Vrije Universiteit Amsterdam

Bachelor Thesis

Building on (Relational) Graph Convolutional Networks with Markov Chains

---

Author: Hristo Petkov (2617110)

1st supervisor: Michael Cochez
2nd reader: Emile van Krieken

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 21, 2020

Abstract

In graph convolutional networks, it is usually unclear how many hidden layers we should use. Choosing a too big number can lead to over smoothing while creating a network with only a few layers can undermine the importance of the topology of the input graph. That is why we build on existing research of two neural network models for graphs - the graph convolutional network (GCN) and the relational graph convolutional network (R-GCN) by introducing the Markov layer and Markov loss. We use them to try to create a system for which the number of hidden layers is irrelevant to its performance. Regarding the Markov layer, we specify two different variations. In the first one, we use the Markov loss to force a GCN or R-GCN to behave like a Markov chain, while in the second approach we use the Markov chain theory to converge the models' state representations to an equilibrium. With this in mind, we claim that reaching a stationary distribution for a state representation of a machine learning model solves the problem of an optimal number of hidden layers. This is so because adding more layers after the system has stabilized does not change its performance. The downside is that we usually observe a worsening of the models' accuracies when using a Markov layer.

1 Introduction

Designing a neural network that produces valuable and accurate predictions can be a complicated and time-consuming task. Depending on the type of input data and end goal, different variations of neural networks are preferred, for instance, a convolutional neural network is most suitable for images and video, a recurrent neural network - for time-series or stock market data, an autoencoder - for compressing data, without losing quality[1]. Regardless of the type, a crucial part of creating an efficient network is hyperparameter tuning. Choosing an appropriate number of the hidden layers is often optimized empirically by trying different values and comparing their accuracies and times for execution. In such a case, it is logical to assume that there exists a linear or another type of relation, which means that increasing the number of hidden layers or choosing a specific one produces the best results. If we focus on the first assumption, we can claim that modeling with an infinitely large number of layers should yield the best results. However, in this thesis, we are using graph convolutional networks, for which adding too many hidden layers leads to a decrease in accuracy because of signals getting lost and averaging behavior. Therefore, it makes sense to assume that we are searching for a specific number, which produces the best results.

To try to find this optimal number of hidden layers without conducting empirical experiments, we propose a novel method of combining graph neural networks [1] with Markov chains [2]. The goal is to use the Markov chains theory of convergence [3] to create a model for which the number of hidden layers is insignificant to its performance. The types of neural networks used are the ones able to model on graphs, such as the graph convolutional network (GCN) [4] and the relational graph convolutional network (R-GCN) [5] with the task of node classification.

We introduce two different approaches for combining the Markov chain theory to such systems. With both, we show that adding specific hidden layers does not change the

[1] https://towardsdatascience.com/types-of-neural-network-and-what-each-one-does-explained-d9b4c0ed63a1

initial performance of the GCN or R-GCN. Depending on the data set, the second method requires some number of hidden layers before the network stabilizes, while usually, the first approach has accuracy oscillating around some value. However, this comes with a price. Compared to the original performance, the models' precision worsens when applying either of the two approaches. Furthermore, we additionally show how the GCN and R-GCN can be used to learn the Markov chain property of convergence with the help of the first method and its Markov loss.

1.1 Research Question

We formulate the following research question which most precisely describes the purpose of this thesis: Is it possible to extend a system, like a GCN or R-GCN, in such a way that changing the number of hidden layers does not influence its actual performance, because its state representation has reached equilibrium? Although we know increasing the number of hidden layers in a GCN or R-GCN would eventually mean a decrease in accuracy, we investigate whether it is possible to create a model, for which we do not need to know the number of hidden layers because after some point increasing that number would make no difference to the outcome of the model.

2 Graph Neural Networks and Markov Chains

2.1 Graphs

When considering GCNs, we denote a graph as $G = (V, E)$. Nodes are represented as $v_i \in V$, while edges as $(v_i, v_j) \in E$. For our research, we work with undirected graphs, hence having $(v_i, v_j) \in E$ does mean $(v_j, v_i) \in E$.

If we assume that n is the number of nodes, then for each graph there exists an adjacency matrix A of size n^*n, which is characterised by $A_{i,j} = 1$ if $(v_i, v_j) \in E$ and $A_{i,j} = 0$, otherwise. We also denote the degree vector as δ and we use it to construct the degree matrix D. It is essentially an identity matrix but with δ for its diagonal entries rather than ones.

2.2 Graph Convolutional Network

The GCN [4] is a machine learning model that operates on graphs. It is used for modeling real-world entities as graphs and predicting their interaction and future behavior. It can also be applied to image classification, community prediction problems, for molecular structure analysis, and others[2].

$$H^{(l+1)} = \sigma(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{1}$$

In its core, the GCN uses the propagation rule Eq. 1 to compute the next layer $l + 1$. In this equation, nodes that are not connected to themselves lose information when propagating through the network. Their representation in the next layer would be only a collection of features of their neighbouring vertices. To include their features, all nodes should have self-loops. The first equation incorporates this requirement in the adjacency

[2]https://missinglink.ai/guides/convolutional-neural-networks/graph-convolutional-networks/

matrix $\widetilde{A} = A + I_N$. By adding an identity matrix with dimensions $n*n$, $\widetilde{A}$ is guaranteed to contain ones at its diagonal entries which ensure self-loops. $\widetilde{D}$ represents the corresponding degree matrix of the changed adjacency matrix $\widetilde{A}$.

Large-scale graphs normally have a set of nodes called hubs. They are characterised with larger degrees. As a consequence when applying Eq. 1, hubs will have substantially larger values in their feature representation than other vertices. This can lead to vanishing or exploding gradients[3]. The authors of the GCN suggest a normalization procedure of the adjacency matrix. This essentially means that each row of $\widetilde{A}$ sums up to one - $\sum_{j\in(0,n)} \widetilde{A}_{i,j} = 1$. Generally, normalization is achieved by matrix multiplication of the type $\widetilde{D}^{-1}\widetilde{A}$, however in [4] it is observed that symmetric normalization: $\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}$ yields more promising results[4].

The second part of Eq. 1 ($H^{(l)}W^{(l)}$) represent the previous layer and the weight matrix of layer l, respectively. Finally, an element-wise activation function $\sigma()$, such as ReLU(x) = max(0,x) or softmax (Eq. 2), is used.

$$softmax(k, x_i) = \frac{e^{xk}}{\sum_{j=1}^{i} e^{x_j}} \tag{2}$$

2.3 Knowledge Graphs

Similar to normal graphs, knowledge graphs [6] contain a set of nodes $v_i \in V$. In addition, they incorporate a set of relations so that each edge between two vertices has a specific type r, where $(v_j, r_n, v_i) \in E$. This type gives further information on the connection between the two nodes. In essence, a knowledge graph G_k can be described as $G_k = (V, E, R)$, where $r_n \in R$ [7].

It makes little sense to think of an undirected knowledge graph because having equal inverse relations is rarely the case. Imagine an edge type *isFatherOf* or even *livesIn*, where the two nodes belong to different classes - person and town, for instance.

