Relating graph auto-encoders to linear models

Solveig Klepper *solveig.klepper@uni-tuebingen.de*
Department of Computer Science and Tübingen AI Center
University of Tübingen

Ulrike von Luxburg *ulrike.luxburg@uni-tuebingen.de*
Department of Computer Science and Tübingen AI Center
University of Tübingen

Reviewed on OpenReview: *https://openreview.net/forum?id=Y1eYplvxrE*

Abstract

Graph auto-encoders are widely used to construct graph representations in Euclidean vector spaces. However, it has already been pointed out empirically that linear models on many tasks can outperform graph auto-encoders. In our work, we prove that the solution space induced by graph auto-encoders is a subset of the solution space of a linear map. This demonstrates that linear embedding models have at least the representational power of graph auto-encoders based on graph convolutional networks. So why are we still using nonlinear graph auto-encoders? One reason could be that actively restricting the linear solution space might introduce an inductive bias that helps improve learning and generalization. While many researchers believe that the nonlinearity of the encoder is the critical ingredient towards this end, we instead identify the node features of the graph as a more powerful inductive bias. We give theoretical insights by introducing a corresponding bias in a linear model and analyzing the change in the solution space. Our experiments are aligned with other empirical work on this question and show that the linear encoder can outperform the nonlinear encoder when using feature information.

1 Introduction

Many real-world data sets have a natural representation in terms of a graph that illustrates relationships or interactions between entities (nodes) within the data. Extracting and summarizing information from these graphs is the key problem in graph learning, and representation learning is an important pillar in this work. The goal is to represent/encode/embed a graph in a low-dimensional space to apply machine learning algorithms to solve a final task.

In recent years neural networks and other approaches that automate learning have been employed as a powerful alternative to leverage the structure and properties of graphs (Tang et al., 2015; Grover & Leskovec, 2016; Ribeiro et al., 2017; Dong et al., 2017; Hamilton et al., 2017a; Chamberlain et al., 2017; Cao et al., 2016; Wang et al., 2016; Chang et al., 2015). For example, Hamilton et al. (2017b) review key advancements in this area of research, including graph convolutional networks (GCNs). Graph auto-encoders (GAEs), first introduced in Kipf & Welling (2016b), are based on a graph convolutional network (GCN) architecture. They have been heavily used and further refined for representation learning during the past couple of years (see Li et al. (2021); Pan et al. (2018); Vaibhav et al. (2019); Davidson et al. (2018), to name a few). Recently, Salha et al. (2020) discovered that the original GAE architecture, which has been used as a basis for several enhancements, can be outperformed by simple linear models on an exhaustive set of tasks. The equivalence or even superiority of (almost) linear models has also been empirically shown for standard GCN architectures for link prediction and community detection (Wu et al., 2019). The question arises as to why we still use and improve these models when simpler models with comparable performance exist. What is the theoretical relation between linear and nonlinear encoders, and does the empirical work provide a reason to replace

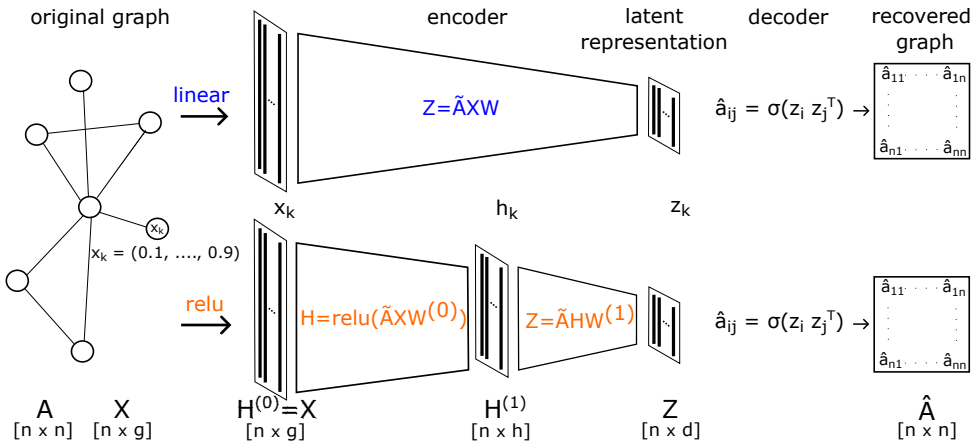

Figure 1: The architecture of a linear auto-encoder and a relu auto-encoder. The only difference is the encoder function: The linear encoder is a simple linear map, whereas the relu encoder is a two-layer GCN network with relu activation in the first layer and linear output activation. In both cases, the input is the graph (A, X). $H^{(1)}$ is the hidden layer of the GCN. A row in the matrix Z gives the latent representation of the respective row/node in the input matrix X. In both cases the decoder is the inner product, and the target is to recover the adjacency matrix A.

graph convolutional networks with linear equivalents? In this paper, we contribute to understanding the underlying inductive bias of the graph auto-encoder model.

Contributions: (1) We draw a theoretical connection between linear and non-linear encoders and prove that, under mild assumptions, for any function $f(A, X)$ on a graph, there exists a linear encoder that can achieve the same training loss. We use this result to show that the representational power of relu encoders is, at most, as large as the one of linear encoders. (2) We investigate the bias that is relevant for good generalization and give theoretical insights into how features influence the solution space of a linear model. (3) We empirically find that the nonlinearity in the relu encoder is not the critical ingredient for generalization. Indeed we discover that if the graph features are taken into account appropriately, the linear model outperforms the relu model test data.

2 Preliminaries

We consider a connected, unweighted and undirected graph G with adjacency matrix $A \in \{0,1\}^{n \times n}$ and a feature matrix $X \in \mathbb{R}^{n \times g}$. The feature matrix X contains g-dimensional feature vectors that describe additional properties of each node of the graph. For example, if the graph consists of a social network, the features could describe additional properties of the individual persons, such as age and income. If we do not have any feature information for the nodes ("featureless graph"), then we set X as the $n \times n$ identity matrix. We always assume that all nodes are connected to themselves, that is, $a_{ii} = 1$ for all i. We call $D \in \mathbb{N}^{n \times n}$ the degree matrix with diagonal entries $d_{ii} = \sum_{j=1}^{n} A_{ij}$ and 0 everywhere else, and $\tilde{A} = D^{-1/2} A D^{-1/2}$ the symmetrically normalized adjacency matrix.

Graph convolutional networks. Inspired by convolutional neural networks, graph convolutional networks (GCNs, Kipf & Welling (2016a)) are one of the most widely used graph neural network architectures. The input consists of a graph (A, X). The graph convolution is applied to a matrix $\tilde{A}$ that encodes the adjacency structure of the graph. This matrix $\tilde{A}$, called diffusion matrix, is the symmetrically normalized adjacency matrix in our case but could be replaced by any matrix encoding the graph structure, for example, the graph Laplacian. In each layer of the network, a node in the graph updates its latent representation, aggregating feature information within its neighbourhood, and learns a low-dimensional representation of the graph and its features. More precisely, in layer l, each node update is some function of the diffusion of the weighted

features: $H^{(l+1)} = \psi(\tilde{A}H^{(l)}W^{(l)})$ with learnable weights $W^{(l)}$, (nonlinear) activation function ψ and $H^{(0)}$ as the feature matrix X. The output of the GCN for each node in the graph is a d-dimensional vector representing this node's embedding. This representation can be used for downstream tasks such as node or graph classification or regression.

