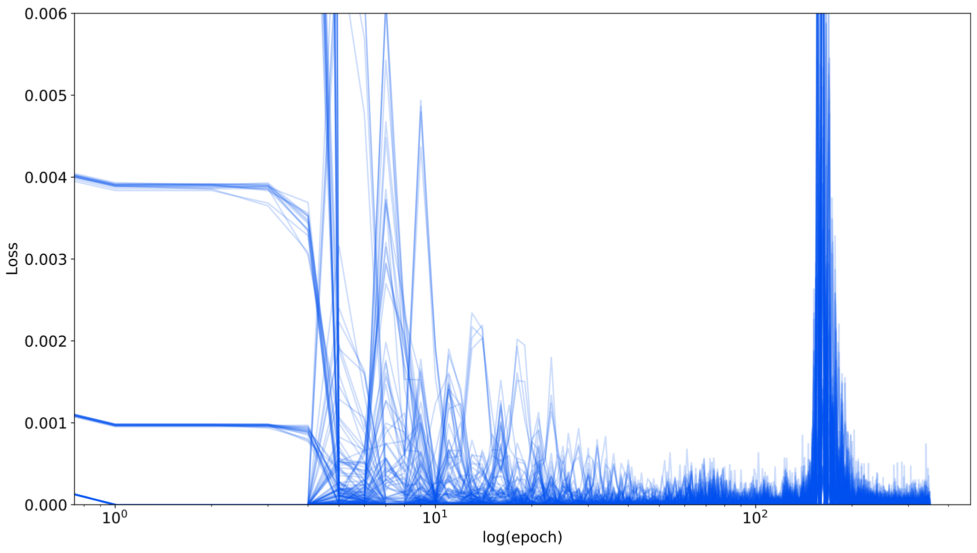


Figure 4.7: Run 1, Squared errors for individual graphs F

There are about 200 isomorphism-unique graphs up to size 6 ($|\mathcal{F}| = 200$). As you can see from the figure 4.7, only errors of few graphs F drag the entire average error up during training. That is why we tried to change loss function for a different run (Run 2) to

$$\mathcal{L}_{\text{maxSE}}(W) = \max_{F \in \mathcal{F}} (\hat{t}(F, G_{\text{train}}) - \hat{t}(F, W))^2$$

in order to push the maximum error down. That modification gave us a hint to another problem that occurred during training where the learning algorithm decreased the error for one graph and increased the error for other graphs since the gradient propagated only for the graph with the largest error, see figure 4.8. A similar effect can be seen in the individual discrepancies for $\mathcal{L}_{\text{MSE}}$ in lesser extent.

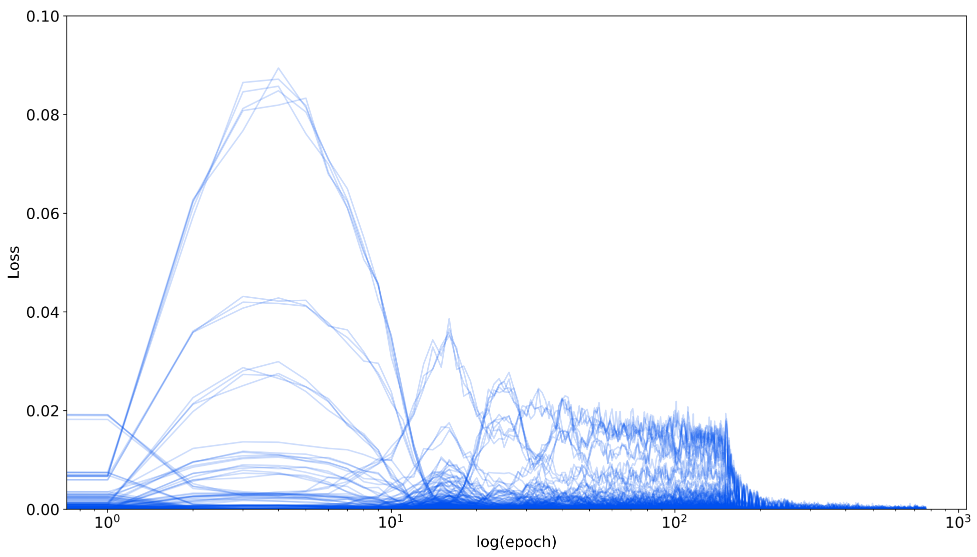


Figure 4.8: Run 2, Squared errors for individual graphs F when training with $\mathcal{L}_{\text{maxSE}}$, learning rate $= 0.001$

In some runs like (Run 3), the errors do not seem to converge for $\mathcal{L}_{\text{maxSE}}$ even though the gradient norm is nowhere near zero, see figure 4.9. This rules out the cases where the learning algorithm would get stuck in local minima or saddle points (configurations with small gradient norms).

