UiO : **Department of Informatics**
University of Oslo

Graph Classification via Neural Networks

An investigation of the effect of network depth on graph classification, and graph classification via latent node representations

Erlend Løken

Spring 2016

Graph Classification via Neural Networks

Erlend Løken

Master's Thesis Spring 2016

Abstract

For a long time, the preferred machine learning algorithms for doing graph classification have been kernel based. The reasoning has been that kernels represent an elegant way to handle structured data that cannot be easily represented using numerical vectors or matrices. An important reason for the success of kernel methods, is the 'kernel trick', which essentially replaces computing the feature representation, with a call to a kernel function, thus saving computation and memory cost. For some of the most successful kernels in the graph domain however, such as graphlets, this is not feasible, and one must compute the entire feature distribution in order to obtain the kernel. We present experimental evidence that using graphlet features presented to different neural networks gives comparable accuracy results to kernelized SVMs. As neural networks are parametric models that scale well with data size and can yield faster predictions than SVMs, our results suggest that they are attractive models for graph classification. Our experiments show that increasing the depth of the network gives a highly significant speedup in convergence, but no effect on accuracy on our datasets. In addition to this, we present an experimental method of using latent node representations from a method called DeepWalk as input to a neural net for graph classification. This method under-performs both kernel based methods and our graphlet based method. Finally we discuss several ways to extend both graphlet based and embedded representation based classification methods.

Acknowledgements

I would first like to thank my thesis advisors, Vinay Jethava, for introducing me to the field of deep learning, and Geir Kjetil Sandve, for guiding my thesis safely to shore. I would also like to thank Sigmund Rolfsjord for fruitful discussions about neural networks. In addition, I would like to thank all the staff and students at Biomedical Informatics at UiO for providing a fun and stimulating environment for research. I also thank Pinar Yanardag of Purdue University for prompt replies to my emails about kernel methods.

I would also like to thank my parents, Anne and Terje Løken, for supporting me and cheering me on during my studies.

And last, but most I want to thank Linnea Fagernæs for tolerating that I spent all those long evening writing and programming after work.

Contents

Part I

Introduction

Chapter 1

Introduction and motivation

In many branches of science, problems have emerged which involve studying a large set of random variables connected by an intricate dependency network. This dependency network may be explicitly encoded in the data, like in social networks, or it may be latent and hence must be inferred from time series observations of the random variables [38]. The mathematical formalism for studying these dependency networks is known as graph theory, and the application of graph theory to real world graphs has spawned the relatively young but very active field of network science. From failure prediction in power grids to toxicology assessment and social network analysis, the number of problems that can be approached via graph analysis is vast. And because graphs are such general objects, advances in the field of graph and network analysis can often instantly be applied to problems in other fields where it is useful to represent data as graphs or where a graph-based structure can be inferred.

Increasingly, when researchers or people in industry need to make sense of complex datasets, be it object recognition in images, speech recognition, speech translation or stock market prediction [40], they turn to machine learning. Originating as a sub-field of artificial intelligence (AI), machine learning explores the construction of systems that can learn from, and make predictions about, data. These machine learning systems are often attempting to model a complex underlying reality but can observe only the data that is presented to them. For a long time it has therefore been a priority for those who build such systems to hand engineer *features* that are appropriate to the task at hand, such as edge detectors for object recognition systems etc. Coming up with features is "difficult, time-consuming, and requires expert knowledge" to quote Andrew Ng [49]. With the advent of what is called *Deep Learning* we are seeing the emergence of algorithms capable of generating features from raw data with no or minumal human intervention [40]. Current state of the art systems in many of these domains now rely on little or no hand engineering of features. In addition, all major commercial speech recognition systems are now based on deep learning methods [25], [12]. Adoption of these types of algorithms help cut the development time of machine learning systems and lowers the need for domain specific expertise when designing the system.

The last couple of years have seen some applications of modern neural networks to graph and network analysis, such as Deep Graph Kernels [62] which attempt to mitigate the *diagonal dominance* problem encountered by conventional graphlet kernels via the use of a context encoding matrix. Other interesting examples include DeepWalk [52] which uses deep learning for generating latent representations of graph vertices, deep locally connected networks which generalizes convolutional neural networks to graphs via multiscale graph clustering and spectral networks which uses a generalization of the Fourier transform on graphs to learn filters over the eigenvalues of the graph Laplacian. But by and large there has for a substantial time been a domination of methods using hand engineered features such as graphlet kernels [57], [58], [22], [4]. A powerful argument for using kernels is that they in some cases are able to computationally cheaply access an implicit higher dimensional feature space by employing what is called the 'kernel trick'. For many graph kernels however, computing the actual feature representations is necessary in order to obtain the kernel. This means that one might as well exploit the entire feature representation as input to a classifier, instead of restricting oneself to kernelized support vector machines.

A common problem in analysis of graph datasets is graph classification. That is, given a set of graphs belonging to mutually exclusive classes, how can we design a system that can learn which class a given graph belongs to by observing either the graph itself or features derived from the graph. We present two contributions to the graph classification problem. The first is an examination into various neural network architectures' capabilities of using graphlet kernels [57] [58] and structurally smoothed graphlet kernels [61] to do graph classification. The second is a method using the latent node representations from DeepWalk [52] as features for graph classification. We show that neural networks are capable of classifying graphs with the same accuracy as SVMs given graphlet input.

Part II

Prerequisites

Chapter 2

Graph Theory

"Mathematics is the art of
giving the same name to
different things."
Henri Poincaré

"The founders of Google
computed the Perron-Frobenius
eigenvector of the web graph
and became billionaires."
Andries E. Brouwer
Willem H. Haemers

Networks are everywhere. Some of them are obvious, such as your social network of friends, family and acquaintances, or the web. Others are slightly more hidden, such as banking or logistics networks. And finally, some are primarily known to specialist researchers such as cell signaling networks in biology or reaction networks in chemistry. If looking closely, one can see that nature and society are permeated by networks at almost every scale except perhaps the very largest (cosmology) and the very smallest (elementary particles) [18]. So it is with good reason that theoretical research into networks and graphs has become a highly active research area over the last decades: general results in network theory can instantly be applied to fields where data is meaningfully represented as networks. The theory and science of networks is relatively young, but its roots go far back and touch many academic domains. The next section will present a short history of network science. Because of the interdisciplinary nature of the field, the jargon can be somewhat confusing. Erring on the side of oversimplifying we can say the difference between graph theory and network science is that graph theory studies abstract hypothetical mathematical structures, whereas network science attempts to model an underlying existing network using the mathematical abstractions from graph theory or qualitative tools from the social sciences. We will use graphs and networks, nodes and vertices, edges and connections interchangeably throughout the thesis.

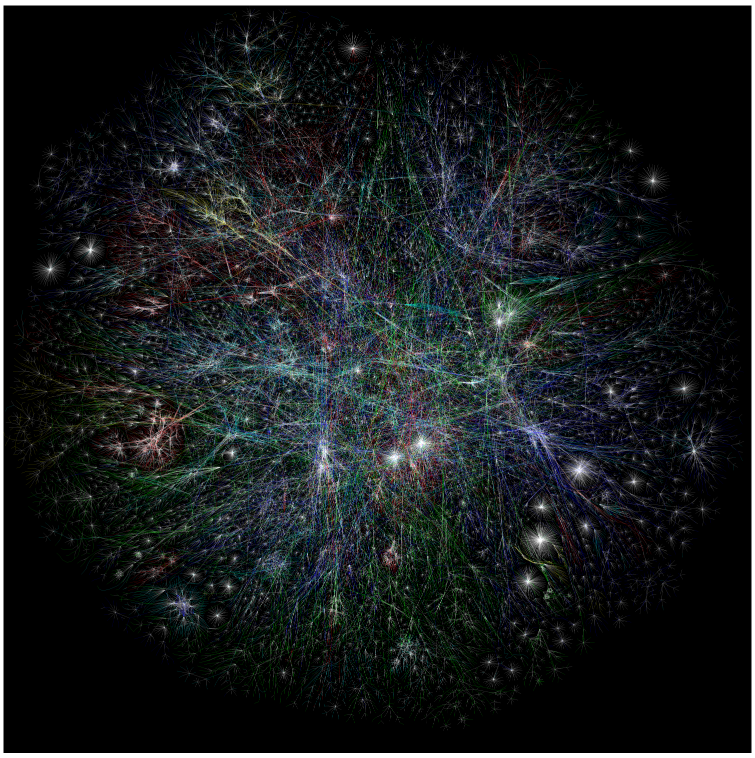

Figure 2.1: Visualization of the internet ca 2005. The bright spots are clusters of densely connected computers. *Source: www.opte.org/maps*

In *A First Course in Network Theory* [18], the authors list several ways to define networks by their connections, ranging from the physical to the more conceptual:

- **Edges representing physical links.** This refers to networks where nodes are physically connected, such as railroad networks, biological neural networks, power grids etc.

- **Edges representing physical interactions.** In some networks, nodes are connected by physical interactions such as the physical interaction among proteins in protein-protein networks.

- **Edges representing 'ethereal' connections.** Some nodes are connected by the exchange of information, irrespective of the exact physical route of the information, as in cellular networks or the web.

- **Edges representing geographic closeness between nodes.** In structures such as maps and biological tissue, we use nodes to denote a region of a surface, and connections to represent the neighbour or nearness relation.

- **Edges representing exchange of mass or energy.** In networks such as food webs, trade networks or metabolic networks, the edges are represented by their transfer of mass or energy between the nodes.
- **Edges representing social connections.** In social networks edges can represent any kind of social tie, such as family, friendship, acquaintance, colleague etc.
- **Edges representing conceptual linking.** Nodes may be conceptually connected to each other such as in academic citation networks or in wikipedia. Almost any given wikipedia article, will lie in a cluster of articles vaguely related to each other.
- **Edges representing functional linking.** Edges can represent functional relationships, such as between parts of certain machine, brain regions etc.

The authors note that these concepts are not disjoint, and that there are cases where it is beneficial to interpret a network using several of these points of view.

2.1 A brief history of graph theory and network science

The branch of mathematics studying graphs stretches back to Eulers 1736 paper on the bridges of the city of Königsberg in Prussia. The city (Now Kaliningrad in Russia) was spread out over both sides of the river Pregel, and two large islands which were connected to each other and the mainland by seven bridges. A puzzle that occupied the people of Königsberg was to find out if it was posssible to have a stroll accross the seven bridges without crossing any of them twice. Euler proved that this was indeed not possible, and with that, lay the seeds for the study of both topology and graph theory. Euler observed that the exact route taken from bridge to bridge on a given landmass was irrelevant. Indeed the only relevant information is the landmasses and their connections. This allows one to abstract the problem into what we today would call vertices (landmasses) and edges (bridges). Euler also observed that during any walk, a person enters a non-terminal landmass as many times as the person leaves it. So if every bridge is to be crossed only once, it follows that each landmass (except the start and ending point of the walk) must have an even number of bridges connecting to it. As all the landmasses in Königsberg are connected by an odd number of bridges, Euler showed that it was impossible to traverse all bridges without crossing one of them twice. Today, a path in a graph which visits each edge only once is called an *Eulerian path.* The invention of graph theory was not much of a Big Bang however. The formalism of what we today call graph theory was invented incrementally over the course of two centuries after Euler's paper. Early important results include William Rowan Hamiltons and Thomas Kirkmans work on hamiltonian paths and cycles. It also found applications in chemistry and it was in this context the term graph was coined, by mathematician James Joseph Sylvester. It took exactly two hundred years from Euler's paper to the publishing of the first book on graph theory by Dénes König, a Jewish Hungarian mathematician. In 1969, the American mathematician Frank Harary also published a textbook on Graph Theory. In it he standardized much of the modern formal terminology and thus broadened the reach of the field to researchers in physics, statistic, social sciences and electrical engineering. Graph theory contains many interesting problems and theorems, a famous one being the *four color theorem*, which can be stated as "given any separation of a plane into contiguous regions, producing a figure called a map, no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color. Two regions are called adjacent if they share a common boundary that is not a corner, where corners are the points shared by three or more regions." This problem was first posed by the South African mathematician Francis Guthrie in 1852 when he noticed that he only needed four colors to color the counties of England. The problem was not solved until over a hundred years later in 1976. Interestingly, it was also the first major theorem to be proved by the aid of a computer.

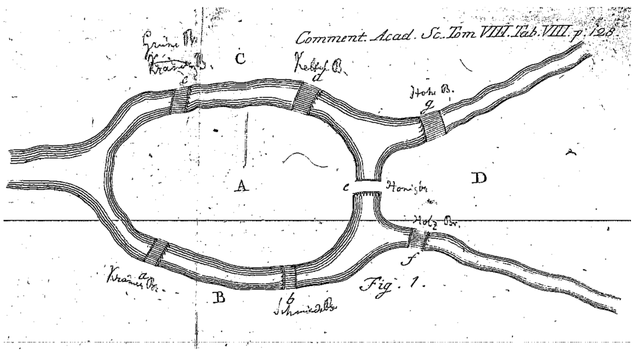

Figure 2.2: Eulers sketch of the Seven Bridges of Königsberg

In 1959-60, Paul Erdõs, Albert Rényi and Edgar Gilbert introduced the concept of random graphs. Up until then graph theory was mostly focused on results concerning graphs where the connectivity is fixed. Conversely, a random graph is a graph where the connectivity is stochastic. As such, the field of random graphs lies at the intersection of probability theory and graph theory and studies distributions over graphs and the properties of random graphs. The first model of a random graph is the *Erdös-Renyi* model $G(n,p)$ where a graph with n vertices is constructed by randomly connecting nodes such that the probability that two nodes are connected is p. The potential for real world applications of their work was understood by Erdös and Renyi, as evidenced by the following prescient quote in their 1960 paper "On the Evolution of Random Graphs" [16]:

> In fact, the evolution of graphs may be considered as a rather simplified model of the evolution of certain communication nets (railway, road or electric network systems, etc.) of a country or some other unit. (Of course, if one aims at describing such a real situation, one should replace the hypothesis of equiprobability of all connections by some more realistic hypothesis.) It seems plausible that by considering the random growth of more complicated structures (e.g. structures consisting of different sorts of "points" and connections of different types) one could obtain fairly reasonable models of more complex real growth processes (e. g. the growth of a complex communication net consisting of different types of connections, and even of organic structures of living matter, etc.).

In the following decades, many candidates for a more realistic hypothesis than the equiprobable connection hypothesis were suggested. Already in 1965, Derek de Solla showed [59] that the numbers of citations a paper received followed a power law, which implied that the citation network was what would later be called a *scale-free* network. A network is called scale-free if the degree distribution of the nodes in the network follows a power law. Interest in scale-free networks increased in 1999 when Albert-László Barabási and Réka Albert independently introduced what we today call the Barabási-Albert model, a model for generating scale-free networks from a process called preferential attachment. The model works as follows: The network begins with a set of m_0 initial nodes that are all connected. Each new node is then connected to $m \leq m_0$ existing nodes with a probability proportional to the connectedness of the existing node. The probability p_i that the new node is connected to the existing node i is

$$p_i = \frac{k_i}{\sum_j k_j} \tag{2.1}$$

where k_i is the degree of node i, and the denominator is the total degree of the network. The result of this is that new nodes have a preference to attach to nodes that already have a high number of connections, leading to the growth of hubs. In parallel with the research in the natural and informational sciences, sociologists started applying networks to the studies of groups of people. In 1933, Jacob Moreno presented the Sociogram [42], a graph based representation of connections in social networks. This method of representation was well received and eventually led to the creation of the field of social network analysis. The interest in studying relational data increased, and after the 1940's what we today call network science was also adopted by social anthropologists, organizational scientists and social psychologists and economists [21]. A famous study is Stanley Milgram's *Small World Problem*. It was the first research to produce evidence suggesting that human society was characterized by short paths, meaning that any two humans on average are removed by only a few degrees of separation. The experiment (technically a series of experiments) was carried out by choosing random people in Omaha and Wichita and asking them to forward a letter to a random person in Boston. If the person knew the Bostonian, he or she was to forward it directly, otherwise the person was to think of another person who might know that person and then forward the letter to them with the same instructions. A postcard was also mailed back to the researchers in order to let them track the letter's journey. From the results the researchers concluded that people in the United States are separated by six people on average, leading to the phrase *six degrees of separation*. Though criticized for having some methodological flaws (selection and non-responsive bias among others), it was nonetheless seen as groundbreaking in shedding a light on the topology human society and remains one of the most cited papers in psychology. Interestingly, the social network site Facebook (1.55 billion monthly users at the moment of writing) estimated the average degree of separation on their entire social network to be 3.57 see Figure 2.3 (with the author of this thesis achieving an average degree of separation of 3.37)[15]. The 90's and onwards saw a surge

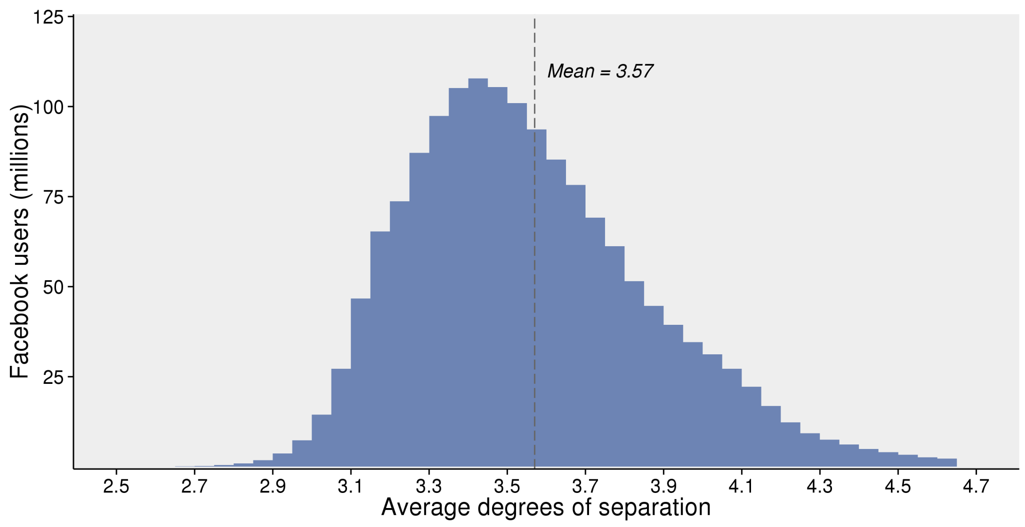

Figure 2.3: The estimated degree distribution of the Facebook social network

in network research in physics and chemistry and biology as it was used to model gene regulatory [34], protein interaction networks[1] and found several applications condensed matter physics and statistical physics [17].