Graph auto-encoders. Graph auto-encoders aim to learn a mapping of the graph input (A, X) to an embedding $Z \in \mathbb{R}^{n\times d}$. This embedding is optimized such that, given a decoder, we can recover the adjacency matrix A from the embedding. As visualized in Figure 1, the encoder consists of a two-layer graph convolutional network with no output activation and hidden layer with size $h \in \mathbb{N}$. The decoder is a simple dot product:

$$\underbrace{Z_{\text{relu}} = \text{GCN}(X, A) = \tilde{A}\ \text{relu}(\tilde{A}XW^{(0)})W^{(1)}}_{\text{encoder}}, \quad \underbrace{\hat{A}_{\text{relu}} = \sigma(Z_{\text{relu}}Z_{\text{relu}}^{\mathsf{T}})}_{\text{decoder}}. \tag{1}$$

Here $Z_{\text{relu}} \in \mathbb{R}^{n\times d}$ is the latent space representation with embedding dimension d, and the matrix $\hat{A}$ is the reconstructed adjacency matrix. Note that $\hat{A}$ is a real-valued matrix that aims to capture the likelihood of each edge in the graph. To obtain a binary adjacency matrix as a final result, we discretize by thresholding the values at 0.5. The loss function optimized by the auto-encoder is the cross-entropy between the target matrix A and the reconstructed adjacency matrix $\hat{A}$. It is minimized over the encoder's parameters $W^{(l)}$. The decoder is a deterministic function and is not learned from the data. For a graph with adjacency matrix A, latent space embedding Z, and with $\sigma(\cdot)$ denoting the sigmoid function, the loss is formalized as follows:

$$l_A(Z) = \sum_{ij=1}^{n} a_{ij} \log(\sigma(z_i z_j^{\mathsf{T}})) + (1 - a_{ij})(1 - \log(\sigma(z_i z_j^{\mathsf{T}}))). \tag{2}$$

Below we compare GAEs based on a relu encoder with a simplified model based on a linear encoder.

3 Relu encoders have at most the representational power of linear encoders

Graph auto-encoders are widely used for representation learning of graphs. However, it has also been observed that linear models outperform them on several tasks. In this section, we investigate this empirical observation from a theoretical point of view. We define a graph auto-encoder that uses a simple linear map as the encoder and prove that the derived model has larger representational power than a relu encoder. To this end, we introduce a linear encoder based on a linear map of the diffusion matrix $Z_{\text{lin}} = \tilde{A}W$, where Z_{lin} is the latent space representation. The loss function and the decoder remain as in the relu model. For a relu encoder, the dimension h of the hidden layer satisfies $g > h > d$ (for the linear encoder, the parameter h does not exist). We now define the solution spaces to help us describe the respective models' behavior. They will represent the embeddings that can be learned by relu and linear encoders, both with and without node features and more general functions on graphs:

$$\begin{aligned}
\mathcal{Z}_{\text{lin}} &= \{Z \in \mathbb{R}^{n\times d} | Z = \tilde{A}W \text{ for } W \in \mathbb{R}^{n\times d}\} \\
\mathcal{Z}_{\text{lin},X} &= \{Z \in \mathbb{R}^{n\times d} | Z = \tilde{A}XW \text{ for } W \in \mathbb{R}^{g\times d}\} \\
\mathcal{Z}_{\text{relu}} &= \{Z \in \mathbb{R}^{n\times d} | Z = \tilde{A}\,\text{relu}(\tilde{A}W^{(0)})W^{(1)} \text{ for } W^{(0)} \in \mathbb{R}^{n\times h}, W^{(1)} \in \mathbb{R}^{h\times d}\} \\
\mathcal{Z}_{\text{relu},X} &= \{Z \in \mathbb{R}^{n\times d} | Z = \tilde{A}\,\text{relu}(\tilde{A}XW^{(0)})W^{(1)} \text{ for } W^{(0)} \in \mathbb{R}^{g\times h}, W^{(1)} \in \mathbb{R}^{h\times d}\} \\
\mathcal{Z}_{f} &= \{Z \in \mathbb{R}^{n\times d} | Z = f(\tilde{A}, X; \theta) \text{ for } \theta \in \Theta\}.
\end{aligned}$$

Here f can be any parameterized function on a graph $f : \mathbb{R}^{n\times n} \times \mathbb{R}^{n\times g} \times \Theta \to \mathbb{R}^{n\times d}$ and Θ can be an arbitrary set of parameters. It is easy to see that $\mathcal{Z}_{\text{lin},X} \subseteq \mathcal{Z}_{\text{lin}}$. In words: when we compare two linear encoders, one with node features and one without node features, the one with node features has a more restricted solution space. The reason is that the feature matrix X is at most rank g and typically has a low rank compared to the diffusion matrix $\tilde{A}$, which restricts the image of $\tilde{A}X$. The same holds for the relu encoder: $\mathcal{Z}_{\text{relu},X} \subseteq \mathcal{Z}_{\text{relu}}$. The relation between the linear encoder and the relu encoder is less obvious. One might guess that the relu encoder can learn more mappings than a linear encoder. However, we now show that this is not the case: we will see that $\mathcal{Z}_{\text{lin}} \supseteq \mathcal{Z}_{\text{relu}}$, focusing on the featureless model. This relation implies that the linear encoder has a larger representational power than a relu encoder.

While this paper focuses on a specific architecture that has been introduced in previous work and is heavily used in practice, some of our insights extend to any encoder function f under mild assumptions on the input graph. Thus, we first state our result in a more general way. We make the following assumptions:

Assumption 1. The number of nodes n is larger than the feature dimension g.

Assumption 2. The rank of the diffusion matrix $\tilde{A}$ is larger than or equal to the embedding dimension d.

Assumption 3. The loss-function l is invariant to orthogonal transformations, that is, $l_A(Z) = l_A(RZ)$ for any orthogonal matrix R.

• Assumption 1 is typically fulfilled in practice. Vertices are typically described by a moderate number of features. Especially for larger graphs, this is a weak assumption.
• Assumption 2 is usually satisfied because the embedding dimension is a choice, and a graph auto-encoder aims to find a low-dimensional representation. We can also think about this assumption in the context of the informational content of our data. If the input matrix is of low rank, it holds little information; we will not gain information by embedding it into a higher dimensional space. Setting the embedding dimension d smaller than the rank of the input matrix $\text{rank}(\tilde{A})$ makes sense.
• Assumption 3 is fulfilled in the graph auto-encoder setting; the loss given in Equation 2 is based on pairwise distances given by the dot-product. This loss is invariant to orthogonal transformations. Generally, one could construct loss functions not invariant to orthogonal transformations. However, orthogonal transformations usually preserve pairwise metrics, and as a result, all loss functions based on a decoder incorporating pairwise distances will also be invariant to orthogonal transformations.

To prove our main theorem, we first observe that given Assumption 2, for any set P of n points in a d-dimensional subset of $\mathbb{R}^n$, we can find an orthogonal transformation that maps the span of A so that it contains all points in P.

Lemma 4. *For $P \in \mathbb{R}^{n\times d}, A \in \mathbb{R}^{n\times m}$ with $\text{rank}(P) \leqslant rank(A)$, there exists an orthogonal matrix R with $\text{span}(P) \subseteq \text{span}(RA)$.*

We can now prove that the linear model can outperform any other function f on the graph G during training.

