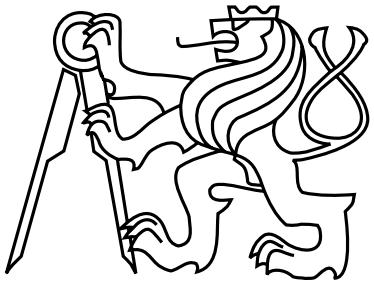
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

Integration of Relational and Deep Learning Frameworks

Bc. Marian Briedon

Supervisor: Ing. Gustav Šourek

Study Programme: Open Informatics

Field of Study: Artificial Intelligence

May 28, 2018

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 25, 2018 .....

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Briedoň** Jméno: **Marián** Osobní číslo: **406463**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Integrace frameworků relačního a hlubokého strojového učení

Název diplomové práce anglicky:

Integration of Relational and Deep Learning Frameworks

Pokyny pro vypracování:

There is a resurgence of interest in neural networks in the machine learning community where deep learning is regularly breaking records on seemingly every possible task. However there is a principal issue with application of these attribute-value learners to relational problems, i.e. problems with inherently structured representations. The aim of this thesis is a research and mainly practical implementation towards succesful applications of existing deep learning frameworks in relational domains.
1) Get an overview of relational learning and neural networks.
2) Review state-of-the-art techniques for applying attribute-value learners to relational problems.
3) Get familiar with existing frameworks for deep learning (e.g. Tensorflow, Torch) and relational inference engines (e.g. PostreSQL, Prolog).
4) Based on your review of existing approaches, propose a method to apply the deep learning frameworks on top of the relational engines.
5) Implement your method(s) using combination(s) of existing engines from both worlds.
6) Benchmark your method(s) from different perspectives (speed, memory) on various tasks (e.g. standard relational learning benchmarks[3] and/or simple games[4]).

Seznam doporučené literatury:

[1] Lise Getoor. 'Introduction to Statistical Relational Learning.' 2007.
[2] Gustav Šourek, Vojtěch Aschenbrenner, Filip Železný and Ondřej Kuželka. 'Lifted Relational Neural Networks.' NIPS CoCo 2015: Cognitive Computation: Integrating Neural and Symbolic Approaches. 2015.
[3] Greg Brockman, et al. 'OpenAI gym.' arXiv preprint arXiv:1606.01540 (2016).
[4] Jan Motl and Oliver Schulte. 'The CTU prague relational learning repository.' arXiv preprint arXiv:1511.03086 (2015).

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Gustav Šourek, Intelligent Data Analysis FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **21.07.2017** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce:
do konce letního semestru 2018/2019

Ing. Gustav Šourek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

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

Abstract

In the recent years, deep neural networks have achieved significant achievements in many subfields of machine learning, such as natural language processing, generating audio files or even lip reading. However, all these neural architectures still have their limitations, for instance, they cannot learn from relational data, which often arise in real world in the form of graphs or databases. On the opposite side there is relational learning, which focuses on interpretable learning from such complex data, where individual learning examples may be differently structured and dependent. Naturally, marrying advantages of both approaches is of a significant scientific interest. The aim of this thesis is on the integration of the deep and relational learning, with a particular focus on a so called templating - a general approach to the integration problem, where relational models serve as templates for automated unfolding of neural networks. Despite its promising properties, at the core of the approach there is an open problem of efficient creation of dynamic neural networks which, being rather unorthodox in standard deep learning, remains largely unsolved. The practical goal of this thesis is to solve this problem via mathematical analysis, custom implementation, and interfacing with modern deep learning frameworks to enhance the integration of the two fields.

Abstrakt

V posledných rokoch dosahujú hlboké neurónové siete významných úspechov v mnohých oblastiach strojového učenia, ako je spracovanie prirodzeného jazyka, vytváranie zvukových súborov alebo dokonca čítanie pier. Všetky tieto neurálne architektúry však stále majú svoje obmedzenia, napríklad sa nemôžu učiť z relačných údajov, ktoré sa často vyskytujú v reálnom svete vo forme grafov alebo databáz. Na opačnej strane je vzťahové učenie, ktoré sa zameriava na interpretovateľné učenie sa z takýchto komplexných údajov, kde jednotlivé príklady učenia môžu byť rôzne štruktúrované a závislé. Samozrejme, o spojení výhod je silný vedecký záujem. Cieľom tejto práce je integrácia hlbokého a relačného učenia s zvlastnim dôrazom na tzv. Templating - všeobecný prístup k integračnému problému, kde relačné modely slúžia ako šablóny pre automatizované vytváranie neurónových sietí. Napriek svojim sľubným vlastnostiam je v jadre templating prístupu otvorený problém efektívneho vytvárania dynamických neurónových sietí, ktoré sú skôr neortodoxné v štandardnom hlbokom učení a zostávajú do značnej miery nevyriešené. Praktickým cieľom tejto práce je vyriešiť tento problém pomocou matematickej analýzy, vlastnej implementácie a prepojenia s frameworky hlbokého učení s cieľom zlepšiť integráciu oboch oblastí.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis talks about statistical relational learning and deep neural networks with a focus on the possibility of their integration. Relational learning [5] is a subfield of machine learning, which focuses on learning the relationships and the structures of the data in an interpretable manner, typically based on relational logic. Relational learning primarily deals with real-world data, which aren't independent nor of a fixed size. The complex nature of data used by relational learning makes it hard to learn by standard classificators and requires specific methods. Statistical relational learning is an extension, which is concerned with domains that exhibit both uncertainty and complex structures. The problem of integration with other models is the use of relations between data as the main information in the process of learning. Statistical relational learning helps with creating the relationships and the structures among the data with a certain probability, allowing to catch and evaluate weaker relations under uncertainty over complex structures over data.

Deep learning is also a subfield of machine learning with algorithms inspired by neural networks. Practically speaking, it is mostly just large neural networks learning from raw data, which allows the deep learning to surpass a lot of other learning algorithms. One of the best features of the neural networks is that their performance doesn't stop with the increase of the data volume. So the increase in hardware power let us use bigger neural networks with larger datasets, which in turn increased the accuracy and the ability to perform on various learning tasks dramatically. In the recent years, it allowed having a big impact on science community, helping to solve difficult problems in multiple subfields of machine learning and artificial intelligence.

The main aim of this thesis is the question whether we can integrate statistical relational learning [6] into the deep learning in a beneficial manner. This is supported by results from other fields, where it had huge success. While deep learning usually works well in the fields with an excessive amount of data, which doesn't change the structure, in statistical relational learning the data can have a very different shape in form of tuples, graphs or logical theories. The problem is thus that this data, that should form the input for the computational graphs, are dynamically changing size and structure. The core issue is to create a method for making the structured data an input of neural networks, which requires the network to somehow adapt to these changes, for instance by building the neural network according to some sort of algorithm.

The integration with statistical relational learning can be divided into three groups; vectorization approaches, relational approaches, and hybrid approaches. The vectorization approaches can be further divided into factorization approaches, neural embeddings [7] approaches and regularizing embeddings methods. Most of the existing papers use the neural embeddings approach since it is the easiest approach to implement with somewhat good results. However this approach mostly simply neglect the relational information within the data. The relational approach, on the other hand, is steered to the opposite side, being too explicit with the relations with poor statistical generalization. Somewhere in the middle, there are the hybrid approaches, based on a combination of the neural embeddings and relational approach. A prominent approach here is the templating, which creates templates in relational languages to be unfolded into the form of neural networks. Using those templates we can encode the relationships in data in the form of neural connections and then learn in a standard manner with gradient descent [8].

However, there are many problems with practical realization of this approach. Even for a single relational example, the learning is not easy. Why is the structure of individual relational examples so big of a problem for neural networks? Because most of the neural network frameworks use only layers as a representation of a neural network. The usual assumption is that the data doesn't change size and structure so then the neural network is easily written in layers. If we want to encode the relations in the neural network we need to be flexible in creating and changing the structure and size of the neural network. This goes against fundamentals of deep learning because the classical neural network only learns values of parameters, not the whole networks. Of course, this approach allowed for great optimization using gpu-acceleration, allowing bigger networks to be computed in a matter of hours instead of days. This lead to great result using more data in relatively small time and the ability to simply increase the hardware power for better working neural networks. Both of those made the deep learning really popular in many fields. However, this doesn't affect our problem of hard-coding relations from data into neural networks, because most of them will not be encoded in simple layers, but in the form of a raw graph where nodes represent each separate neuron and edges represent the inputs/outputs of the neurons as induced by the relations. So a different approach to the coding of the neural networks will be needed.

The templating approach focuses on creating neural networks, which have the specific structure and shared weights of the neural network for each example. This is a different approach to the artificial neural networks currently used. All the nodes are separated into layers. The layers are interconnected using all available edges, this is called fully-connected layer. The input of such layer is limited to the layer before it. All of these things allow for easy transformation into matrix multiplications, where it is not needed to change the structure or the size during the computation process. This doesn't hold for the networks created by templating approach for the integration with the deep learning.

In this thesis, we propose a solution for faster and easier evaluation of neural networks created by the templating approach. There are two main approaches to deal with the differences and the problems. One is to compute each neuron individually. The other one is to first compute matrices of the weights and use those matrices for computing the neural network. Both of the approaches are tested using random graph generator to compare the efficiency and the building time of the neural networks. The random graphs have the same number of layers and have a uniform distribution of nodes in the layers. The generated

graphs have the different density for the purpose of testing the difference in a number of edges. Using different density is also to test some of the implementations.