When translating this to the adjacency matrix A_k of the knowledge graph G_k, we should have twice as many relations because of inverse relations. They can also be seen as the indegrees of V. Furthermore, another issue is the dimensionality of A_k. Due to relations holding valuable information, there should be an adjacency matrix per relation. Hence, A_k has a size of $p*n*n$ Eq. 3.

$$p = |R| + |R^{-1}| = 2|R|. \tag{3}$$

2.4 Relational Graph Convolutional Network

For modeling on knowledge graphs, we use the R-GCN developed by [5]. It is an extension of the GCN with the difference it manages to incorporate relations through an enhanced propagation rule Eq. 4 [5]. Furthermore, the R-GCN is used for both link prediction and node classification, whereas the GCN is primarily used only for the latter. In general, the use cases of the GCN and R-GCN match.

[3]https://towardsdatascience.com/how-to-do-deep-learning-on-graphs-with-graph-convolutional-networks-7d2250723780

[4]https://tkipf.github.io/graph-convolutional-networks/

$$h_i^{(l+1)} = \sigma\left(\sum_{r\in R}\sum_{j\in N_i^r}\frac{1}{c_{i,r}}W_r^{(l)}h_j^{(l)} + W_0^{(l)}h_i^{(l)}\right) \tag{4}$$

Although both Eq. 1 [4] and Eq. 4 share some similarities, there are a few key differences. As already indicated, the last equation manages to include relations. Additionally, Eq. 4 shows the computation of a single element of the new layer, whereas Eq. 1 derives the whole next layer in a single step. Hence, according to Eq. 4, we need to compute the feature representation of each next hidden neuron separately to propagate through an R-GCN layer. Nevertheless, this is an implementation detail. The matrix formulation, as in Eq. 1, can be derived here, too.

In the GCN section, we mentioned two issues - the problem with losing information when self-loops are not present and with hubs. As both networks are analogous, these problems also apply to the R-GCN. Adding self-loops is not as straightforward as in regular graphs. The authors of [5] decide to include a new relation to each node in the data. Essentially, they add an identity matrix to A to ensure that each entity keeps its current information when propagating from layer l to $l + 1$. This increases relations to Eq. 3 + 1.

When observing Eq. 4 we see σ which is again either ReLU(x) = max(0,x) or softmax (Eq. 2). For each relation the R-GCN has a separate weight matrix. Hence, $W_r^{(l)}$ indicates the matrix for relation $r \in R$ with the suitable dimensionality for layer l. Then, $h_j^{(l)}$ expresses the number j node that is connected to $h_i^{(l)}$ through relation r, such that $(h_i^{(l)}, r, h_j^{(l)}) \in E$. Therefore, here $\sum_{r\in R}\sum_{j\in N_i^r} W_r^{(l)}h_j^{(l)}$ means that for a node $h_i^{(l)}$, each of its neighbours' features of relation type r ($h_j^{(l)}$) are multiplied by a weight matrix for that same relation type ($W_r^{(l)}$). This is repeated for all the relations node $h_i^{(l)}$ has any neighbours.

Returning to the second issue - the problem with hubs, Eq.4 tackles this complication by $c_{i,r}$, which is the number of neighbouring indices of node i under relation r. Depending on whether $r \in R$ or $r \in R^{-1}$, $c_{i,r}$ is either the number of outdegrees or indegrees, respectively. Finally, after both sums the self-loops are added once: $W_0^{(l)}h_i^{(l)}$.

When imagining a system such as a R-GCN, it is not clear what connection it may have with Markov chains and how would they be used to boost its performance. Before explaining the similarity between the two, we give a short description of the theory behind Markov chains.

2.5 Markov Chains

A Markov chain is a stochastic model that describes sequential events. It has the property that every current event depends only on the one that precedes it. Markov chains are composed of a current state vector v_0 and a transition matrix M [2].

It is convenient to think of this concept as a graph, where nodes represent all possible states and the transition matrix holds the probabilities of moving from one particular state to another. Figure 1 illustrates a simple graph that we use to explain Markov chains. Here, vector v_0 indicates that for state zero, there is a 100 % chance for being at node A and 0 % probability for being at any other node. The transition matrix M is simply the adjacency matrix of the graph. However, each entry represents probabilities. That is

why each column of the transition matrix shown in Figure 1 (b) should sum up to one. A normalization procedure, similar to the one described in the previous sections, is applied.

Each new state v' is computed by the equation $v' = M * v$. Hence, state one v_1 should be $\begin{pmatrix} 0 \\ 0.5 \\ 0 \\ 0.5 \end{pmatrix}$. If we look at the graph in figure 1, we can reach a logical explanation of why the state changes in such a way. Provided that we start at node A, we can either go to node B or node D. Therefore, half of the times, we end up at B and the other half at D for state 1. Because we start with 100 % probability of being at node A, we end up with the results represented at v_1.

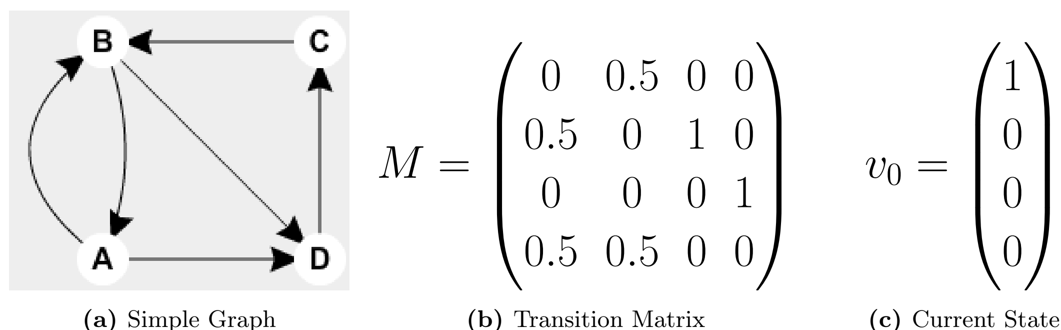
By continuously computing new states we reach an approximation of the unique state V, which is also known as the stationary distribution of the Markov chain. This vector has the property that further multiplying it with the transition matrix does not change the current state of the Markov chain (Eq. 5). Therefore, after some number of iterations, we can deduct that the Markov chain stabilizes, reaching an equilibrium. Furthermore, the initial state does not influence the system's convergence. All variations of v_0 can reach the stationary distribution, although some would require a larger number of multiplications with the transition matrix.

$$V = M * V \tag{5}$$

2.5.1 Regular Markov Chains

Not all Markov chains have a stationary distribution. Certain conditions determine whether the system can reach an equilibrium. However, if a Markov chain is regular, it always has a current state that satisfies Eq. 5.