The advent of the information age and its accompanying interconnectedness and exponential growth in data production has further increased the interest in network science. Data from the web now fuel large amounts of research into networks in general and social networks. Examples include modeling the topology of the world wide web [19], predicting cascades of content sharing on social media sites [7], the recurrence of cascades [8] and how connectivity suddenly grows in real world networks [13]. As a recognized field of it's own, however, network science is very young (Cambridge University Press' *Network Science Journal* was only founded in 2013). This is because, as the first editorial of the journal notes[5], network science came about as a result of people in a wide array of sciences focusing on interdependent relations, and borrowing from what had been done in other fields in order to research this.

2.2 Graph Classification

With graph structured data becoming ubiquitous, algorithms for mining and analyzing them for the purpose of doing statistical inference is of growing importance. Problems of interest include graph classification, which the topic of this thesis, node classification, graph compression for compact representation of data, graph clustering for finding important dense subnetworks and link/edge prediction for the prediction of graph evolution. Graph classification and regression has been of special importance in bio- and chemoinformatics, both are fields with large datasets of graph structured data. In areas like drug development for example, it is important to identify molecules that are active towards intended targets but not towards other targets. In 2012 the pharmaceutical company Merck organized a Kaggle competition offering \$40 000 to the creators of the most accurate algorithm for molecular activity [45]. Interestingly, graph classification has also been used to help programmers debug noncrashing buggy code by identifying program regions that lead to faulty code [43].

2.3 Key Concepts from Graph Theory

This section covers the basic mathematical formalism and terminology of graph theory and several key concepts useful for the material further on. A graph is a representation of a set of objects and their relations. Formally it is an ordered pair $G = (V, E)$ where $V = \{v_1, v_2,...,v_n\}$ is the ordered set of vertices (or nodes) and $E \subseteq V \times V$ is the set of edges between the vertices. Two edges are called *adjacent* if they are connected by an edge, or more formally; two vertices $v_i, v_j \in V$ are *adjacent* iff $(v_i,v_j) \in E$. Similarly, two vertices are *incident* if they share a vertex, i.e. all edges $(v,v_i) \in E$ are *incident* on v. We denote the size of the graph by $|V|$, in this case n. If

$(v_i, v_j) \in E \iff (v_j,v_i) \in E$ the graph is called *undirected*, otherwise it is *directed*.

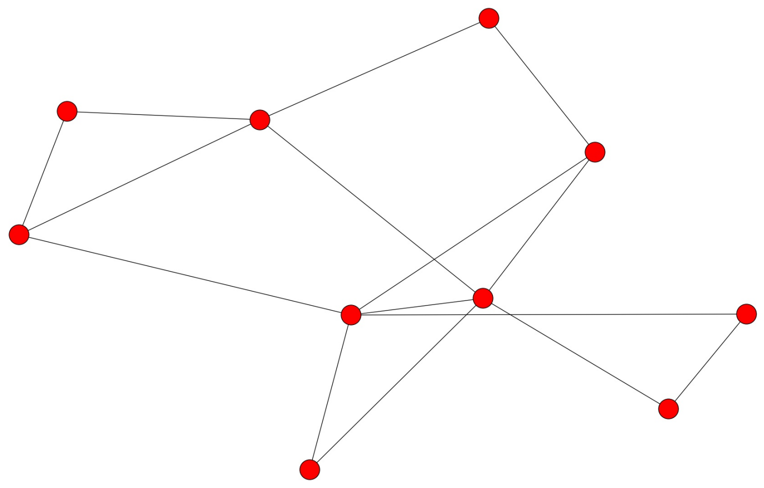

Figure 2.4: A random graph

Given a graph $G = (V_G,\ E_G)$ and a graph $H = (V_H,\ E_H)$ we say that H is a *subgraph* iff there is an injective mapping $\alpha : V_H \rightarrow V_G$ s.t. $(v,w) \in E_H \iff (a(v),\ a(w)) \in E_G$, and denote it by $H \sqsubseteq G$. That is, if V_H is a subset of V_G and E_H is a subset of E_G, then H is a *subgraph* of G. If for every $v_i, v_j \in V_H$ we have that $(v_i,v_j) \in E_H \iff (v_i,v_j) \in E_G$ we say that H is an *induced* (or full) subgraph of G. An intuitive way to understand the concept of induced subgraphs is that they are the subgraphs that you can obtain by deleting nodes and their incident edges from G. If all pairs of vertices of an undirected graph are adjacent, we say that the graph is *complete*. An *induced subgraph* that is complete is called a *clique*. If the clique is not a subset of any other clique, it is called a *maximal clique*. We say that the *neighbourhood* of a graph $N(v)$ is defined as $N(v) = \{v_i \in V|(v,v_i) \in E\}$. We define the degree of a vertex as $deg(v) = |N(v)|$. An edge is called a *self loop* if it is of the form (v_i,v_i). If a graph is *undirected* and has no *self loops* we say that is is a *simple graph*.

We say that two graphs G and H are *isomorphic* if there exists a bijective mapping $\beta : V_G \rightarrow V_H$ such that $(v_i,v_j) \in E_G \iff (\beta(v_i),\ \beta(v_j)) \in E_H$. Deciding if two graphs are isomorphic is a non-trivial problem that holds an interesting place in computational complexity theory. It is clearly in NP, while it is suspected not to be NP-complete and not to be in P.

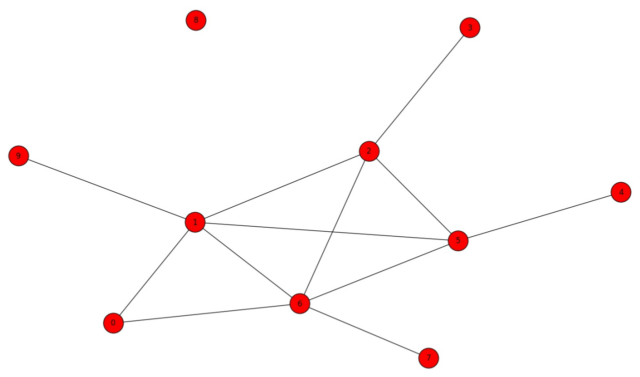

Figure 2.5: A random graph with a maximal clique consisting of nodes 1,2,5,6

2.4 Matrix Representations of Simple Graphs

Graphs can often be represented using matrices. This is very convenient because by employing matrix representations we can turn certain analysis of graphs into a linear algebra problems, for which we have a vast machinery of efficient algorithms and methods at our disposal. Indeed it has been show that many important properties of a graph is encoded in the characteristic polynomial, eigenvalues, and eigenvectors of its matrix representations [23]. We therefore review some of the most important matrix representations in this section.

Adjacency Matrix A simple graph G of size n can be represented as an $n \times n$ matrix A where the entries $a_{i,j} = 1$ iff $(v_i, v_j) \in E$ and 0 otherwise. The adjacency matrix of a simple matrix is symmetric.

Incidence Matrix The incidence matrix of a graph is a $|V| \times |E|$ binary matrix M with $m_{i,j} = 1$ iff v_i is incident upon edge e_j and 0 otherwise.

Degree Matrix The degree matrix of a graph contains information about the degree of each vertex. It is a diagonal matrix D such that $d_{i,j} = deg(v_i)$ for $i = j$.

The Graph Laplacian The graph laplacian is defined as $L = D - A$ for a simple graph, where A is the adjacency matrix and D is the degree matrix.

The entries of L are thus given by

$$l_{i,j} = \begin{cases} deg(v_i) & \text{if } i = j \\ -1 & \text{if } i = j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Chapter 3

Machine Learning & Neural Networks

"Let the machine take care of the machines, and I'll go spend more time with my family, or golf."

Mark Goddard

In order to understand how neural networks can be applied to the problem of graph classification, it is necessary to have an understanding of Machine Learning in general, and Artificial Neural Networks in particular. The former is a sub-field of Artificial Intelligence (A.I.) which deals with methods for making computers learn patterns from data, and the latter is a particular set of data structures and algorithms used to achieve this goal. Machine learning is commonly divided into three main types [27]:

- **Supervised Learning** In this type of learning we are trying to learn a hypothesis $h : X \to \mathbf{y}$ which maps from the input data domain to the outputs (or training signal). If the output set $\mathcal{Y}$ is finite, we say that the task is a *classification* task, if it is continuous we say that it is a *regression* task.

- **Unsupervised Learning** If all we have is the input data X and no output data to guide our training, the task is called *unsupervised* learning. The focus in this type of learning is to discover hidden structure in the data. Common problems in unsupervised learning are *dimensionality reduction* and *clustering* of the input data.

- **Reinforcement Learning** In reinforcement learning we have a situation where the learning algorithm interacts with an environment and is trying to learn how to behave. The cues for whether or not the algorithm is behaving optimally is given only occasionally in the form of (usually) scalar reward values. Reinforcement learning can be viewed as a a form of semi-supervised learning where the training signal is sparse and delayed. It can also be viewed as planning in a domain with stochastic transitions.

As the focus on this thesis is graph classification, the scope of our discussion of machine learning extends only to the supervised learning problem. We will start this chapter by a quick overview of the theory of statistical learning, followed by an introduction to several neural network architectures such as logistic regression, multilayer perceptron and convolutional neural networks. The chapter then finishes with a discussion on the specifics of training neural networks regarding optimization, computing the gradient, and regularizing the network.

3.1 Statistical Learning Theory

Statistical learning theory provides a theoretical framework for the discussion of machine learning, grounding it in probability theory and statistics [44], [20]. This section presents some basic concepts from this field which allows us to compare different machine learning methods and test them in a theoretically rigorous fashion. Again, we restrict the discussion to supervised learning, as it is the most relevant part for this thesis. For the mathematical description of neural networks and backpropagation, we will borrow notation from Michael Nielsen's book *Neural Networks and Deep Learning* [50].

Let $\mathcal{X}$ be the vector space of possible inputs and $\mathcal{Y}$ be the space of all possible outputs, and let $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$ be the product space. We assume that there is a hidden distribution $P(\mathcal{Z}) = P(\mathcal{X},\mathcal{Y})$ over this product space. Denote by $\mathcal{H}$ the *hypothesis space* of functions $f : \mathcal{X} \to \mathcal{Y}$. The total product space $\mathcal{Z}$ is unobserved, but we have a training set $S = \{(\mathbf{x_1},y_1),(\mathbf{x_2},y_2),...,(\mathbf{x_n},y_n)\} = \{z_1,z_2,...z_n\}$ of n i.i.d samples drawn from $P(\mathcal{Z})$. The supervised learning problem in its essence, is to use the training data from S to pick a hypothesis f_H from $\mathcal{H}$ such that $f_H(\mathbf{x}) \approx y$ for hitherto unseen data from $P(\mathcal{Z})$. For our purposes, f_H usually takes the form of a parametrized function.

3.1.1 Training & Model Performance

Conditioning the model to draw reasonable hypotheses' from $\mathcal{H}$, is called training, and consists of finding the model parameters that maximizes or minimizes a certain performance criterion $J(\theta)$, where θ is the set of model parameters, that quantifies how the model performs. How a model is trained vary across model families. Certain linear models for instance, allow for closed form estimation of optimal parameters for certain performance criterions. For neural networks however, iterative approaches employing numerical optimization methods must be used. As the focus of this thesis is classification, this discussion will primarily cover performance measures relevant to this. Functions of this type are referred to as loss functions and penalize erroneous classification and reward correct classification. Several loss functions have been suggested in the machine learning literature. An intuitive performance measure for a classifier is the *zero-one loss*, or misclassification rate:

$$J = \frac{1}{N}\sum_{i=1}^{N} I_{f(x_i)\neq y_i}$$

where N corresponds to the number of training examples, I is the indicator function:

$$I_x = \begin{cases} 1, & \text{if } x \text{ is True} \\ 0, & \text{if } x \text{ is False} \end{cases}$$

and $f(x_i)$ and y_i refer to the model output and correct label for example i, respectively. We see that by minimizing this objective function, we get parameters that maximizes the accuracy on the training set S. A serious drawback with the zero-one loss however, is that is prohibitively expensive to optimize for large models as it is not differentiable. This has led to the adoption of other loss functions such as Mean Square Error and Cross Entropy:

$$J = \frac{1}{2N} \sum_{i=1}^{N} ||f(x_i) - y_i||^2 \tag{3.1}$$

$$J = -\sum_{i=1}^{N} \log \left(f(\mathbf{x_i})^{y_i} (1 - f(\mathbf{x_i}))^{1-y_i} \right) \tag{3.2}$$

While both are used frequently for classification purposes in both research and industry, cross-entropy, which corresponds to the negative log-likelihood of a series of Bernoulli trials, is a more natural fit for the classification setting.

3.1.2 Model complexity, generalization and regularization

A sufficiently complex model (complex in the sense of number of trainable parameters, also referred to as *model capacity*) can be taught to learn its training data perfectly. But it is the ability of the method to perform well on unseen data, called *generalization*, that matters for most practical supervised learning tasks. If a model performs well on training data, but poor on test data, we say that the model has overfitted. This means that the model is putting to much weight on the peculiarities of the training set. If a model fails to get good performance on both the training and test set we say that the model underfits. A model that underfits lacks the complexity necessary to express the relationship between the features and the labels. A model generalizes well when its performance on the training set translates to the test set.

A rigorous method for finding optimal model capacity given a problem, or dataset, is still an open problem in machine learning. A common heuristic method for solving this problem is to build models with excessive complexity and then penalize model complexity through a process known as regularization [11]. Regularization refers to an extensive set of methods applied to model training that reduce overfitting, and is a highly active research area. A common method for regularization is the introduction of

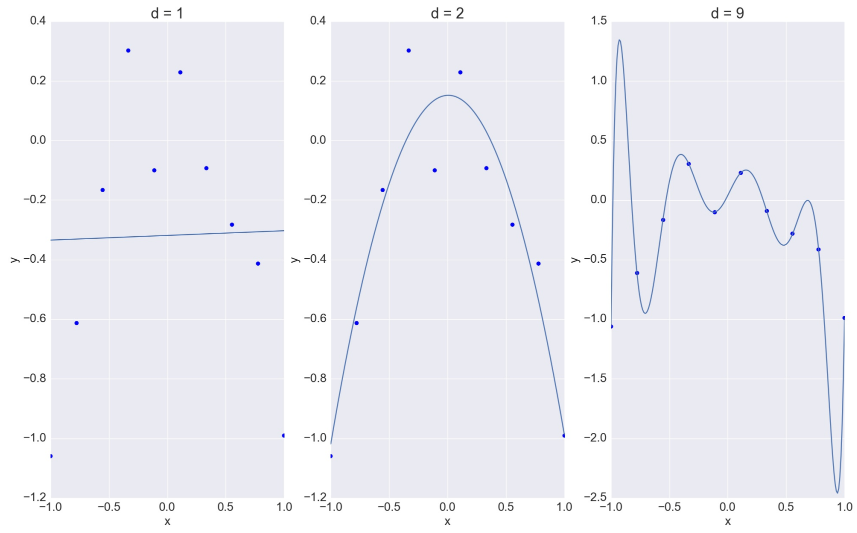

Figure 3.1: An illustration of overfitting. The function $f(x) = -x^2 + \epsilon$ ($\epsilon \sim N$(0,1)) is fitted with polynomials of degrees 1,2 and 9. The model in the leftmost figure is unable to capture the nonlinear relationship and underfits. The model in centre figure achieves a good fit. The model in the rightmost figure fits all datapoints perfectly but oscillates significantly between them, indicating overfitting.

a regularization term to the loss function. This regularization term is often a function of the the model parameters, thus large parameter values (which tend to increase model variance and overfitting) are penalized during training for giving the model a higher loss. For machine learning algorithms trained using numerical optimization, a regularization technique called early stopping is also often employed. Early stopping refers to terminating the training procedure when the model stops improving on the validation criteria. There are several variants of this technique, and they will be discussed in more detail in the section on Neural Network Training.

3.1.3 Model Evaluation

Evaluating a given model has two primary steps, the first is to select an evaluation criterion. The second step is to validate the model's performance in a methodologically sound fashion.

3.1.3.1 Performance Criteria

For the binary classification task, which corresponds to many of the experiments in this thesis, there a several evaluation criteria available, most of which relate to the relative cost of the error types seen in table 3.1. A Type

Table 3.1: Statistical decision making and error-types

		Predicted Condition	
		Predicted Positive	Predicted Negative
True Condition	Positive	True Positive	False Negative Type II Error
	Negative	False Positive Type I Error	True Negative

I error refers to predict positive when the truth is negative (rejecting a true null hypothesis), Type II error refers to predicting negative when the truth is positive (failing to reject a false null hypothesis). There is usually a trade-off between these two errors, which is why choosing a suitable evaluation criterion is important. The most common evaluation criterion reported on several benchmarking datasets in machine learning (MNIST, CIFAR-10, CIFAR-100) is accuracy: $ACC = \frac{\sum \text{True Positive } + \sum \text{True Negative}}{\text{Total Population}}$. This is a sensible measure to pick if the cost of false negatives and false positives are reasonably similar. In practical applications however, the cost of the two types of errors can be very different. In cancer detection systems for instance, it can be acceptable to increase the risk of false positives in order to minimize the number of false negatives (undiagnosed cancers). In such a case, a criterion like false negative rate $FNR = \frac{\sum \text{False Negatives}}{\sum \text{Condition positive}}$ can be a good choice.