The frameworks used for implementation of the approaches and for the testing are Tensorflow [9], Dynet [10] and Pytorch [11]. All of those frameworks use eigen [12] as a library for mathematical operation acceleration. The next logical thing was to create a custom framework using eigen and coded in c++. This framework has tested also the usage of sparse matrices for representing the matrices of the weights. First, we test frameworks using cpu-acceleration on both approaches. The next thing is to test the frameworks on the usage of the gpu [13]. The first approach is not suitable for this test as it has a lot of small mathematical operations, compared to computing matrix multiplications, so it is not performed.

The first chapter chapter 2 is about artificial neural networks and how they work. In chapter 3 we discuss relational learning and the statistical relational learning. The discussion about the integration of relational learning and deep learning is then in chapter 4. In the chapter 5, there is the explanation of possible approaches to the problem of integration. The approaches, suggested in chapter chapter 5, are then experimentally evaluated in chapter 6. The results are then discussed in section 6.5.

Chapter 2

Artificial neural networks

Artificial neural networks [14] (ANNs) are a subfield of the machine learning. They were inspired by the functions of the nerve cells in the animal brain. The similarity is that the basic structure of both are neurons, both of them have multiple inputs and one output. This is were the similarities end. One of the other views on the topic of neural network is a multi-layer linear separator. Using multiple inter-connected separators achieves the ability to separate otherwise linearly inseparable data. The artificial neural networks are mainly used for supervised learning, because the standard architectures require the input data to be labeled. The need for labeling is one of the biggest restrictions on the input data. Artificial neural networks have the ability to learn classifications of the labeled data, which is one of the defining parts of this approach. The other one is getting better result using more of the input data. This means it outperforms other methods of the machine learning in the domains with plenty of data. They can solve a big number of tasks from image recognition to natural language processing. ANNs can be used for many tasks with different modus operandi. The convolution neural network uses shared values to compress the image without losing the complexity, recurrent neural networks uses stored values during computation to emulate sequences of signals and self-organizing maps for dimensionality reduction.

The ann is made out of artificial neurons which function as linear classifiers. The neuron consists of weighted sum and activation. A reader can learn more about neurons in section 2.2. Each edge between artificial neurons can transmit a signal from the sending to the receiving end. The artificial neuron that receives the signal can process it and then signal to artificial neurons connected to it. In common ANN implementations, the signal at a connection between artificial neurons is a real number, and the output of each artificial neuron is calculated by a non-linear function of the sum of its inputs. Artificial neurons and connections typically have a weight that adjusts as the learning proceeds. The weight increases or decreases the strength of the signal at a connection. Artificial neurons may have a threshold such that only if the aggregate signal crosses that threshold is the signal sent. Typically, artificial neurons are organized in layers. Different layers may perform different kinds of transformations on their inputs. Signals travel from the first (input) to the last (output) layer, possibly after traversing the layers multiple times. The output layer neurons may omit the activation function for the purpose of regression.

2.1 Deep learning

Deep learning [15] is a class of machine learning methods, primarily focusing on raw data. The deep learning is used in many domains of problems from supervised (classification) to unsupervised (pattern analysis). It uses the gathering of knowledge to build a hierarchy of concepts from simpler to more complicated ones. The features of this graph are that the concepts are built upon each other, forming many layers and creating deep graphs. Because of the hierarchical structure of features, these methods are also called Deep structured learning or Hierarchical learning. Deep learning also has a big potential for transfer learning and a feature extraction.

The classic approach used human experts to preprocess the data to learn from. The humans used various algorithms to extract features, attributes, and models form the raw data. So the machine learning needs a programmer to first create a relevant dataset using different methods, then the dataset is used to learn the actual model. The deep learning, on the other hand, learns from raw data. Using raw data without the help of a human to find the important features to finish task makes the algorithm to look for features, which aren't normally used by other methods. This approach to using a raw data as input, makes the deep learning fully automatic, bringing both the effective and the innovative approach to old and new problems of machine learning. The other side of using raw data is that we need a lot of it to be efficient. This means we can use raw data in the fields of machine learning where there is an abundance of data. Using a lot of data to train the neural network allows for better results and the neural network produces better models with more data used. With this in mind researchers developed convolutional neural networks. The idea is to use many different layers in one neural network for different purposes. One good example is image recognition where the first layers are used to learn models and to extract features, then it uses different layers to complete the neural network. In this thesis, we will not focus on the convolutional neural networks, but use similar methods to integrate deep learning with statistical relational learning.

In deep learning [16], each level learns to transform its input data into a slightly more abstract and composite representation. In an image recognition application, the raw input may be a matrix of pixels; the first representation layer may abstract the pixels and encode edges; the second layer may compose and encode arrangements of edges; the third layer may encode a nose and eyes; and the fourth layer may recognize that the image contains a face. Importantly, a deep learning process can learn which features to optimally place in which level on its own. (Of course, this does not completely obviate the need for hand-tuning; for example, varying numbers of layers and layer sizes can provide different degrees of abstraction.

The "deep" in "deep learning" refers to the number of layers through which the data is transformed. More precisely, deep learning systems have a substantial credit assignment path (CAP) depth. The CAP is the chain of transformations from input to output. CAPs describe potential causal connections between input and output. For a feedforward neural network, the depth of the CAPs is that of the network and is the number of hidden layers plus one (as the output layer is also parameterized). For recurrent neural networks, in which a signal may propagate through a layer more than once, the CAP depth is potentially unlimited. No universally agreed threshold of depth divides shallow learning from deep learning, but

most researchers agree that deep learning involves CAP depths greater than 2. The CAP of depth 2 has been shown to be a universal approximator in the sense that it can emulate any function. Beyond that, more layers do not add to the function approximator ability of the network, but the extra layers help in learning of the features.

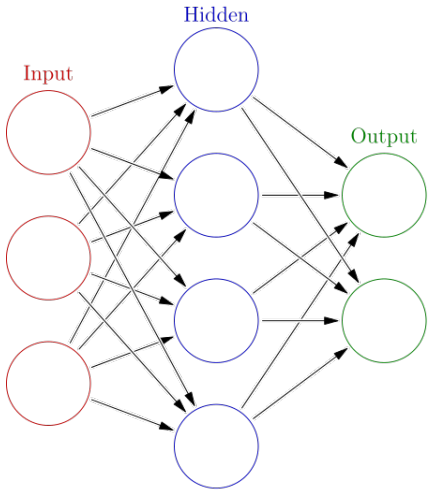


Figure 2.1: Artificial neural network with multiple layers [1]

2.2 Neuron

An artificial neuron is a mathematical function created to simulate a work of biological neurons. Building elements of an artificial neural network are artificial neurons. Usually each input is separately weighted, and the sum is passed through a non-linear function known as an activation function [17]. The transfer functions usually have a sigmoid shape, but they may also take the form of other non-linear functions, piecewise linear functions, or step functions. They are also often monotonically increasing, continuous, differentiable and bounded. The thresholding function has inspired building logic gates referred to as threshold logic; applicable to building logic circuits resembling brain processing.

The equation of the neuron

$$y_n = \varphi(\sum_{i=0}^{m} w_{ni} * x_i) \tag{2.1}$$

In the section 6.3 above is seen the neuron. The φ represent the activation function, the w represent weight of the input and x,y are input and output respectively.

The basic unit of artificial neural network is neuron. It is like the node of graph. It receives multiple inputs from various sources but most importantly from other neurons and have only one output. Usually the neuron consists of activation function and a weighted sum. The activation function makes the neuron non-linear and resembles a fuzzy logic variable with values from 0 to 1. Activation function needs to be non-linear function converting

number from input to output. This is for training, the usual way to train neural network is through backpropagation [18] using the derivatives of the activation functions. So the derivation of the activation functions needs to be efficiently computed for the speed of the training neural network. All of them have different derivatives, which can take different time to calculate. The other expect is how big the derivation can happen to be so for this reason is used learning rate, which is different for all the activation functions. The learning rate allows to change the absolute change of the weight allowing to regulate the absolute change of the weights in neural network.

Activation functions

- sigmoid $1/(1 + e^{-x})$, a sigmoid function is real-valued, monotonic, and differentiable. Sigmoid having a non-negative first derivative $x * (1 - x)$, which is bell shaped
- tanh, function looks like sigmoid is real-valued and differentiable and has derivation $1 - tanh^2$
- rectified linear unit (ReLU), the newest activation function $f(n) = max(x, 0)$, which doesn't have a derivation so for derivation we use very smooth aproximation $f(x) = log(1+e^x)$, which is called softplus function. It's derivative is logistic(sigmoid) function.

The problem of sigmoid is that the peak of the derivation is only 0,25 so the derivation value rapidly decreases towards lower levels making the multilayer network almost impossible to train. This effect is called vanishing of the derivation and makes the effect of using mostly two or three layers of linear layers. So for the bigger network is recommended to use relu as an activation function.

2.3 Representation of artificial neural networks

The mathematical representation of a neural network is a graph with nodes and edges. However, the neural network frameworks use layers for representing neural networks. The benefit of the layer representation is that the same operation can be used on multiple data inputs. This helps when using gpu acceleration to compute large networks. This enabled rapid growth in usage of neural networks. It was a lot of times faster than anything else, enabling the use of large neural networks with fast learning time. This method is used primarily for static graphs were the model is created once and then used for learning. However, we need to change a graph dynamically, which means we need some extra instructions for changes of the neural network model. This creates problems with layer-wise interpretation because usually, the graph changes substantially. This means we will need to either compute beforehand what changes happen in which layer and then apply them, or use a different representation of the neural networks.