Regularity requires certain conditions, one of which includes that the transition matrix, being a probabilistic representation of the system, should have the property $\sum_0^i M_{i,j} = 1$. Strictly speaking, each column of M should sum up to one. Additionally, a current state vector is a representation of the probability of being at some node. Therefore, it should also sum up to one. Moreover, the world of Markov chains represent a rather fixed system, where the events change, but the probabilities to move from one to another remain the same. In other words, the transition matrix should be unchanged throughout the process

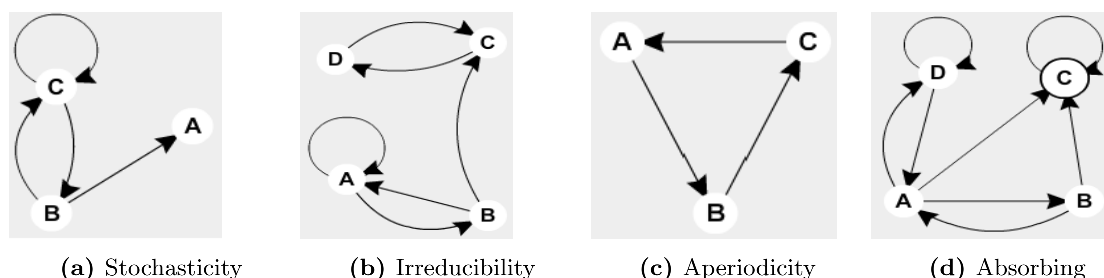


(a) Simple Graph **(b)** Transition Matrix **(c)** Current State

Figure 1

of computation of new current states.

Firstly, if we observe figure 1 (a) we see that node A has no outgoing edges, which means the sum $\sum_0^i P_{i,0}$ is zero. A possible way to fix this is to include a connection between vertices A and C, for example. Secondly, a regular Markov chain should be irreducible. In figure 1 (b) we see a graph of four nodes. When transitioning from vertex B to C there is no path backward. For a graph to be irreducible there should exist a path from every two nodes of that graph. Subgraphs (A,B) and (C,D) are connected with only a single directed edge, which means that returning to (A,B) is not possible. A solution would be to add an edge from either vertex C or D to vertex A or B.



(a) Stochasticity **(b)** Irreducibility **(c)** Aperiodicity **(d)** Absorbing

Figure 2

Finally, a Markov chain should only have aperiodic nodes to be regular. A vertex is periodic if any return to that same vertex occurs in only multiples of an integer. An example is figure 1 (c). If we start at node A, after exactly $3 * x$ iterations we return at our starting point. To make (c) aperiodic we need to break the cycle by introducing self-loops, for example[5]. Then, any irreducible and aperiodic Markov chain is guaranteed to converge to a stationary distribution [3].

Regarding relevance, Markov chains find application in many fields and algorithms such as biology, statistics, PageRank, speech recognition, and others [8].

2.6 PageRank

Before Google's solution to web crawling - PageRank, search engines had difficulties returning a suitable website. Their flaw was that they just counted the number of terms in a web page that is similar to the search. This meant that including popular terms on a website even if it has no information about them would significantly boost the website's ranking. Additionally, other common pitfalls included intentional or accidental spider traps, which are explained shortly.

Structurally, PageRank can be viewed as a Markov chain model [9]. However, real-world cases usually do not satisfy the conditions of irreducibility and aperiodicity. Some particular examples can include dead ends and spider traps. The prior is similar to the first problem discussed for Markov chains and shown in figure 1 (a). Node A does not have any outgoing edges which imply that once a process reaches that state it loses its ability to move to another vertex. Furthermore, no matter how many steps are needed at some

[5]Figure 1 (c) with self-loops at every node is aperiodic because starting from any node we can go back to our starting position by any integer. For 1 and 2 iterations, we only use the self-loops, for over 3 iterations we can either traverse the graph, use self-loops or make a combination of both.

point we reach node A. Intuitively, this indicates that if we try to estimate the probabilities of the system after convergence, we end up with only zeroes. We illustrated the second issue of spider traps in figure 1(d). In Markov chain theory it is analogous to the case of absorbing Markov chains. It also resembles the problem of dead ends with the difference that node C has only one outgoing edge - a self-loop. Again, when we reach that vertex there is no path to any other. The difference between the two lies within the outcome of the convergence of the system. In the case of spider traps node, C would get all probability as eventually, a state would end in that vertex. Hence, all nodes will have zero probability except for the spider trap, which will have one.

$$v' = \beta P v + (1 - \beta) e/n \tag{6}$$

PageRank addresses these issues by introducing teleportation [9, 10]. Its concept is to disrupt the natural order of events - an idea which is highly useful when stuck in a dead-end or spider trap. When looking at the equation of teleportation (Eq. 6), we see that the first part $v' = \beta P v$ is similar to Eq. 5 except for the term β. It stands for the probability to teleport, usually in the range of 0.8 to 0.9. The second part consists of $(1 - \beta)e/n$. Here, $\mathbf{e}$ is a vector of ones with the size the same as v' and n is the number of nodes. From a broader perspective, the first part can be interpreted as a natural continuation of PageRank when following the concepts of Markov chains, whereas e/n can be understood as a random teleport to any other node with probability equal to one divided by the number of vertices. Additionally, both parts are connected with the teleport probability, which defines the chance of choosing either the normal course of events β or teleportation $(1 - \beta)$.

Regarding convergence of a system, teleportation introduces the notion of randomness in the rather closed world of Markov chains and Eq. 5. To some degree, Eq. 6 manages to solve the problems of reducibility and periodicity (both dead ends and spider traps are special cases of the prior). No matter if a system has dead ends or spider traps, all vertices receive some probability as a result of randomness. Nevertheless, it is not distributed evenly. Most of the probability is absorbed by the spider traps depending on the value of β. The case is similar for dead ends, too. All nodes with outgoing edges receive some probability, however, a big part of it is lost. If we imagine a process being stuck at a dead-end node, its only option is to teleport, which depends on β. The difference between the two lies in the fact that if Eq. 6 is used on a graph with dead ends, eventually the values of v' are not going, to sum up to one, whereas in the case of spider traps they are going to.

3 Implementations

For applying the Markov chain theory, we have taken two directions, which differ fundamentally. Before that, we need actual models to make the experiments. Furthermore, we want to create our own networks and that is why we spent some time coding in plain PyTorch. We still followed closely Eq. 1 and Eq. 5 for the GCN and R-GCN, respectively, however, there are some subtle differences between ours and the original networks, which are described next.

3.1 GCN

Our Graph Convolutional Network has the task of semi-supervised node classification with label frequency the same as in [4]. Initially, we were transforming the data in a format suitable for the GCN ourselves, but we decided to use the DGL library because it conveniently prepares all three citation data sets (Cora, Citeseer, and Pubmed [11]), which are used in the original paper. Our model is trained for 200 epochs, with L2 regularization, 16 hidden neurons per hidden layer, a learning rate of 0.01, and cross-entropy loss. The main difference with the original is that we do not use dropout and a validation set. Therefore, we also skip the early stopping procedure. Finally, when showing how the GCN performs without the Markov properties we take the average 50 accuracies because there is usually a slight bias of around $\pm$ 1.2%.

3.2 R-GCN

Similarly, we use our Relational Graph Convolutional Network for node classification. The main difference is we create a multidimensional tensor which serves as the adjacency matrix of the graph. As there are r different relation we can make r adjacency matrices each of which captures the topology of the graph when viewing it with only one relation. Therefore, we have a r*n*n tensor, where each matrix is normalized separately. We also have a weight matrix for each relation as indicated in [5]. From here, our understanding of the propagation rule in a R-GCN is for each relation multiplying its adjacency matrix with the current feature's representation and then with the corresponding weight matrix. We end up with r different matrices for the next layer, which we sum up into the final version of the next layer.