Theorem 5 (Minimal Loss). *Consider a fixed graph $G = (A, X)$. Let $\tilde{A}$ be the diffusion matrix of G and let $Z \in \{Z \in \mathbb{R}^{n\times d} \mid Z = f(A, X; \theta) \text{ for } \in \Theta\}$ be any latent space representation of any graph function $f(A, X; \theta)$. Let l be a loss function and assume that Assumptions 1-3 are satisfied. Then the linear model can find a a latent space representation with loss at least as good as the relu encoder.*

Proof. With Lemma 4 it holds that for any $Z \in \mathbb{R}^{n\times d}$ and any matrix $\tilde{A} \in \mathbb{R}^{n\times m}$ with $\text{rank}(\tilde{A}) \geqslant \text{rank}(Z)$, there exists an orthogonal matrix $R \in \mathbb{R}^{n\times n}$ such that all points in latent space representation Z are in the image of $R\tilde{A}$:

$$\exists R \text{ orthogonal s.t. } \{R\tilde{A}W \mid W \in \mathbb{R}^{m\times d}\} \supseteq \{Z \mid Z = f(A, X; \theta) \in \mathbb{R}^{n\times d}\}.$$

This implies that there exists a weight matrix $W \in \mathbb{R}^{m\times d}$ with $l(Z) = l(R\tilde{A}W) = l(\tilde{A}W)$. In other words, the minimal loss over the set of possible solutions that is learnable by the linear model is smaller or equal to any loss over the solution space achievable by the function f:

$$\inf\{l(Z_f) \mid Z_f \in \mathcal{Z}_f\} \geqslant \inf\{l(Z_{\text{lin}}) \mid Z_{\text{lin}} \in \mathcal{Z}_{\text{lin}}\}$$

□

Theorem 5 considers a quite general statement. The graph auto-encoder architecture is one particular instance of a function f. Given the specific architecture of the auto-encoder we can show that the solution space of the relu-encoder is contained in the solution space of the linear model.

Corollary 6 (Representational power of GAEs). *For any (trained) graph auto-encoder in $\mathcal{Z}_{\text{relu}} \supseteq \mathcal{Z}_{\text{relu},X}$, there exists an equivalent, featureless linear encoder in $\mathcal{Z}_{\text{lin}}$ that can achieve the same training loss because we have $\mathcal{Z}_{\text{lin}} \supseteq \mathcal{Z}_{\text{relu}}$.*

We refer to Appendix A for the proof. Note that if we assume that $\operatorname{rank}(AF) > d$, then the rank of AF is larger than the embedding dimension d. Theorem 5 and Corollary 6 hold for the solution space of the linear model, including features: $\mathcal{Z}_{\text{lin},\mathtt{X}}$. We discuss the influence of the features on the representational power of the linear model in Section 4.

Corollary 6 has a simple but strong implication: in principle, the (featureless) linear model is more powerful than the traditional GAE (with or without features). Here "more powerful" means that the linear model can achieve better training loss.

The issue of representational power. Note that we do not claim that

$$\{f : \mathbb{R}^{n\times g} \to \mathbb{R}^{n\times d} \mid f(A, X) \text{ is any nonlinear GNN}\} \subseteq \{f : \mathbb{R}^{n\times g} \to \mathbb{R}^{n\times d} \mid f(A, X) = A\, W \text{ for } W \in \mathbb{R}^{g\times d}\},$$

which would mean that the function class of linear models contains the function class of nonlinear models. However, in the setting of a graph neural network, we usually have a single graph as an input with a fixed number of nodes n. In this setting, the goal is to find an embedding of this fixed graph. Our result says that $\mathcal{Z}_f \subseteq \mathcal{Z}_{\text{lin}}$ and in particular $\mathcal{Z}_{\mathtt{relu},\mathtt{X}} \subseteq \mathcal{Z}_{\mathtt{relu}} \subseteq \mathcal{Z}_{\mathtt{lin}}$. In words: The solution space of the nonlinear model is contained in the solution space of the linear one, showing that the representational power of the linear model is larger than the one of the nonlinear model.

Generalization properties. Note that Corollary 6 does not claim that the linear model without features outperforms the relu model. This corollary only considers the training loss. Thus we can not derive anything about the test performance or the solution that an optimization algorithm might find. So far, we are just consider training loss and the existence of a weight matrix W. From the above discussion, it is apparent that the linear encoder is less restricted, which means that it introduces a weaker inductive bias than the relu encoder. Note that this is independent of the fact that we used the relu activation function or the depth of the convolutional network. Regarding the test performance, the question is whether the restriction that the relu encoder puts on the solution space induces a "good" inductive bias that helps improve the model's test error. As we see below, we empirically find that when using features, a relu encoder does outperform the (featureless) linear encoder on test data. The relu activation and the features both restrict the solution space and introduce an inductive bias. Next, we investigate which of these two ingredients helps to restrict the representational power in a meaningful way.

4 The inductive bias of adding features

This section investigates how features influence the solution space of the linear model and the inductive bias that we introduce by adding features. While Corollary 6 shows the superior representational power of the linear model without features, in Section 5, we will observe improved test performance for models that do use feature information. Whenever the features are low rank compared to the diffusion matrix, they restrict the solution space and introduce an inductive bias that facilitates learning and can improve generalization if they hold helpful information. We observe this behaviour for both considered architectures, the relu and the linear encoder. Consequently, we conjecture that the presence of features, and not the relu nonlinearity, introduces the necessary bias to restrict the solution space in a meaningful way. However, some care is needed. As we will see, adding features can also harm performance if they contradict the structure of the target graph. In order to quantify the harm, we derive a measure for misalignment between graph structure and feature information. Based on this, we give theoretical insights about the restriction of the solution space when including features in the linear model.

Intuitively, we want to call a graph A and the corresponding features X "aligned" if they encode "similar" information. In our setting, this would be the case if A is "close to" XX^T: considering the features $X \in \mathbb{R}^{n\times g}$ as node embeddings, the matrix XX^T reconstructed adjacency matrix in a similar way as the dot product decoder. In other words, nodes should be close together in the feature space if and only if an edge connects them. Since the adjacency matrix is symmetric, a decomposition $A = YY^T$ for $Y \in \mathbb{C}^{n\times n}$ always exists. The error that we might introduce by adding features then comes from the fact that XX^T is a (low rank) approximation of $YY^\mathsf{T} = A$: Even for perfectly aligned features, the feature dimension g is typically much

smaller than n. Note that if we consider one graph and two different feature matrices with the same span, both features restrict the solution space in the same way and give rise to the same optimal embedding:

Proposition 7 (Features with the same span induce the same solution space). *Let $\tilde{A} \in \mathbb{R}^{n \times n}$ be the diffusion matrix of a fixed graph and $U, F \in \mathbb{R}^{n \times g}$ two different feature matrices for this graph. If $\text{span}(U) = \text{span}(F)$, then $\mathcal{Z}_{\text{lin},U} = \mathcal{Z}_{\text{lin},F}$.*

Intuitively, this means that if U and F span the same space, the corresponding models can, in principle, learn the same mappings and output the same embeddings.