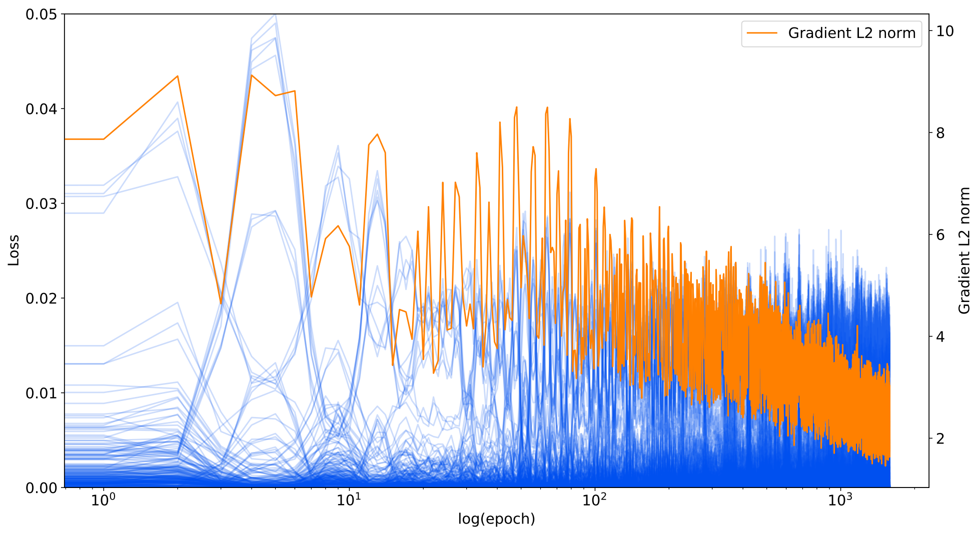


Figure 4.9: Run 3, Oscillating individual graphs F errors when training with $\mathcal{L}_{\text{maxSE}}$

This might still be a case of a high learning rate but we lowered it for Run 3 and when we visually inspect the network graphon in figure 4.10, it does not really change that much which leads us to believe that the learning rate is fine.

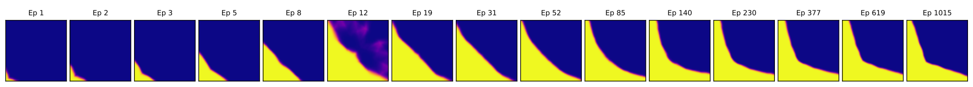


Figure 4.10: Run 3, Non-converging network visualization during training

This tug-of-war problem between density discrepancies of different F's was also somewhat observed in another run (Run 4) for $\mathcal{L}_{\text{MSE}}$, seen in figure 4.11. This run also displays a long plateau which was finally overcome at the end of the training. During the plateau, a small gradient norm can be seen which indicates either of local minimum, saddle point, or a vanishing gradient.

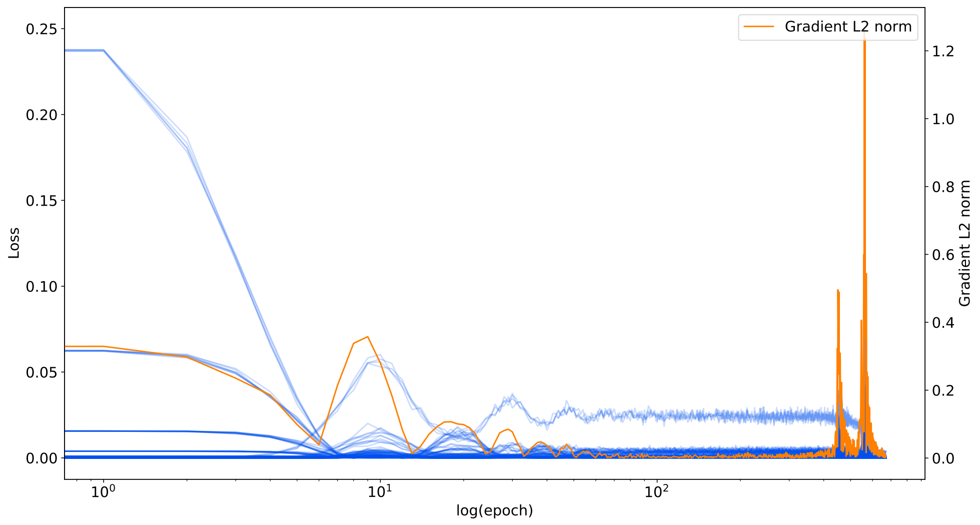


Figure 4.11: Run 4, Squared error of some graphs F plateau

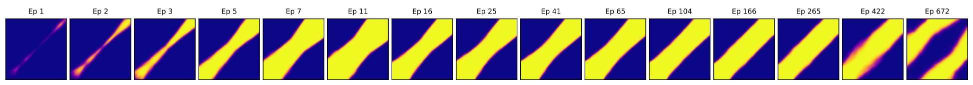


Figure 4.12: Run 4, Neural network development over the course of learning

4.3.2 Growing Uniform Attachment Graphon

In this experiment, we used the same architecture as in 4.3.1 for Run 1.

The results can be seen in figures 4.13, 4.14, and 4.15.