For dynamically changing graphs, it is best to use a mathematical representation with the representation of changes, or simply a totally different graph after each change. This will require an approach rapidly different to that of standard neural network frameworks. It is not preferable to create whole new neural networks for every change that occurs in the representation. For this reason, it is necessary to represent changes to a graph separately and also store the labeled data for each stage of the dynamic neural network. This results in

faster deployment of dynamically changing graphs because we don't need to program each change separately, allowing the user to develop faster different neural networks. The only problem of this is the size of memory it requires to run properly.

2.4 Different types of artificial neural networks

Machine learning tries to deal with a lot of problems in multiple domains. So each domain of problems has separate attributes and representation. The specific domain includes processing of signals and patterns in image recognition or natural language processing. For each specific task in each domain, there were specific neural networks created, according to the needs of each domain. This section presents multiple approaches to the problem of neural networks and explains the changes and special parts to deal with those problems. The difference come in many shapes, like sharing values, changing dynamically structure of the neural network or reusing the outputs as the inputs for the neuron in the same layer or in the layer before.

2.4.1 Feedforward neural networks

It's the simplest neural network where the information goes only one way – forward from input nodes to output nodes through hidden nodes. No cycles or loops are allowed. So, the graph of a neural network is of a directed acyclic type, allowing easy transformation into layers in most cases. The advantages of the feed-forward neural network [19] is in the signals going only one way, making the neural network computation fast and easy. Most of the feed-forward neural networks use multiple layers. The hidden layer usually serves the purpose of abstracting higher level features from the data. The layers are usually fully-connected and most of the time they don't share weight values.

2.4.2 Convolutional neural networks

It takes input in form of matrix, the whole matrix shares weight values. Next thing it applies filters to the input. The filters are only parts of the initial matrix, this means that we either zero-pad or other technique to make the initial input smaller and the relevant data to be seen. The smaller the parts to some degree of the initial picture, the better the accuracy of the network. But with smaller parts increases the number of parts exponentially making the balance between the computational time and the accuracy. On those parts of the image is applied some sort of pooling making it smaller in size allowing to shrink the input into small parts without losing the complexity. When the input shrinks to the size usable by the fully-connected neural network, we apply than the fully-connected neural network. This points to the fact that convolutional neural network [20] are primarily used to shrink the data size while keeping the information loss to a minimum.

2.4.3 Recurrent neural networks

Recurrent neural network [21] is a special type of neural network. Its specialty lies in that it reuses outputs of some neurons and use them as inputs for other neurons in the same

or lower layers. This also means we can expand representation of input data from vectors to sequences. This special attribute of the recurrent neural networks allows the network to have a temporal behaviour. This allows them to be applicable to task requiring the sequence recognition such as handwriting recognition or speech recognition. The huge downside of this is that it need substantially more computational power than the acyclic neural networks. The neurons inside recurrent neural network need to store the inner state for the purpose of backpropagation, because we use the same neurons more than one time. This inner states are called blocks. Most of the blocks have structured state and the inner states can be part of the neural network, the example of which is the Long short term memory.

Long short-term memory [22] (LSTM) neurons are used as blocks to built layers of a recurrent neural network (RNN). Long short-term memory(LSTM) neurons have the ability to store previously used values for enhancing the gradient descent. Using those stored values make the gradient descent to be relevant in the next evaluation creating the notion of using values from the previous training, allowing to simulate the sequences.

2.4.4 Dynamic neural networks

The dynamic neural network frameworks are focused on dynamically changing the structure of the neural network with each successful run. This approach needs three methods, the init, the forward and the backward pass over the graph. The forward pass is to define the structure of the neural network using global variables during the evaluation and the backward pass for the backpropagation and training of the neural networks. The init is to initialize the number of used variables and the basic structures of the neural networks. The init part makes adding new neurons and layers problematic, so the creator of the neural network needs to know how many neurons and layers is needed to correctly encode the neural network. This means the neural network can only change the structure not the size because we declared the size of the neural network in the initial phase. The other side is that it can use part of the initialized neurons for the neural network representation. The focus is on light-weighted representations of the neural networks and not so much on optimization for the neural network computation. Letting the programmer have more freedom in a way of building the neural network comes at a cost of optimizing such neural networks by the framework. Good examples of the dynamic neural network frameworks are Pytorch and Dynet.

The static neural networks are the opposite of the dynamic neural networks, thus the focus of them is to encode only one model of the neural network. Encoding only one model of the neural networks, which doesn't change during the usage of the neural network, allows for better optimization and predefined structures. The optimization of the neural network takes time, making the build and initialization time much longer than the dynamic neural network. Example of the static neural framework is Google's Tensorflow.

Thus, the biggest problem of dynamic networks is that they have very slow computation and most of them don't use multiprocessing for computation. Small size and relatively sparse neural networks make them only slower using gpu acceleration. This makes them comparably slow, and mostly faster to use with custom written software than any existing framework. The best way to interpret and implement their computational graph is to compute each neuron separately in a stream of them w.r.t. their topological sorting.

2.5 Training and optimization of neural network

In the training phase, the correct class for each record is known (this is termed supervised training), and the output nodes can, therefore, be assigned "correct" values – "1" for the node corresponding to the positive class, and "0" for the others. (In practice it has been found better to use values of 0.9 and 0.1, respectively.) It is thus possible to compare the network's calculated values for the output nodes to these "correct" values and calculate an error term for each node (the "Delta" rule). These error terms are then used to adjust the weights in the hidden layers so that, hopefully, the next time around the output values will be closer to the "correct" values.

A key feature of neural networks is an iterative learning process in which data cases (rows) are presented to the network one at a time, and the weights associated with the input values are adjusted each time. After all, cases are presented, the process often starts over again. During this learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of input samples. Neural network learning is also referred to as "connectionist learning," due to connections between the units. Advantages of neural networks include their high tolerance to noisy data, as well as their ability to classify patterns on which they have not been trained. The most popular neural network algorithm is back-propagation algorithm proposed in the 1980's [23].

Once a network has been structured for a particular application, that network is ready to be trained. To start this process, the initial weights (described in the next section) are chosen randomly.

The network processes the records in the training data one at a time, using the weights and functions in the hidden layers, then compares the resulting outputs against the desired outputs. Errors are then propagated back through the system, causing the system to adjust the weights for application to the next record to be processed. This process occurs over and over as the weights are continually tweaked. During the training of a network, the same set of data is processed many times as the connection weights are continually refined.

Note that some networks never learn. This could be because the input data do not contain the specific information from which the desired output is derived. Networks also don't converge if there is not enough data to enable complete learning. Ideally, there should be enough data so that part of the data can be held back as a validation set.

The core of backpropagation is to apply the chain rule through all of the possible paths in our network, till the neural network is trained. The problem is that the number of the paths between input and output is exponential. This can be dealt using dynamic programming to store and reuse the results of inner neurons. In training, we backpropagate the errors and using the stored results we can easily calculate the gradient of a neuron. This process is related to the evaluation of the neural network, but it propagates the errors instead of the results. [24]

So the general equation for backpropagation is

$$\frac{\partial E_{total}}{\partial weight} = \frac{\partial E_{total}}{\partial output} * \frac{\partial output}{\partial sum} * \frac{\partial sum}{\partial weight} \tag{2.2}$$

derivation of error according to the derivation of the output of the node

$$\frac{\partial E_{total}}{\partial output} = (output - target) \tag{2.3}$$

derivation of sigmoid(logistic) function , here comes the derivation of the activation function

$$\frac{\partial output}{\partial sum} = (output * (1 - output)) \tag{2.4}$$

the last part is derivation of the sum according to the weight meaning the weight value gets out of the backpropagation and doesn't change the value of the total derivation

$$\frac{\partial sum}{\partial weight} = (1 * input * weight^{1-1} + 0 + 0 + ...) = input \tag{2.5}$$

so together we get

$$\frac{\partial E_{total}}{\partial weight} = (output - target) * (output * (1 - output)) * input \tag{2.6}$$

We can combine and calculate two of the derivations into one and use it for the whole node, this is called delta. It is used in backpropagation for faster calculation, because the other significant part is the input of the neuron.

$$\partial delta = \frac{\partial E_{total}}{\partial output} * \frac{\partial output}{\partial sum} = (output - target) * (output * (1 - output)) \tag{2.7}$$

For the next layers underneath.

$$\frac{\partial E_{total}}{\partial output} = \partial delta * weight \tag{2.8}$$

For the next neuron we need to multiply the output of the delta by the weight, this is caused by the shape of the neural network which is composed function of all the neurons it goes through. So the delta have to be multiplied by the derivation of the output plus the weight of the output in the next neuron. This makes the fully-connected neural networks so fast because we don't need to calculate the weight changes for all individual neurons, but instead using two vectors we create a matrix that is subtracted from the weight matrix. This way we use the backpropagation multiple times making it faster than any other representation of the artificial neural network.

Chapter 3

Relational learning

The first attempts to learn logical concepts were done using symbolic AI, which didn't work well. After that, they shifted to using statistical models on the independently and identically distributed samples. The boost to accuracy on low-level tasks was significant enough to make it standard in the community. The difficulties are in the real world domains, which don't have independent data or aren't identical in terms of size and structure. Example domains include natural language, biological, social, or computer networks. All domains exhibiting a structure that isn't fixed for all samples aren't i.i.d., for example, knowledge bases and graphs. The biggest database of the knowledge in the real world is the Internet. The data comes in a lot of different structures and are interlinked. Every entity has its own set of attributes and can belong to multiple types of entities. Learning the variously structured data samples, which are often part of a bigger structure is covered by relational learning.