Proof. We show that for every weight matrix W_U with $Z = \tilde{A}UW_U$, there exists a weight matrix W_F with $Z = \tilde{A}FW_F$. From $\text{span}(U) = \text{span}(F)$ it follows that $\text{span}(\tilde{A}U) = \text{span}(\tilde{A}F)$. We define corresponding linear maps $f_U, f_F : \mathbb{R}^d \to \mathbb{R}^n$ with $f_U(x) = \tilde{A}Ux$ and $f_F(x) = \tilde{A}Fx$. Since $\text{Im}(f_U) = \text{Im}(f_F)$ we know that every point $f_U(x) = y$ has at least one pre-image in f_F. Let $W_U = [w_{U_1}, ..., w_{U_d}]$ and $Z = [z_1, ..., z_d]$ be such that $z_i = f_U(w_{U_i})$. For every z_i there exists at least one w_{F_i} with $f_F(w_{F_i}) = z_i = f_U(w_{U_i})$. Since this holds for any matrix W_U we can conclude that $\mathcal{Z}_{\text{lin},U} = \mathcal{Z}_{\text{lin},F}$. □

We have seen that two different feature matrices with the same span restrict the solution space equivalently. Next, we derive a condition for the alignment between features X and graph structure $\tilde{A}$ that quantifies when features are potentially helpful. To do so, we investigate two settings. In the first setting, the features align with the graph structure and can introduce a valuable restriction of the solution space. In the second setting, the features contradict the graph structure and restrict the solution space in such a way that the optimal embedding can no longer be recovered. The intuition for helpful features corresponds with the one for good embeddings: If points are connected in the graph, they should have similar feature vectors. Consequently, if we were not to embed the nodes into a low-dimensional space but were simply interested in any latent representation of the graph, the features themselves would represent desirable embeddings.

Proposition 8 (Features cannot hurt if they perfectly align with the graph structure). *Let X be the feature matrix of the graph with diffusion matrix $\tilde{A}$ and let the embedding dimension coincide with the feature dimension g. If $\text{Im}(\tilde{A}X) = \text{Im}(X)$, then there exists a weight matrix W with $X = \tilde{A}XW$.*

This proposition shows that if $\text{Im}(\tilde{A}) = \text{Im}(\tilde{A}X)$, then the model can recover the features.

Proof. Since $\tilde{A}X$ and X span the same subspace, we can find a w_2 for any w_1 with $Xw_1 = \tilde{A}Xw_2$ and $w_1, w_2 \in \mathbb{R}^g$. We interpret every column in a matrix as an independent vector to derive that for any matrix W_1 we can find a matrix W_2 with $XW_1 = \tilde{A}XW_2$ and $W_1, W_2 \in \mathbb{R}^{g \times g}$. □

The proposition suggests that if the features encode the same structure as the graph's adjacency matrix, they do not negatively restrict the solution space. We now consider the opposite case: if the node relations encoded by their features contradict the adjacency structure of the graph, we can neither recover the features nor the adjacency matrix. In this setting, features restrict the function space in a harmful way, preventing the easiest and trivially optimal embedding. From these observations, we then derive a definition to measure the alignment between a graph's structure and node features.

Proposition 9 (Features can hurt if they do not align with the graph structure). *Let X be the graph's feature matrix, and assume that the embedding dimension coincides with the feature dimension, $d = g$. Let $\tilde{A}$ be the diffusion matrix and let $Y \in \mathbb{R}^{n \times d}$ be a matrix such that $\tilde{A} = YY^T$. Assume that the graph can be perfectly represented in g dimensions, that is, there exists a $Z \in \mathbb{R}^{n \times g}$ with $ZZ^T = \tilde{A}$. Let $\text{rank}(Y) = \text{rank}(\tilde{A}) = g$.*

1. *If $\text{Im}(\tilde{A}X) \neq \text{Im}(X)$, then there exists no weight matrix W with $X = \tilde{A}XW$.*
2. *If $\text{Im}(\tilde{A}X) \cap \text{Im}(X)^{\perp} \neq \emptyset$, then there exists no weight matrix W with $Y = \tilde{A}XW$.*

The intuition is that if the adjacency matrix and the node features define two different structures, then neither of the structures can be encoded by the linear model. The alignment of the structures is captured by the condition on the image of $\tilde{A}X$ and X.

Proof of Part 1. We prove the contraposition: If there exists a weight matrix W with $X = \tilde{A}XW$ then $\mathrm{Im}(\tilde{A}X) = \mathrm{Im}(X)$. Let $\tilde{A} \in \mathbb{R}^{n\times n}$ and let $X \in \mathbb{R}^{n\times g}$ be full rank. Let W be such that $X = \tilde{A}XW$. Since X is full rank, the rank of W is full and thus $\mathrm{Im}(X) = \mathrm{Im}(\tilde{A}XW) = \mathrm{Im}(\tilde{A}X)$. □

Proof of Part 2. For $Y = \tilde{A}XW = YY^\mathsf{T}XW$ to hold we need W to be the $g \times g$ inverse of (Y^TX). We prove that if $\mathrm{Im}(\tilde{A}X) \cap \mathrm{Im}(X)^\perp \neq \emptyset$, then $Y^\mathsf{T}X \in \mathbb{R}^{g\times g}$ is of rank smaller g and thus not invertible. We again prove the contraposition: If $X^\mathsf{T}Y$ is full rank then $\mathrm{Im}(\tilde{A}X) \cap \mathrm{Im}(X)^\perp = \{0\}$. Note that $Y^\mathsf{T}X$ is full rank by assumption and thus: $\mathrm{Im}(\tilde{A}X) = \mathrm{Im}(YY^\mathsf{T}X) = \mathrm{Im}(Y)$. Generally it holds that $\mathrm{Im}(X)^\perp = \ker(X^\mathsf{T})$. Let $w \in \mathrm{Im}(Y) \cap \ker(X^\mathsf{T})$. Since w is in $\mathrm{Im}(Y)$ there exists a $v \in \mathbb{R}^d$ with $w = Yv$. Since w in $\ker(X^\mathsf{T})$, it follows that $X^\mathsf{T}Yv = X^\mathsf{T}w = 0$. This implies $v = (X^\mathsf{T}Y)^{-1}0 = 0$ and thus $w = 0$ and thus $\mathrm{Im}(Y) \cap \ker(X^\mathsf{T}) = \{0\}$. □

Note that the dot product is invariant under orthogonal transformations; thus, for the features, every orthogonal transformation can be absorbed in the weight matrix. Proposition 9 also holds for orthogonally transformed features and embeddings. Thus given the dot-product as a decoder there is no other embedding $\tilde{Y} \neq Y$ that recovers the adjacency structure of the graph.

Based on all these insights, we propose a misalignment definition between a graph and its node features. Intuitively, we want to say that A and X are misaligned if $\tilde{A}X$ and X do not span the same subspace.