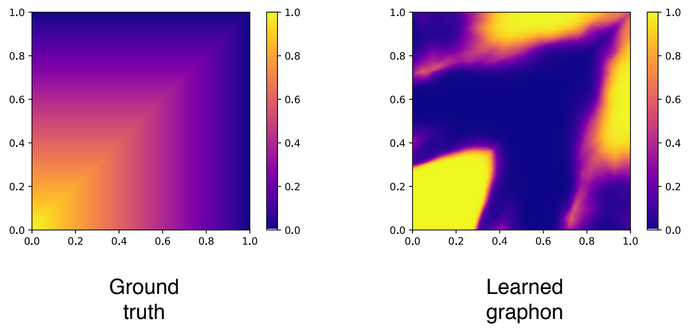


Figure 4.13: Run 1, Ground truth and learned graphon

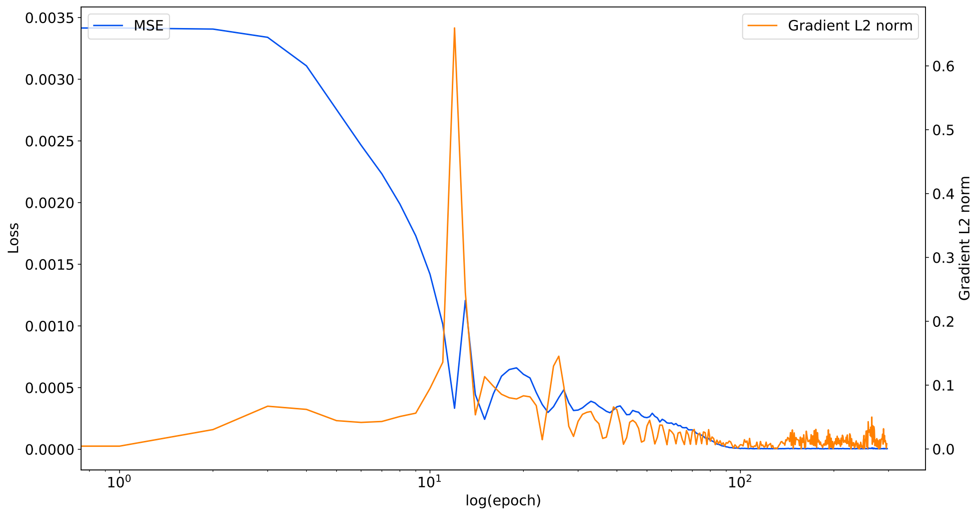


Figure 4.14: Run 1, MSE loss convergence

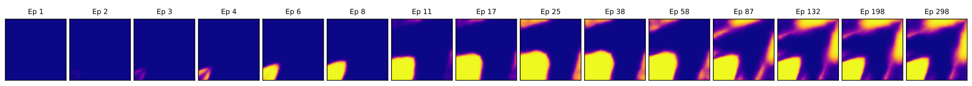


Figure 4.15: Run 1, Neural network development over the course of learning

One thing to note here is that the resulting graphon does not really look like the ground truth one, however the loss is still small so this might be a case of weak isomorphism.

Interestingly, when in Run 2 we adjusted the architecture to be much simpler to 2 hidden layers with 32 neurons each, the resulting graphon converged even quicker and resembled the ground truth more from the visual perspective. See the results of the second run 4.16, 4.17, and 4.18.