3.1.3.2 Cross-Validation

In order to evaluate the performance of a model in a sound way it is common to split datasets into training and validation sets. The training set is used for estimating model parameters, and the validation set is used to gauge the model's performance on unseen data. Some people include a third set, the test set, that is not presented to the model until all hyperparameter tuning is done. The reason for this is that information can *leak* from the validation set and into to the model during repeated training and validation attempts as one eventually ends up trying to optimize against the validation set. There is a trade-off in deciding how much of the data to use for training and how much to use for validation. The more data one uses for training, the better the model will be at generalizing. However, this will lead to greater uncertainty in the model's performance on unseen data, as this now is estimated form a smaller sample. A class of methods that attempt to mitigate this trade-off this is called cross-validation. There are many variations of cross-validation, we will only discuss the one applied in the experimental part of this thesis, stratified K-fold cross-validation. This is performed by partitioning the dataset into K equally sized *folds* in such a way that the statistical properties of the response value of all the folds are roughly the same. For binary classification this means equal proportions of both classes in each fold. After the dataset is partitioned, the model is trained on $K-1$ folds, and evaluated on one fold, K times. The folds are rotated in such a way that each fold is used exactly once as validation data. In addition to sidestepping the training/testing trade-off, cross validation gives researchers K measures of model performance, which can be used to infer expected performance on unseen data, and the variance of the performance. For a thorough discussion of the different types of cross-validation and the properties of the validation estimates we refer the reader to [27].

3.2 Neural Network Architectures

The history of neural networks is long, and involve researchers and works spanning multiple generations, so we will not have the space to deal with it extensively here. For a good summary of the history we recommend Jürgen Schmidhuber's excellent review article [56].

The recent resurgence in neural network research can to a great extent be explained by the creation of neural architectures and hardware capable of mitigating the effect of what is known as the *vanishing gradient problem*, which refers to an exponential weakening of the training signal in the number of layers, leading to slow convergence. In 1997 Hochreiter & Schmidhuber presented the Long Short-Term Memory model [31], which implements a gating system for 'locking' the network state during training. Yann LeCun and others created Convolutional Neural Networks [10], [41], [39], which drastically reduce the numbers of parameters needed in the network, which again speeds up training. In 2006 Geoffrey Hinton published the Deep Belief Network [30]. A neural network where the each layer was pretrained using unsupervised learning, and later fine-tuned using back propagation. In addition to the theoretical contributions from machine learning researchers, developments in the hardware space and the creation of very large datasets have contributed significantly towards the feasibility of deep learning as a technology. The advent of highly parallel GPU computing, in particular, has significantly reduced the training time for most neural architectures.

This section will discuss the different neural network architectures deployed in our experiments.

3.2.1 Logistic Regression

Logistic regression is simple machine learning model that can be thought of as a one-layer neural network for binary classification (no hidden layers):

$$P(y = 1|\mathbf{x}) = f(\mathbf{x}) = \frac{1}{1 + \exp(W^T\mathbf{x})}$$

$$P(y=0|\mathbf{x}) = 1 - f(\mathbf{x})$$

Where $\mathbf{x}$ is the training example and y is the example label. While not really a neural network, the Logistic Regression (or Softmax regression in the multinomial case) is useful to analyze because many other neural network architectures have a logistic regression/softmax as the output layer. This means that by comparing the results of logistic regression with more complex architectures we can gauge the importance of the feature learning performed in the hidden layers of the more complex architectures. Logistic regression is usually trained numerically by minimizing the cross-entropy via stochastic gradient descent, which will be explained further into this chapter.

3.2.2 Multilayer Perceptron

This family of neural networks have many names in addition to Multilayer Perceptron (MLP); vanilla neural networks, fully connected network, or simply feedforward neural network. It refers to neural networks consisting of L layers, an input layer, an output layer and $L-2$ hidden layers. Each neuron in layer l is connected to each neuron in layer $l-1$ via a matrix W^l of trainable weights.

More formally: Let $(\mathbf{x}, y)$ be training data sampled from some unknown data generating distribution $P(\mathbf{X},\mathbf{Y})$. Let $f_{\boldsymbol{\theta}}$ be a parametrized family of functions describing how a model will behave on new examples from the data generating distribution. Training the model means adjusting the parameter set $\boldsymbol{\theta}$ given the training data. Let $J \in \mathbb{R}$ denote the aforementioned objective function describing the loss associated with the algorithms prediction $\hat{y} = f_{\boldsymbol{\theta}}(\mathbf{x})$ and the target $\mathbf{y}$. With these definitions in mind we can define a feedforward neural network as a set of layers acting on the output of the previous layers. We define the j-th neuron in the l-th layer as

$$a_j^l = \sigma\left(\sum_k w_{j,k}^l a_k^{l-1} + b_j^l\right) \tag{3.3}$$

where the sum runs over the K neurons in the previous layer, $w_{j,k}^l$ denotes the weight mapping from the neuron a_j^l to the neuron a_k^{l-1}, b_j^l denotes a trainable bias term and $\sigma(\mathbf{x})$ is the non-linear activation function (usually sigmoid, hyperbolic tangent or rectified linear unit) which is applied element-wise to its inputs. In vector notation

$$\mathbf{a}^l = \sigma\left(W^l \mathbf{a}^{l-1} + \mathbf{b}^l\right) \tag{3.4}$$

Additionally, we define $a_j^1 = x_j$, that is, the first layer is the training example presented to the network. From this definition we see that we can interpret a neural network as a sequence of nested affine transformations with non-linearities stacked between them. The final layer is then used as input in the cost function. Showing the input as a single vector is a simplification done for notational simplicity, in most actual implementations of such neural networks, the data is presented to the network in mini-batches consisting of several training examples. This is done in part because mini-batches tend to give more stable gradient estimates than single examples, and in part because mini-batch training utilizes matrix-matrix multiplication which is highly efficient on modern GPUs [29]. Like logistic/softmax regression, these models are usually trained by numerically minimizing a cost function via stochastic gradient descent. It is worth noting that for a long time, training architectures with multiple hidden layers was impossible due to the computational cost of calculating the gradient of the cost function. This was only made feasible by the discovery of the backpropagation algorithm [55], which is discussed in detail in section 3.3.2.

3.2.3 Convolutional Neural Network

An architecture that is related to the multilayer perceptron and now widely adopted for many supervised learning problems is the Convolutional Neural Network [40]. The CNN architecture differs from that of the MLP in the following important ways [36]:

Layers are composed of volumes. Layers in a multilayer perceptron have one dimension, often referred to as width, which corresponds to the number of neurons in the layer. In convolutional neural networks however, layers are organized in volumes that have width, height, and depth.

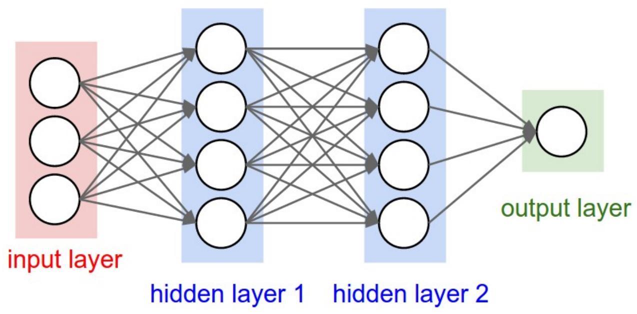

Figure 3.2: A traditional neural network with layers represented as columns of neurons (Picture from Stanford's CNN for Visual Recognition course: http://cs231n.github.io/convolutional-networks/).

Different types of layers. An MLP only implements one type of layer, consisting of a non-linear activation function that transforms the weighted input to the layer. A CNN usually implements several types of layers such as convolutional layers, pooling layers and fully connected layers (MLP's). The pooling layer stands out in that it does not contain any trainable parameters,

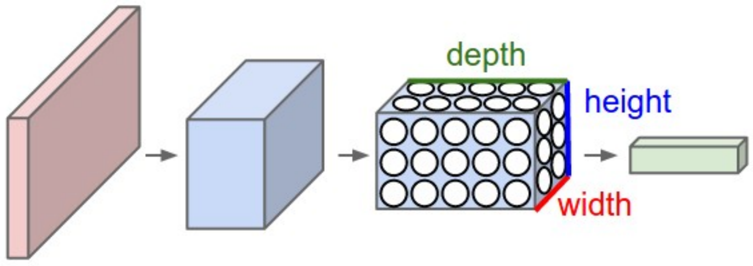

Figure 3.3: A CNN: Notice how the layers are represented as volumes. The different *filters* are stacked along the depth dimension (Picture from Stanford's CNN for Visual Recognition course: http://cs231n.github.io/convolutional-networks/).

but instead implements a downsampling of the input data from the previous layer that serves to create an invariance to small shifts and distortions in the feature, while at the same time reducing the overall dimension of the representation [40].

Local connectivity. Each neuron in an MLP is fully connected, meaning that there is a weight from every input to the neuron. In a convolutional neural network, the neurons are normally only connected to a small region of the input volume, called the neurons *receptive field*. The implication of this is that we get a drastic reduction in the number of weights that need to be trained, which speeds up training and memory consumption. If we imaging an MLP being trained to classify a $300 \times 300 \times 3$ images (the third dimension is for color data), each neuron in the first layer needs $300 \times 300 \times 3 = 270\ 000$ parameters to be trained. Contrast this with a neuron in the input layer of a convolutional neural network with a $10 \times 10 \times 10$ receptive field which only has 300 parameters that need to be trained.

Parameter sharing Another technique that reduces the number of parameters needed in a CNN is parameter sharing. This refers to the fact that for a given depth of a layer, all the neurons along the width and height dimension share the same parameters. This constraint means that we can view the forward pass of the network in each depth dimension as a convolution (hence the name) of the inputs with a *filter* i.e. the weights.

The architectural aspects of CNNs listed here have several important implications for their performance. One is that with the reduction in trainable parameters achieved from the use of a receptive field and parameter sharing, they can be scaled more easily to larger datasets. Another consequence is that it is easier to explicitly see what a CNN actually is learning. Each of the *filters* in the lower layers typically learn basic features. In image recognition this typically corresponds to something akin to Gabor filters etc. The next layer of the network then combines these features to build more abstract high level features. This process continues up until the last output volume of the network which holds the class probabilities for a given input. For a thorough discussion of the mathematical operations that separates CNNs from conventional feedforward architectures we refer the reader to [14].

3.2.4 Maxout Networks

A relatively novel innovation in neural networks is the Maxout network [24]. These networks are designed to leverage a highly effective regularization technique, dropout, that is described in section 3.3.3.2. In a maxout network, a hidden layer implements

$$a_i^l = \max_{j \in 1,K} z_{i,j} \tag{3.5}$$

where $z_{i,j} = {a_i^{l-1}}^T W_{i,j} + b_{i,j}$, $W \in \mathbb{R}^{d \times m \times k}$ and $b \in \mathbb{R}^{m \times k}$. That is, instead of passing the result of an affine transformation elementwise through an activation function like relu, the network computes K such affine transformations and for each element in the hidden layer, choose the max of the K outputs. The authors of [24] shows that a maxout layer with arbitrarily many affine components and just two hidden units can approximate any given convex function. In a sense this lets the network learn which activation function is the most appropriate for the task at hand, and then implement it.

3.3 Training Neural Networks

This section covers the peculiarities of training neural networks. It starts with a brief discussion of how gradient-based neural network training relates to optimization and a description of some of the specific optimization algorithms used. This is followed by a description of the backpropagation algorithm, the dominating method for finding the gradient of the objective function. After that follows a discussion of how to do regularization in neural networks.

3.3.1 Optimization and neural network training

Training neural networks is done by minimizing a cost function subject to model parameters. As such, the theoretical framework of mathematical optimization has been of significant importance when developing the methods for training neural networks. However, as discussed in the upcoming book *Deep Learning* [2], there are several differences between traditional optimization and neural network training that need to be taken into consideration in order to successfully train these models. For instance we might be interested in minimizing a performance measure P, such as the misclassification rate. However, as this is undifferentiable it is an inefficient choice of loss function, we therefore choose to indirectly minimize P via a proxy measure J. Another difference is that we can only compute a proxy of the cost function as we do not have access to the full data generating distribution $P(\mathcal{X},\mathcal{Y})$. So instead of observing the expectation of the cost function according to the data generating distribution:

$$J(\theta) = \mathbb{E}_{(\mathbf{x},y)\sim P} L\left(f(\mathbf{x};\boldsymbol{\theta}),y\right) \tag{3.6}$$

we observe the average of cost function for the empirical distribution obtained from the training set S:

$$J^*(\theta) = \mathbb{E}_{(\mathbf{x},y)\sim \hat{P}} L\left(f(\mathbf{x};\boldsymbol{\theta}),y\right) \tag{3.7}$$

where $\hat{P}$ is the empirical data distribution, L is the loss function for a single data point and $f(\mathbf{x};\boldsymbol{\theta})$ is the predicted output of the model given data $\mathbf{x}$ and parameter values $\boldsymbol{\theta}$. An additional layer of abstraction is added by the fact that we don't want to minimize 3.7 directly as this tends to lead to overfitting from memorization of the dataset. Another peculiarity is that because the objective function can be decomposed as a sum of differentiable functions (one per training sample), and the accuracy of the estimation of the gradient scales sub-linearly in the number of examples, the most efficient training algorithms are not based on evaluating the cost function over the entire dataset, but rather on smaller batches. This lead to the development of the most common class of optimization methods for training neural networks: Stochastic Gradient Descent. For a thorough discussion on the potential pitfalls and challenges of training neural networks we refer to [2].

3.3.1.1 Training Algorithms

This section covers Stochastic Gradient Descent (SGD), one of the most common optimization algorithms for training neural networks, SGD with momentum and AdaGrad which is a more advanced optimization algorithm.

Stochastic Gradient Descent
SGD is essentially normal gradient descent evaluated on minibatches of the data instead of the the entire training set. It is show in [2] that sampling minibatches from the data generating distribution and averaging over them can yield unbiased estimates of the gradient. The SGD algorithm is:

Algorithm 1: Stochastic Gradient Descent

```
Input: Learning rate $\epsilon_k$, Initial parameter $\boldsymbol{\theta}$
while stopping criterion not met do
    Sample a minibatch of $m$ examples $(\mathbf{x},y)$ from the training set S.
    Compute gradient estimate $\hat{\mathbf{g}} \leftarrow +\nabla_{\boldsymbol{\theta}}\frac{1}{m}\sum_i L(f(\mathbf{x}^{(\mathbf{i})};\boldsymbol{\theta}),y)$.++
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\mathbf{g}}$++
end
```

SGD with Momentum

The objective of deep architectures often have so called *ravines* that are steep in directions not leading to a minima, and relatively flat in the direction of the minima [2]. In order to speed up convergence it is common to add a momentum term to the gradient update. This introduces a degree of inertia in the gradient updates that can speed up convergence. Pseudocode for stochastic gradient descent with momentum is given below.

```
Algorithm 2: Stochastic Gradient Descent with Momentum
  Input: Learning rate $\epsilon_k$, momentum parameter $\alpha$, initial parameter $\boldsymbol{\theta}$,
         initial velocity $\boldsymbol{v}$
  while stopping criterion not met do
      Sample a minibatch of $m$ examples ($\mathbf{x}$,$y$) from the training set S.
      Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_{\boldsymbol{\theta}}\frac{1}{m}\sum_i L(f(\mathbf{x}^{(\mathbf{i})};\boldsymbol{\theta}),y^{(i)})$
      Compute velocity update: $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \epsilon\hat{\mathbf{g}}$
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end
```

AdaGrad

When training neural networks with vanilla SGD it is common to set what is called a *learning rate schedule.* This is because empirical experiments have shown benefits to use a large learning rate in the beginning of training and the slowly decreasing it. AdaGrad, short for Adaptive Gradient, is an optimization method that adjusts the learning rate independently for each parameter at each step, based on previous previous gradients. A weakness in its design is that it accumulates squared gradients in the denominator, making the learning rate effectively zero with enough training. Pseudocode for AdaGrad is given below.

```
Algorithm 3: The AdaGrad Algorithm
  Input: Global learning rate $\epsilon$, small constant $\delta$, initial parameters $\boldsymbol{\theta}$
  Initialize gradient accumulation variable $r = \mathbf{0}$.
  while stopping criterion not met do
      Sample a minibatch of $m$ examples ($\mathbf{x}$,$y$) from the training set S.
      Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_{\boldsymbol{\theta}}\frac{1}{m}\sum_i L(f(\mathbf{x}^{(\mathbf{i})};\boldsymbol{\theta}),y^{(i)})$
      Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + g \odot g$
      Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta+\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end
```

RMSProp

A very popular optimization method that attempts to rectify the problems with the vanishing learning rate is RMSProp [29]. Interestingly, it has never been published properly, so it is usually cited from a set of slides from a Coursera online course on Deep Learning.

```
Algorithm 4: RMSProp
Input: Global learning rate $\epsilon$, decay rate $\rho$, initial parameters $\boldsymbol{\theta}$, small
       constant $\delta$
Initialize accumulation variables $\mathbf{r} = \mathbf{0}$.
while stopping criterion not met do
    Sample a minibatch of $m$ examples $(\mathbf{x}, y)$ from the training set S.
    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_{\boldsymbol{\theta}} \frac{1}{m} \sum_i L(f(\mathbf{x^{(i)}}; \boldsymbol{\theta}), y^{(i)})$
    Accumulate squared gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho)\, \mathbf{g} \odot \mathbf{g}$
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end
```

3.3.2 Backpropagation

The most common way of obtaining the gradient used for the various optimization algorithms mentioned is Backpropagation. Fundamentally, it is an algorithm for calculating derivatives quickly, and it has many applications beyond the training of neural networks. Because of this, it has been reinvented multiple times throughout history, in different academic disciplines. It was the 1986 paper *Learning Representations by Back-Propagating Errors* [55] that demonstrated how effective the method was for learning neural networks and lead to its wide spread adoption in the machine learning community [50]. The application independent name for the method is reverse-mode differentiation [51]. The focus in this exposition of backpropagation is on simplicity rather than generality, as we believe this is better for gaining an intuitive understanding for what the algorithm does. We will therefore present it specifically in the case of obtaining gradients for a multilayer perceptron with MSE for cost function. For an excellent and thorough discussion on the more general aspects of back propagation we refer the reader to [2].