Modeling on the larger graphs and creating the 3-D adjacency matrix requires a lot of computational power, which limits our ability to model on the big knowledge bases. Moreover, we use Colab and our personal computer which has the GeForce GTX 1050 video card and 16 GBs of memory. This further restricts our abilities for testing. That is why we limit ourselves to only the AIFB and the MUTAG data sets [12]. Our choice for graphs is influenced by [5] as we are trying to replicate their system. We also take graphs from the code of the original R-GCN implementation, which removes relations that were used to create entity labels. As suggested in [5], we create a unique one-hot vector for each node in the graph, which serves as the input to the model.

The basis-decomposition regularization reduced the performance of our model and we decided to drop it. The hyperparameters remain similar to the original implementation - cross-entropy loss, Adam [13] with a learning rate of 0.01, two hidden layers with 16 hidden neurons, L2 regularization, and 50 epochs. Interestingly, it almost always reaches 100% train accuracy even before the last epoch. Because we work with a considerately smaller train and test sets we observe a large bias of around $\pm$3.5% between different model trainings on the AIFB data set. That is why we measure the performance of our R-GCN by averaging 50 different test accuracies.

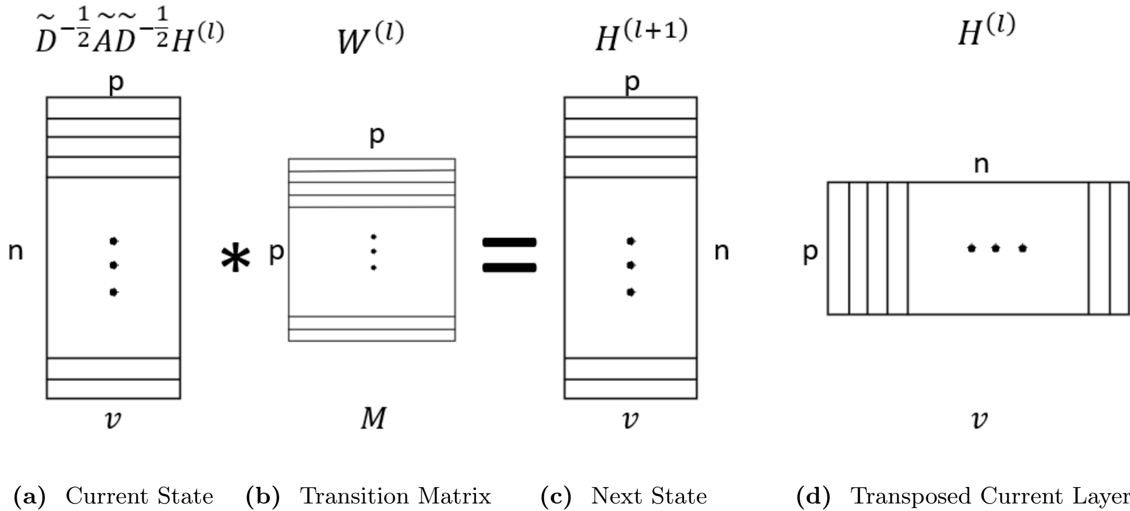
4 Applying Markov Chain and PageRank Theory

We propose two approaches for applying the Markov chain theory to both the GCN and R-GCN. The first one focuses on the structure of neural networks in general and can be used on different variations, while the second one exploits the topology of the input graph and is specifically designed for the two models we use. The main difference between the two methods is how we interpret the transition matrix and the current state of a Markov chain.

4.1 First Approach

Since we have a single weight matrix when modeling on regular graphs, we can assign it to be the transition matrix of a Markov chain. Figure 3 (b) presents it for p number of features in the field of neural networks and p number of nodes in the realm of Markov chains. On its top, we observe $W^{(l)}$, which means it is the weight matrix of a GCN, while on the bottom, we see M, hence it is the transition matrix of a Markov chain too.

Figure 3 (a) shows the other part of a Markov chain - the current state (we will shortly explain why it is in the form of a matrix). It is also the result of multiplying the normalized adjacency matrix with the current layer of the GCN. From here, finding the next layer (figure 3(c)) is multiplying figures 3 (a) and 3 (b).



(a) Current State **(b)** Transition Matrix **(c)** Next State **(d)** Transposed Current Layer

Figure 3

Regarding modeling on knowledge graphs, we need to take into account that we have multiple adjacency matrices and weight matrices. This means that for every next layer we have r different matrices like the one in figure 3 (c). We sum them up to obtain the next layer. Since we already have a single matrix for the current layer, the first approach for combining the Markov chain theory to R-GCNs looks similar to the one described for GCNs. On a side note, we can consider a more abstract point of view that we have r GCNs stacked together, where r is the number of relations.

We now introduce the Markov layer, which is the collection of current states and next states accompanied by a transition matrix. For example, the first three matrices fall into

the same Markov layer. Furthermore, it would also include all new next states.

With the notion of a Markov layer, we can introduce the Markov loss and the Markov depth. The latter is the number of multiplications with the transition matrix. For instance, we have a Markov layer of depth one in figure 3. This number increases when we multiply the next state with the transition matrix. However, this would not change the Markov layer. Regarding the Markov loss, it is the mean absolute error (MAE) between the first and the last next state of the Markov layer. Hence, it sets boundaries for the Markov depth because if it is less than two, the Markov loss is the MAE between the same two matrices - figure 3 (c).

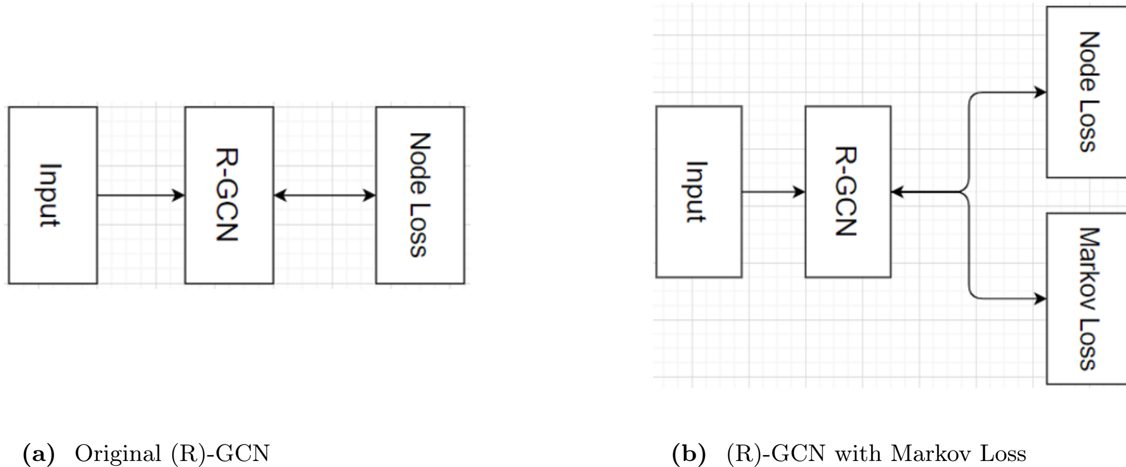
We use the Markov loss to optimize the weight matrix of the GCN or R-GCN in such a way that we can reach the stationary distribution v in a few multiplications. The number of temporary new states before v depends solely on the transition matrix because we know from section 2 that a Markov chain can reach v no matter of the initial state. In an ideal scenario, we have v for the current state after only one multiplication with figure 3 (b). Having a stationary distribution for the current layer of a GCN or R-GCN, is equivalent to the Markov loss being zero. When this happens, we talk about a stabilized system.

$$FinalLoss = \gamma * ModelLoss + (1 - \gamma) * MarkovLoss \tag{7}$$

The first approach incorporates two different losses (Markov and model), which we need to combine. This gives us the advantage that we can alter the natural flow of events of the models by giving more priority to one of the losses. Priority or weight of a loss is a value between zero and one, which is defined by γ in Eq. 7. For example, by decreasing γ, we give more weight to the Markov loss, or we force the model to search for an approximation of the stationary distribution v.