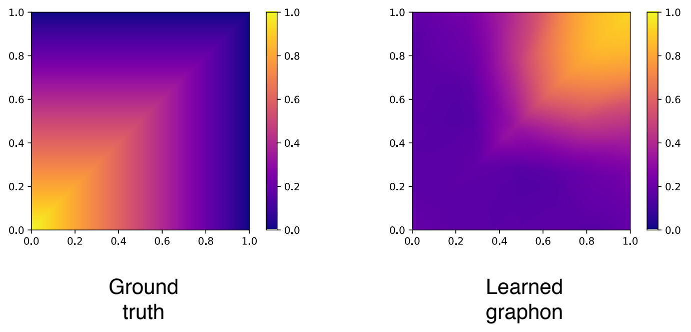


Figure 4.16: Run 2, Ground truth and learned graphon

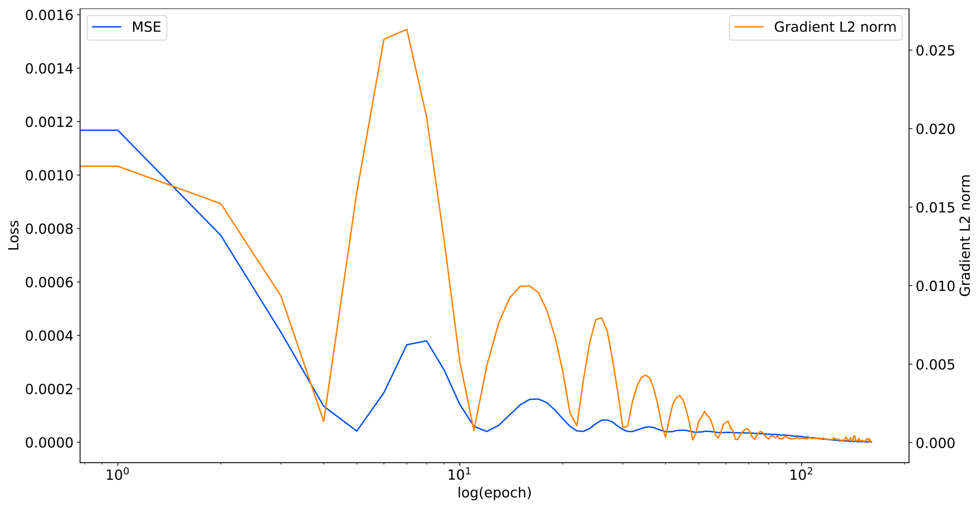


Figure 4.17: Run 2, MSE loss convergence

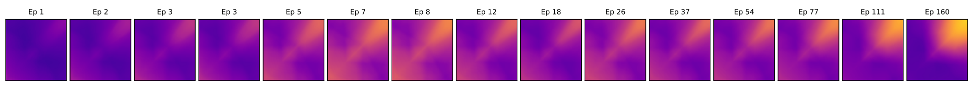


Figure 4.18: Run 2, Neural network development over the course of learning

4.3.3 Constant 0.5 Graphon

This graphon is the simplest in terms of function complexity, although modeling it might reveal if homomorphism densities for small graphs F overfit on basically a noise graph and if they model the unwanted noise.

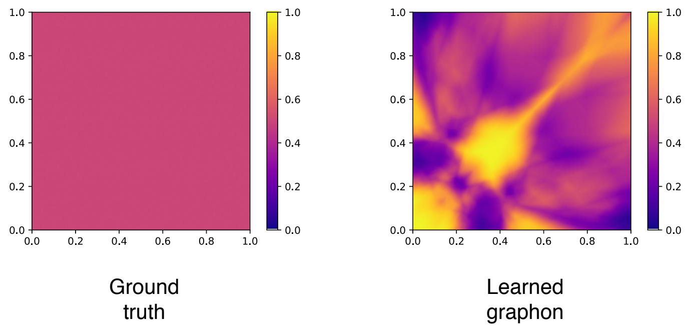


Figure 4.19: Ground truth and learned graphon

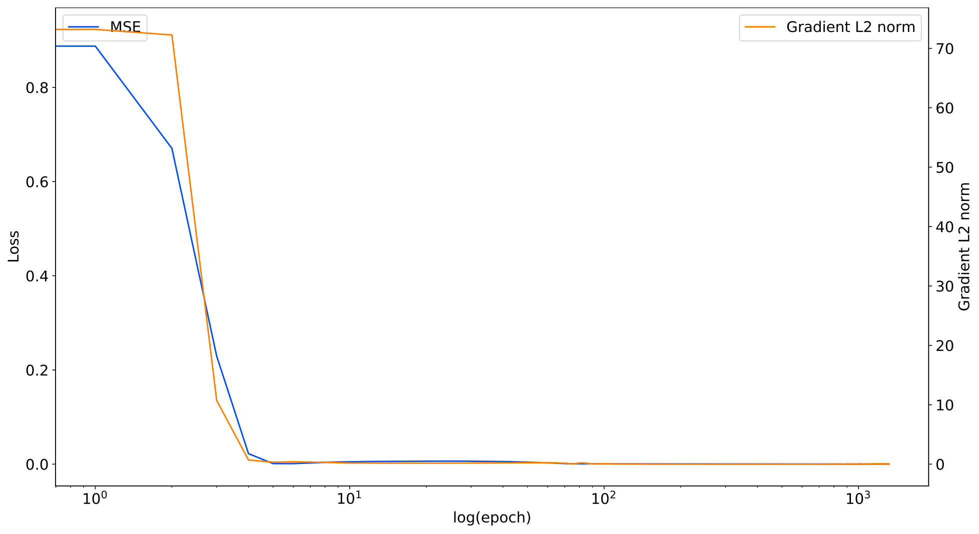


Figure 4.20: MSE loss convergence

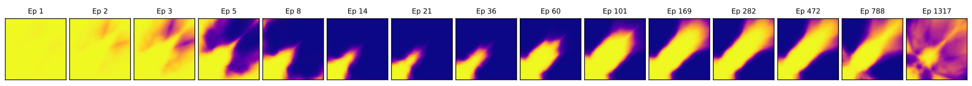


Figure 4.21: Neural network development over the course of learning

The convergence plot 4.20 tells us that there were no major problems with learning. The resulting learned graphon in 4.19 is a little bit suspicious because it does contain quite a big clique or almost a clique in there. This, again, might be due to weak isomorphism but it should be very unlikely for a random graph to contain such a big clique. We hoped to try to use our learned graphons for clustering based on subset connectedness and cluster notions for graphons introduced in [EBW16]. However, looking at our learned graphon, the subset $\langle 0.3, 0.5\rangle$ would definitely be in one cluster at around level 0.9 which means that that set of "vertices" is highly connected. Such connectedness for a random graph with edge probability 0.5 should be very unlikely which is why we didn't delve into clustering yet.

4.4 Improvement Experiments

This section briefly describes improvement attempts to our base algorithm together with their results.