So most of the community switched to propositional, fixed sized data from the relational data. An approach that stayed on the relational side is Inductive logic programming, which uses the language of the First-order logic (FOL). It has the ability to use background knowledge as well as being transparent to people. The syntactic and semantic language biases can be dealt with prior to learning. All of this comes at the cost of efficiency. The general concept learning problem is undecidable and the decidable subproblems are at least NP-complete. The other problem of the relational learning is uncertainty and noise in the data. Learning based on logic doesn't have any methods to deal with those problems. The uncertainty of the data rises from multiple things such as attributes, types and membership of the relationships. To deal with the issue, researchers proposed Statistical relational learning, which is discussed further in section 3.2.

3.1 Logic

3.1.1 Propositional logic

Propositional logic is mainly concerned with propositions and use of logical connectives and the truth value of them. They use to decompose the statements into components, which are then used to determination of the truth value of the whole statement. Using

propositional logic, we can define relationships such as a tree is in a forest and leaves are on the tree. Suppose we believe that both statements are true, then using implication rule we get that leaves are also in the forest. The overall expressiveness of the language is small due to the problem of naming every single possibility, which makes the domain to increase exponentially. For example, that we wanted to say that, in general, if one person knows a second person, then the second person also knows the first. We can't do this in propositional logic which can't formulate rules. The positive is the decisiveness of the propositional logic.

3.1.2 First order logic

First-order logic has a similar structure to propositional logic. The difference is that in first-order logic, instead of propositions in the propositional case, we use "predicates" possibly describing sets of entities (a,b,c,..), instead of only one proposition. In the predicates, we may use variable symbols X to represent entities (a,b,c,...) (like the way we use variable symbols to represent numbers in elementary algebra). The entity variable X can then be described by two quantifiers: "for all" and "there exists".

For example in propositional logic we represent sentence *Socrates is a man*, we use two expressions in first-order logic:

- "There exists X such that X is Socrates."
- "X is a man."

In this way, we used quantifier "There exists", which tells us that at least one Socrates in the set of variable **X**. Then we have two predicates such as "is Socrates" and "is a man". Using variables and two quantifiers we can reduce the number of propositions needed to cover more complex scenarios. This feature allows us to use complex logical models and most of the time to decide their truth value, but it is proven that it can't decide all clauses by Alan Turing and Alonzo Church.

3.2 Statistical relational learning

The majority of learning algorithms focuses on propositional data, which assumes independence and fixed structure. Nevertheless, the real data aren't propositional in most cases but are relational. Relational data are neither independent nor have fixed structure. This means relational data are composed of different classes, which have a different set of attributes. In the real world, everything has relations with almost every other thing creating a huge number of relational tables. The structure of relational data enables to have additional information, which is used to show correlations and relationships between entities. Statistical Relational Learning [6] (SRL) is a branch of machine learning that tries to model real world using relational data. SRL can deal with more complex problems than the relational learning to use statistical models to predict uncertainties. SRL model shows the relationships between data but can also show dependencies of attributes in different relational tables.

SRL uses first-order logic to describe relational properties of a class of entities in more general manner, than the relational learning. This is achieved using probabilistic graphical

models to model the uncertainty, some methods go even further building upon the methods of inductive logic programming. The field of usability of the Statistical relational learning is not limited only to learning it can also reason and represent knowledge.

The template is a relational rule-set, which is used to unfold the statistical models, in our case neural networks. The templated models are also called lifted models, and they define patterns, which can unfold specific ground models.

Chapter 4

Integration of deep learning and relational learning

Integration of the deep learning and relational learning has been attempted many times in the recent years. A lot of the works were quite successful but were either slowed to calculate or very specific in the field of application. This thesis focuses on how to find a general way to enhance faster neural computation of the integration using existing frameworks, for which we later introduce two approaches. In this section, we explain what are possible approaches to the integration of the deep learning within statistical relational learning, and what is beneficial to them.

Relational learning and statistical relational learning to have the ability to learn logic models. Those models can express depth and complexity of the real world. This results in the data they used to be non-trivial and having relationships encoded in them. The deep learning had many breakthroughs in the field using real data. The integration of those two should bring an easy way to learn from relational data using methods developed by the deep learning.

4.1 Vectorization approach

Vectorization approach focuses on using vectors for representation of the data. In the vector, we represent all the combinations of the data and their relationships. The aim of this is to turn complex data into data, which satisfies as an input data into the neural network. This allows using the neural networks directly on the vector representation of the data, without changing the neural network in any way. This combination is effective for the reason that it needs a small change in representation and can use any and all of the algorithms and possibilities of the deep learning. The problem with this approach is to create a sophisticated algorithm for representing multidimensional problem into single fixed size representation, which is not only hard but generally impossible. Vectorization approach deals with the problem with groups of methods such as factorization, neural embeddings, and regularizing embeddings. Factorization approach views relations as the products of different facts. Using factorization it breaks down the relations into small parts, which encodes into vectors. Neural embeddings use a set of relational features to be encoded into artificial neural

networks, making them capable to learn from relational data. Regularizing embeddings focuses on regularization of the vectors, creating additional information to represent complex relations between data attributes.

4.2 Relational approach

The relational approach focuses on building hierarchies from relational features. Relational features are part of the relational logic. It is formally defined as a minimal set of literals such that no local variable occurs both inside and outside that set. This means that one term can have multiple relational features. The relational features are usually set of occurrences of one literal in the entire rule. Their hierarchy can be defined as the minimum overlapping features in the set. Learning the hierarchies form the features can be problematic, for the definition of the hierarchy. Introducing hierarchy into the relational data is beneficial for deep learning. The hierarchy can define the structure of the data, which can then be used to improve results of the deep learning methods.

4.3 Templating

Templating is part of hybrid approaches. It combines neural embeddings with relational feature hierarchies creating new approach towards the relational data. Using relational feature hierarchies to create templates of neural networks and neural embeddings to set weights of those neural networks. The learning part is in sharing the weights in the neural networks, which are created from the template. So the fact that each data creates its own neural network means it is used mostly for the problems which have small domains and very complex structure. This restriction can't be principally bypassed using faster computation frameworks.

4.3.1 Lifted Relational Neural networks

An example of templating approach is that of Lifted Relational Neural networks (LRNN) [25]. LRNN is a method of deep lifted relational neural network learning, in which the structure of neural networks is unfolded from a set of weighted rules. The template is created by using those rules and training and testing examples. The distinguishing feature is that for the neural network construction it uses also the examples and not only the relational logic rules.

Process of the neural network unfolding

- 1. For every ground atom there is atom neuron. Inputs of an atom neuron are the aggregation neurons and fact neurons. The weights of the input neurons are the respective weights from the rules.

- 2. For every ground fact, there is a fact neuron, which has no input and always outputs a constant value.

- 3. For every ground rule, there is a rule neuron. It has the atom neurons as inputs, all with weight 1.
- 4. For every rule with weights and every valid substitution, there is a aggregation neuron. Its inputs are all rule neurons with all weights equal to 1.

So to summarize the LRNN will always have a hierarchically repeated structure of 3 layers. With the input represented by the ground facts, leading into patterns represented by the rule neurons, which are further aggregated by aggregation neurons, finally leading into atom neurons representing true statements. It follows that the neural network can change the structure and grow in size with different examples, but only to bigger layers and not in the maximal depth. The best way to optimize it for bigger data is to use gpu-acceleration, which is further discussed in subsection 6.2.2.

4.3.2 The Problem

The general problem of templating is to create fast dynamic neural networks. The meaning of dynamic in this context is dynamically changing size and shape. To change the shape and size usually requires changing the whole structure of the neural network model. In most frameworks, it will be very slow to change the shape and size multiple times. This leads to a need for the library to quickly and efficiently change the model of the neural network according to some algorithm. This will make the templating approach reasonably fast to use w.r.t. comparable approaches.

This problem has two parts, having to represent each neural network and have to change and evaluate models of changing neural networks. The problem of a representation is that most of the frameworks use a layer representation for neural network model, but to effectively change size and shape we need to pinpoint changes in each neuron. This means we need to represent the model but also to have a special term for most neurons. The amount of information we need to store is huge, from the inner values to the look of neural networks making it demanding in terms of memory. The effectiveness of the representation depends on the ability to represent changes to the graph for quickly changing models, and to be easily loaded.

Chapter 5

Approach

Our approach to the problem of dynamic neural networks is based on reading the templated network as a graph (e.g. out of some file) and then creating the neural network model according to a given standard. We need to create the neural network on the level of individual nodes, not layers. This means that the optimization of the computing process of the neural network is different, as we will not need to multiply matrices and vectors to calculate the whole layers of the neural network. Even if we precalculate the layers out of the network, we would get sparse matrices which will be less efficient in multiplication that the fully connected. The efficiency of converting a graph into layer representation will depend on how interconnected the graphs are and if the calculation for each node isn't faster.

So we have two main approaches; to create sparse matrices for a transformed layer representation, and to evaluate the neural network individually by its neurons. Comparison of these two fundamental approaches is in this section. First, the approach of matrices is explained and some calculations explaining the ideas behind this approach. Next, the focus shifts to an explanation of ideas behind the individual neurons approach and the features it provides. Then we go through some representative cases to determine the speed of evaluation of the approaches. The cases differ with the density of graphs, the numbers of nodes, the inclusion of "jumping neurons" (a concept explained later in subsection 5.3.3), and patterns of sharing neurons' parameters within and across multiple neural networks.