So far, we have defined a few terms. Now, we focus on the constraints of the first approach and how to deal with them. The first problem we face is the state of the current layer and weight matrices of a GCN or R-GCN. Markov chains have positive values, representing probabilities. Hence, each entry in the current state and the transition matrix should be positive. Furthermore, the prior should sum up to one, as well as each column of the transition matrix. Therefore, a normalization procedure for both parts of a Markov chain is required. It includes subtracting for each column of a weight matrix or a transition matrix, its smallest value to all other values in the same column. Then, we normalize column-wise to sum up to one. Regarding hidden layers or current states, we do not need to worry about negative values because we use ReLU (Eq. 1 and Eq. 4), which produces only positive numbers or zero. Hence, we only normalize the matrix to sum up to one.

The normalization procedure includes changing the model's parameters and state representations, which can lead to overfitting. That is why in the first approach we decide to compute the two losses in parallel. Figure 4 (a) shows the normal execution of a R-GCN. It is directly taken from [5], but it can be applied to GCNs, too. The input is fed into the neural network and the system's task is to optimize the performance of the GCN or R-GCN. With figure 4 (b), we illustrate the parallelism of both losses. We create copies of the current layer and the weight matrices, which we normalize. Therefore, the Markov loss does not influence the state of the current layer, because it does not change it explicitly. We use the loss to alter the results of backpropagation with the help of Eq. 7. In a sense, we change the current layer implicitly. This implicit change has some benefits. We do



(a) Original (R)-GCN

(b) (R)-GCN with Markov Loss

Figure 4

not need to worry about linearity with the first approach, as we do not use an activation function for the Markov loss. We will see how this changes in the second method.

Earlier, we defined the normalization of figures 3 (a) and 3 (b) as column-wise. However, we have drawn rows in the figures. The more accurate interpretation of figure 3 is $v^T = v^T M^T$. We normalize the matrices by rows, which is the same as taking the transpose. If we want to strictly follow Eq. 5, we can take the transposed current layer depicted in figure 3 (d). It again does not look exactly like a current state, because of its dimensionality. When we sum it by the rows, we obtain the original current state, which represents the probability of each feature. Nevertheless, the structure of the matrix in figure 3 (d) keeps more information because for each node we have the distribution of the probabilities of its features. That is why we can say that with the first approach, we perform a feature optimization procedure.

We also need to follow a few other constraints, some of which are described in paragraph 2. Firstly, we need square weight matrices. Secondly, we also need to deal with aperiodicity and irreducibility and lastly, the weight matrices should be the same for a Markov layer, no matter the depth.

The first constraint is rather logical when thinking about Markov chains, but it imposes some restrictions on graph neural networks. If we want to create a network that reduces or increases the number of features (autoencoder), we need to create intermediate layers that produce square weight matrices. Then, we can use the Markov layer to compute the Markov loss. This is certainly a disadvantage as it can also increase the execution time of the system.

We also do not have a lot of control over the second issue. Having a reducible or periodic graph means the transition matrix or the weight matrix of the GCN or R-GCN is not regular. Our approach to this problem is to propose the ingenious solution of Google to add teleportation. Hence, the problem of non-regular Markov chains is transferred to PageRank. Furthermore, we argue that no matter the input graph we never have dead ends. For GCN we use undirected graphs and for R-GCN we have inverse relations. Plus, both models have self-loops. The lack of dead ends does not solve the second issue and we still need teleportation, however, the sum of every state is always one.

4.2 Second Approach

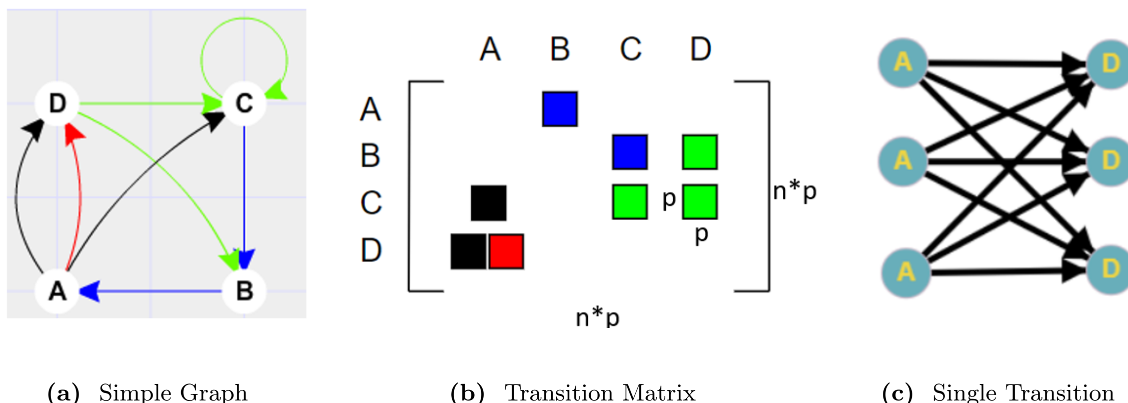
The first approach is somewhat an addition to an existing R-GCN or GCN layer. With the second method, we change the layers of the models, while the weight matrices remain the same.

Some of the biggest differences are how we interpret the current state and the transition matrix. When we look at figure 5 (a) we see a simple knowledge graph with 4 nodes. Each vertex has some connection to the other ones, where the color of the edge represents the relation type. This means we have four relations in the graph. To make the Markov layer, we create a specific weight matrix for it, which is shown in figure 5 (b).

The problem is that each node in the graph has p number of features. Therefore, the transition from node A to D would look like figure 5 (c). From the picture we can deduct that p is three because of the number of identical nodes. Each feature of node A is connected to all the features of node D with some probability which we can obtain from the black weight matrix. The color black comes from the connection type between A and D. This applies to the whole figure 5 (b). We use the specific weight matrix for the transition from any two nodes if an edge between them exists. For example, node C has a blue connection with vertex B and a green connection to itself. Hence, the transition matrix for the column of node C has two weight matrices with their corresponding colors and positions as shown in the figure.

The second method is also applicable to GCNs. If we imagine we do not have relations then the graph has only one type of edges. This is certainly a simplification as the transition matrix would look the same as in figure 5 (b) but without the colors.

The current state of this Markov chain is similar to the one used in the first approach. We take the matrix in figure 3 (a) which has n rows and p columns and we transform it to (n*p) rows. Basically, we just change its dimensionality. Because we are creating a Markov chain we need to normalize its parts. As the problem is similar to the one in the first approach, we apply the same procedure for the current state - the n*p vector, and for the big transition matrix. However, we need to mention that with this method we are not dealing with transposed matrices[6].