4.4.1 Multiple-Gradient Descent Algorithm

To try to mitigate the oscillations of $\mathcal{L}_{\text{maxSE}}$ and to some extent of $\mathcal{L}_{\text{MSE}}$, we have applied the Multiple-Gradient Descent Algorithm (MGDA) [Dés12] which is an algorithm that, given gradients $\nabla\mathcal{L}_i(W)$ w.r.t. multiple different objectives $\mathcal{L}_i$, finds a common steepest direction of descent that optimizes at least one objective without worsening the others. This is exactly what we need when we set $\mathcal{L}_i = (\hat{t}(F_i, G_{\text{train}}) - \hat{t}(F_i, W))^2$. The algorithm works by finding the minimum norm element ω^* in a convex hull of the gradients $\nabla\mathcal{L}_i(W)$ and then using it to update the weights $\hat{\theta} \leftarrow \hat{\theta} - \alpha_k \cdot \omega^*$. We approximately solved the optimization task

$$\omega^* = \arg\min_{w} ||\omega||_2$$

$$\omega = \sum_{i=1}^{k} \beta_i \cdot \nabla\mathcal{L}_i(W)$$

$$\beta_i > 0, \forall i \in \{1, \ldots, k\} \qquad \sum_{i=1}^{k} \beta_i = 1$$

with a basic genetic algorithm. The network parameters then converge to a so called Pareto-stationary point which means that $\omega^* = \mathbf{0}$. Unfortunately, this descent algorithm almost always plateaus with a high $\mathcal{L}_{\text{maxSE}}$ by finding an $\omega^* = \mathbf{0}$ or very close to the zero vector. That suggests that the network is usually already in a Pareto-stationary configuration when it plateaus.

4.4.2 Skip Connections and Batch Normalization

Skip connections, popularized by residual networks or ResNets [HZRS16], are connections in a deep network that connect the output of a layer to an input of another while skipping one or more layers in between, see figure 4.22.

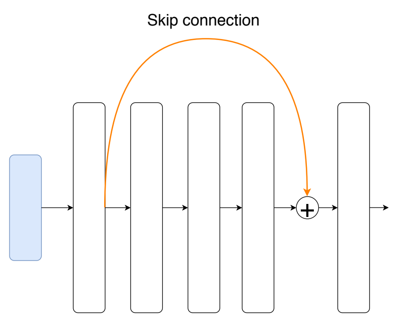


Figure 4.22: Skip connection adding an output of a layer to the output of another

Inspired by skip connections, we modified the hidden layers of our architecture from 4.3.1 to become residual blocks, i.e. the input to hidden layer x was added to its output $h(x)$:

$$h_{\text{residual}}(x) = h(x) + x.$$

It is called a residual block because it adds an output of a function to the input x, thereby modeling what is left in x to model - the residual. Skip connections help with the vanishing gradient problem by introducing shortcuts along which the gradient can flow back to even distant layers from the output of the network.

After this modification, an issue arose where the sigmoid activation at the end of our network became saturated very quickly and the graphon put out either 0 almost everywhere or 1 almost everywhere. This was most probably caused by better gradient propagation which caused the distribution of hidden layer outputs to shift drastically. To correct for this problem, we added a batch normalization [IS15] layer after the last hidden layer which normalizes its output to have a mean of 0 and a variance of 1. This keeps the majority of the input into the sigmoid centered around 0, thus preventing saturation. Otherwise, the architecture stayed the same - 10 hidden layers 64 neurons each, LeakyReLU activations.

We trained this modified network on the complete bipartite graphon for several runs with different random seeds and it does seem to be more robust against plateaus. See figures 4.23 and 4.24 for an example run.

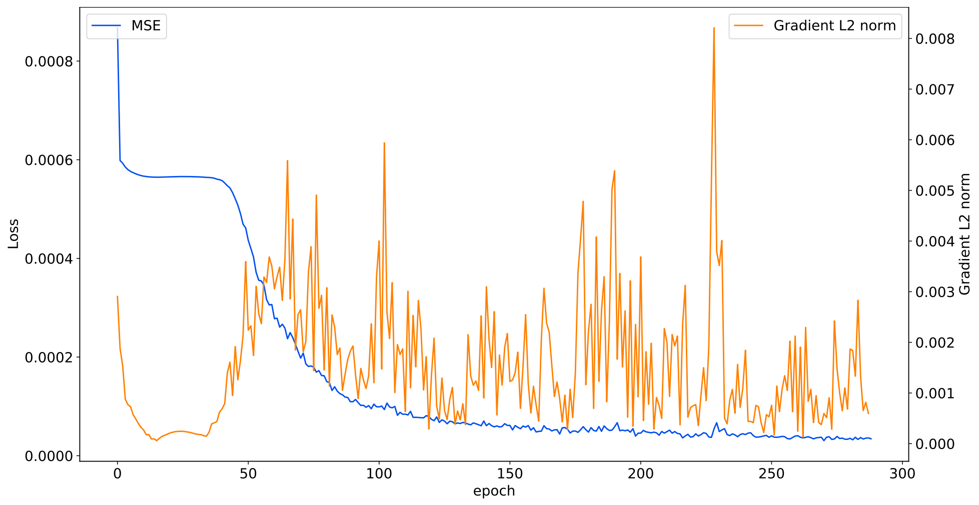


Figure 4.23: MSE loss convergence

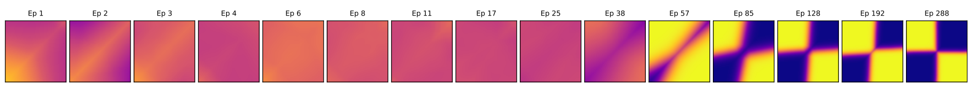


Figure 4.24: Neural network development over the course of learning

Figure 4.25 shows the $\mathcal{L}_{\text{MSE}}$ distribution statistics across 50 random training runs on the complete bipartite graphon (random in the sense of different seeds for weight initialization, $\hat{t}$ approximation etc.). As seen in the figure, the skip connections together with batch normalization improved not only the initial loss, but also the loss convergence in the average case without plateauing.