5.1 Matrix approach

The matrix approach creates the standard representation used by most of the existing frameworks. The approach is about creating matrices that represent weights for the layers. The approach needs to calculate matrices for each layer, based on the input structure of that layer. First, it is needed to calculate the input of the layer, the next step is to create a weight matrix according to the input of the layer and the position of inputs of neurons in that layer. The bias can be added after multiplication, to the result. If the graph can't be parsed into layers and has jumping neurons, we need to build input vectors for every layer. The evaluation of the graph starts by preparing the input vector, which is needed for the multiplication. The input vector is taken from the outputs of previous neurons or is the input of the whole neural network. Subsequently, the input vector is used for matrix

multiplication and the output is achieved by applying the activation function on the result from matrix multiplication. The last part is to store the output of each neuron, which is subsequently used in the next layer. The problematic part is to store and load vectors for each layer, which enhances speed. this part is needed to deal with the jumping neurons.

```
Data: graph
for layer<layers do
    for node->nodes do
        if node.layer == numberOfLayer then
            nodeOfLayer.add(node);
        end
        for input->nod.input do
            foundInInputVector = True;
            for input-> inputOfLayer do
                if input == layerInput then
                    foundInInputVector = False;
                    break;
                end
            end
            if found==True then
                inputOfLayer.add(input);
            end
        end
    end
    for i:=0 to nodeOfLayer.size do
        node nod = nodeOfLayer[i];
        for input->nod.input do
            positionInInputVector = 0;
            for l:=0 to inputOfLayer.size do
                if inputOfLayer[l] == input then
                    break;
                else
                    positionInInputVector++;
                end
                layerMatrices[numberOfLayer][positionInInputVector]=1;
            end
        end
    end
end
```

Algorithm 1: An algorithm for creating matrices from the graph representation

For matrix multiplication, most of the deep learning frameworks use a library called Eigen [12]. One of the ideas to increase the speed of the evaluation and the backpropagation of the sparse neural network is by using sparse matrices. The sparse matrices are basically an array of triplets, so the smaller they are, the faster the computation of the neural network is. The bigger problem could be their crosswise multiplication and subtraction from dense matrices. This can cause the sparse matrices to be slower overall than the dense ones. This

effect can be seen in the experiments.

The prerequisite of this approach is to use topological sort and add a layer value to each attribute. The next step is to calculate the input of the layers and nodes that belong to that particular layer. Using those we create the weight matrix, which is used for the representation of the graphs. The pseudocode1 below showcases an algorithm which creates the weight matrices along with layer inputs and layer nodes. To change the classic neural network evaluation and backpropagation is to add one element-wise multiplication with the scheme of the neural network. Adding this multiplication we ensure that the values set as zeroes at the beginning of the neural network evaluation stay zeroes. This means no new edges are added during training of the neural network.

The matrix representation can be done using two types of matrices, the normal dense ones, and the sparse ones. The sparse matrix is mostly represented as the array of triplets, array of sparse columns or array of sparse rows. All of those interpretations have one in common, they are arrays. So most of the operation will be set as operations with arrays, which can be potentially slow. Another disadvantage is that the representations are not interchangeable. This means if the operation is between two different representations, then it is significantly slower compared to the operation between the same types. The problematic part of representing the sparse matrices is that most of the deep learning frameworks don't support using sparse matrices in their code.

5.2 Graph approach

```
Data: list of neurons in graph
sortedNeurons=TopologicalOrdering(neurons);
for neuron->sortedNeurons do
    inputVector=();
    if neuron.input.length==0 then
        inputVector.add(inputNeuralNetwork);
    else
        for input->neuron.input do
            inputvector.add(neurons[input]);
        end
    end
    result=activationFunction(inputVector+neuron.bias);
    neuron.output=result;
end
```

Algorithm 2: An algorithm for graph approach

The graph approach is about evaluating each neuron individually. The algorithm works the way that, first is to store the list of topologically ordered list of neurons. The second is to go through all of the neurons using topological order. Topological order ensures that every input of being calculated neuron has been calculated before. This approach is easy to program and understand. It doesn't need to have things precalculated for it, which is the biggest plus to this approach. This means it can efficiently adapt to changes and is better for the recurrent neural networks. The problem is it doesn't share the computed vectors of

the deltas, meaning it needs to calculate it multiple times. Meaning it is typically slower than the matrix approach, especially with the higher number of the nodes. This approach is suitable more for dynamically changing or recurrent neural networks rather than a static neural network.

5.3 Performance indicators

The key indicators that affect the performance of individual approaches are introduced here. They were extracted based on the problems we explored from experiments with classic frameworks for the neural networks. We analyzed them and experimented to prove which frameworks were good at dealing with them, and for the cases that didn't suit any of the existing frameworks, we propose our own method of dealing with them. The general issue was defined in subsection 4.3.2. In this section the focus shifts on how to deal with it, and to explain to the reader the root causes and some of the possible solutions. The main indicators of the evaluation time are the density of the graph and number of nodes in the graph. Those two indicators are interconnected, with this I mean that density and number of nodes in graph together creates the number of edges in that particular graph.

5.3.1 Density

The density of a directed simple graph is defined as $D = |E|/|V| * |V - 1|$ and it goes from 0 to 1. The overall number of edges is unsatisfactory for most of the neural networks because the neural networks are structured in interconnected layers and don't usually have edges connecting more than the two neighboring layers. This means that standard equation will overestimate them in most cases, so there is a need for the better equation for counting the density. We propose to count the maximum amount of the edges as the sum over all connected layers as an input of layer times the output of the layer. The equation is shown at the equation 5.3.1.

$$D = |E|/|V| * |V - 1| \tag{5.1}$$

$$density = \sum_{i=0}^{layers} edges/[output * input] \tag{5.2}$$

It goes from 0 to 1, an upper limit is 1 because the maximum amount of edges in a directed graph the maximum amount of edges one node can have is V-1, but if we use directed each node can have the same number of edges, because it will be the some of receiving edges and sending edges. Using directed edges we only double the possible edges compared to the undirected case.

We propose to count the percentage per connected layers with the equation being connections / output*input, whereby *output* we mean the output of the previous layer and by input we mean the input of the layer we count the density for. So the definition of the density of a layer is defined, now let's define the density for the whole graph which is the $connections/AllEdges$. The equation for the density takes into an account the standard equation for the directed graph density and changes it according to our need. The change is in a reduction of the maximum number of edges and is done by summing all possible edges

through the whole network. The problem in using standard density is the fact that the numbers would be relatively low compared to the actual density. That's why we proposed different equation.

5.3.2 Size

The number of nodes and number of edges is the best indicator to determine the computation time for the graph of the neural network. So the density is determined by the size of layers, but the overall distribution of the nodes into layers also matters. If we align bigger layers near each other and put most of the neurons into those big layers, the overall number of edges increases. This means that even if two networks have the same number of neurons and the same number of layers, the number of edges can be different. The number of edges is one of the best indicators for the computation time in the matrix approach. The upper limit to the number of edges is the second power of the number of nodes. So the density can change the overall number of the edges but can't stop the rapid growth of the edges compared to the nodes.

5.3.3 Jumping neurons

Jumping neuron is a neuron which has input from other layers than the layer right in front of him. It can't be encoded into a classical fully-connected neural network. So we need to encode the neural network with more edges. These neural connections make the creation of matrices for the layers much more difficult. The complication comes in terms of using more space for the matrices. For the frameworks, we need to code it into them. The general idea is to code one layer at a time and use an output of a layer as an input of a different layer. The only way is to create an input vector for each layer using jumping neurons. Because of this is essential to store the outputs of each neuron and to generate vectors describing the input nodes of each layer. The added code for creating the input vector and storing the results of each neuron is making the overall computation slower.

5.3.4 Multiple graphs

In this case, the graphs of the neural networks need to share neurons or values of some subparts. These demands are explained in section templating approach. So for sharing of the values we will use a list of triplets and encode the shared neurons in labels. Then we need to change the values of the shared edges every time we change the graph. So the time will increase, because we need to change the values in the matrices from the first approach, or use the graph approach for sharing neurons. The one approach to test this is by creating one big neural network and use the smaller parts for the learning process. The one proposition is to build one big graph and, by using gpu-acceleration and zero-padding for the input and output, we execute the graph. The only problem seemed to be the biases producing nonzero output, so we removed them. The system with only one big graph works pretty well. For sharing values throughout the course of actions we need to create an algorithm for the extraction of required values and placing them on the next matrix. This seems to be reasonably fast for changing values. So the biggest indicator of which of those two methods to use is how interconnected the graphs are. The zero-padding of the input can decrease the

computation time by a little, but it will not stop from multiplying the matrices even if most of the vector is full of nulls. The zero-padding is done by using the graphs input and output, before creating big graph, and according to the places of the neurons from the graphs. Next we increase the inputs and outputs to match the big graph inputs and outputs by adding zeros, such that the only non-zero numbers match the neurons of the smaller graph.

5.4 Deep learning frameworks

Deep learning frameworks are created for purpose of accelerating the learning process of the deep neural networks. For this reason, they rely on the gpu acceleration using cuda libraries. The deep learning networks offer preprogrammed different layers, activation functions and tools for creating, training and testing of the deep learning neural networks. The explanation of deep learning neural networks is in section 2.1. There are many deep learning neural networks. For the purpose of creating and testing dynamic neural networks, is needed only part of the deep learning frameworks. The best known dynamic deep learning frameworks are pytorch and dynet. The best known deep learning neural framework is probably the Tensorflow, which has a library called tensorflow-fold for dynamic neural networks. The deep learning frameworks use automatic gradient computation so a user is required to code only the forward part of the neural network. Most of the deep learning frameworks have programmed optimizers of the neural networks coded by users. This means they can automatically change the neural network computation time for the purpose of optimizing the evaluation time. This technique is mostly used by deep learning frameworks focusing on the static graphs because to optimize the graph after every change would be time demanding.