(a) Simple Graph **(b)** Transition Matrix **(c)** Single Transition

Figure 5

[6]The difference is that the transition matrix of approach two has its columns sum up to one, while the one of the first method has its row sum up to one.

An issue with the second approach is the dimensionality of the transition matrix. We have n number of nodes and each entry of figure 5 (b) holds a p*p matrix. Therefore, it is (n*p) * (n*p). For our experiments, we need to create a new one for every epoch. Although we manage to use it for smaller graphs, it can easily lead to scalability issues. However, the advantage of this approach is that we manage to integrate the adjacency matrix. Hence, it also depends on the topology of the input graph and is far more relatable to the particular structure of the GCN and R-GCN.

Some of the other drawbacks include the need for square weight matrices and the issues with periodicity and reducibility. We use the same solution as in the first approach. For the prior, we again propose the use of an intermediate layer which produces the required dimensions for the weight matrices, while for the second issue - teleportation. If we look at the transition matrix of the Markov layer (figure 5 (b)) we can observe another complication - the connections between nodes A and D. Vertex A has a double connection with vertex D with two different relations. This means we need to put two weight matrices in the same place. We propose in such situations taking the sum of both matrices. Then, we argue that the resulting entry at location M_{AD} holds information of all relations that node A has with node D.

With everything explained so far we introduce the Markov layer for the second approach. While in the first method it is more of a whole Markov chain, here a Markov layer is only one multiplication between the current state and the transition matrix. It surely bears a stronger resemblance to a normal neural network layer. If we have met all the requirements and constraints, after some number of Markov layers the system would stabilize because an approximation of the stationary distribution is reached. Then, adding more Markov layers would not make any difference to the performance of the GCN or R-GCN. However, we need to change the activation function. It is understandable that because of the Markov chain properties the transition matrix from figure 5 (b) and the current state would only have positive values. In contrast to the first approach, where we left the normal execution of the GCN or R-GCN, in the second method we have overwritten the normal layers of the model with the special Markov layers. Hence, choosing ReLU(x) makes no sense. That is why we create the activation function PositiveReLU(x) = max(0.1,x). It is the same as ReLU but with an added bias of 0.1. That value can be chosen to be everything bigger than zero.

5 Experiments

Despite all the theory and concepts explained, our task is to classify nodes in citation networks and knowledge bases. We are particularly interested in applying Markov chains and generalizing the problem of the number of hidden nodes, but we also look for considerably accurate models.

The code of the six systems (plain GCN and R-GCN and each with the two approaches) is available at `https://gitlab.com/petkov.1231/bachelor-project`. We mainly train and test on the citation data sets Cora and Citeseer [11] for the GCN. The task is semi-supervised node classification with only 140 nodes for Cora and 120 nodes for Citeseer, similar to [4]. Both approaches require a lot of computational power. Besides, the citation network Pubmed has too many nodes and edges and that is why we are unable to make

extensive testing on it. Regarding our R-GCN input, we utilize the AIFB and MUTAG data sets [12]. Because of the way we create the adjacency matrix of each graph we are not able to load the other two networks used in [5] for node classification.

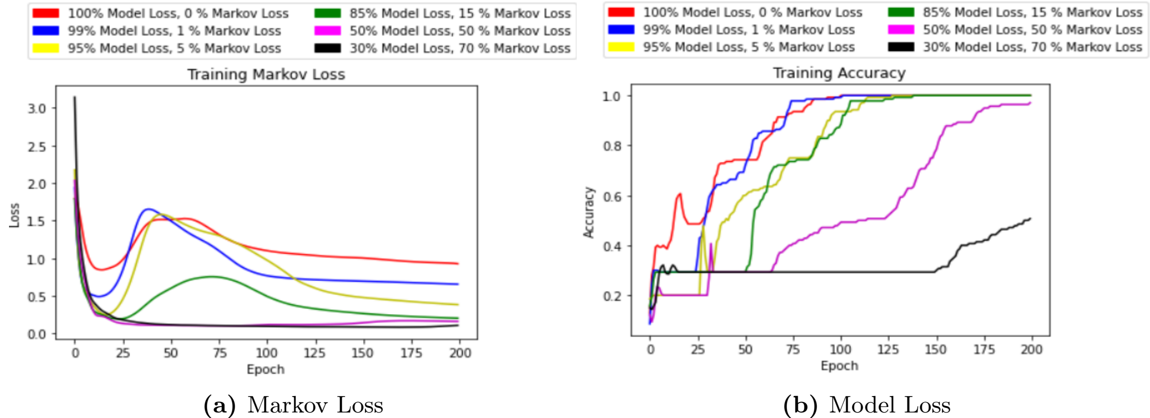
We set two baselines for our experiments. Firstly, we aim at having implementations of the GCN and R-GCN with similar or at least comparable accuracies to the original models. That is particularly important because otherwise, we cannot claim that any results we obtain from the two approaches would be relatable to any other systems for machine learning on graphs. Secondly, we try to create an implementation with the property that after some point adding more hidden layers is meaningless because theoretically and practically the model does the same operations. While the test accuracy is not our primary focus and we do expect some worsening in this area, we still want to create a system that manages to classify nodes relatively correctly.

5.1 First Approach Results

Here, we try to optimize the Markov loss. As it is the difference between the current state/layer and an approximation of the stationary distribution if we have a loss of zero the model is stabilized. We start with some testing for γ in Eq. 7. The Markov depth is initially kept at three.

Figure 6 (a) shows how the Markov loss changes for each epoch on the Cora data set. Each line depicts different values of γ as described in the legend of the graph. For example, the blue line shows the optimization of the Markov loss when $\gamma = 0.99$. Although we leave only 1% priority to the Markov loss and 99% priority to the normal loss we still see that the blue line decreases. In general, the trend is that with more epochs the values of the Markov loss are reduced. The weight γ defines how low the Markov loss can go. For example, for $\gamma = 0.95$ we reach a lowest point of around 0.4, while for $\gamma = 0.85$ - 0.2. There is, however, some inconsistencies between epoch 25 and 75. If we look at figure 6 (b) we see the opposite for the training accuracy. The model struggles to learn in the first epochs, but between the 25 and the 75 epoch the different lines increase their values substantially. For example, the green line has some inconsistencies in the Markov loss graph around the 60th epoch, while in figure 6 (b) we see that around the 50th - 60th epoch it starts learning. In general, for all lines the first 25 epochs the Markov loss takes priority, then for the next 50 to 75 epochs the model focuses on the training accuracy and for the last 100 epochs, the GCN optimizes the Markov loss again. That is to say that the two losses interfere with each other. The results are similar for the other citation network - Citeseer.

Previously we mentioned that the value of the Markov loss is computed by taking the MAE from the first and the last next states. However, originally it does not start from 3 for Cora as illustrated in figure 6. To have this value, we multiply it with 500 000. Citeseer has a similar loss of around 1.5 after normalization with the scalar. The value of 500 000 is chosen to have close values of the Markov loss and the model loss, which starts from around 2. For modeling on the two knowledge graphs, the constant is chosen to be 3 000 000. This produces a Markov loss of around 6 for AIFB and 1.5 for MUTAG with a model loss of around 1.5 on both data sets. Our arguments for the normalization procedure are that having negligibly small values for the Markov loss leads to a final loss mainly composed of the model loss. In this scenario, the model loss might overpower the other one. Furthermore, it is easier to view and compare them if they are of similar scales.