The general intuition behind backpropagation is as follows: pass training examples trough the network from the input layer all the way to the output layer. At the output layer, compute the error for the cost function used in the network. Then, propagate the error backwards from the output layer all the way to the first hidden layer and use it to compute the gradient. The error terms can then be used to obtain the derivative of the cost with regard to the individual weights and biases in the network. More formally, assume a multilayer perceptron with L hidden layers. Let W^l denote the weight matrix s.t. $W^l_{j,k}$ is the weight for the connection from the k-th neuron in the $(l-1)$-th layer to the j-th neuron in the l-th layer. Denote by b^l the bias vector s.t. $\mathbf{b}^l_j$ is the bias term for j-th neuron in the $l-th$ layer. Let $\mathbf{a}^l = \sigma\left(W^l \mathbf{a}^{l-1} + \mathbf{b}^l\right)$ be the activation vector for the l'th layer (The first hidden layer has the activation vector $\mathbf{a}^2 = \sigma\left(W^l \mathbf{x} + \mathbf{b}^l\right)$ where x is the training example). In order to ease notation we denote by $\mathbf{z}^l$ the weighted neuron input $\mathbf{z}^l = W^l \mathbf{a}^{l-1} + \mathbf{b}^l$. We define the cost function as

$$J = \frac{1}{2n} \sum_x ||y(x) - a^L(x)||^2 \tag{3.8}$$

where n is the number of training examples, $y(x)$ and $a^L(x)$ is the label and network output for training example x respectively. The objective is to find $\frac{\partial J}{\partial W^l_{j,k}}$ and $\frac{\partial J}{\partial b^l_j}$ such that they can be used for minimizing J. We define the error of neuron j in layer l as

$$\delta^l_j = \frac{\partial J}{\partial z^l_j} \tag{3.9}$$

or in vector notation $\boldsymbol{\delta}^l = \nabla^l_{\mathbf{z}} J$. Backpropagation offers a computationally inexpensive way to find these errors for all L layers, and then relate them to $\frac{\partial J}{\partial W^l_{j,k}}$ and $\frac{\partial J}{\partial b^l_j}$. This is primarily done through four equations. The first equation computes the error of the output layer:

$$\delta^L_j = \frac{\partial J}{\partial z^L_j} \tag{3.10}$$

which by applying the chain rule can be expressed as

$$\delta^L_j = \sum_k \frac{\partial J}{\partial a^L_k} \frac{\partial a^L_k}{\partial z^L_j} \tag{3.11}$$

As $\frac{\partial a^L_k}{\partial z^L_j} = 0$ for $j \neq k$, this simplifies to

$$\delta^L_j = \frac{\partial J}{\partial a^L_j} \frac{\partial a^L_j}{\partial z^L_j} \tag{3.12}$$

which, since $a^L_j = \sigma\left(z^L_j\right)$, we rewrite as

$$\delta^L_j = \frac{\partial J}{\partial a^L_j} \sigma'\left(z^L_j\right) \tag{3.13}$$

or

$$\boldsymbol{\delta}^L = \nabla_{\mathbf{a}^L} J \odot \sigma'(\mathbf{z}^L) \tag{3.14}$$

in vector notation, where $\odot$ denotes the Hadamard product (elementwise matrix multiplication: $(\mathbf{s} \odot \mathbf{t})_{i,j} = s_{i,j} t_{i,j}$).

The second equation relates the error of layer l to the error of layer $l+1$, thus allowing for the backwards propagation of errors. The error of node j in layer l is defined as

$$\delta_j^l = \frac{\partial J}{\partial z_j^l} \tag{3.15}$$

Using the chain rule, we can write this in terms of layer $l+1$

$$\delta_j^l = \sum_k \frac{\partial J}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{3.16}$$

where the sum is over the parent nodes in layer $l+1$. Noting that $\frac{\partial J}{\partial z_k^{l+1}} = \delta_k^{l+1}$ we rewrite this as

$$\delta_j^l = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} \tag{3.17}$$

Recalling the definition of weighted neuron input we can rewrite z_k^{l+1}

$$z_k^{l+1} = \sum_j w_{k,j}^{l+1} \sigma(z_j^l) + b_k^{l+1} \tag{3.18}$$

and differentiate, in order to obtain

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{k,j}^{l+1} \sigma'(z_j^l) \tag{3.19}$$

Inserting this in 3.17 we get

$$\delta_l^j = \sum_k w_{k,j}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \tag{3.20}$$

or in vector notation:

$$\boldsymbol{\delta}^l = \left(\left(W^{l+1}\right)^T \boldsymbol{\delta}^{l+1} \right) \odot \sigma'(\mathbf{z}^l) \tag{3.21}$$

The third equation relates the derivative of the cost wrt. the bias of layer l, $\frac{\partial J}{\partial b^l}$ to the errors. We begin by noting that

$$\frac{\partial J}{\partial b_j^l} = \frac{\partial J}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} \tag{3.22}$$

Simplifying the two first factors on the right hand side via the chain rule, we get

$$\frac{\partial J}{\partial b_j^l} = \frac{\partial J}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \frac{\partial z_j^l}{\partial b_j^l} \tag{3.23}$$

Recalling that $z_j^l = \sum_k w_{j,k}^l a_k^{l-1} + b_j^l$, we see that

$$\frac{\partial z_j^l}{\partial b_j^l} = 1 \tag{3.24}$$

Inserting in 3.23 we get that the derivative of the cost wrt. the bias is exactly the error

$$\frac{\partial J}{\partial b_j^l} = \delta_j^l \tag{3.25}$$

The fourth equation expresses the derivative of the cost wrt. an individual weight in terms of error, and is found in much the same way

$$\frac{\partial J}{\partial w_{j,k}^l} = \frac{\partial J}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{j,k}^l} \tag{3.26}$$

Again simplifying the two first derivatives via the chain rule we get

$$\frac{\partial J}{\partial w_{j,k}^l} = \frac{\partial J}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{j,k}^l} = \delta_j^l\frac{\partial z_j^l}{\partial w_{j,k}^l} \tag{3.27}$$

Evaluating $\frac{\partial x_j^l}{b_j^l}$ and inserting in 3.27, finally yields

$$\frac{\partial J}{\partial w_{j,k}^l} = a_j^{l-1}\delta_j^l \tag{3.28}$$

With these four equations in hand we can outline a simple version of the backpropagation algorithm in pseudocode

Algorithm 5: Backpropagation

```
Input: Training example $\mathbf{x}$, corresponding label $\mathbf{y}$
Assign training example to be the first layer: $\mathbf{a}^1 \leftarrow \mathbf{x}$
for $l = 2,3,...,L$ do
    Compute weighted neuron input: $\mathbf{z}^l \leftarrow W^l\mathbf{a}^{l-1} + \mathbf{b}^l$
    Compute neuron activation: $\mathbf{a}^l \leftarrow \sigma(\mathbf{z}^l)$
end
Compute loss: $J \leftarrow \frac{1}{2}||\mathbf{y} - \mathbf{a}^L||^2$
Compute error of final layer: $\delta^L \leftarrow \nabla_{\mathbf{a}} J \odot \sigma'(\mathbf{z}^L)$
for $l = L-1, L-2,...,2$ do
    Compute error of layer $l$: $\boldsymbol{\delta}^l \leftarrow \left(\left(W^{l+1}\right)^T \boldsymbol{\delta}^{l+1}\right) \odot \sigma'(\mathbf{z}^l)$
    Compute gradient wrt. weights: $\frac{\partial J}{\partial W^l} \leftarrow \boldsymbol{\delta}^l {\mathbf{a}^{l-1}}^T$
    Compute gradient wrt. biases: $\frac{\partial J}{\partial \mathbf{b}^l} \leftarrow \boldsymbol{\delta}^l$
end
return $\nabla_{W,\mathbf{b}} J$
```

3.3.3 Regularization

Because of their potentially high complexity and number of model parameters, neural networks can be prone to overfitting unless measures are taken. This section discusses some specific regularization techniques popular for regularizing neural networks.

3.3.3.1 Parameter Norm Regularization

A very common type of regularization is what is called *parameter norm regularization.* They work by limiting the representational capacity of the model by adding a penalty function of the parameter norm to the objective function.

$$\tilde{J}(\boldsymbol{\theta};\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{\theta};\boldsymbol{X},\boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

The two most common ones are $L1$ and $L2$ regularization with the respective penalty terms $\Omega(\theta) = \frac{1}{2}||\boldsymbol{w}||_1$ and $\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{w}||_2$. Speaking loosely we can say that the difference is that $L1$ regularization promotes sparse parametrization while $L2$ spreads the penalty more evenly throughout the weight matrix. For a thorough discussion of parameter norm regularization we refer the reader to [2]

3.3.3.2 Dropout

Many machine learning competitions are won not by a single model, but by averaging the output of an ensemble of models [29]. One framework for doing this is what is called bagging, which is short for Bootstrap Aggregation. The essence of bagging is to train k different models on one of k different datasets generated by sampling with replacement from the original dataset. At test time all models are evaluated on the input data and the output is averaged. If the errors of the k models are not perfectly correlated, this will lead to a more robust prediction than any one of the k models would be able to give [27]. The drawback from this is that it is computationally expensive at test time. Dropout attempts to approximate this process for a single neural network. It

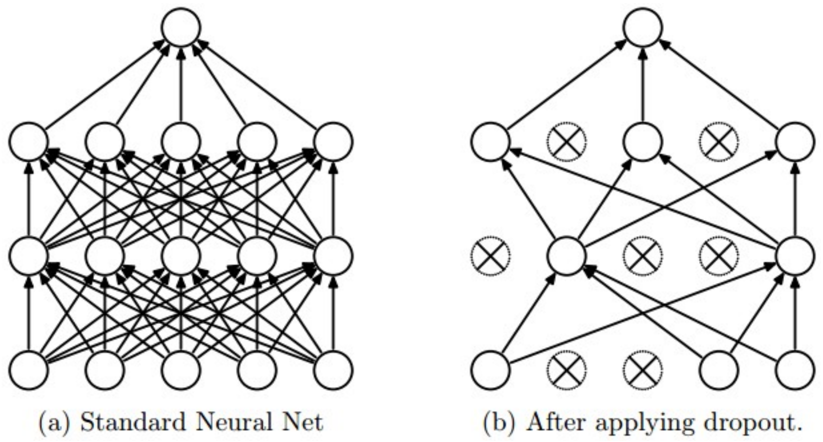

Figure 3.4: Illustration of the dropout idea. Nodes are removed during training time (shown in b)), giving rise to a constrained, sparser network. This prevents the hidden units from co-adapting to much. During testing and deployment the full network pictured in a) is activated. (Picture from the Dropout paper [60]).

works by removing a fraction of the neural nets' nodes at random at training time. This can be done by multiplying the output of a unit by zero. The effect of this is to effectively train by sampling from the exponential number of sub-nets of the full net during training, which forces each unit to be more independent and not co-adapt excessively. At test time, the full network is deployed. The primary difference between traditional bagging and dropout is model independence, with dropout, the sampled models share parameters. As dropout effectively reduces the width of the network, the authors suggest widening the hidden layers of networks when applying dropout, in order to maintain the representational capacity of the network. If n is the optimal size of a layer where dropout is applied, and p is the probability of retaining a neuron at training time, then the authors suggest that the layer should be widened to n/p. Another issue is that dropout introduces a significant amount of noise in the gradients. Srivastava et al. suggest mitigating this by increasing the learning rate by a factor of $10 - 100$, and using a momentum value between $0.95 - 0.99$. They suggest using retention probability p from $0.5to0.8$.

A less commonly discussed version of dropout (but presented in the same paper) is Gaussian dropout [60]. Ordinary dropout can be viewed as multiplying the activation units with Bernoulli random variable which takes the value 1 with probability p and 0 with probability $1 - p$. This can be generalized to other probability distributions. Srivastava et al. reports better results on the MNIST dataset for Gaussian Dropout than Bernoulli dropout. The Gaussian dropout was performed by multiplying the activation units with a Gaussian random variable with $\mu = 0$ and $\sigma = \sqrt{\frac{1-p}{p}}$ where p is the corresponding probability from the Bernoulli Dropout experiment.

3.3.3.3 Batch Normalization

A relatively new method for improving Neural Network training is the popular Batch Normalization [33]. Batch normalization works by normalizing layer input for each training mini-batch. This prevents *covariate shift*, the tendency for the distribution of layer inputs to change during trading. Batch Normalization has shown itself to have several beneficial effects on neural networks. It allows one to train models with much higher learning rates, thus increasing the time to convergence. It is also shown to prevent overfitting in the model. As the whitening of the data is done per batch, and batches are randomized per epoch, batch normalization can be considered a very light type of data augmentation (discussed in 3.3.3.5). Since a naive normalization of input can reduce the representational power of the layer (normalizing input to a sigmoid constrains the output to be in its linear region), two trainable parameters γ and β are added to provide an affine transformation of the normalized value. Batch normalization considers values of x over a mini-batch $\mathcal{B} = \{x_1, x_2,...,x_m\}$. Mini-batch mean is calculated as

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3.29}$$

and mini-batch variance is calculated as

$$\sigma^2_{\mathcal{B}}\frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}}) \tag{3.30}$$

Finally the input is normalized

$$\hat{x} = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{3.31}$$

and transformed

$$y_i = \gamma\hat{x} + \beta \tag{3.32}$$

Note that the whitening is done per dimension individually across each batch, and not over the full batch and all dimensions. Additionaly, batch normalization is a differentiable transformation, which means that back-propagating errors through the transformation works.

3.3.3.4 Early Stopping

Overfitting generally means that the algorithm has gone beyond learning the function that generated the data and instead is learning the noise in the data. This can be monitored by evaluating the algorithm on a validation set after each epoch. A model with high representational capacity will typically have an error that decays rapidly in the start for both the training and validation set. When the error is decreasing for the training set but no longer decreasing for the validation set, that is a strong indicator that the model is starting to overfit. At this point, we can stop training the model.

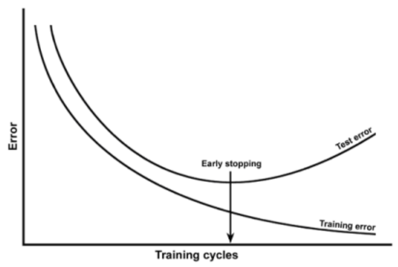

Figure 3.5: Early stopping illustration

There are several ways of doing early stopping in practice. We will in this section review the method used for the experiments in this thesis: using the quotient of generalization loss and progress [53]. Let E be the loss function of the neural network. Let $E_{tr}(t)$, $E_{va}(t)$ be the loss for the training set and the validation set after epoch t, respectively. Let

$$E_{opt}(t) = \min_{t' \leq t} E_{va}(t')$$

denote the minimum loss obtained on the validation set from epoch 0 to t. Let

$$GL(t) = 100 \cdot \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right)$$

denote the generalization loss at epoch t. Define a training strip of length k to be a sequence of k epochs $n+1$,...,$n+k$ where n is divisible by k. Then let training progress be defined as

$$P_k(t) = 1000 \cdot \left(\frac{\sum_{t'=t-k+1}^{t} E_{tr}(t')}{k \cdot \min_{t'=t-k+1}^{t} E_{tr}(t')} - 1 \right)$$

This measures the ratio of the average training ratio to the minimum training error for the strip. Unless the training is globally unstable, this will converge to zero as the number epochs aproaches infinity. The stopping criterion is defined as

$$PQ_\alpha : \text{stop after the first end-of strip epoch } t \text{ with} \frac{GL(t)}{P_k(t)} > \alpha$$

3.3.3.5 Data Augmentation

The best way to get a neural network to generalize well is to feed it large amounts of non-redundant data. However, for many problems data might be scarce. In this case one can simulate a larger dataset by augmenting the existing data. Augmentation methods include rotation, translation, cropping, and adding additive noise. For some tasks, such as classification and object recognition, this works very well. An explanation for this could be that the objective of a neural network in both of these settings is to recognize different instances of an object despite the large amount of variations inherent in the class or object.

Part III

Related Research, Thesis Contribution and Results

Chapter 4

Related Research

"There is nothing so practical as
a good theory"
Ludwig Boltzmann

4.1 Kernel Methods For Graphs

Formally, the graph classification problem is to correctly assign a label from the set $\mathcal{Y}$ of possible labels (in the case of binary classification $\mathcal{Y} = \{-1,1\}$) to graphs from the ordered set of n graphs $\mathcal{G} = \{G_1, G_2,...,G_n\}$ with a corresponding ordered set $\mathcal{L} = \{y_1, y_2,...,y_n\}$ given features from the graphs in $\mathcal{G}$. In this chapter we will summarize previous research addressing this problem. Much of the previous research that most closely resembles ours is based on kernel methods [57], [58], [62], [22], [4], we therefore devote this section to a very brief summary of kernel methods in general and some of the work most related to this thesis.