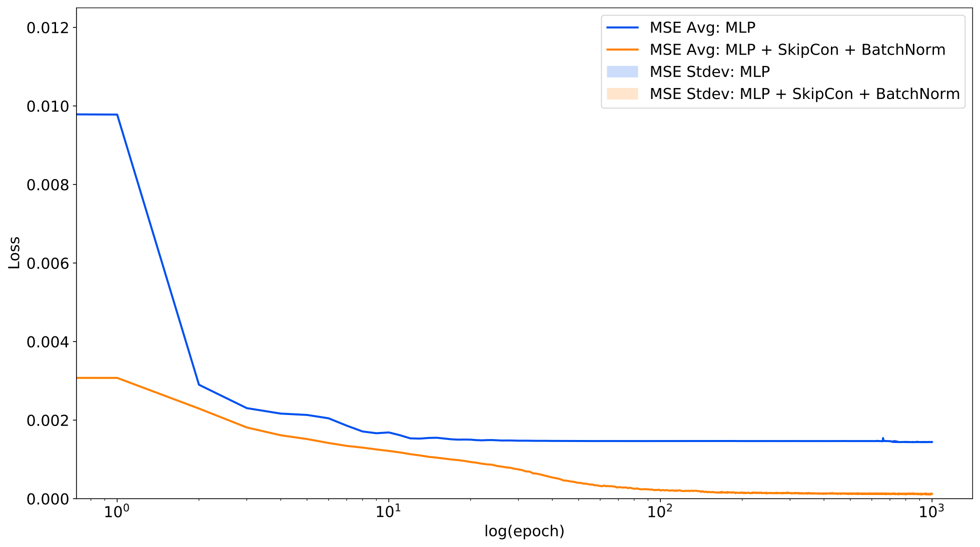


Figure 4.25: $\mathcal{L}_{\text{MSE}}$ average and standard deviation over 50 different runs with different seeds improvement

4.5 Auxiliary Results

In this section, we compare the learned graphons to the ground truth graphons by comparing parameters that were not explicitly learned.

We sampled two graphs of size 5000 vertices each, from the ground truth graphon and the best learned graphon respectively and compared the vertex degree distribution of the sampled graphs.

4.5.1 Complete Bipartite Graphon

As seen in figure 4.26, the ground truth degrees are concentrated in two values - the two vertex partitions. Each node in each partition is connected to every node in the other partition. Due to the randomness of sampling, the partitions are not equally sized which is why the degrees concentrate around two values instead of one. The values correspond to the sizes of the two partitions. The vertex degrees of the learned graphon concentrate around the value 2500 which is the size of one of the partitions they were equally sized. We conclude that the degree distribution was learned quite well in this case.

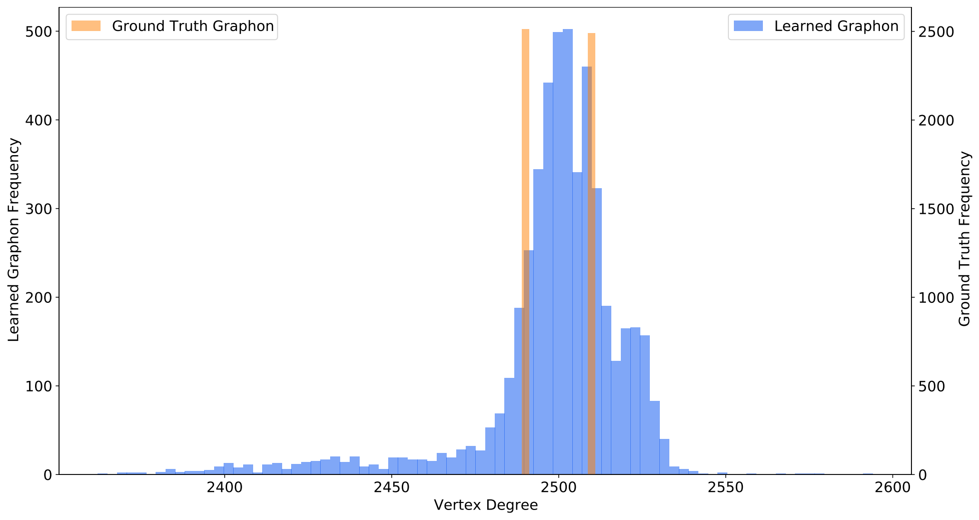


Figure 4.26: Degree Distribution Comparison

4.5.2 Growing Uniform Attachment Graphon

Figure 4.27 shows that the degree distribution of the growing uniform attachment graphon was learned decently. The network didn't quite capture low degree vertices. The graphon was learned by an MLP with skip connections and batch normalization with 1 hidden layer with 8 neurons.