Definition 10 (**Misalignment of graph and features).** Let $\tilde{A}$ be the symmetrically normalized adjacency matrix and $\tilde{X}$ the normalized feature matrix such that each column has l_2-norm 1. We measure misalignment of a graph A and its features X using the distance $d_{\text{algn}} : \mathbb{R}^{n\times n} \times \mathbb{R}^{n\times g} \to \mathbb{R}$, where the arccos function is applied to every entry in the matrix:

$$d_{\text{algn}}(A, X) \mapsto \mathrm{Tr}(\arccos(\tilde{A}\tilde{X}\tilde{X}^\mathsf{T})).$$

We got inspired to this measure by the geodesic distance on the Grassmannian manifold, which measures the principal angles of two subspaces AX and X. If d_{algn} is low, the alignment between A and X is high. It is easy to see that with this formulation, $d_{\text{algn}}(A, X) = 0$ if $A = XX^T$. This is consistent with our intuition that the features and structure of the graph are perfectly aligned if they encode the same structure.

The generalization error of linear encoders is task dependent. Learning and generalization can only work if the learning architecture encodes the correct bias for the given task. We now consider two tasks to demonstrate the negative and positive impact of the bias that we introduce when using features.

In recent work, the most common task is **link prediction**. For this task we consider a graph given by its adjacency matrix $A \in \{0, 1\}^{n\times n}$ and a feature matrix $X \in \mathbb{R}^{n\times g}$. We construct a train graph $\bar{A}$ by deleting a set of edges from the graph, which we later, during test time, would like to recover. We train to minimize the loss $l(\bar{A}, X)$, while the test loss is given by $l(A, X)$. In our opinion, link prediction is a somewhat strange task when talking about generalization. We assume that the input graph is incomplete or perturbed. During training, we optimize the GAE for an adjacency matrix that we do not wish to recover during test time (because we also want to discover the omitted edges). However, this cannot be encoded in the loss of the auto-encoder (compare Eq. (2)). For the encoding to discover the desired embedding, the necessary bias to prevent the model from optimally fitting the training data must be somewhere else in the model. We claim that the features introduce this necessary bias, and we evaluate the link prediction task because it is so prominent in the literature.

We consider a second task, **node prediction**. For this task, the is graph given by its adjacency matrix $A \in \{0, 1\}^{n\times n}$ and the feature matrix $X \in \mathbb{R}^{n\times g}$. We construct a train graph $\bar{A}, \bar{X}$ by deleting a set of nodes from the graph A and the corresponding features from the feature matrix X. We train to optimize $l(\bar{A}, \bar{X})$. We then compute the test loss on the larger graph A, X: $l(A, X)$ with the goal of predicting the omitted nodes. The goal is to learn a mapping from the input graph to the latent space that generalizes to new, unseen vertices.

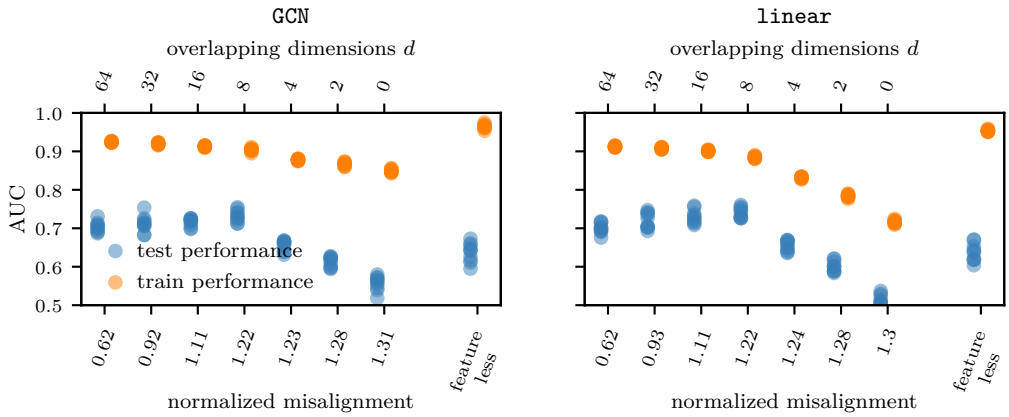

Figure 2: When features help. In the link prediction task, features help to regularize if they align with $\tilde{A}$ because they encode the target structure. Along the x-axis we plot the alignment between the graph and the features with two different scores: on the bottom, we plot misalignment as measured by d_{algn}, described in Definition 10 (low is good alignment, high is bad alignment). On the top, we show the alignment given by the number of overlapping dimensions of the two subspaces. The figures show train and test performance for the link prediction task on the synthetic dataset described in Section 5. We embed the points into eight dimensions. We plot values for ten independently sampled graphs. The left figure shows results for the architecture with the relu encoder and the right figure shows results for the linear model.

Intuitively it is clear that, depending on the task, the usefulness of the introduced bias might vary. We empirically evaluate the influence of features and compare the performance of the linear and the nonlinear encoder with and without the bias of features.

5 Empirical evaluation

In this section, we evaluate the influence of node features and the role of their alignment both for linear and relu encoders. We will see that the results support the theory that linear encoders outperform relu encoders.

Setting. We consider two models for empirical evaluation. The first is the relu encoder defined in Equation (1) based on the implementation of Kipf & Welling (2016b), and the second is the linear encoder. Since the relu encoder has two layers and thus two weight matrices, we realize the weights for the linear model accordingly: $Z_{\text{lin}} = \tilde{A}XW^{(0)}W^{(1)}$. This definition does not contradict the one from above; we can simply choose $W = W^{(0)}W^{(1)}$. Regarding the representational power of the model, this makes no difference. However, from an optimization perspective, it can influence the training (see Saxe et al. (2013)), and we aim to compare the models as fair as possible. We use the same objective function as Kipf & Welling (2016b), weighting the edges to tackle the sparsity of the graph and regularizing by their mean squared norm to prevent the point embeddings from diverging. We train using gradient descent and the Adam optimizer for 200 epochs. Due to sparsity, the accuracy of the recovered adjacency matrix is not very insightful. As is done in previous work, we measure the performance of the auto-encoders by the area under the receiver operating characteristic curve (AUC). To get a representative metric we use all present (positive) edges from the graph and uniformly sample the a number of what we call 'negative' edges, that is, edges that are not present in the graph's adjacency matrix. As a result, we only use some of the entries of the adjacency matrix for evaluation and get a balanced set of present and non present edges in the graph. To get representative results, we run every experiment 10 times. In real-world datasets, we randomize over the test set, drawing different edges or nodes each time. For the synthetic datasets, we draw ten independent graphs from the same generative model described below, then construct the test and train sets on these.

Datasets. We use the following generative model to get a **synthetic dataset** with aligned features. We draw $n = 1000$ data points from a standard Gaussian distribution in $g = 64$ dimensions. We normalize the