The main motivation for the use of kernel methods is nicely explained in Hofman et al. (2008) [32]: the theory and algorithms for problems with linear dependencies are well developed. However, many real world datasets exhibit non linear dependencies. Kernel methods solve this by computing the dot product in a (usually high-dimensional) feature space, and thus implicitly embedding the data in a high dimensional space where linear estimation methods can perform well. And as long as the kernel can be computed directly, there is no need to explicitly compute the feature representation, this is called the *kernel trick*.

To describe kernel methods formally we borrow definitions and notation from [32]. Let $\mathbf{x},\mathbf{x}' \in \mathcal{X}$. A kernel function is then a real valued function $k(\mathbf{x},\mathbf{x}') = \langle \boldsymbol{\phi}(\mathbf{x}),\ \boldsymbol{\phi}(\mathbf{x}')\rangle \in \mathbb{R}$, where $\boldsymbol{\phi}$ maps into the feature space. Typically $k(\mathbf{x},\mathbf{x}') = k(\mathbf{x}',\mathbf{x})$ and $k(\mathbf{x},\mathbf{x}') \geq 0$, which means it can be interpreted as a similarity measure. Given a kernel, k and inputs $x_1,x_2,...,x_n \in \mathcal{X}$, we define the Gram matrix as the $n \times n$ matrix such that $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$. A real $n \times n$ matrix K_{ij} satisfying $\sum_{i,j} c_i c_j K_{ij} \geq 0$ for all $c_i \in \mathbb{R}$ is called positive definite. If equality occurs only for $c_i = 0\ \forall i$ we say the matrix is strictly positive definite. A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ which for all $n \in \mathbb{N},\ x_i \in \mathcal{X},\ i \in [n]$ gives rise to a positive definite Gram matrix is called a positive definite kernel, or a *Mercer* kernel. A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ which for all $n \in \mathbb{N}$, and distinct $x_i \in \mathcal{X}$ gives rise to a strictly positive definite Gram matrix is called a strictly positive definite kernel. If a kernel is a Mercer kernel, it is possible to employ the previously mentioned kernel trick and replace all computations of the inner product feature representation with a call to the kernel function $k(\mathbf{x},\mathbf{x}')$. This lets the learning algorithm operate in an implicit higher dimensional space without adding significant cost to memory and runtime. We here present some kernels applied to graph classification that relate to our work, before ending with a section on the shortcomings of kernel methods versus deep architectures.

4.1.1 Random Walk Kernels

The idea behind random walk graph kernels [22], [4], is to measure the similarity of two graphs G and G' by counting their number of *matching walks*. A walk is defined as a sequence of vertices where repetition of the vertices are allowed and two walks *match* if they are the same in two graphs. If two graphs have many matching walks, then they are similar. The walks of length k in G can be computed by taking the adjacency matrix of G; A to the power of k. We then get that $A^k_{ij} = c$ means that there exists c walks of length k between v_i and v_j. In order to get the number of matching walks, the authors utilize the product graph $G_x = (V_x, E_x)$ defined as:

$$V_x = \left\{ \left(v_i, v_r^{'}\right) : v_i \in V, v_r^{'} \in V' \right\}$$

$$E_x = \left\{ \left(\left(v_i,v_r^{'}\right), \left(v_j,v_s^{'}\right) \right) : (v_i, v_j) \in E \wedge \left(v_r^{'},v_s^{'}\right) \in E^{'} \right\}$$

The product graph consists of pairs of identically labeled vertices and edges from G and G', so an edge is only present in G_x if and only if the corresponding nodes are adjacent in both original graphs. The adjacency matrix of G_x can be computed by taking the *Kronecker product* of the adjacency matrices of the original graphs:

$$A_x = A \otimes A'$$

and since performing a walk on the product graph corresponds to performing simultaneous walks on the G and G', we can now compute the common walks from A^k_x. The random walk kernel can then be computed as (though there are many various implementations based on the same principle):

$$k(G,G') = \frac{1}{|G||G'|} \sum_k \lambda^k \mathbf{e}^T A_x^k \mathbf{e} = \frac{1}{|G||G'|} \mathbf{e}(I - \lambda A_x)^{-1} \mathbf{e}$$

where λ is a decay factor $0 \leq \lambda \leq 1$ that ensures the sum converges, $\mathbf{e}$ is a vector with all elements set to 1 and I is the identity matrix.

Random walk kernels in general suffer from two problems. The adjacency matrix of the direct product graph matrix A_x is of size $|V| \times |V'|$, and even if it is sparse, powers of the matrix can be dense. For large graphs this can lead to significant runtime and memory requirements. The other issue is that iteratively visiting the same set of nodes for small substructures can artificially boost the similarity measure and limit the overall expressiveness of the kernel [4].

4.1.2 Shortest Path Kernels

A different approach is to measure similarity by computing paths (walks where $v_i \neq v_j$ iff $i \neq j$ $\forall i,j \in \{1,....,k\}$). Computing all paths of a graph is NP-hard, as is finding the longest paths. Finding the shortest path however, can be done in polynomial time. The idea behind the shortest-path graph kernel [4] is to transform the original graphs G and G' into *shortest-paths* graphs S and S'. S contains all the same nodes as G, and there is an edge between all nodes in S that are connected by a walk in G. Every edge between v_i and v_j in S is labeled with the shortest path between them. The shortest-path kernel is then defined as:

$$k(S,S') = \sum_{e \in E} \sum_{e' \in E'} k^{(1)}_{walk}(e,e')$$

where $k^{(1)}_{walk}(e,e') = 1$ if the label of e equals the label of e', and zero otherwise.

4.1.3 Graphlet Kernels

The previous research that most closely relates to the research presented in this thesis is the work on graphlet kernels by Shervashidze et. al [57]. They define *graphlets* to be non-isomorphic sub-graphs of size $k \in \{3,4,5\}$ Then they define a graphlet kernel as follows:

Defintion: Graph Kernels *Given two graphs G and G' of size $n \geq k$, the graphlet kernel is defined as*

$$k_g(G,G') = f_G^T f_{G'} \tag{4.1}$$

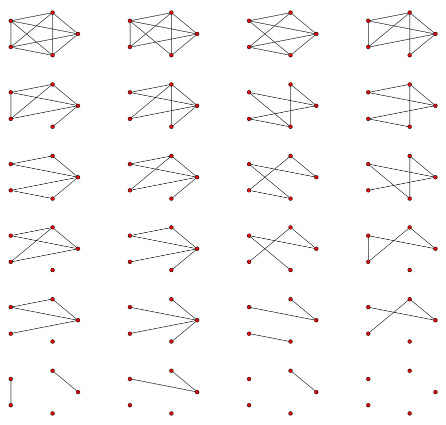

Figure 4.1: All size-5 graphlets

where f_G is a vector of length N_k whose $i-th$ component corresponds to the number of times graphlet(i) occurred in G

In order to eliminate the effect of the size of the graph on the kernel, the counts are normalized:

$$D_g = \frac{1}{\#all\ graphlets\ in\ G} f_G \tag{4.2}$$

resulting in the normalized graphlet kernel:

$$k_g(G,G') = D_G^T D_{G'} \tag{4.3}$$

As counting all graphlets of size k on a graph with n vertices has $O(n^k)$ complexity, they introduce the following sampling theorem in order to ease the computational burden of creating the kernel:

Theorem 6 (from [57]) *Let D be a probability distribution on the finite set $A = \{1,...,a\}$. Let $X = \{X_j\}_{j=1}^m$ with $X_j \sim D$ for a given $\epsilon > 0$ and $\delta > 0$*

$$m = \left\lceil \frac{2(\log 2 \cdot a + \log \frac{1}{\delta})}{\epsilon^2} \right\rceil \tag{4.4}$$

This theorem is then applied via setting A to be the set of all size-k graphlets and assuming that they are distributed according to an unknown distribution D. By letting m be the number of graphlets randomly sampled from the graph, theorem 6 guarantees that the empirical distribution obtained $\hat{D}^m$ is at most ϵ distance away from the true distribution D with confidence $1 - \delta$.

4.1.4 The Structurally Smoothed Graphlet Kernel

Graph kernel methods such as [57], [58], [22], [4] suffer from what is called the *diagonal dominance problem.* As the number of features grow, only a few of these features will be common across graphs. As a consequence, many graphs in the dataset are similar to themselves but not to other graphs, resulting in a diagonal dominance in the kernel matrix. The structurally smoothed graphlet kernel attempt to remedy this by smoothing the graphlet frequency vector in a way that respects the underlying topological relationship between graphlets of different sizes. This is achieved via the Kneser-Ney [37] smoothing method, a method originally invented for natural language processing. Kneser-Ney smoothing computes the probability of an event by discounting raw counts by a fixed mass and then re-distributing this mass according to a base distribution

$$P_{KN}(e_i) = \frac{\max(c_i - d,0)}{\sum_j c_j} + \sum_{j=1}^{n} |\{e_j : c_j > d\}| \frac{d}{\sum_j c_j} P_0(e_i) \tag{4.5}$$

where c_i is the number of times event e_i occurs in the data, $d \geq 0$ is the discounting parameter, $P_0(e_i)$ is the probability of e_i under the chosen base distribution and $\sum_{j=1}^{n} |\{e_j : c_j > d\}|$ is a normalization factor denoting the number of events the discount is applied to. In order to obtain a base distribution with desirable properties for graphlets, Yanardag and Vishwanathan build a directed acyclic graph (DAG) where the empty nodes are graphlets and edges represent if they are one edit-distance away. The DAG is organized in $k + 1$ levels. A level l contains all the graphlets of size l. The parent nodes of a graphlet g_j on level l are graphlets on level $l - 1$ that can be created by deleting a node (and the attached edges) from g_j. Correspondingly, the child graphlets of g_j are graphlets on level $l + 1$ that can be created by adding a node and one or more edges to g_j. The edge weight between g_j and one of it's parents g_i is defined as

$$w_{i,j} = \frac{s_{i,j}}{\sum_{g_{j'} \in \mathcal{C}_{g_i}} s_{i,j'}} \tag{4.6}$$

where $s_{i,j}$ denotes how many times g_i occurs as a sub-graph of g_j and $\mathcal{C}_{g_i}$ denote all the children of graphlet g_i. This can then be used alongside the DAG to define a base distribution over the graphlets. The probability of the root and the subsequent graphlet is 1. The probability of a graphlet g_j in the following layers is defined as

$$P_0(g_j) = \sum_{g_i \in Pa(g_j)} w_{i,j} P_0 g_i \tag{4.7}$$

where g_i denotes a graphlet in the preceding layer and $Pa(g_j)$ is the set of parent graphlets of g_j. With this, the authors define graphlet probability in their Structural Kneser-Ney (SKN) framework as

$$P_{SKN}(g_j) = \frac{\max(c_j - d,0)}{\sum_{g_{j'} \in \mathcal{G}_{k+1}} c_{j'}} + \frac{d}{\sum_{g_{j'} \in \mathcal{G}_{k+1}} c_{j'}} \sum_{g_{j'} \in \mathcal{G}_{k+1}} |\{g_{j'} : c_{j'} > d\}| \sum_{i \in Pa(g_j)} P_0(g_i) \frac{w_{i,j}}{\sum_{g_{j'} \in \mathcal{C}_{g_i}} w_{i,j'}} \tag{4.8}$$

We see that the raw count of all the graphlets are discounted by d, the total mass is then redistributed to all the graphlets according to the base distribution. The discount parameter is tuned by using a Pitman-Yor process [61].

4.1.5 Deep Graph Kernels

Yanardag, Vishwanathan also have another method aimed at mitigating the effect of diagonal dominance [62]. The method is based on building a context encoding matrix M that learns co-occurence relationships between the underlying features. Formally: Instead of the canonical graph kernel

$$k(G,G') = \phi(G)^T \phi(G')$$

they propose a kernel

$$k(G,G') = \phi(G)^T M \phi(G')$$

where M is the $|V| \times |V|$ matrix encoding the relationships between features and V represents the vocabulary of features. They propose two methods for constructing M. The first is based on using the fact that some features such as graphlets have an inherent *edit-distance* relationship which means that one can obtain size $k+1$ graphlets by adding nodes or edges from size k graphlets and vice-versa. This edit-distance can then be used to encode similarity between features into M. The second method is based on using recently introduced neural language models [46][47] (discussed in detail in section 4.2.1.1) to learn latent representations $\boldsymbol{\phi} \in \mathbb{R}^{|V|}$ of the features. These latent feature representations are then used to construct a diagonal matrix M by computing M_{ii} as $\boldsymbol{\phi}_i^T \boldsymbol{\phi}_i$ and $M_{ij} = 0$ for $i \neq j$.

4.1.6 The Shortcomings of Kernel Methods

Deep learning researchers have criticised kernel methods for being fundamentally un-scalable to larger and more complex machine learning tasks [3] (such as AI), primarily stemming from kernel methods in general being shallow two-layer architectures. The authors break down the limitations to four points.

- The first point is based on a depth-breadth tradeoff in circuit design [26], where many computations can more cheaply be computed by deeper architectures than by shallow ones. This suggests that deeper learning architectures can more cheaply represent complex functions.
- The second point is that kernels can be inefficient at representing functions that have many variations, as is often the case with high dimensional real world data. Many kernels imply a local smooth function around each training example, which means one needs a large amount of data in order to appropriately cover the function with locally smooth pieces.
- Most learning algorithms for kernels are quadratic in the number of training data or worse. In general, large scale datasets require linear- or sub-linear-time learning algorithms.
- The most important point stems from the problem of shallowness (first point) and locality (second point). The authors argue that the combination of locality and shallowness make kernel machines too inefficient in the number of inputs and trainable parameters, to learn more complex tasks such as the ones being currently tackled by deep architectures.

The authors note however that kernel machines do have a place in AI, as kernel machines that take in kernels trained by deep architectures (as opposed to pre-specified by humans) are quite powerful.

4.2 Neural Network based approaches

This section is presents some recent contributions to graph classification via neural networks.

4.2.1 DeepWalk

DeepWalk gets its inspiration from a class of models from natural language modeling, word2vec. As it is such a critical part of DeepWalk, we will give a presentation of word2vec here, with notation and explanation based on [54]:

4.2.1.1 word2vec

In 2013, a new and very successful type of language model, commonly known as word2vec, was introduced [46], [47]. These models embed one-hot encoded words into a lower-dimensional continuous vector-space based on their context (where context is defined as the C preceding and succeeding words). Though often referred to as belonging to the class of deep Learning algorithms, word2vec models are in fact shallow two-layer networks. To further increase confusion they are often referred to as if they were one monolithic entity, whereas they actually encompass a small space of algorithms with much variation in composition and performance.

The key to the success of word2vec is its capability to build word representations that encode grammatical and semantic information. In many language processing systems, words are treated as atomic units; one-hot encoded vectors of length V, where V is the size of the vocabulary. This yields two problems if one wants to use text as input to machine learning systems: sparsity, and lack of *meaning.* Sparsity means a machine learning algorithm needs significantly more data in order to perform. The second question is more serious. Atomic representations provide no useful information about relationships between words. Knowing the atomic representation for cat, gives you no indication of what the atomic representation of kitty is. Word2vec provides a way to get representations that are dense, and where the set of word representations encode a structure in the vector space that respects meaningful relationships between words. This allows a model to leverage information about entities it knows a lot about when processing data about entities it knows little about. In a 2013 paper *Linguistic Regularities in Continuous Space Word Representations* [48], it was shown, among other things, that

$$\text{vec(King) - vec(Man) + vec(Woman)}$$

produces a vector very close to Vec(Queen). This means that the representations are capable of encoding analogies. We now give a short summary of the two main models for generating representations in the word2vec framework, Continuous Bag-of-Word (CBOW) and Skip-Gram.