(a) Markov Loss **(b)** Model Loss

Figure 6: GCN on the Cora Data Set

The results of γ testing for the R-GCN are illustrated in figure 7. The graph of AIFB is almost the same as MUTAG's, so it is not shown. Here, the two losses are optimized in parallel unlike the case of GCNs. The Markov loss in figure 7 has a stable trend of decreasing with the increase of epochs. The inconsistency comes from how γ relies to the minimum reached value. Although we can argue that usually a higher γ poses stronger restriction to the amount of Markov loss (the red line comes at around 0.75 and the blue one at around 0.6 at the 50th epoch), we see that the purple line has slightly higher loss than the yellow one throughout the whole execution of the R-GCN. This trend is particularly odd for the green and black ones. We would imagine that forcing the model to optimize the Markov loss or choosing $\gamma = 0$ would perform the best, however we see that between epoch 15 to 35 the green line has lower loss than the black. After that, they have similar values, which we still did not expect.

GCN (Markov depth = 3)			R-GCN (Markov depth = 3)		
Method	Citeseer	Cora	Method	AIFB	MUTAG
Original [4]	**70.3%**	**81.5%**	**Original [5]**	**95.83%**	**73.23%**
2-layered	67.47	79.19	2-layered	89.16%	69.59%
3-layered	63.33%	75.92	75% Model Loss	87.22%	69.59%
99% Model Loss	61.48%	74.68%	50% Model Loss	87.33%	69.59%
95% Model Loss	59.20%	72.77%	10% Model Loss	85.83%	68.29%
85% Model Loss	55.42%	70.19%	1% Model Loss	79.89%	66.79%
50% Model Loss	46.54%	64.8%	0.1% Model Loss	72.94%	67.65%
30% Model Loss	44.24%	47.15%	0% Model Loss	26.39%	24.53%

Table 1: Results First Approach

So far we have demonstrated the ability of our two models to learn the Markov property (Eq. 5) by fixing the depth of the Markov layer at three. The corresponding accuracies are shown in table 1. Decreasing the priority of the model loss influences the models' performances. It is worth explaining why the GCN needs three-layered architecture. One of the constraints of both approaches is the need for a square weight matrix. The Cora and Citeseer networks come with 1,433 and 3,703 input features, respectively. We could create

weight matrices with such dimensions, however, as the first method is computationally costly we reduce the hidden neurons to 16 as in [4]. This reduction is made possible with an extra hidden layer with weight matrix of dimensions (1433,16) for Cora. The disadvantage is we lose accuracy of our GCN model - from 67.47% to 63.33% for Citeseer (table 1). On the contrary, the R-GCN does not come with input features. Hence we can create the unique one-hot encoded matrix of size (n,16). Furthermore, we believe we satisfy our baseline to have the GCN and the R-GCN models with performance relatable to the original ones. On all four data sets we have worse accuracies, but we think that the difference is not substantial.

Finally, we test on how the depth influences the performance of our models. Although in the first approach we consider a Markov layer the whole Markov chain, increasing the depth can be viewed as adding a hidden layer. The concepts are similar - we multiply the normalised copies of the matrices in figures 3 (a) and (b). The difference is that we do not use an activation function as there is no such thing in a Markov chain.

Layer Depth	GCN		R-GCN	
	Citeseer	Cora	AIFB	Mutag
3	59.20%	72.77%	85.83%	68.29%
4	59.76%	72.79%	85.67%	68.35%
5	59.08%	72.89%	86.67%	67.94%
6	59.63%	73.68%	86.44%	67.31%
8	58.98%	73.61%	85.44%	67.71%
10	59.45%	73.22%	85.56%	67.59%

Table 2: Markov Depth Accuracy

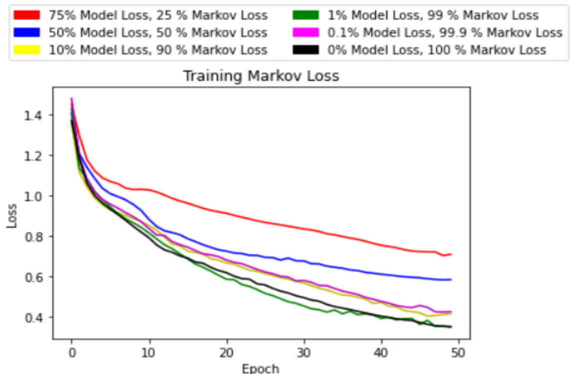


Figure 7: Markov Loss on MUTAG Data Set (R-GCN)

The results are available in table 2. While we initially fixed the depth to make testing for γ, now we do the opposite. Basically, we fix $\gamma = 0.95$ for the GCN (yellow line, figure 6) and $\gamma = 0.01$ for the R-GCN (green line, figure 7) to test on the Markov depth. We choose these values because of our second baseline - we need to take into account model accuracy, too. Indeed, we do not change the current layers of the GCN and R-GCN, but we influence the outcome with the Markov loss. The results indicate that increasing the number of hidden layers inside the Markov layer does not change the models' performances.

5.2 Second Approach Results

We give a more concrete solution to the research question with our second approach. However, we need to solve the problem of linearity caused by the non-negative values of a Markov chain. As we modify the current layer in the second method, we need to find a specific activation function. Using ReLU makes no sense because it would not produce any different results. We perform testing with our R-GCN to find the optimal minimal value of ReLU, which is illustrated in table 3. Clearly, our model gives the best results on both data sets when PositiveReLU(x) = max(0.1,x). We still consider this temporary solution to a bigger issue.

Positive ReLU	Accuracy	
	AIFB	MUTAG
0	50%	66.18%
0.05	52.78%	69.12%
0.1	55.56%	69.12%
0.15	52.78%	63.24%
0.2	51.39%	63.24%
0.3	47.22%	64.71%

Table 3: Min. Value PositiveReLU

Layer Depth	R-GCN			
	AIFB		MUTAG	
	Original (ours)	Markov	Original (ours)	Markov
1	89.16%	55.56%	69.59%	69.12%
2	90.61%	55.56%	69.82%	69.12%
3	88.89%	55.56%	68.71%	69.12%
4	84.5%	55.56%	69.47%	69.12%
5	73.72%	55.56%	65.97%	64.71%
7	43.57%	55.56%	64,26%	64.71%
10	41.67%	55.56%	66,18%	64.71%

Table 4: Results R-GCN on 2nd Approach

As already mentioned, with the second approach we give a different definition to a Markov layer. We drop the concept of the Markov loss, while the Markov depth is somewhat the Markov layer itself. The final results are shown in Table 4. Before viewing them, we need to explain what we mean by layer depth. It is the number of hidden layers. However, they share the same weight matrices.

Moving to the results in Table 4, we see our R-GCN has the best performance on both data sets with a layer depth of 2. If we try increasing the number of hidden layers, we observe that even without the Markov chain property (Eq. 5) the difference between the accuracies starts to become smaller and smaller, which is what we want to prove with this thesis. By giving a larger layer depth, we see that the performance of our R-GCN oscillates around 66% for MUTAG and converges at around 41% for AIFB[7].