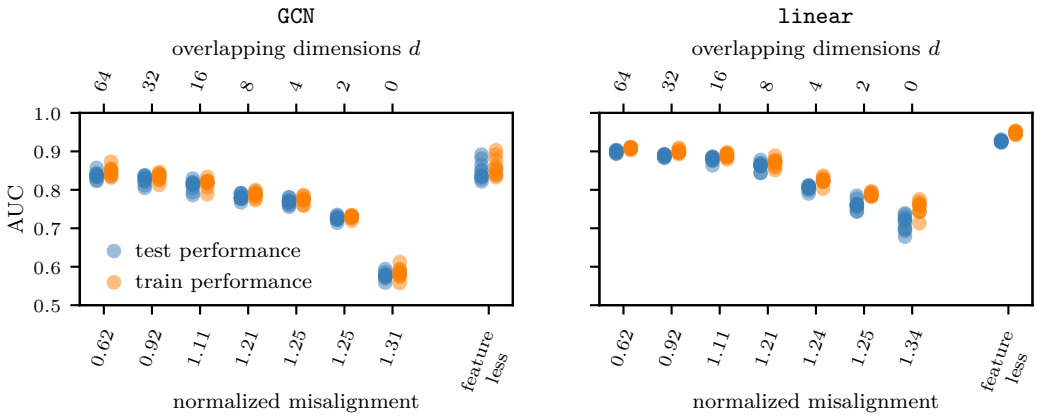

Figure 3: When features harm. Adding features can harm the node prediction task performance, preventing optimal graph fitting. The x-axis shows the alignment between the graph and the features with two different scores: we plot the misalignment as measured by d_{algn}, described in Definition 10 (low is good alignment, high is bad alignment). On the top, we show alignment given by the number of overlapping dimensions of the two subspaces. The figures show train and test performance for the node prediction task on the synthetic dataset described in Section 5. We embed the nodes into eight dimensions. We plot values for ten independently sampled graphs. The left figure shows results for the architecture with the relu-encoder and the right figure shows results for the linear model.

points to lie on the unit sphere and store their coordinates in a feature matrix $X \in \mathbb{R}^{n \times g}$. We then construct the graph by thresholding the matrix $XX^\top$ to get a 0-1-adjacency matrix of density between 0.01 and 0.02. With this construction, the feature matrix and the graph adjacency matrix are as aligned as possible, given the dimensional restriction. We generate data that simulates misalignment as follows. We aim to construct features that overlap in d dimensions with the graph's optimally aligned, true features. Recall that changing the span does not restrict the solution space, so we can replace the features with any other basis of the same space. The feature matrix X is spanned by $\{u_1, ..., u_g\}$. We construct a perturbed feature matrix $X_{\texttt{perturbed}}$ that has the same rank as X and spans the first d dimensions of the space; $\text{span}(u_1, ..., u_d)$, but is orthogonal to the remaining space; $\text{span}(u_{d+1}, ..., u_g)$. We do so by calculating the singular value decomposition $X = U\Sigma V^\top$, then choose the first d columns and the last $g - d$ columns from U. Let $U = [u_1, ..., u_n]$, we set $X_{perturbed} = [u_1, ..., u_d, u_{n-(g-d)}, ..., u_n]$. Note that the vectors $u_{n-(g-d)}, ..., u_n$ lie completely in the orthogonal complement of $\text{span}(u_1, ..., u_g)$, which means that the span of the optimal features X and the perturbed features $X_{\texttt{perturbed}}$ overlap in exactly d dimensions. We calculate the misalignment between the graph A and the perturbed features $X_{\texttt{perturbed}}$ as in Definition (10). By this construction, we do not fix the degree of rotation of the non-overlapping dimensions. As a result, the misalignment values for an overlap of a specified dimension d can vary slightly in different runs. However, as we normalize the values with regard to the dimensions, we still get insights for quantitative comparison. For example, we get a misalignment value d_{algn} of about 0.92 for an overlap of 32 dimensions on the synthetic dataset with 64 dimensions overall. This corresponds to an overlap of approximately 50% of the dimensions. We get a misalignment value d_{algn} of approximately 1.11 for an overlap of 16 dimensions, corresponding to an overlap of approximately 25% of the dimensions.

As **real world datasets** we consider three standard benchmarks: `Cora`, `Citeseer`, and `Pubmed`. Similar to previous work, we embed the nodes into 16 dimensions when considering the link prediction task. For the node prediction task, embedding into 16 dimension turns out to be too simple a task. We thus use 4 as the embedding dimension. For graph statistics of all used datasets and more details on the experimental setup, see Appendix B.

When features help for link prediction. In Figure 2, we plot the AUC performance of both encoder variants, relu and linear, both with and without features, on the link prediction task. We observe that

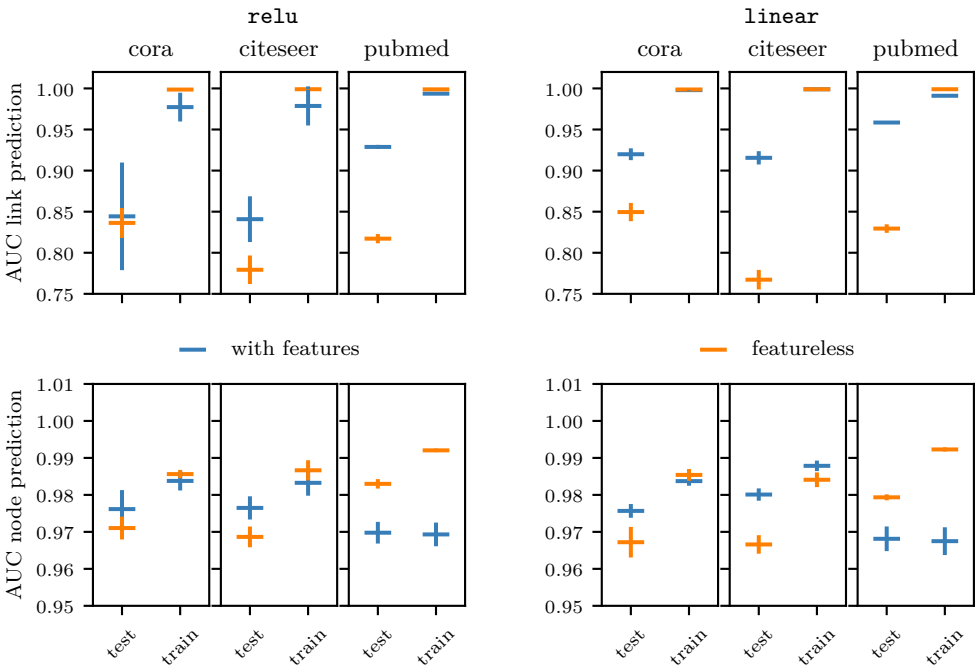

Figure 4: Train and test performance for the three real-world datasets `CoraCiteseer`and `Pubmed`. Vertical lines indicate standard deviation. We consider relu and linear encoder models with and without features. Top: Link prediction task; we embed points into 16 dimensions. Bottom: Node prediction task, points are embedded into four dimensions. The normalized misalignment scores are 0.67 for `Cora`, 0.69 for `Citeseer`and 0.86 for `Pubmed`.

both models' test performances usually decrease with increasing misalignment between the features and the graph. Adding features decreases the train performance, indicating that features restrict the solution space. Features may encode similarities in the feature space that are not present in the (incomplete) graph. This information restricts the solution space and can help to recover those missing structures. Moreover, the featureless model shows high train performance while the test performance is low. This behavior is expected in link prediction as optimal train performance implies sub-optimal test performance. In Figure 4, we observe the same behavior for the real-world datasets. The features add information about the target structure and improve the test performance. We suggest the following perspective: In the link prediction task, features act as a regularizer preventing the model from encoding the training adjacency matrix optimally. If they encode the desired graph structure, this regularization makes the model more robust to perturbations in the adjacency matrix, which is basically the task for link prediction. Here we assume the input graph is incomplete or perturbed, and we want to be robust against these perturbations and still recover the original graph. If the features do not encode the desired structure, including them can harm the performance. The relu encoder, where the features are passed through a weight multiplication and a relu activation, turns out to be more robust to false information, possibly learning to ignore (parts) of the features. The linear model can not compensate for erroneous feature information, which becomes visible in Figure 2, where, with more significant misalignment, the training error for the relu encoder drops slowly compared to the linear encoder. We can suspect slight overfitting in both settings when the overlapping dimension exceeds the embedding dimension. The effect is declining test performance, even for very aligned features. In this case, the introduced bias via the features might be too weak.