CBOW

Continuous Bag-of-Words takes as input a set of C context words $w_{i-\frac{C}{2}},...,w_{i-1},w_{i+1},w_{i+\frac{C}{2}}$ from a vocabulary of size V, and outputs a word w_i. As such it can be viewed as predicting a word given its context. The model consists of an input layer, a hidden layer, and an output layer, with corresponding weight matrices $W \in \mathbb{R}^{V \times N}$ for the input-to-hidden connections and $W' \in R^{N \times V}$ for the hidden-to-output connections, producing the parameter set $\theta = (W,W')$. The input is presented to the network as one-hot encoded vectors $\mathbf{x}_i$. The hidden layer is calculated as

$$\mathbf{h} = \frac{1}{C} W \cdot (\mathbf{x}_{i-\frac{C}{2}} + ,.., + \mathbf{x}_{i-1} + \mathbf{x}_{i+1} + ,..., + \mathbf{x}_{i+\frac{C}{2}}) \tag{4.9}$$

Which, since the vectors $\mathbf{x}$ are one-hot encoded, amounts to copying the activated row from the weight matrix W. The rows of W correspond to the N-dimensional vector representation of the word (referred to as the *input word*, w_I), these are then averaged

$$\mathbf{h} = \frac{1}{C} \cdot (\mathbf{v}_{w_{i-\frac{C}{2}}} + ,.., + \mathbf{v}_{w_{i-1}} + \mathbf{v}_{w_{i+1}} + ,..., + \mathbf{v}_{w_{i+\frac{C}{2}}}) \tag{4.10}$$

The hidden matrix is then multiplied with the hidden-to-output matrix giving a score u_j for each word in the vocabulary.

$$u_j = \mathbf{v}'^T_{w_j}\mathbf{h} \tag{4.11}$$

In order to obtain the posterior distribution of the words, a softmax function is used with the scores as input

$$p(w_j|w_I) = y_j = \frac{\exp(u_j)}{\sum_{j'=1}^{V}\exp(u_{j'})} \tag{4.12}$$

Maximizing the log of this give us the cost function

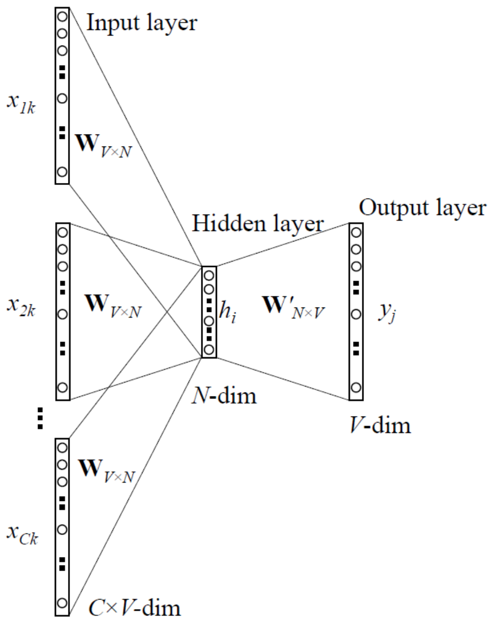

Figure 4.2: Continuous bag-of-words model. Image from [54]

$$J(\theta) = -\mathbf{v}'^T_{w_O}\cdot\mathbf{h} + \log\sum_{j'=1}^{V}\exp(\mathbf{v}'^T_{w'_j}\cdot\mathbf{h}) \tag{4.13}$$

Skip-Gram
Skip-gram is in a way the mirror image of CBOW as it takes a word as input and maximizes the probability of the context. This means that instead of sharing a input-to-hidden matrix W across input words, as in CBOW, a hidden-to-output matrix W', is shared across C output vectors. The input is a one-hot encoded vector which is multiplied by W, yielding the hidden layer h

$$h = W_{(k,\cdot)} = v_{w_I} \tag{4.14}$$

This is then the input to the C output layers

$$u_{c,j} = u_j = \mathbf{v}'^T_{w_j} \cdot \mathbf{h}, c = 1,2,...,C \tag{4.15}$$

Finally, the posterior distribution is computed for all C words in the context using the softmax function.

$$p(w_{c,j} = w_{O,c}|w_I) = y_{c,j} = \frac{\exp u_{c,j}}{\sum_{j'=1}^{V} \exp u_{j'}} \tag{4.16}$$

The cost function then becomes

$$\begin{aligned} J &= -\log p(w_{O,1},w_{O,2},...,w_{O,C}|w_I) \\ &= -\log \prod_{c=1}^{C} \frac{\exp u_{j*_c}}{\sum_{j'=1}^{V} \exp u_{j'}} \\ &= -\sum_{c}^{C} = 1u_{j*_c} + C \cdot \log \sum_{j'=1}^{V} \exp u_{j'} \end{aligned} \tag{4.17}$$

where $j*$ is the index of the True label.

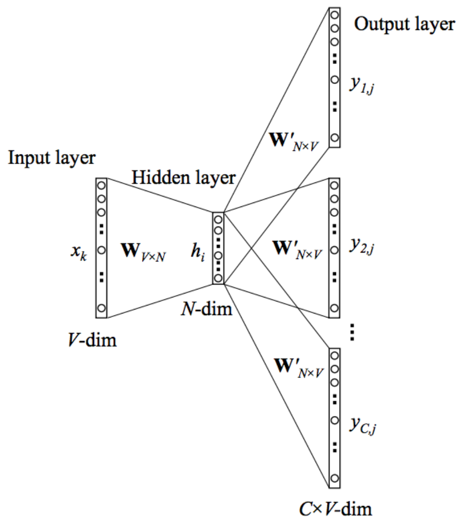

Figure 4.3: Skip-gram model. Image from [54]

Hierarchical Softmax Computing C Softmax normalizations for large vocabularies are quite expensive. There has therefore been a focus on developing efficient approximations to full softmax, such as hierarchical softmax and negative sampling [46]. Hierarchical softmax builds a binary Huffman tree based on the frequencies of the words in the vocabulary. Then, the posterior probability of a given word is a computed as a chain of multiplications from the root of the tree to the leaf node containing the word.

$$p(w|w_I) = \prod_{j=1}^{L(w)-1} \sigma\left([n(w,j+1) = ch(n(w,j))] \cdot v'^{T}_{n(w,j)} v_{w_I}\right) \tag{4.18}$$

To quote [46]: "In probabilistic terms, one N-way normalization is replaced by a sequence of O(logN) local (binary) normalizations."

4.2.1.2 DeepWalk

Building on this framework, Perozzi, Al-Rfou and Skiena present DeepWalk [52], a method for learning latent node representations in networks that can be used for multi label classification. They adopt the word2vec framework by treating vertices as words, and random walks on the network as sentences that provide the context. DeepWalk uses a set of short truncated random walks as the corpus and the graph vertices as vocabulary. It uses SkipGram to build the representation and hierarchical softmax to approximate the softmax function of the final layer. The model is trained by stochastic gradient decent, using backpropagation to compute the gradient, much like in a conventional neural network. The representations are tested as input to a one-vs-all logistic regression on the multi-label classification task. Perozzi et al. show that the representations generated by DeepWalk produce F1 scores up to 10% higher than competing methods for datasets where label data is sparse. It also in generally outperforms baseline methods while using significantly less data.

Algorithm 6: DeepWalk

```
Input: graph G(V,E), window size $w$, embedding size $d$, walks per
       vertex $\gamma$, walk length $t$
Sample $\Phi$ from $\mathcal{U}^{|V|\times d}$
Build a binary tree $T$ from $V$
for $i = 0,1,...,\gamma$ do
    $\mathcal{O}$ = Shuffle($V$)
    foreach $v_i \in \mathcal{O}$ do
        $\mathcal{W}_{v_i}$ =RandomWalk($G$,$v_i$,$t$)
        SkipGram($\Phi$, $\mathcal{W}_{v_i}$, $w$)
    end
end
return Vertex representation matrix $\Phi \in \mathbb{R}^{|V|\times d}$
```

4.2.2 Spectral Networks and Deep Locally Connected Networks

A long way into this thesis, we were made aware of two papers showing very promising approaches to learning deep representations of graphs: *Spectral Networks and Deep Locally Connected Networks on Graphs*[6] and *Deep*

Convolutional Networks on Graph-Structured Data[28]. These papers present two novel approaches to generalizing Convolution Neural Networks to graphs. As the authors explain[28], deep learning models have shown themselves to be extremely successful in the fields of speech recognition, image classification, object recognition, translation etc. because they are able to exploit stationarity and compositionality through local features. Stationarity is important because it is what gives rise to the translation invariance of features, which implies the possibility for using weight sharing. This is what CNNs exploit when a trained feature detector is convolved over the input signal (weight sharing in space). It is also what RNNs exploit when they recognize the same phoneme at different times in a speech signal (weight sharing in time). Compositionality is important because it is what allows for the use of low level features to build successively more complex representations of the input signal in the higher layers of the network. The authors note that one can think of these signals as being defined on a low dimensional grid, where stationarity is defined via the translation operator on the grid, locality is defined via the metric of the grid and compositionality is obtained from down-sampling[28]. It is thus possible to think of these signals as special cases of data defined on a low-dimensional graph. General graph data may have much higher dimensionality and not respect the properties of stationarity and compositionality in the same way as data defined on grids.

This section describes the methods introduced in [6] and [28]. Two methods are proposed, in order to generalize CNNs to graph-structured data. [6] introduces *Spectral Networks* and *Deep Locally Connected Networks* and [28] extends this to higher dimensional cases, and cases where the graph structure is latent in the data and must be inferred.

4.2.2.1 Deep Locally Connected Networks

The first architecture proposed is the Deep Locally Connected Network. Define a graph $G = (\Omega, W)$, where Ω is the set of the graph's m vertices and W is a $m \times m$ symmetric and non-negative matrix. Deep Locally Connected Networks performs a *spatial construction* by performing a multi-scale hierarchical clustering on G, and defining layers of local receptive fields on this clustering. Let K be the number of scales in the clustering and $\Omega_0 = \Omega$. For each k = 1,2,...,K, define Ω_k as a partition of Ω_{k-1} consisting of d_k clusters. Let $\mathcal{N}_k = \{\mathcal{N}_{k,i}\ i1,2,...,d_{k-1}\}$ be the collection of neighborhoods around each cluster in Ω_{k-1}. Where a neighborhood is defined as

$$N_\delta(j) = \{i \in \Omega \colon W_{i,j} > \delta\}$$

for a given threshold δ.

Let x_1 be a real signal defined in Ω, and f_k be the number of filters created at layer k. The network transforms a f_{k-1} dimensional signal indexed by

Ω_{k-1} into a f_k dimensional signal indexed by Ω_k as follows:

$$x_{k+1,j} = L_k h \left(\sum_{i=1}^{f_{k-1}} F_{k,i,j} x_{k,i} \right) \ , \ (j = 1,..,f_k) \tag{4.19}$$

where $F_{k,i,j}$ is a $d_{k-1} \times d_{k-1}$ sparse matrix with nonzero entries in the locations given by $\mathcal{N}_k$, h is an element-wise activation function and L_k outputs the result of a pooling operation over each cluster in Ω_k.

4.2.2.2 Spectral Networks

The second architecture generalizes the convolution operator via the spectrum of the graph-Laplacian. As with Deep Locally Connected Networks, we view the input signal x as a real signal defined on $G = (\Omega, W)$, where $\Omega = \{v_1,v_2,...,v_m\}$ and W is an $m \times m$ symmetric and non-negative similarity matrix. The graph Laplacian is defined as

$$L = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}} \tag{4.20}$$

where $D = W \cdot \mathbf{1}$. Denote the eigenvectors of L as $U = (u1, u2,...,u_m)$. The graph convolution of x with a filter g on G is then defined as

$$x * g = U^T(Ux \odot Ug) \tag{4.21}$$

where $\odot$ denotes element-wise multiplication and U denotes the diagonal matrix of eigenvalues. Learning filters on a graph is equivalent with learning spectral multipliers $w_g = (w_1, w_2,...,w_m)$

$$x *_G g = U^T(\mathrm{diag}(w_g)Ux) \tag{4.22}$$

Filters in conventional CNNs are defined to have localized spatial support on the input signal. In order to construct filters with this property in the Fourier domain, the authors exploit the fact that in the grid, fast spatial decay corresponds to smoothness in the frequency domain since

$$\left| \frac{\partial^k \hat{x}(\xi)}{\partial \xi^k} \right| \leq C \int |u|^k |x(u)| du \tag{4.23}$$

where $\hat{x}(\xi)$ is the Fourier transform of x. The author in [6] suggest using a smoothing kernel $\mathcal{K}^{N \times N_0}$ to obtain smoothed spectral multipliers

$$w_g = \mathcal{K} \tilde{w}_g \tag{4.24}$$

Pseudocode for training a Spectral Network from [28] is given below:

```
Algorithm 7: Train Graph Convolution Layer
  Input: Graph Fourier Transform matrix $U$, similarity kernel $\mathcal{K}$,
         weights $w$
  Forward Pass:
  Fetch input batch $x$ and gradients w.r.t outputs $\nabla y$
  Compute interpolated weights: $w_{f',f} = \mathcal{K}\tilde{w}_{f',f}$
  Compute output: $y_{s,f'} = U^T(\sum_f Ux_{s,f} \odot w_{f',f})$
  Backward Pass:
  Compute gradient w.r.t. input: $\nabla x_{s,f} = U^T(\sum_f' \nabla y_{s,f'} \odot w_{f',f})$
  Compute gradient w.r.t. interpolated weights:
  $\nabla w_{f',f} = U^T(\sum_f' \nabla y_{s,f'} \odot x_{s,f})$
  Compute gradient w.r.t. weights: $\nabla \tilde{w}_{f',f} = \mathcal{K}^T \nabla w_{f',f}$
  return $\nabla \tilde{w}_{f',f}$
```

Chapter 5

Thesis Contributions

This chapter covers the thesis contributions to the graph classification problem. The primary challenge when doing graph classification via convolutional neural networks or other deep learning architectures is the choice of input representation. The general aspect of this problem is that graphs are very general objects and have several equivalent representations and also lend themselves to a wide array of feature extraction methods such as the kernels in [22], [4], [57]. The more specific aspect of the problem is that the feedforward neural networks investigated here can only handle fixed input dimensions due to the need for a fixed-size weight matrix from the last convolutional layer to the first fully connected- or output layer. A second and perhaps more serious issue is that in CNNs, spatial relationships in the input data representation matter, and this is not always the case for raw graph representations such as adjacency matrices (where the overall graph structure is invariant to shuffling of both rows and columns).

We present two contributions to graph classification, representing two very different ways of attacking the problem of graph representration. The first method is based on learning to distinguish graphlet frequency vectors [57] and is essentially an investigation into deeper networks ability to learn higher order features from graphlet distributions. The second method uses the collection of high-dimensional node representations obtained by DeepWalk [52] to build compressed node representations for graph classification.

5.1 The Data

MUTAG (Debnath et al., 1991) is a data set of 188 mutagenic aromatic and heteroaromatic nitro compounds labeled according to whether or not they have a mutagenic effect on the Gram-negative bacterium Salmonella typhimurium. The dataset is not balanced, but split 0.665/0.335.
PTC (Toivonen et al., 2003) contains 344 chemical compounds tested for carcinogenicity in mice and rats. The classification task is to predict the

carcinogenicity of compounds. The dataset is fairly balanced and is split 0.558/0.442.
NCI1 and NCI109 represent two balanced subsets of data sets of chemical compounds screened for activity against non-small cell lung cancer and ovarian cancer cell lines respectively (Wale and Karypis (2006) and http://pubchem.ncbi.nlm.nih.gov).
D&D is a data set of 1178 protein structures (Dobson and Doig, 2003). Each protein is represented by a graph, in which the nodes are amino acids and two nodes are connected by an edge if they are less than 6 Angstroms apart. The prediction task is to classify the protein structures into enzymes and non-enzymes. The dataset is not balanced, and is split 0.4134/0.5866.
REDDIT-MULTI-5K is a dataset of 5000 graph representations of threads on reddit.com. Nodes corresponds to users and there is an edge between them two nodes if either of the users have commented on at least one of each others posts. The classification task is to identify which subreddit a given graph/discussion belongs to. The data is equally split among five classes.
COLLAB is a dataset of 5000 ego-networks obtained by compiling 3 public collaboration datasets; high-energy physics, condensed matter physics and astro physics. The classification task is to identify which field the ego-network of a given researcher belongs to. The dataset is not balanced.
IMDB-MULTI is a dataset of 1500 ego networks of actors and actresses. The nodes are actors and actresses and there is an edge between two nodes if two actors appear in the same movie. The classification task is to determine if the collaboration graph belongs to a comedy, romance or sci-fi movie. The dataset is balanced.
IMDB-BINARY is equivalent IMDB-COMEDY-ROMANCE-SCIFI to except that there are 1000 networks and the categories are action and romance movies. The dataset is balanced.
Due to time constraints we were unable to evaluate our first method on REDDIT-MULTI-5K, COLLAB, IMDB-MULTI and IMDB-BINARY.

5.2 Hardware & Software

All experiments were run on a desktop with an Intel $i7-6700K$ CPU, 32GB Ram and a GTX Titan X GPU. All models were built using either Tensorflow or Keras.

5.3 Neural Networks on Graphlet Frequency Distribution Vectors

Our first approach attempt to classify graphs by learning to distinguish their graphlet frequency distribution vectors via neural networks. This method relies extensively on the work of Shervashidze et al. [57], in particular the sampling theorem and the use of graphlets as underlying features. The hypothesis is that as the graphlet kernel in eq. 4.1 is computed, some of the information in the component graphlet frequency vectors is lost. We therefore instead feed the frequency vectors directly to the various neural network architectures. In addition to the standard graphlet frequency vector, we also test on the smoothed frequency vector from [61], using the same hyperparameter as them. We deploy four different architectures. Logistic/softmax regression, a multilayer perceptron with a single hidden layer, a CNN with two convolutional layers and a CNN four convolutional layers. The motivation for using several architectures is to observe if the deeper architectures are able to extract higher level features and thus converge faster to a good accuracy, as well as seeing if the convolution operator brings any benefits.

5.3.1 Input data

For each graph in our dataset $\mathcal{G} = \{G_1, G_2, ..., G_n\}$ we compute the graphlet frequency distribution vector defined as

$$D_g = \frac{1}{m} f_G \tag{5.1}$$

where m is defined as in eq. 4.4 and f_G is defined as in eq. 4.1. For graphlets of size $k = 5$, $D_g \in \mathbb{R}_{\cancel{k}}{}^{34}$ etc. This vector essentially serves as an estimate of the probability mass function of the distribution of graphlets in the graph. Figure 5.1 shows a graph model of a protein and its corresponding graphlet distribution. The respective locations of the different graphlet counts in eq. 5.1 and eq. 5.1, are somewhat arbitrary but in general more connected graphlets are located farther out in the vector. In [61] a topological sorting of the graphlets according to their edit distance is presented, however all that is required is that the ordering is consistent across graphs.

5.3.2 Network Architectures

Four separate architectures were built and tested: a one layer logistic regression, a multilayer perceptron with one hidden layer and two convolutional nets: one with two convolutional layers and one with four convolutional layers.

Logistic Regression The softmax has 34 input units (number of size $k = 5$ graphlets), which feeds to a sigmoid function.

Multilayer Perceptron The MLP has the following architecture

- **Input layer** 34 input units.