5.4.1 Custom framework

The custom solution [3] is programmed in the c++ using eigen [12] library to accelerate computing of mathematical operations. It has its own part for reading graphs in gml format. The graph is read by going through lines looking for regex of the nodes and edges. Upon finding the regex lines are extracted and processed into parts of a graph. This solution seems to be faster than the solution used by python library networkx [26]. In the occasion of loading multiple graphs, we store the graphs with the labeled data into a map. The next thing is to find the shared neurons of the graphs and prepare the structure for extracting the shared values from one graph and implanting them to the other one. It can process different activation functions based on the attributes of nodes in the graph.

The next thing is to process the graphs into matrices. The pseudocode for this is in the matrix approach Section 6.2 for processing a graph into matrices. The last part of the code is an evaluation of the neural network and sharing values from the previously evaluated neural networks. For evaluation of the networks, we first run the feed-forward part with storing the results, which are then used in the backpropagation part of the evaluation. To keep the zero-valued parts of the weights not changing, we need an update matrix, which is created by multiplying input vector with delta vector, to be element-wise multiplied with the matrix representing the edges.

```
Data: list of small graphs; input data in small graphs; output data in small graphs;
      List of shared neurons
[1] MyProcedure maxLayer=maximum(graphs.maxLayers);
neuronLayers=;
for graph-> graphs do
    inputLayer=inputLayerNodes-> graph for node->inputLayer do
        neuronLayers[0].add(node);
    end
end
for i=1 to maxLayers do
    for graph-> graphs do
        for node->graph.layer[i] do
            if sharedNeurons.conatins(node) then
                sharedNeuronNumber=0;
                for node->neuronLayers[i do
                    sharedNeuronNumber++;
                end
                neuronLayers[i].add(sharedNeuronNumber).addNewEdges(node.edges);
            else
                neuronLayers[i].add(node);
            end
        end
    end
end
totalInput=neuronLayers[0].length;
totalOutput=neuronLayers[maxLayer].length;
inputData= outputData= inputsize=0;
outputSize=0;
for graph->graphs do
    for data->graph.input do
        inputData.add(zerropadd(data,inputSize,totalInput))
    end
    for data->graph.output do
        outputData.add(zerropadd(data,OutputputSize,totalOutput))
    end
    inputSize+=graph.input[0].length;
    outputSize+=graph.output[maxLayers].length;
end
```

Algorithm 3: An algorithm for creating one big neural network from the small neural networks

5.4.2 Pytorch

Pytorch [27] is a framework currently developed by Facebook. It's dynamic neural network framework fully written in python using torch as a core. The main focus of this framework is on the easy implementation of changes. Because it is fully written in python, its implementation is easy and it works well with native python code and even can use different libraries such as numpy or scipy. Three parts of the computation graph are init, forward and backward pass, enabling to change dynamically the structure of the neural network. Having the forward part makes it easier to use recurrent neural networks. Another great feature is that to change computation from the cpu to gpu, it is very easy by setting the model, the variables and layers onto the gpu by adding "*.cuda*()" to the existing code. The downside is that it uses prepared modules for the auto-gradient computation, so creating a new one is very problematic. Those facts make it perfect to be used in the graph approach, but the framework is difficult to use for the matrix approach. Coder has to define a new class of layers, which takes in the initialization matrices, in which the structure of the computation graph is encoded. The class needs also the backward part for the auto-gradient feature. The next part is to prepare the input for each of the matrices in the structure. The positive side of this framework is the easy switch from using cpu to gpu.

5.4.3 Tensorflow

Tensorflow [9] is a deep learning neural framework focusing on the static computation graph. The focus of the implementation is to deliver the most optimized a neural network. For this purpose, it has many blocks and units programmed to help the user use the correct features. The main advantage lies in using tensors and the ability to inscribe everything from scratch, without using any blocks. This code will be optimized no matter the size and way of writing it down. The cost for this optimization is the time needed to optimize any added tensors to the computation graph.

Tensorflow uses static data flow graph for evaluating the neural network. This allows the tensorflow to optimize the structure of the data-flow graph to perform better than the data-flow proposed by the user. The tensorflow [28] is used to see the limit of computation achievable on the generated graphs. Because it is static deep learning framework, it is impossible to change the structure or share the values of neurons without rebuilding the graph each time is shared weight or changed the structure. The only thing is to reuse variables and called it sharing. This can be done pretty easily. The tensorflow is used to see the limit in optimization using cpu on a single example. The test will only show how good can the neural networks be optimized automatically.

5.4.3.1 Tensorflow fold

Tensorflow fold [29] is a special library created for tensorflow to allow the use of dynamic data. The two main things added in this library are *while* cycle and *concat*. The special thing about this neural network is that it can change inner sizes of the layers, which is useful for the computation of sequential data with different structures. The changing of inner structure is meant as the structure can enlarge or reduce to fit the size of the incoming data. However, the change of structure as changing the number of layers or adding and removing

layers is not included. This could be easily applied to the templating approach, the problem is sharing weights and that the neural network has to be static in terms of structure. The only way how to use this is to write such neural network which could evaluate whole templates as inputs. This is a very hard task, which doesn't have a reasonable probability of success. For this reason, we didn't try to use this as one of the approaches. The library also stopped being developed after few months [2]. In figure 5.1 it is shown a schematics of how the fold works.

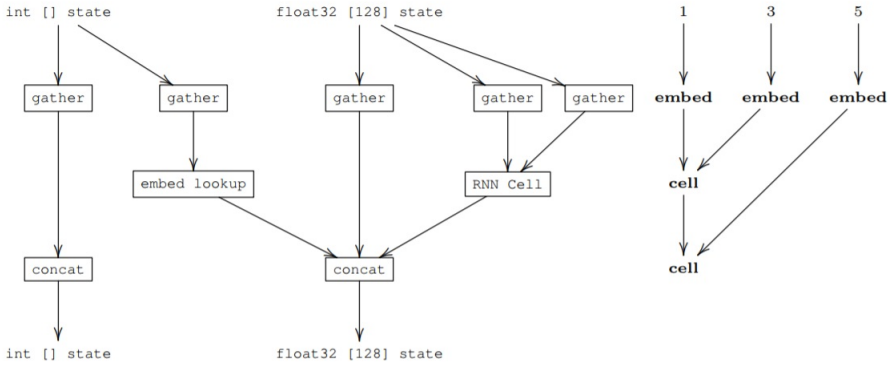


Figure 5.1: Tensorflow fold [2]

5.4.4 Dynet

DyNet [10] is deep dynamic neural network framework specializing in a dynamic declaration strategy. The transparent computation graph construction allows for fast evaluation of the neural network. This is achieved by using procedural code to compute the network output. This allows the user to dynamically change the network structure for each input. This allows for huge freedom for the user to build their own neural networks in an idiomatic way. This means we first declare model and then using the model we add variables to the model. The next thing is to define the computation graph. The user can define different computation graphs using them as functions. The special attribute of the dynet is sparse updates. Sparse updates in context of Dynet means that only weights that are non zero are applied during backpropagation.

Chapter 6

Experiments

The evaluation metrics of testing of the approaches are primarily a time of evaluation, to neglect the difference in each run we use summed time for 100 iterations of training on the graph. The accuracy of those neural networks is not shown, because we use randomly generated graphs with randomly generated data. The accuracy is only to show if the back-propagation works correctly. The metrics we are being interested in are primarily a time to create a table and run 100 iterations of training. This two has the best correlation between each other. They will allow us to observe the difference different density.

The experiments were conducted on a notebook running Ubuntu 16.04 with Intel core i7 6700 HQ cpu 2.6 GHz, Nvidia gtx970m graphic card and 8gb ram ddr4. For creating the implementation of deep learning frameworks is used python 3.6, pytorch 0.4, tensorflow 1.8 and dynet 2.0.

6.1 Graph and data generation

For the generic experiments, we created a graph generator, which generates computation graphs based on selecting a number of neurons for input, layers, a number of nodes and probability for edge to be in the graph. We first generate the layers with the nodes.

1. distribute uniformly nodes into layers
2. go through layers and randomly assign inputs according to a defined density
3. add jumping neurons take them at random from the 2nd to the last layer
4. check if the nodes in layers other than the output have at least one output edge, if they don't have at least one add an output edge to a randomly selected neuron from the layer directly above them

Then we use random distribution to determine how many edges passes to the final graph. The problem of using uniform distribution is that it usually creates a big chunk of neurons at the first layers and leaves rest with just a small number. That's why it was decided to use random distribution over set layers instead. This change ensured the distribution of neurons to be uniform for all layers. Using uniform distribution is to ensure to minimize the overall

number of edges. During the random choice of the edges we can't guarantee that all neurons will have at least one output, this holds especially in the case of sparse graphs. Thus we take the inner neurons without outputs and connect them to the next layer, where we choose the output neurons randomly. This addition makes the graph of the created neural network to be connected. The only nodes having no following output neurons are the ones in the last layer. This makes it easy to use in the matrix approach, where the last matrix always produces the output of the whole neural network. [30]

For the purpose of testing, we generated 147 generated graphs. Graphs were created with increasing number of nodes from 50 to 2450. The graphs have three different settings of the density. The densest graphs have around 90% density, next graphs have around 55% density and are called normal, the last have 25% density and are representation for sparse graphs. The setting of these percentages of density was determined, by the fact that the really dense but not fully connected neural network would have around 90% and that the graph using only every 4th edge is considered sparse, but still has a healthy number of inputs for the neurons. The normal density is near the average of dense and sparse densities.