When features harm. In Figure 3, we visualize the performance of the models in the node prediction task. We observe similar behavior for the architectures using features as in the previous task. As expected, the performance decreases for an increasing distance of alignment d_{algn}. Interestingly, the test performance stays close to the train performance, which indicates that both models generalize well to unseen nodes and larger

graphs. Supporting our intuition, we observe that for both encoders, the featureless versions outperform the architectures using features. In the node prediction task, restricting the representational power by adding features harms the model's performance. Our experiments for the node prediction task show that featureless models can outperform the versions including features. Contrary to the link prediction task, in the node prediction setting, we assume that the given graph adjacency matrix already encodes the target structure of every node in the graph. Even if the features align optimally with the adjacency matrix, they do not hold additional information that is not already encoded in the adjacency matrix. In this setting, features still add a bias, but assuming that the given adjacency matrix is noiseless, they restrict the solution space in an undesired way.

In practice and real-world data, we usually do not know whether the given features encode the correct structure or if the graph is complete. So, not all of the intuition discussed above directly applies. We compute the normalized distance as given by d_{algn} and observe that the alignment is generally very good, with values of 0.6709 and 0.6940 for `Cora` and `Citeseer`, respectively. Misalignment is still good but slightly worse for `Pubmed` with a value of 0.8577, which indicates that almost 50% of the dimensions are misaligned. For the three benchmarks and the node prediction task, embedding into 16 dimensions already achieves nearly perfect recovery for all models. This reinforces our discussion about Assumption 2 being weak to no restriction. Usually, one would choose the smallest possible embedding dimension that shows promising results. See Appendix C for the performance when embedding into 16 dimensions. We embed the data into only four dimensions to get more insightful results. In Figure 4, we see that for the `Cora` and the `Citeseer` datasets, the models using features perform slightly better than those without features. This behavior could indicate noise in the input adjacency matrix, which the feature information regularizes. For the `Pubmed` dataset, where the features are slightly less aligned, not using the features seems to perform better. However, this difference is minimal. Compared to the synthetic dataset and `Pubmed`, the two networks from `Cora` and `Citeseer` have significantly higher feature dimensions in relation to the number of nodes possibly encoding redundancy. In summary, our findings indicate that the features encode mostly the correct structure, and adding them can improve optimization even for features with almost no additional information.

6 Conclusion

This work presents a theoretical perspective on the representational power and the inductive bias of graph auto-encoders. We consider a relu architecture and a linear architecture for the encoder. We prove that the linear encoder has greater representational power than nonlinear encoders. Theorem 5 also extends to more advanced models with similar encoder architecture and even to other nonlinear graph functions. Based on our experiments and empirical work in the literature, the nonlinear structure of the relu auto-encoder does not improve learning compared to the linear encoder. Our evaluations support the idea that the introduced bias from the nonlinearity is not the crucial ingredient to reducing the solution space in a meaningful way. On the other hand, the features can introduce a powerful inductive bias to both encoder architectures, improving their test performance. Whether features improve training and test performance heavily depends on the task we want to solve. For the two example tasks we consider in this paper, features help with link prediction but do not help node prediction. However, supporting the idea that linear encoders have larger representational power in training and can generalize, the linear encoder outperforms the nonlinear one in both tasks.

Acknowledgments

This work has been supported by the German Research Foundation through the Cluster of Excellence Machine Learning New Perspectives for Science" (EXC 2064/1 number 390727645) and the International Max Planck Research School for Intelligent Systems (IMPRS-IS).

References

Shaosheng Cao, Wei Lu, and Qiongkai Xu. Deep neural networks for learning graph representations. AAAI Conference on Artificial Intelligence, 2016.

Benjamin Paul Chamberlain, James Clough, and Marc Peter Deisenroth. Neural embeddings of graphs in hyperbolic space. *preprint arXiv:1705.10359*, 2017.

Shiyu Chang, Wei Han, Jiliang Tang, Guo-Jun Qi, Charu C. Aggarwal, and Thomas S. Huang. Heterogeneous network embedding via deep architectures. International conference on knowledge discovery and data mining (KDD), 2015.

Tim R. Davidson, Luca Falorsi, Nicola De Cao, Thomas Kipf, and Jakub M. Tomczak. Hyperspherical variational auto-encoders. Conference on Uncertainty in Artificial Intelligence (UAI), 2018.

Yuxiao Dong, Nitesh V. Chawla, and Ananthram Swami. Metapath2vec: Scalable representation learning for heterogeneous networks. International conference on knowledge discovery and data mining (KDD), 2017.

Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. International conference on Knowledge discovery and data mining (KDD), 2016.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. Neural Information Processing Systems (NeurIPS), 2017a.

William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *preprint arXiv:1709.05584*, 2017b.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *preprint arXiv:1609.02907*, 2016a.

Thomas N Kipf and Max Welling. Variational graph auto-encoders. *NeurIPS Workshop on Bayesian Deep Learning*, 2016b.

Xuelong Li, Hongyuan Zhang, and Rui Zhang. Adaptive graph auto-encoder for general data clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, Lina Yao, and Chengqi Zhang. Adversarially regularized graph autoencoder for graph embedding. International Joint Conference on Artificial Intelligence, 2018.

Leonardo F.R. Ribeiro, Pedro H.P. Saverese, and Daniel R. Figueiredo. Struc2vec: Learning node representations from structural identity. International conference on knowledge discovery and data mining (KDD), 2017.

Guillaume Salha, Romain Hennequin, and Michalis Vazirgiannis. Simple and effective graph autoencoders with one-hop linear models. European Conference, ECML PKDD, 2020.

Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. pp. 1067–1077. International conference on world wide web (WWW), 2015.

Vaibhav, Po-Yao Huang, and Robert Frederking. Rwr-gae: Random walk regularization for graph auto encoders. *arXiv preprint arXiv:1908.04003*, 2019.

Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. International conference on Knowledge discovery and data mining (KDD), 2016.

Felix Wu, Tianyi Zhang, Amauri H. Souza Jr., Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks. International Conference on Machine Learning (ICML), 2019.

A Proofs

In this Section, we provide the proofs of Lemma 4 and Corollary 6 and discuss the consequences.

Lemma 4 (Orthogonal Transformation). For $P \in \mathbb{R}^{n\times d}, A \in \mathbb{R}^{n\times m}$, $\text{rank}(P) \leqslant rank(A)$, there exists an orthogonal matrix R with $\text{span}(RA) \supseteq \text{span}(P)$.

Proof of Lemma 4. $\text{rank}(Z) \leqslant rank(A) = k$.
$\text{Im}(A)$ is spanned by an orthogonal normal basis $\{u_{i=1,\dots,d,\dots k}\}$
$\text{Im}(Z)$ is spanned by an orthogonal normal basis $\{v_{i=1,\dots,d}\}$.