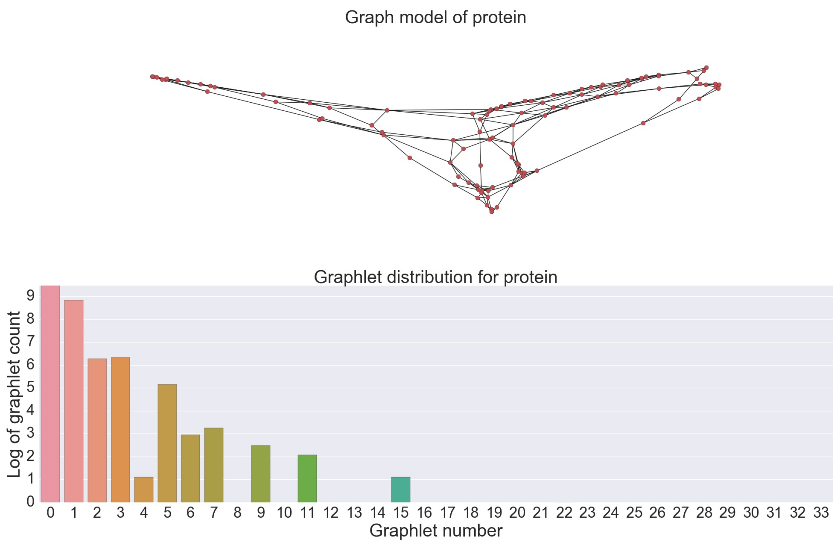

Figure 5.1: Graph model of protein and its corresponding graphlet distribution.

- **Hidden layer** 40 hidden units with rectified linear activation units (relu).
- **Output layer** Sigmoid output layer

CNN with four convolutional layers The deepest of the two networks has the following architecture.

- **1st Convolutional Layer** 64 filters of size 34×1 (same as input), followed by a Relu layer.
- **2nd Convolutional Layer** 64 filters of size 17×1, followed by a Relu layer and a Max pooling layer with kernel size: 2×1
- **3rd Convolutional Layer** 32 filters of size 9×1, followed by a Relu layer and a Max pooling layer with kernel size: 2×1
- **4th Convolutional Layer** 16 filters of size 5×1, followed by a Relu layer and a Max pooling layer with kernel size: 2×1
- **1st Fully Connected Layer** Fully connected layer with 128 hidden units and Relu activation functions.
- **2nd Fully Connected Layer** Fully connected layer with 64 hidden units and Relu activation functions.
- **Output Layer** Sigmoid ouput layer.

CNN with four convolutional layers The shallow of the two networks has the following architecture.

- **1st Convolutional Layer** 32 filters of size 17×1, followed by a Relu layer and a Max pooling layer with kernel size: 2×1.
- **2nd Convolutional Layer** 16 filters of size 9×1, followed by a Relu layer and a Max pooling layer with kernel size: 2×1
- **1st Fully Connected Layer** Fully connected layer with 64 hidden units and Relu activation functions.
- **Output Layer** Sigmoid output layer.

5.3.3 Training and Validation

The performance of each network is validated using 5-fold cross validation. The networks are trained with cross entropy as the cost function and AdaGrad as the optimization algorithm with a learning rate of 0.01. For each fold the network is trained for 1000 epochs with a batch size of $n/4$ where n is the number of examples in the fold. In order to avoid overfitting several regularization methods were deployed simultaneously. L2-regularization was performed on the weights, early stopping as described in 3.3.3.4 and dropout of 50% was performed between each layer.

5.3.4 Hyperparameter tuning and network design

Hyperparameters where tuned on the NCI1 dataset, as this is one of the larger datasets. Once satisfactory parameter values where found, they where kept constant during training and validation on the other datasets.

5.4 Graph Classification via Compressed Latent Node Representations

As mentioned in the introduction, a major goal in neural network research is to be able to build end-to-end neural architectures without relying on specialized features such as the graphlet frequency vectors of the previous method. In this section we present an advancement in this direction for graph classification, by applying collections of the latent node representations generated by DeepWalk [52], which is described in detail in section 4.2.1.2. DeepWalk uses Skip-Gram to build latent node representations in a continuous vector space by performing a series of random walk on the graph and maximizing the probability of a given walk given a node in the walk. Our method then uses the collection of these node representations as input to a highly regularized and relatively shallow neural network. The current implementation is a two-step process. For each graph in our dataset $\mathcal{G} = \{G_1, G_2,...,G_n\}$ we use DeepWalk to compute Φ, the matrix of vector node representations. This matrix is then passed to the neural network for classification. For graphs with more than 150 vertices, we sampled node representations of 150 vertices without replacement instead of building the full Φ.

5.4.1 DeepWalk

As input to DeepWalk we used the edgelist of the graphs. We trained the model using standard settings from the paper. At each vertex in the graph, 10 random walks of length 40 are started. The window-size for Skip-Gram was set to 5, and the dimensionality of the node representations is 64.

5.4.2 The Neural Network Architecture

The node vector representation matrix Φ is of size $|V| \times N$, where $N = 64$ in our case. In order to deal with varying input size we compute the size of the largest graph in the dataset $K = \max_g \mathcal{G}$, and insert Φ in a $K \times N$ empty matrix Φ^*. The first layer of the network consists of K parallel MaxOut layers that each take a row vector $\mathbf{x} \in \mathbb{R}^{64}$ from Φ^* and output a compressed node representation $h(\mathbf{x}) \in \mathbb{R}$ where h represents the MaxOut transformation. These compressed node representations are then merged into a single hidden maxout layer of width K. This layer is then followed by another maxout layer of size K, before a final output layer with a sigmoid (softmax in the case of multinomial classification) activation function.

5.4.3 Training and Validation

The performance of each network is validated using 5-fold cross validation. The networks are trained with cross entropy as the cost function, RMSProp as the optimization algorithm with a learning rate constant at 0.001, $\rho = 0.9$ and $\epsilon = 1\text{e} - 06$ as the optimizer. For each fold the network is trained for 2000 epochs with a batch size of $n/4$ where n is the number of examples in the fold. In order to avoid overfitting several regularization methods were deployed simultaneously. L2-regularization was performed on the weights, early stopping as described in 3.3.3.4 and batch normalization and dropout with $p = 0.5$ is performed before the second hidden layer and before the output layer.

5.4.4 Hyperparameter tuning and network design

Neural networks often need extensive hyperparameter tuning before they perform satisfactory. The inherent danger in this is that it can introduce severe data snooping bias if the tuning is done using to much of the data.

We used half of the $NCI1$ dataset for tuning, as this is one of the larger datasets we had access to. The decision to split the first layer into K layers learning compressed node representations was made in order to reduce network capacity and thus prevent overfitting. For datasets containing graphs with more than 150 vertices, fully connecting the input to the first hidden layer would translate to $150 \times 150 \times 64 + 150 = 1\ 440\ 150$ trainable parameters. This is excessive for datasets containing between 188 and 4000 graphs. By constraining the units of the first hidden layer to each only see one node representation, this number was cut to $150 \times 64 + 150 = 9\ 750$.

Early attempts using softmax and vanilla multilayer perceptrons failed to yield satisfactory results due to excessive and fast overfitting. This led to the decision to design a highly regularized network. The introduction of batch normalization between hidden layers proved crucial to obtain the results. With batch normalization, each sample is normalized in accordance to the batch it is currently in (batches are shuffled between epochs), this leads to a very light form of data augmentation that gives an illusion of training om more data. Conventional hidden layers were abandoned in favor of MaxOut layers which leverage and enhances the ensemble training effects of dropout.

Chapter 6

Results

6.1 Neural Networks on Graphlet Frequency Distribution Vectors

As stated in the machine learning chapter, the benefit of using deeper architectures is that if the data allows it, they can build higher level representations of the data that improve performance. Usually this means complex non-obvious features that are combinations of transformations of the raw input data. The plots in figure 6.1, 6.2, 6.3, 6.4, and 6.5 shows how validation loss developed with number of epochs for the benchmark datasets. The training losses are averaged over 10 folds and both smoothed and unsmoothed frequency distributions are used, as indicated by the dotted and full lines.

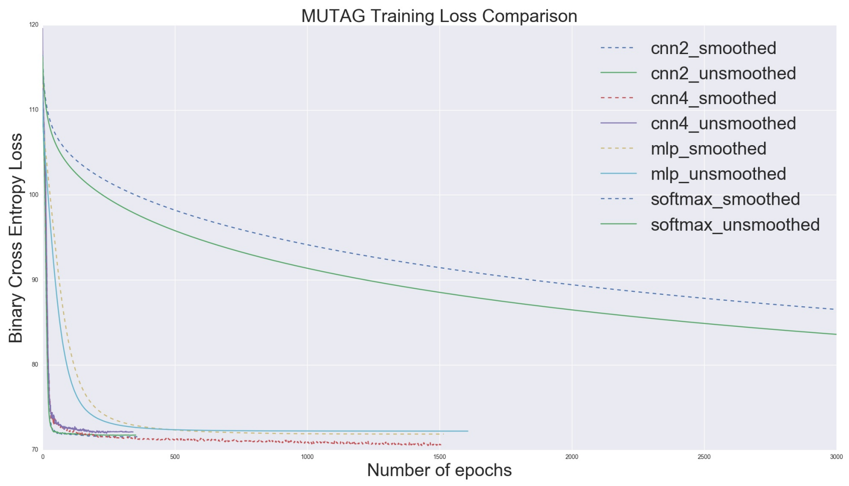

Figure 6.1: Validation loss on the MUTAG dataset.

From the plots we see that there is a qualitative difference in the convergence of the neural networks versus the softmax, with the cnn

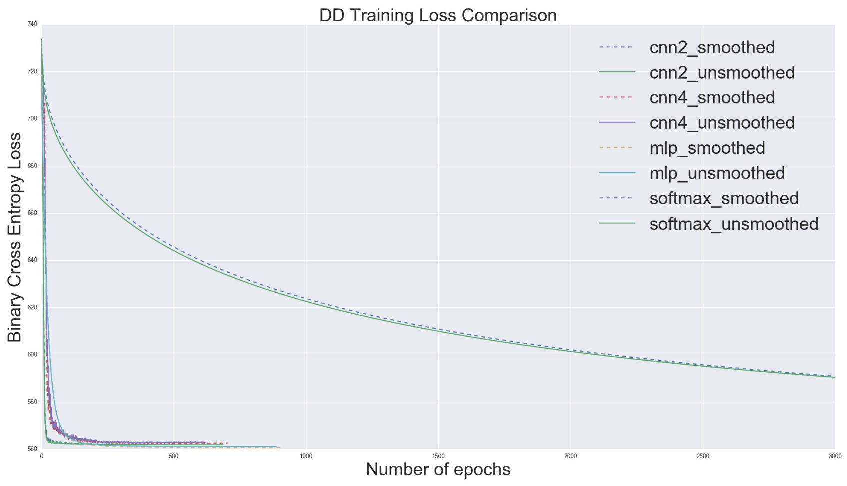

Figure 6.2: Validation loss on the DD dataset.

architectures in general reaching the lowest loss levels. On all the dataset except PTC, the neural network architectures manage a rapid convergence which it then improves on until the early stopping criterion is triggered, whereas the softmax algorithm takes significantly more epochs to reach comparable loss levels. When considering that the final layer of all four architectures is identical (softmax/sigmoid output), we can view all the preceding layers as essentially feature generators for a logistic/softmax regression. In this light, the significant difference in speed of convergence both in epochs and time indicates that learning of non-obvious features is taking place. When looking at table 6.1 and 6.4 however, a more complicated picture appears. We find that all architectures report similar accuracies on all datasets, and all are comparable to the various graphlet kernel methods. We speculate that the explanation for the phenomenon of divergence in speed but convergence in accuracy can in part be because of the deeper architectures' ability to relatively quickly compute salient features. Note that accuracies are comparable to results in [57] and [61]. In addition, our training and validation times are significantly lower than reported in [57]. Though this of course somewhat affected by the development we have seen in hardware.

6.2 Compressed Latent Node Representations

This method performs markedly worse than the graphlet based one. In the chapter on Further Research 7, we present two sections that discuss why this probably is the case, and potential remedies. The first considers using negative sampling from all classes, instead of just the graph the node is in. The second considers embedding entire walks in a vector space, and not just nodes.

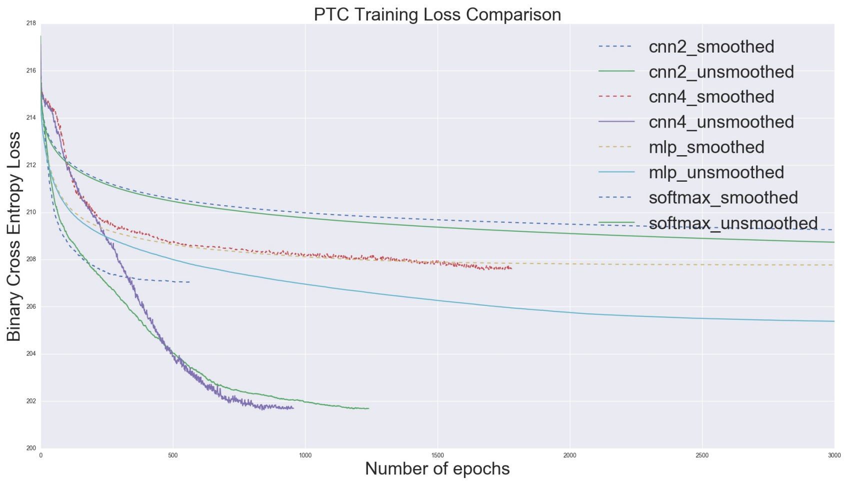

Figure 6.3: Validation loss on the PTC dataset.

Table 6.1: Accuracy for unsmoothed graphlet density

Accuracy Results (Unsmoothed)				
DATASET	Softmax	MLP	CNN2	CNN4
MUTAG	81.92% (+/- 4.30)	80.84% (+/- 2.95)	80.31% (+/- 2.21)	79.27% (+/- 3.26)
PTC	56.98% (+/- 6.4)	57.27% (+/- 1.56)	56.68 (+/- 1.45)%	55.52% (+/- 0.99)
DD	75.13% (+/- 1.63)	75.47% (+/-1.81)	74.96% (+/- 1.73)	75.13% (+/- 1.28)
NCI1	62.63% (+/- 1.15)	62.60% (+/- 1.02)	62.34% (+/- 1.02)	62.55% (+/- 1.28)
NCI109	62.30% (+/- 0.88)	62.59% (+/- 0.84)	61.96% (+/- 0.83)	62.20% (+/- 0.79)

Table 6.2: Time for unsmoothed graphlet density per fold in seconds

Training and validation time (Unsmoothed)				
DATASET	Softmax	MLP	CNN2	CNN4
MUTAG	123.66	10.69	3.39	3.57
PTC	41.63	2.38	11.90	13.06
DD	70.77	10.79	4.27	50.90
NCI1	121.44	40.86	7.62	131.82
NCI109	131.32	131.12	9.70	160.48

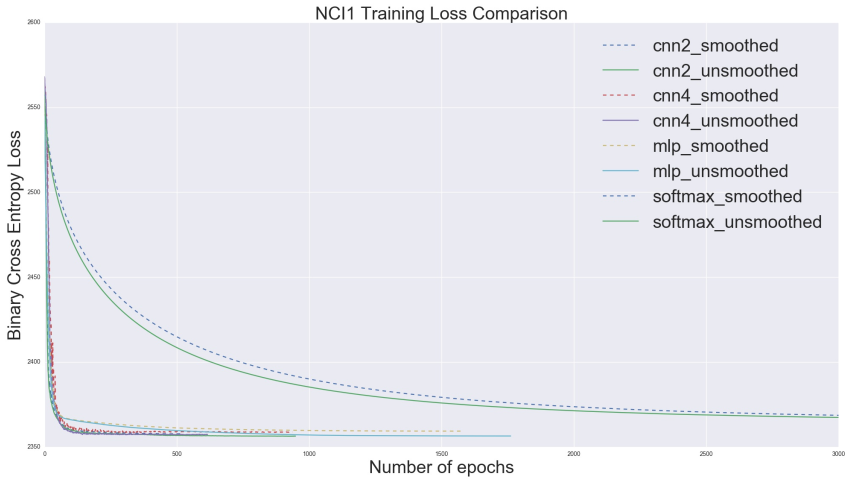

Figure 6.4: Validation loss on the NCI1 datset.

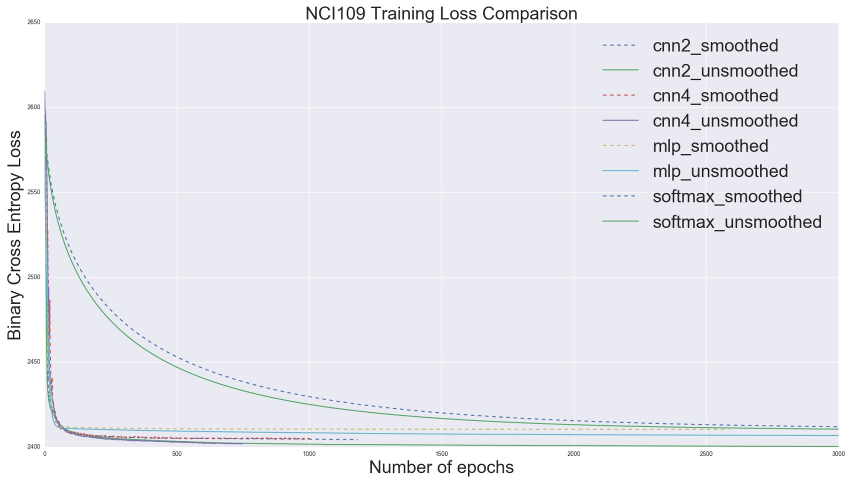

Figure 6.5: Validation loss on the NCI109 dataset.