6.1.1 Loading data

We used in python networkx library [26] for working with graphs this includes loading them and using them as storage for the attributes of the nodes. The loading time is directly correlated to the number of edges, this is demonstrated in Figure 6.1. For the custom framework, we write our own parser of the graphs. To load and save randomly created graphs of neural network we decided to use GML [31].

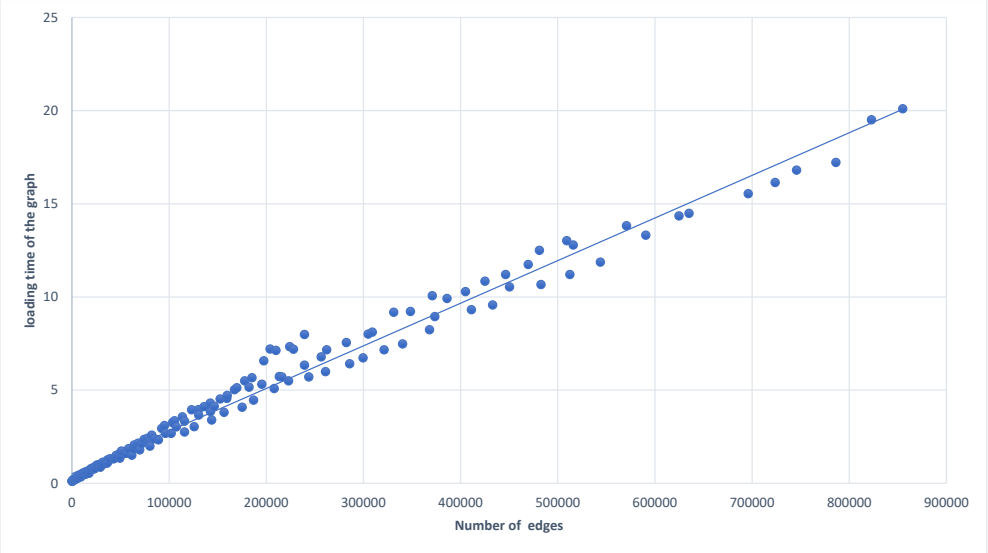


Figure 6.1: Graph showing correlation between loading time of the graph and number of edges inside the graph

6.2 Testing the matrix approach

This section talks about the matrix approach. The implementation of matrix approach can have two parts, using either dense or sparse matrix. The problem of the sparse matrix is that it is not implemented in most of deep learning frameworks. This approach is used nowadays in many fully-connected layers of the neural networks. This means it is obvious that it is one of the best possible representation of the neural network. The reason behind this is that there are many libraries dealing with the problem of matrix multiplication, which can increase the performance of this particular approach. In this section, we test only representation of one neural network as sharing values in the dynamic framework is easy and take much less time than the actual evaluation of the neural network.

6.2.1 Dense matrix testing

In this subsection, we tested implementations of the matrix approach on three different density settings. All the tests are done on cpu. This test is meant to showcases training only one neural network, without sharing neuron values. We compare each of the implementations on three sets of the graphs showing the effectiveness on the different settings of the graphs. In the next three graphs, we can see the comparison between the implementations. We can see small differences in the frameworks.

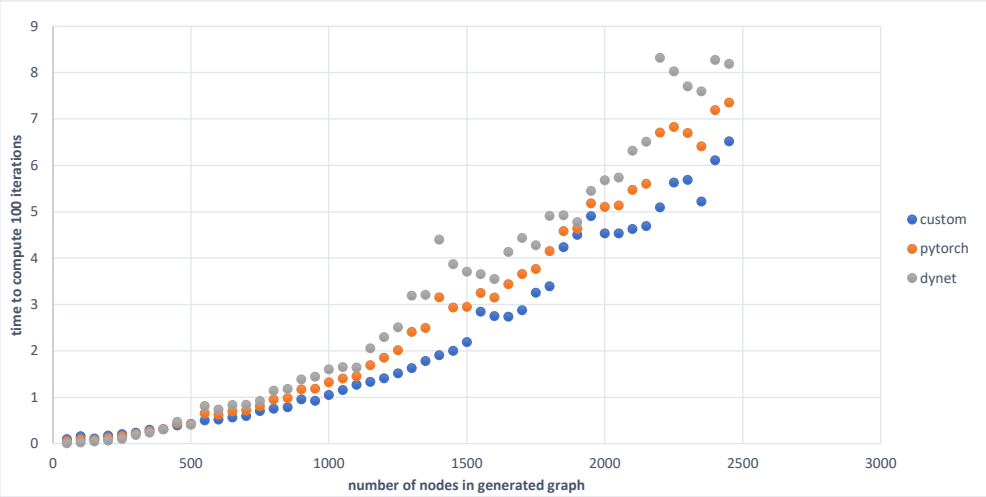


Figure 6.2: Graph showing different implementations tested on dense graphs

The difference showcased between the dense graphs and the normal graphs are explained by the density changing the overall number of edges in the graph.

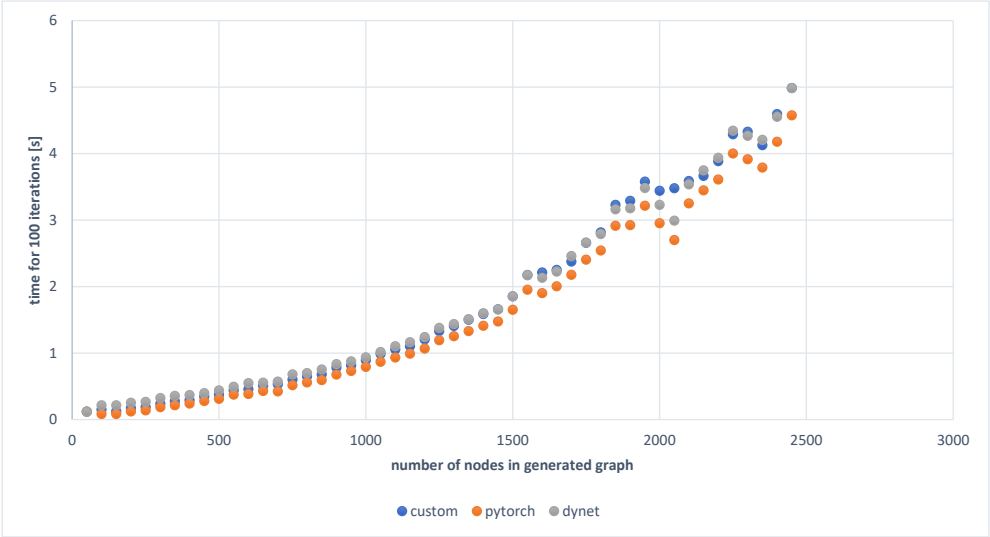


Figure 6.3: Graph showing different implementations tested on normal graphs

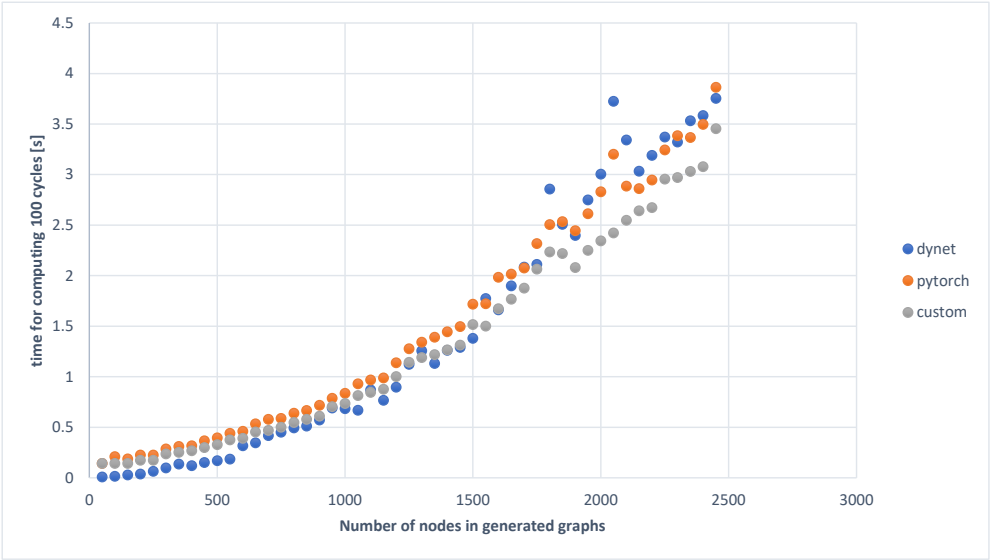


Figure 6.4: Graph showing different implementations tested on sparse graphs

6.2.2 Cpu vs Gpu

We know that gpu is nowadays used for acceleration of computing speed of neural networks. This holds for most of the neural network architectures, the only problem is small simple neural networks. In this subsection, we will discuss the problem of choosing when to use neural network trained on gpu or on cpu. The best ability of gpu is a fast multiplication of large matrices using the SIMD [32], where we multiply the vectors with whole matrices,

where the weakest point is the time needed to send the data to the gpu.