Let $U := \left(\begin{array}{ccc|c} | & \cdots & | & \\ u_1 & \cdots & u_d & \cdots \\ | & \cdots & | & \end{array}\right)$ and $V := \left(\begin{array}{ccc|c} | & \cdots & | & \\ v_1 & \cdots & w_d & \cdots \\ | & \cdots & | & \end{array}\right)$.

Then with $R = V \left(\begin{array}{c|c} \mathbb{I}_d & 0 \\ \hline 0 & 0 \end{array}\right) U^T, \text{span}(Z) \subseteq \text{span}(RA)$.

Corollary 6 (Representational power of GAEs). For any (trained) graph auto-encoder in $\mathcal{Z}_{\text{relu}} \supseteq \mathcal{Z}_{\text{relu},X}$, there exists an equivalent, featureless linear encoder in $\mathcal{Z}_{\text{lin}}$, that can achieve the same training loss: $\mathcal{Z}_{\text{lin}} \supseteq \mathcal{Z}_{\text{relu}}$.

Proof of Corollary 6. Let $Z_{\text{lin}}(W) = \tilde{A} \cdot W$ for $W \in \mathbb{R}^{n\times f}$ be the latent representation of the linear model with weights W. We can write the latent representation for any GAE as a linear function of the form $Z_{\text{relu}}(W) = \tilde{A} \cdot W$ for some matrix $W \in \mathcal{W}$, where $\mathcal{W}$ is the set of matrices that can be represented by the relu term in the function: $\mathcal{W} = \{\text{relu}(\tilde{A}XW^{(0)})W^{(1)} : W^{(0)} \in \mathbb{R}^{n\times d}, W^{(1)} \in \mathbb{R}^{d\times f}\} \subseteq \mathbb{R}^{n\times f}$. We now define the possible latent embeddings that the two models can learn. For the relu encoder we have $\mathcal{Z}_{\text{relu}} = \{Z_{\text{relu}}(W) \text{ for all } W \in \mathcal{W}\}$, and for the linear model we get $\mathcal{Z}_{\text{lin}} = \{Z_{\text{lin}}(W) \text{ for all } W \in \mathbb{R}^{n\times g}\}$. Since $\mathcal{W} \subseteq \mathbb{R}^{n\times g}$, the set of learnable embeddings of the relu encoder is a true subspace of all learnable embeddings of the linear encoder: $\mathcal{Z}_{\text{relu}} = \{\tilde{A}W : W \in \mathcal{W}\} \subseteq \{\tilde{A}W : W \in \mathbb{R}^{n\times g}\} = \mathcal{Z}_{\text{lin}}$. It follows that, given any loss l, the linear model can achieve at least as good performance as the relu encoder: $\inf_{Z_{\text{relu}}\in\mathcal{Z}_{\text{relu}}} l(Z_{\text{relu}}) \geqslant \inf_{Z_{\text{lin}}\in\mathcal{Z}_{\text{lin}}} l(Z_{\text{lin}})$.

Consequences. To better understand the implications of Corollary 6, it is instructive to engage in the following thought experiment. Consider the standard supervised learning setting with a training data matrix $U \in \mathbb{R}^{n\times d}$ and labels $y \in R$. For any function (say a deep neural network) $f : \mathbb{R}^{n\times d} \to \mathbb{R}^n$, it is possible to express $f(U) = UU^+f(U) = UW$, for some $W \in \mathbb{R}^{d\times 1}$ if and only if U has a right inverse. This is a mild requirement if $d > n$. Note that this is precisely the "underdetermined" or "high-dimensional" setting where it is possible to find a linear map that perfectly fits the training data to the function values. If $d < n$, the right inverse cannot exist; therefore, one cannot find such a W. Let us contrast this with our setting of GNNs in Corollary 6. Since we consider graph networks, the input is the adjacency matrix $\tilde{A} \in \mathbb{R}^{n\times n}$; that is, we have feature dimensions equal to sample size ($n = d$). If $\tilde{A}$ is full rank, one can similarly find a linear map that perfectly fits the outputs of any nonlinear mapping of the input data. While it may seem surprising at first glance, it is intuitively clear that in high dimensions, one can always find a linear map that perfectly fits the outputs of any nonlinear mapping. If the model is restricted to the input data domain, we can attain the same "training loss". But even if $\tilde{A}$ was not full rank, in the graph auto-encoder setting, the GCN's last layer has linear activation, restricting the solution space in exactly the same way the linear model does.

B Experimental details

For our empirical evaluation we consider one synthetic dataset and three real world citation networks; `Cora`, `Citeseer` and `Pubmed`. Table 1 shows the graph statistics for the considered datasets.

dataset	nodes	edges	density	dim. features
`Cora`	2708	5278	0.00143	1433
`Citeseer`	3327	4614	0.00083	3703
`Pubmed`	19717	44324	0.00023	500
Synthetic	~ 1000	$\sim 10000 - 20000$	$\sim 0.01 - 0.02$	64

Table 1: Graph statistics of the real world datasets and averaged values in the synthetic setting.

For every dataset we compute the performances for two different tasks; node and link prediction, and four different models: using the relu encoder and the linear encoder once with features and also without features. So we compute performances in eight different settings for each dataset.

Every graph is split into train, validation and test sets with a ration of 70/10/20. So training always happens on either 70% of the edges and all nodes for the link prediction task, or on only a set of 70% of the nodes for the node prediction task. Test performance is then evaluated on 90% of the edges / nodes. We do not use the parts of the graph used for validation.

The code for the relu graph auto-encoder is publicly available with an MIT license.

We run the experiments on an internal cluster on *Intel XEON CPU E5-2650 v4* and *GeForce GTX 1080 Ti*. All experiments on the synthetic dataset take about 9 hours on single CPU and single GPU. Experiments for `Cora` and `Citeseer` take about 4 and 5 hours respectively. For the `Pubmed` dataset, which is the largest one, running all 8 setups took about 4 days on a single CPU and two GPUs.

C Additional experiments

For the real world datasets and the link prediction task we show result for embedding into 4 dimensions in the main paper. Table 2 show results when embedding into 16 dimensions. All models show near to optimal performance which indicates that embedding into 16 dimensions is barely a restriction and too simple of a task.

	cora			
	edge		node	
	linear	GAE	linear	GAE
features	0.91 ± 0.0088	0.88 ± 0.0123	0.99 ± 0.0009	0.99 ± 0.0014
featureless	0.84 ± 0.0078	0.85 ± 0.0122	0.98 ± 0.0027	0.98 ± 0.0016

	citeseer			
	edge		node	
	linear	GAE	linear	GAE
features	0.92 ± 0.0087	0.84 ± 0.0264	0.99 ± 0.0010	0.99 ± 0.0012
featureless	0.78 ± 0.0131	0.78 ± 0.0149	0.98 ± 0.0015	0.98 ± 0.0014

	pubmed			
	edge		node	
	linear	GAE	linear	GAE
features	0.96 ± 0.0018	0.92 ± 0.0037	0.98 ± 0.0006	0.97 ± 0.0055
featureless	0.83 ± 0.0056	0.83 ± 0.0049	0.99 ± 0.0009	0.99 ± 0.0006

Table 2: Test performance in terms of AUC for the three real world datasets `Cora` `Citeseer` and `Pubmed` with given standard deviations. We consider relu and linear encoder models with and without features. Top row: Link prediction task, points are embedded into 16 dimensions. Bottom row: Node prediction task, points are embedded into 16 dimensions.