Table 6.3: Time for smoothed graphlet density per fold

Training and validation time (Smoothed)				
DATASET	Softmax	MLP	CNN2	CNN4
MUTAG	92.23	10.95	2.52	18.48
PTC	68.14	3.08	11.47	41.31
DD	233.16	8.33	16.39	17.63
NCI1	198.63	24.05	15.64	89.35
NCI109	210.19	14.80	32.26	71.50

Table 6.4: Accuracy for smoothed graphlet density

Accuracy Results (Smoothed)				
DATASET	Softmax	MLP	CNN2	CNN4
MUTAG	80.84% (+/- 3.22)	80.84% (+/- 2.95)	80.31% (+/- 2.21)	76.63% (+/- 3.80)
PTC	59.59% (+/- 1.13)	59.60% (+/- 1.02)	57.86% (+/- 1.31)	57.56% (+/- 0.71)
DD	75.22% (+/- 1.66)	75.56% (+/-1.83)	75.13% (+/- 1.58)	75.81% (+/- 1.47)
NCI1	62.60% (+/- 1.21)	62.41% (+/- 1.05)	62.41% (+/- 0.67)	62.26% (+/- 1.08)
NCI109	62.44% (+/- 0.85)	62.39% (+/- 0.86)	61.71% (+/- 0.84)	62.05% (+/- 0.58)

Table 6.5: Accuracy for embedded node representations

Accuracy Results (Smoothed)			
DATASET	Accuracy	Standard Error	Average Time (seconds)
MUTAG	85.12%	0.2%	12.49
PTC	58.15%	1.5%	89.38
DD	71.4%	1.6%	490.17
NCI1	58.27%	1.05%	608.38
NCI109	57.6%	1.12%	785.11
COLLAB	65.2%	0.06%	755.09
IMDB-BINARY	22.8%	0.08%	444.51
IMDB-MULTI	43.73%	0.17%	178.86
REDDIT-MULTI	31.90%	0.09%	784.38

Chapter 7

Discussion and Further Research

The biologist and complex systems scientist Stuart Kauffman has coined a concept he calls the adjacent possible, meaning the state space of configurations made possible by a system's current state. It has been described as a 'shadow future, hovering on the edges of the present state of things, a map of all the ways in which the present can reinvent itself' [35]. The term has been adopted by many fields, such as innovation theory, where it is used to ask what products can built, given what components, supply chains and customer sentiments exist. In the spirit of the adjacent possible, this chapter contain ideas for further research that unfortunately did not emerge until the main ideas in this thesis were fully explored, and thus did not make it to the thesis contributions due to time constraints.

7.1 Multiscale graphlet distributions

One of the main issues with using graphlets as features is the trade-off between modeling the local and global topology of the graph [61], [62]. Large graphlets can capture more global structure, but as the graphlet size increases the number of graphlet increase exponentially, with only a few of them being represented in a given graph. As mentioned, this often leads to a very sparse representation and often, diagonal dominance.

We propose a multi-scale graphlet distribution built by combining the hierarchical clustering from the Deep Locally Connected networks from [6] and conventional graphlets from [57]. The distribution would be built as follows. A graph $G = (\Omega, W)$ is completely defined as the set of nodes Ω and the similarity matrix W (adjacency matrix, if the graph structure is explicit and unweighted). As in [6], consider K scales. Set $\Omega_0 = \Omega$ and for each $k = 1,...,K$, define Ω_k, a partition of Ω_{k-1} into d_k clusters, and a collection of neighborhoods around each element of Ω_{k-1}. For a given graphlet size l, compute the graphlet distribution of Ω_0. For each successive Ω_k, treat each cluster as a node, connected to other nodes by the cluster neighborhood relation. Then compute the graphlet distribution over this graph. This will allow for a graphlet distribution over a successively downsized graph, which as the scale increases should allow for the capture of more and more global features of the graph. Experimental research will need to be carried out in order to analyze what graphlets to use at different graph scales etc.

7.2 Negative sampling from a mixed corpus

DeepWalk is built for generating representations for multi-label classification. It is intuitive that the representations work well for this task, as inference is done on the same graph that is used for building the representations. It is somewhat more surprising that these representations work as well as they do for graph classification. Representations are built using information that is local in the dataset, but used to do inference globally over the whole dataset. What essentially happens when one is using Skip-Gram is that the vector representation of the nodes are 'pushed' towards their correct contexts and 'pulled' away from incorrect contexts. An idea for further research could be to explore the effect of using negative sampling where the negative samples were walks obtained from graphs not belonging to the desired class, or a mix of both. If certain walks are completely absent in a graph, this should manifest itself as a context that the node representations are pushed away from. Given that two classes have a different topology that can be seen in the walks it permits, this should give better node representations.

7.3 Embedding walks instead of nodes

The vector representations of nodes in DeepWalk encode topological data from the graph they belong to, but it might be even more effective to embed entire walks and use them as features for classification. As mentioned, DeepWalk is built for generating representations to do node inference. There is most likely a substantial amount of redundancy in using the collection of all node representations as features for a graph. Redundancy hurts classifier performance, especially for small datasets.

7.4 Recurrent Neural Networks

The novelty of DeepWalk lies in generalizing a model originally designed for language modeling, to graphs. There is another class of neural networks, which we have not discussed in this thesis, that are frequently used for language modeling, namely recurrent neural networks (RNN). For a thorough discussion of RNNs, we refer to Alex Graves' PhD dissertation *Supervised Sequence Labelling with Recurrent Neural Networks*, from where we will borrow some notation. RNNs are neural networks that process sequences of information and where neurons feed their own state information back into themselves, and to other neurons at the next time-step. Formally, consider an RNN with a size N input layer, one hidden layer of size H, and K output units. Denote by W^I and W^H the input-to-hidden and hidden-to-hidden connections respectively. The network takes an input sequence $\mathbf{x}$ of length T. Let x_i^t be the value of input i at time t and a_h^t be the i-th hidden unit at time t. The hidden unit is then defined as

$$a_h^t = \sigma(\sum_{i=1}^{I} w_{i,h} x_i^t + \sum_{h'=1}^{H} w_{h',h} a_{h'}^{t-1}) \tag{7.1}$$

where σ is the nonlinerarity applied elementwise to every neuron in the hiden layer. These so called 'vanilla' RNNs proved to be difficult to train. This led to the development of more advanced architectures such as the Long Short-Term Memory model (LSTM) [31] and the Gated Recurrent Unit (GRU) [9], which are now common in state of the art implementations of sequence processing. Notable contributions are the word prediction work by Zaremba, Sutskever and Vinyals [63] and the sequence-to-sequence modeling in [9]. In [63], the task is to predict the next word given a sentence on the Penn Tree Bank dataset. In [9], the task is to map a sequence of arbitrary length to a different sequence to arbitrary length, the method is applied to statistical machine translation where the input sequence is a sentence in one language and the output sequence is the translated sentence in another language. In both papers, the words that the sentences are composed of are embedded in a continuous vector space à la word2vec in order to increase performance.

There are several possibilities for using RNNs to extend this research further. One possible direction for further research could be suggest to extend the language-to-graph analogy of DeepWalk and pass walks of densely represented nodes to recurrent neural networks such as LSTMs for graph classification. Indeed it could be interesting to see results even without node embedding. Especially for graphs with labeled nodes, the network should be able to infer some measure of graph topology without node embeddings.

Bibliography

[1] Alberto-Lazlo Barabasi and Zoltan Oltvai. “Network biology: understanding the cell’s functional organization”. In: *Nature Reviews Genetics* 5 (2 2004), pp. 101–113.

[2] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. “Deep Learning”. Book in preparation for MIT Press. 2015. URL: http://www.iro.umontreal.ca/~bengioy/dlbook.

[3] Yoshua Bengio et al. *Scaling learning algorithms towards AI*. 2007.

[4] Karsten M. Borgwardt and Hans-peter Kriegel. “Shortest-Path Kernels on Graphs”. In: *In Proceedings of the 2005 International Conference on Data Mining*. 2005, pp. 74–81.

[5] ULRIK BRANDES et al. “What is network science?” In: *Network Science* 1 (01 Apr. 2013), pp. 1–15. ISSN: 2050-1250. DOI: 10.1017/nws.2013.2. URL: http://journals.cambridge.org/article_S2050124213000027.

[6] Joan Bruna et al. “Spectral Networks and Locally Connected Networks on Graphs”. In: *CoRR* abs/1312.6203 (2013). URL: http://arxiv.org/abs/1312.6203.

[7] Justin Cheng et al. “Can Cascades Be Predicted?” In: *Proceedings of the 23rd International Conference on World Wide Web*. WWW ’14. Seoul, Korea: ACM, 2014, pp. 925–936. ISBN: 978-1-4503-2744-2. DOI: 10.1145/2566486.2567997. URL: http://doi.acm.org/10.1145/2566486.2567997.

[8] Justin Cheng et al. “Do Cascades Recur?” In: *CoRR* abs/1602.01107 (2016). URL: http://arxiv.org/abs/1602.01107.

[9] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *CoRR* abs/1406.1078 (2014). URL: http://arxiv.org/abs/1406.1078.

[10] Le Cun et al. “Handwritten Digit Recognition with a Back-Propagation Network”. In: *Advances in Neural Information Processing Systems*. Morgan Kaufmann, 1990, pp. 396–404.

[11] “Deep Learning: Online course”. Udacity. 2016.

[12] L. Deng and D. Yu. “Deep Learning: Methods and Applications”. In: *Foundations and Trends in Signal Processing* 7 (2014), pp. 3–4.

[13] R. M. D'Souza and J. Nagler. "Anomalous critical and supercritical phenomena in explosive percolation". In: *Nature Physics* 11 (July 2015), pp. 531–538. DOI: 10 . 1038 / nphys3378. arXiv: 1511 . 01800 [cond-mat.dis-nn].

[14] Vincent Dumoulin and Francesco Visin. *A guide to convolution arithmetic for deep learning*. cite arxiv:1603.07285. 2016. URL: http://arxiv.org/abs/1603.07285.

[15] Sergey Edunov et al. *Three and a half degrees of separation*. URL: https://research.facebook.com/blog/three-and-a-half-degrees-of-separation/.

[16] P. Erdős and A Rényi. "On the Evolution of Random Graphs". In: *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*. 1960, pp. 17–61.

[17] E. Estrada. "Graph and Network Theory in Physics". In: *ArXiv e-prints* (Feb. 2013). arXiv: 1302.4378 [math-ph].

[18] Ernesto Estrada and Philip A. Knight. *A First Course in Network Theory*. 2015. ISBN: 978-0-19-872645-6.

[19] M. Faloutsos, P. Faloutsos, and Faloutsos C. "On Power-Law Relationships of the Internet Topology". In: *Social Networks* 1.1 (1979), pp. 5–51.

[20] Cucker Felipe and Steve Smale. "On the Mathematical Foundations of Learning". In: *Bulletin of the American Mathematical Society* 3| (1 2002), pp. 1–49.

[21] Linton C. Freeman. *The Development of Social Network Analysis: A Study in the Sociology of Science*. BookSurge Publishing, 2004.

[22] Thomas Gärtner, Peter Flach, and Stefan Wrobel. "On graph kernels: Hardness results and efficient alternatives". In: *IN: CONFERENCE ON LEARNING THEORY*. 2003, pp. 129–143.

[23] C. Godsil and G. Royle. *Algebraic Graph Theory*. Graduate Texts in Mathematics. Springer Verlag, 2001. ISBN: 9780198520115.

[24] Ian J. Goodfellow et al. "Maxout networks". In: *In ICML*. 2013.

[25] Awni Hannun et al. "Deep Speech: Scaling up end-to-end speech recognition". In: (2004). arXiv: 1412.5567.

[26] Johan Håstad. *Computational Limitations of Small-depth Circuits*. Cambridge, MA, USA: MIT Press, 1987. ISBN: 0262081679.

[27] Trevor Hastie, Robert Tibshirani, and Robert Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer Series in Statistics. Springer Verlag, 2009. ISBN: 9780387848587.

[28] Mikael Henaff, Joan Bruna, and Yann LeCun. "Deep Convolutional Networks on Graph-Structured Data". In: *CoRR* abs/1506.05163 (2015). URL: http://arxiv.org/abs/1506.05163.

[29] Geoffrey Hinton. "Overview of mini-batch gradient descent." Course on Neural Networks. 2000.

[30] Geoffrey E. Hinton and Simon Osindero. "A fast learning algorithm for deep belief nets". In: *Neural Computation* 18 (2006), p. 27.

[31] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[32] Thomas Hofmann, Bernhard Schölkopf, and Alexander Smola J. "Kernel Methods in Machine Learning". In: *The Annals of Statistics* 36 (3 2008), pp. 1171–1220.

[33] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). URL: http://arxiv.org/abs/1502.03167.

[34] H. Jeong et al. "Lethality and centrality in protein networks". In: *Nature* 411 (6833 2001), pp. 41–42.

[35] Steven Johnson. *The Genius of the Tinkerer*. Ed. by The Wall Street Journal. [Online; posted 25-Sept-2010]. Sept. 2010. URL: http://www.wsj.com/articles/SB10001424052748703989304575503730101860838.

[36] Andrej Karpathy. *CS231n Convolutional Neural Networks for Visual Recognition - Convolutional Neural Networks (CNNs / ConvNets)*. URL: http://cs231n.github.io/convolutional-networks/.

[37] R. Kneser and H. Ney. "Improved backing-off for M-gram language modeling". In: *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*. Vol. 1. May 1995, 181–184 vol.1. DOI: 10.1109/ICASSP.1995.479394.

[38] Mladen Kolar et al. "Estimating Time-Varying Networks". In: *The Annals of Applied Statistics* 4 (1 2010), pp. 94–123.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems*. 2012, pp. 1090–1098.

[40] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (2015), pp. 436–444.

[41] Yann Lecun et al. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE*. 1998, pp. 2278–2324.

[42] Manuel Lima. *Visual Complexity: Mapping Patterns of Information*. Princeton Architectural Press, 2013. ISBN: 1616892196, 9781616892197.

[43] Chao Liu et al. “Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs”. In: *Proceedings of the 2005 SIAM International Conference on Data Mining, SDM 2005, Newport Beach, CA, USA, April 21-23, 2005.* 2005, pp. 286–297. DOI: 10.1137/1.9781611972757.26. URL: http://dx.doi.org/10.1137/1.9781611972757.26.

[44] Ulrike Von Luxburg and Bernhard Schölkopf. “Statistical learning theory: Models, concepts, and results”. In: *Handbook for the History of Logic, Vol. 10: Inductive Logic. Elsevier, 2011. 13 A.C. Tamhane Y. Hochberg. Multiple Comparison Procedures.* John Wiley & Sons, 1997.

[45] *Merck Molecular Activity Challenge.* URL: https://www.kaggle.com/c/MerckActivity.

[46] Thomas Mikolov et al. “Distributed Representations of Words and Phrases and their Compositionality”. In: (2013). arXiv: 1310.4546.

[47] Thomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: (2013). arXiv: 1301.3781.

[48] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. “Linguistic Regularities in Continuous Space Word Representations”. In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT-2013).* Association for Computational Linguistics, May 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=189726.

[49] Andrew Ng. *Machine Learning and AI via Brain Simulations.* URL: http://www-cs-faculty.stanford.edu/~uno/abcde.html.

[50] Michael A. Nielsen. *Neural Networks & Deep Learning.* 2015.

[51] Christopher Olah. *Calculus on Computational Graphs: Backpropagation.* URL: http://colah.github.io/posts/2015-08-Backprop/.

[52] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. “DeepWalk: Online Learning of Social Representations”. In: (2015). arXiv: 1403.6652.

[53] Lutz Prechelt. “Early Stopping - but when?” In: *Neural Networks: Tricks of the Trade, volume 1524 of LNCS, chapter 2.* Springer-Verlag, 1997, pp. 55–69.

[54] Xin Rong. “word2vec Parameter Learning Explained”. In: *CoRR* abs/1411.2738 (2014). URL: http://arxiv.org/abs/1411.2738.

[55] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Neurocomputing: Foundations of Research”. In: ed. by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Chap. Learning Representations by Back-propagating Errors, pp. 696–699. ISBN: 0-262-01097-6. URL: http://dl.acm.org/citation.cfm?id=65669.104451.

[56] J. Schmidhuber. “Deep Learning in Neural Networks: An Overview”. In: *Neural Networks* 61 (2015). Published online 2014; based on TR arXiv:1404.7828 [cs.NE], pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003.

[57] Nino Shervashidze et al. “Efficient graphlet kernels for large graph comparison”. In: *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics*. 2009, pp. 488–495.

[58] Nino Shervashidze et al. “Weisfeiler-lehman graph kernels”. In: *The Journal of Machine Learning Research* 12 (2011), pp. 2539–2561.

[59] Derek J. de Solla Price. “Networks of Scientific Papers”. In: *Science* 149.3683 (1965), pp. 510–515. ISSN: 0036-8075. DOI: 10.1126/science.149.3683.510. eprint: http://science.sciencemag.org/content/149/3683/510.full.pdf. URL: http://science.sciencemag.org/content/149/3683/510.

[60] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2627435.2670313.

[61] Pinar Yanardag and S. V. N. Vishwanathan. “The Structurally Smoothed Graphlet Kernel”. In: *CoRR* abs/1403.0598 (2014). URL: http://arxiv.org/abs/1403.0598.

[62] Pinar Yanardag and S.V.N. Vishwanathan. “Deep Graph Kernels”. In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2015, pp. 1365–1374.

[63] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. “Recurrent Neural Network Regularization”. In: *CoRR* abs/1409.2329 (2014). URL: http://arxiv.org/abs/1409.2329.