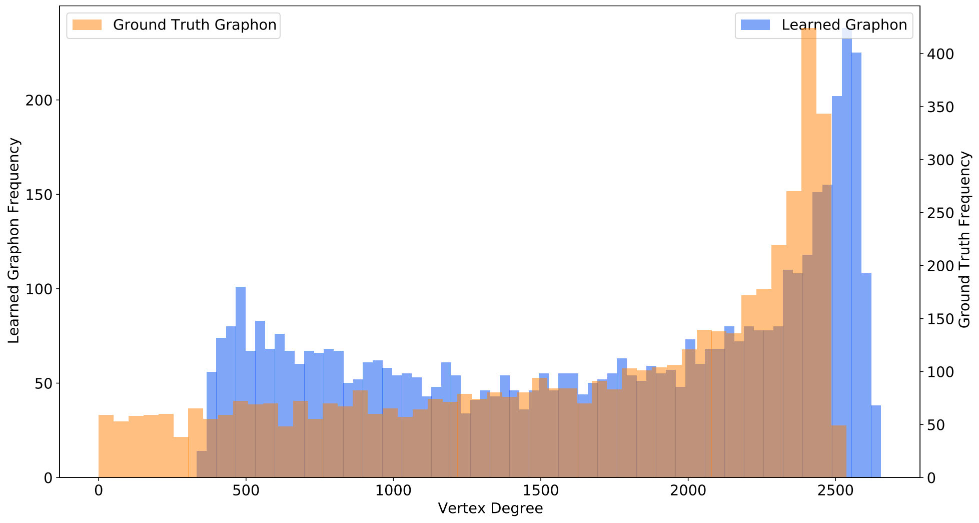


Figure 4.27: Degree Distribution Comparison

4.5.3 Constant 0.5 Graphon

In figure 4.28, the ground truth distribution looks like a normal distribution with a mean of 2500, a vertex is on average connected with about half of the other ones. The learned graphon distribution has the highest mode also around 2500 but it is roughly bi-modal with a heavy left tail and with a concentration around 2800 which is probably the clique mentioned in 4.3.3.

The degree distribution was not learned well in this case. This would be fine if the loss for this graphon was somewhat high but it was very low $(2 \cdot 10^{-6})$ which may be due to the homomorphism densities for the few graphs $\mathcal{F}$ not being enough to identify the ground truth graphon.

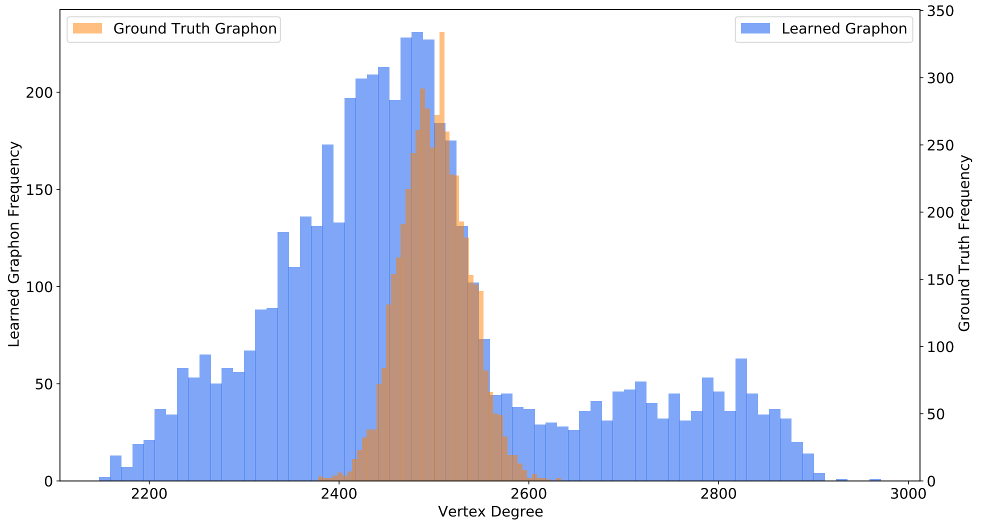


Figure 4.28: Degree Distribution Comparison

Chapter 5

Conclusion

To conclude the thesis, let us wrap up what we have done first. We went over graphon theory that introduces the graphon, an object representing the limit of graph sequences that converge structurally. Convergence in structure is encoded in homomorphism densities with which we train a neural network to represent a graphon. To that end, we defined a gradient-based learning algorithm, that minimizes the squared errors between the ground truth homomorphism densities and densities produced by the trained neural network. We then analyzed the resulting representations and convergence of the training process.

Now let's summarize the results. While some of the representations were promising, the results as a whole are far from perfect. The complete bipartite graphon was learned well visually and in terms of vertex degree distribution which is impressive that this was done only based on homomorphism densities. The same thing cannot be said about the constant graphon that had a big mismatch in degree distribution.

The high-level goal of this thesis was to explore this direction of trying to utilize the learning capability neural networks to model large graphs structure with graphons. The overall conclusion is that it is definitely possible, at least for some types of graphons, but the modeling process requires some tuning for different graphons.