The difference between the two input graphs is that with increasing the number of hidden layers the results on AIFB show a significant drop, reaching 43.57% and 41.67% accuracy for seven and ten layers, respectively. On the contrary, on MUTAG even with a large layer depth, we observe a similar performance to the best accuracy of around 66%. This characteristic of the graphs turns out to be very important for our Markov layer performance.

The column named "Markov" in Table 4 represents our findings for the second approach. As we can see, applying the Markov chain property to the R-GCN stabilizes the system. On the AIFB data set, we get a constant accuracy of 55.56%, no matter the layer depth, while on MUTAG we start with an accuracy of 69.12%, but after 4 hidden layers, it decreases to 64.71%. It is safe to assume that adding more Markov layers does not change the performance of the R-GCN.

We think there is a correlation between the expected final Markov accuracy and the accuracy of the original R-GCN with a lot of hidden layers. On AIFB there is a huge drop in performance after a layer depth of seven. We believe this influences the low value of the Markov accuracy. Conversely, we see a relatively good Markov accuracy on MUTAG because there is no drastic decrease in the performance of the original model. Further

[7]We would like to say that the actual results might be slightly different because when testing on the plain R-GCN we get results with big bias. We do average 50 different test results for each entry in the tables, however, one might need to pick a larger number.

research is necessary, where multiple data sets are used, to prove or disprove our claim of correlation.

We use a value of 0.85 for the teleportation β in Eq. 6 for both approaches on the two models. While on the first approach changing β does not influence the results on the second method a higher teleportation probability yields better output.

We tested the second approach on our GCN, however, we did not obtain any valuable results. Initially, we were skeptical about changing the models' parameters and current states because it might cause problems with gradient descent. We think this is the case for the GCN as it cannot learn during training. Here, our model usually does not change the training accuracy no matter of the epoch. We get constant training accuracy of 16.67% and testing accuracy of 7% on Cora.

6 Related Work

In this section, we provide a brief overview of related work on the two main fields for our thesis. The first one is the (relation) graph convolutional networks. The second is the models that try to solve the problem of hidden layers.

A lot of research has been done in the area of graph neural networks. However, we concentrate on [4, 5]. As we have reported their work in section 2, we describe the other field. The first paper that caught our attention creates a system that can be viewed as an infinite depth network for normal[8] neural networks [14]. The advantage of the deep equilibrium model (DEQ) is the substantial memory reduction of up-to 88%. It does not require storing of intermediate activation values, because of its logistics of the backpropagation procedure, which uses constant memory due to implicit differentiation. The authors note the model can be viewed as in infinitely deep network and can also be viewed as a single-layer network. Furthermore, stacking multiple DEQ does not create extra representation power over a single DEQ.

The next paper works on graphs and it introduces a novel approach to computing current states and the learning algorithm for GNNs by a constraint optimization task, solved in the Lagrangian framework [15]. The weights and the current states are simultaneously updated up to a stationary condition. The authors rely on backpropagation for the computations and on constraints, which express the diffusion mechanism. Compared to most of GNN models, their system requires less memory. The DEQ still has an even better performance with regards to memory, however, it is not developed for input in the form of graphs.

The last paper explores the GCN model and its limits [16]. We mention it because the authors talk about how the hidden layers influence the performance of the system. The idea is that using a lot may lead to an over smoothed output. Furthermore, vertices from different clusters may become indistinguishable, and adding more layers increases the complexity when training.

7 Future Research

Here, we focus on the limitations we faced. The first one is the already mentioned issue with linearity in the second approach. While we think that PositiveReLU(x) = max(0.1,x) can

[8]Here, by normal we mean neural networks that do not operate on graphs

work well enough for this thesis, we also believe it is worth exploring other functions. The reason is we only make empirical experiments, which means that the nonlinear function might not work for other cases. It would be interesting to see more testing on PositiveReLU and how other activation functions work in our scenario and with other knowledge bases.

The second issue is the complication with scalability. Although this relates more to the second approach, where using a larger graph is almost impossible, we also note that the first approach is very computationally heavy, too. In the second method, we create a new sparse tensor for the big (n*p) * (n*p) matrix for every epoch. The initializations take more time than the computations. We believe that optimization can be applied to reduce the time used.

With the first approach, we create the Markov loss and we optimize it, while with the second method we directly stabilize the current layer of the R-GCN by Eq. 5. These two approaches are interchangeable. We can easily transform the second approach to create the same Markov loss and vice versa. That is to say, the scope is too broad to explore every variation in this thesis, especially since we test on two different models. With this in mind, we suggest that in future research the goals can be changed to possibly obtain more valuable results.

Finally, we have researched only node classification. There are two other tasks when modeling on graphs. It would be interesting to see if any valuable results can be obtained for creating embeddings and for link prediction.

8 Conclusion

In this thesis, we have created our implementations of the GCN and R-GCN, which perform similar to the originals, and the Markov layer, which builds on top of these two models. Using two different approaches, we have shown how to create a model, for which after some point the number of hidden layers does not influence its performance. We support our system with the theoretical foundation of Markov chains and their explanation of convergence to a stationary distribution. We have also demonstrated how to make a GCN or R-GCN learn the Markov property (Eq. 5) by introducing the Markov loss and optimizing it. Although we have experienced a worsening of the model accuracy, we believe that with further research this problem can be overcome.

References

[1] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[2] Charles Miller Grinstead and James Laurie Snell. *Introduction to Probability*, pages 405–452. American Mathematical Soc., 2012.

[3] A Freedman. Convergence theorem for finite markov chains. *Proc. REU*, 2017.

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

[5] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

[6] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, José Emilio Labra Gayo, Sabrina Kirrane, Sebastian Neumaier, Axel Polleres, et al. Knowledge graphs. *arXiv preprint arXiv:2003.02320*, 2020.

[7] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. *SEMANTiCS (Posters, Demos, SuCCESS)*, 48:1–4, 2016.

[8] Philipp Von Hilgers and Amy N Langville. The five greatest applications of markov chains. In *Proceedings of the Markov Anniversary Meeting*, pages 155–158. Citeseer, 2006.

[9] P Ravi Kumar, Alex KL Goh, and Ashutosh Kumar Singh. Application of markov chain in the pagerank algorithm. *Pertanika Journal of Science and Technology*, 21:541–554, 2013.

[10] Anand Rajaraman and Jeffrey David Ullman. *Link Analysis*, page 139–175. Cambridge University Press, 2011.

[11] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[12] Petar Ristoski, Gerben Klaas Dirk De Vries, and Heiko Paulheim. A collection of benchmark datasets for systematic evaluations of machine learning on the semantic web. In *International Semantic Web Conference*, pages 186–194. Springer, 2016.

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

[14] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. In *Advances in Neural Information Processing Systems*, pages 690–701, 2019.

[15] Matteo Tiezzi, Giuseppe Marra, Stefano Melacci, Marco Maggini, and Marco Gori. A lagrangian approach to information propagation in graph neural networks. *arXiv preprint arXiv:2002.07684*, 2020.

[16] Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.