We would like to mention potential future directions to take to improve on this kick-off. The code could definitely be optimized so that homomorphism densities could be iteratively computed without the GPU VRAM bottleneck. Different network architectures from computer vision like convolutional layers could be modified and adapted to fit this problem. The learning process and network architecture should be revisited to make them more robust against plateaus. A learning process based on homomorphism frequencies could be looked at to model sparse graphs.

Bibliography

[BCL$^+$08] Christian Borgs, Jennifer T Chayes, László Lovász, Vera T Sós, and Katalin Vesztergombi, *Convergent sequences of dense graphs i: Subgraph frequencies, metric properties and testing*, Advances in Mathematics **219** (2008), no. 6, 1801–1851.

[Dés12] Jean-Antoine Désidéri, *Multiple-gradient descent algorithm (mgda) for multiobjective optimization*, Comptes Rendus Mathematique **350** (2012), no. 5-6, 313–318.

[Drc] Jan Drchal, *Statistical Machine Learning course slides, Faculty of Electrical Engineering at Czech Technical University in Prague*, `https://cw.fel.cvut.cz/b181/_media/courses/be4m33ssu/anns_ws18.pdf`, accessed on 2020-08-10.

[EBW16] Justin Eldridge, Mikhail Belkin, and Yusu Wang, *Graphons, mergeons, and so on!*, Advances in Neural Information Processing Systems, 2016, pp. 2307–2315.

[GB10] Xavier Glorot and Yoshua Bengio, *Understanding the difficulty of training deep feedforward neural networks*, Proceedings of the thirteenth international conference on artificial intelligence and statistics, 2010, pp. 249–256.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio, *Deep sparse rectifier neural networks*, Proceedings of the fourteenth international conference on artificial intelligence and statistics, 2011, pp. 315–323.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016, `http://www.deeplearningbook.org`.

[GGR98] Oded Goldreich, Shari Goldwasser, and Dana Ron, *Property testing and its connection to learning and approximation*, Journal of the ACM (JACM) **45** (1998), no. 4, 653–750.

[Gla16] Daniel Glasscock, *What is a graphon?*, arXiv preprint arXiv:1611.00718 (2016).

[HN90] Pavol Hell and Jaroslav Nešetřil, *On the complexity of h-coloring*, Journal of Combinatorial Theory, Series B **48** (1990), no. 1, 92–110.

[Hoe14] Wassily Hoeffding, *Probability inequalities for sums of bounded random variables*, Wiley StatsRef: Statistics Reference Online (2014).

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, Proceedings of the IEEE international conference on computer vision, 2015, pp. 1026–1034.

[HZRS16] ______, *Deep residual learning for image recognition*, Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[IS15] Sergey Ioffe and Christian Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, arXiv preprint arXiv:1502.03167 (2015).

[Kat09] Helmut G. Katzgraber, *Introduction to monte carlo methods*, arXiv preprint arXiv:0905.1629 (2009).

[KB14] Diederik P Kingma and Jimmy Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980 (2014).

[Lov12] László Lovász, *Large networks and graph limits*, vol. 60, American Mathematical Soc., 2012.

[MHN13] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng, *Rectifier nonlinearities improve neural network acoustic models*, Proc. icml, vol. 30, 2013, p. 3.

[Pol64] Boris T. Polyak, *Some methods of speeding up the convergence of iteration methods*, USSR Computational Mathematics and Mathematical Physics **4** (1964), no. 5, 1–17.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams, *Learning representations by back-propagating errors*, nature **323** (1986), no. 6088, 533–536.

[Ros60] Frank Rosenblatt, *Perceptron simulation experiments*, Proceedings of the IRE **48** (1960), no. 3, 301–309.

[WM03] D. Randall Wilson and Tony R. Martinez, *The general inefficiency of batch training for gradient descent learning*, Neural networks **16** (2003), no. 10, 1429–1451.

Appendix A

List of Used Abbreviations

- **ANN** Artificial Neural Network
- **CPU** Central Processing Unit
- **DAG** Directed Acyclic Graph
- **DFFNN** Deep Feed-Forward Neural Network
- **GHz** Giga Hertz
- **GPU** Graphic Processing Unit
- **LeakyReLU** Leaky Rectified Linear Unit
- **MGDA** Multiple-Gradient Descent Algorithm
- **MLP** Multi-Layer Perceptron
- **MSE** Mean Squared Error
- **PDF** Portable Document Format
- **PReLU** Parametric Rectified Linear Unit
- **RAM** Random Access Memory
- **ReLU** Rectified Linear Unit
- **SGD** Stochastic Gradient Descent
- **VRAM** Video Random Access Memory
- **XOR** Exclusive OR
- **maxSE** Maximum Squared Error

Appendix B

Contents of the Attached Memory Card

```
/
├── readme.txt ..... Text file describing contents of the files
├── src
│   ├── nngraphons ........ Python package with all learning code
│   └── notebooks .............. Experiments in Jupyter notebooks
└── thesis
    ├── vuhuyhoa_thesis.pdf ................... This thesis in PDF
    └── src ....................... LaTeX source code of this thesis
```