Figure 6.5: Pytorch using matrix approach on gpu

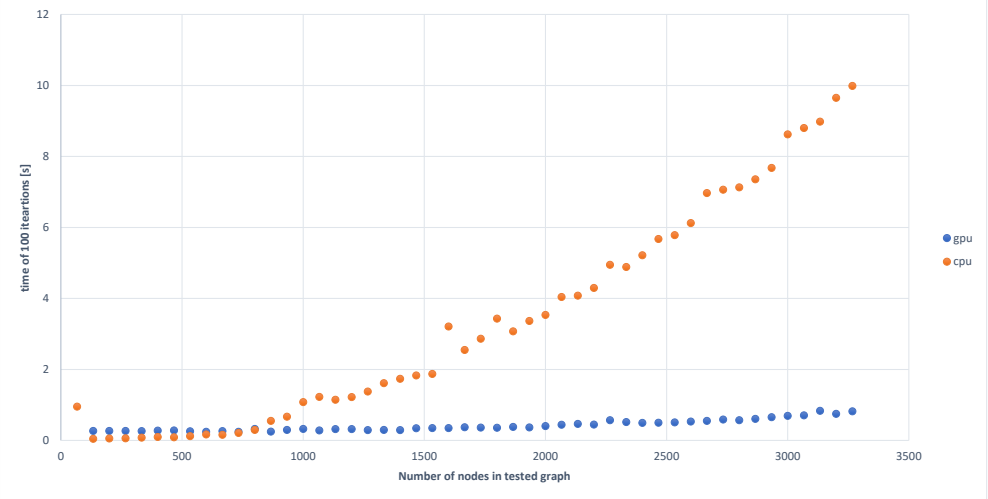


Figure 6.6: Comparison between cpu and gpu using tensorflow

The main difference is in that the gpu needs some time to transfer the results to gpu and from gpu, while cpu doesn't. The transferring creates a constant overhead which can't be surpassed, making the neural network having high minimum time to compute, while the cpu time depends purely on the size of the neural network. The difference can be seen in the figure 6.6. The difference is same for the other implementations. This means that while the cpu has a lower overall time necessary to initialize and doesn't need to transfer the data, it has problems maintaining the overall time while training, which rises much faster than the gpu one. This implies that is better to use gpu for larger neural networks and keep cpu

for small ones. There are multiple ways how to increase the speed of the computations on Intel processor. Intel processors have instruction set SSE [33], which is a type of SIMD. This significantly speeds up the computation of the matrix multiplication. Another thing is that the gpu is getting slower with increasing the number of layers compared to the number of nodes. The gpu is viable only for the matrix approach because in the graph approach we use many separate operations, which implies many computations slowing the gpu down.

In the figure 6.6 we can see the small amount of time increase with respect to the number of nodes, this phenomenon is caused by the fact, that gpu uses primarily SIMD instructions allowing for faster matrix multiplications, which is the main source of computation. Its downside is that it needs the same time to create a handle and send data to the gpu-card making it painfully slow on small data compared to cpu. In the subsection 6.2.1 you can see the graph showing the dependency of time on the number of nodes used by cpu acceleration. On first notice, we can see this graph resembles more exponential function rather than any other. Compared to the gpu it has more steep increase, while it significantly beats the gpu on small graphs, it can't compete with the gpu on the rest. The maximum time for gpu is around 0.9 seconds and for cpu, it is 12 seconds, that's the difference characterizing how well can gpu handle the increase in data size.

6.2.3 Sparse matrix testing

The only currently available implementation of the sparse matrix approach is by using eigen and coding the implementation by yourself. The deep learning frameworks don't have coded the sparse matrices into its blocks. The pytorch have sparse tensor [34] but not all of the function works, but more importantly it doesn't work with autogradient, this means we can't use sparse tensors as weights, because they are not seen as variables for the autogradient algorithm. This results in them not training during training. Dynet has sparse vectors [35], but only in c++ version and can be used only as an input for the neural networks.

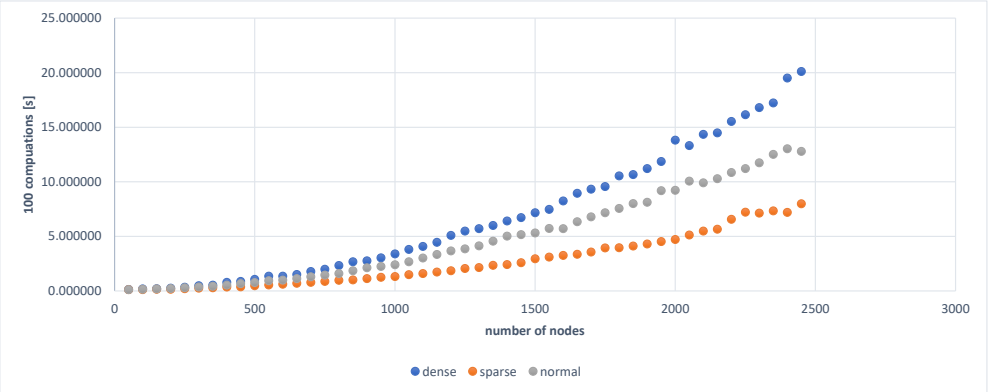


Figure 6.7: Matrix approach with sparse matrices

The graph in the Figure 6.7 discuss how efficient is the sparse solution in the test set. The graph shows the expected thing that the computation times increases exponentially to

the number of nodes in the graph. This fact doesn't change with the overall density of the graph, which only slightly slows the exponentially growing computation time. This effect is caused by the fact that the number of edges doesn't increase linearly compared to the number of nodes in the graph. The number of edges is defined by the structure of the neural network and by the number of nodes in the inner layers because the maximum number of edges is done by summing the sizes of layers near each other. So the upper limit to growth is a second power of the number of nodes in a graph.

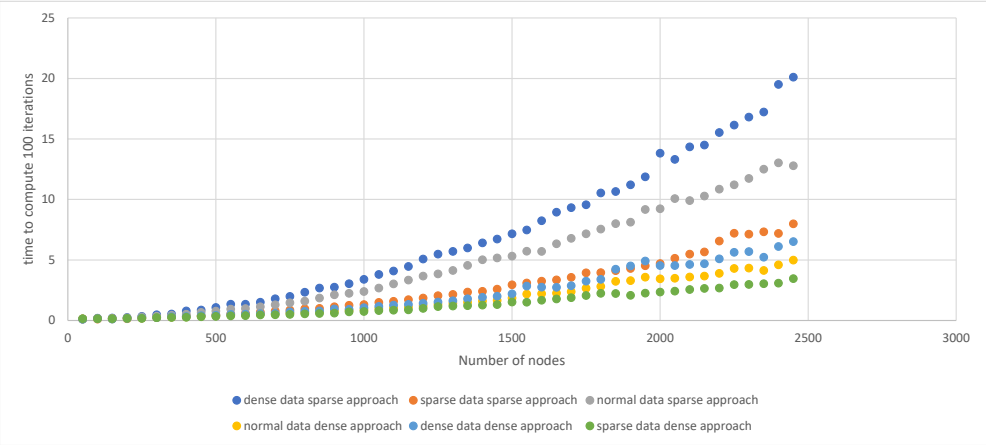


Figure 6.8: Comparisons of custom framework [3]

The figure 6.8 compares the efficiency of our custom solution, one using dense matrices and the other one using sparse matrices. The dense means that it uses normal matrices in the code and runs on the dense graphs of the set. The time doesn't change much with respect to the density of the graphs. For the comparison, we use the code with sparse matrices and only the sparse part of the set. We can see that they are almost identical, the biggest problem of the sparse matrices is updating the weights during backpropagation, compared to the classic matrices. This makes usage of the sparse matrices in the form of neural network less efficient than the classic one. This result is unintuitive because the overall number of operation should be smaller, but multiplying the vector input with the output vector creates a dense matrix, which is then multiplied element-wise using the matrix representing the structure of the neural network. After this, the update is done by adding two sparse matrices, during this operation, it is necessary to check if the addition didn't bring some new members.

6.3 Testing the graph approach

In this subsection it is shown how efficiently the different neural frameworks deal with the graph approach. On the figure 6.11 the is the graph of the pytorch using graph approach. It is obvious that it didn't work too well. The time needed to compute the independent neurons is alarming. This shows that pytorch doesn't optimize as well as tensorflow. The users using pytorch must do all the optimization by themselves.

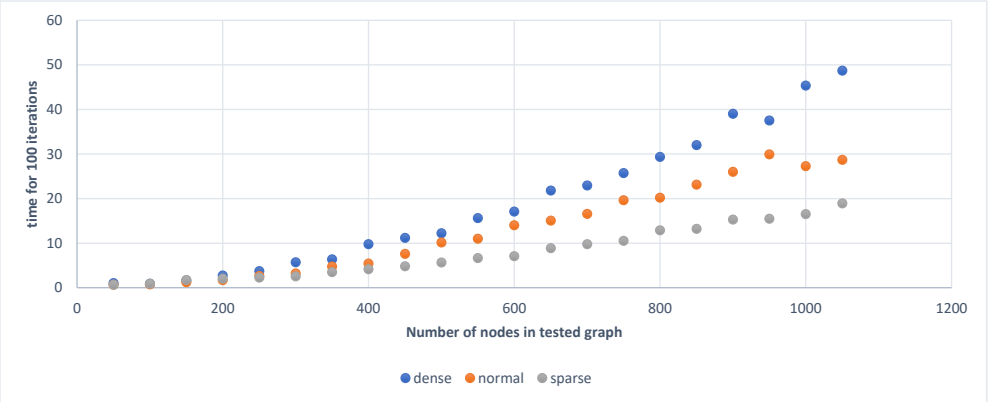


Figure 6.9: Pytorch graph approach

Dynet as shown at figure 6.10 seems to deals best with the graph approach, surprisingly getting superb times, even compared to the matrix approach. This fact is confirmed by the fact that the other implementation couldn't compute the 2000 node graph 100 times in less than 30 seconds, which lead to their end. This can be reasoned through its direct approach to computation. The only optimization of the neural network is done by the user of the framework.

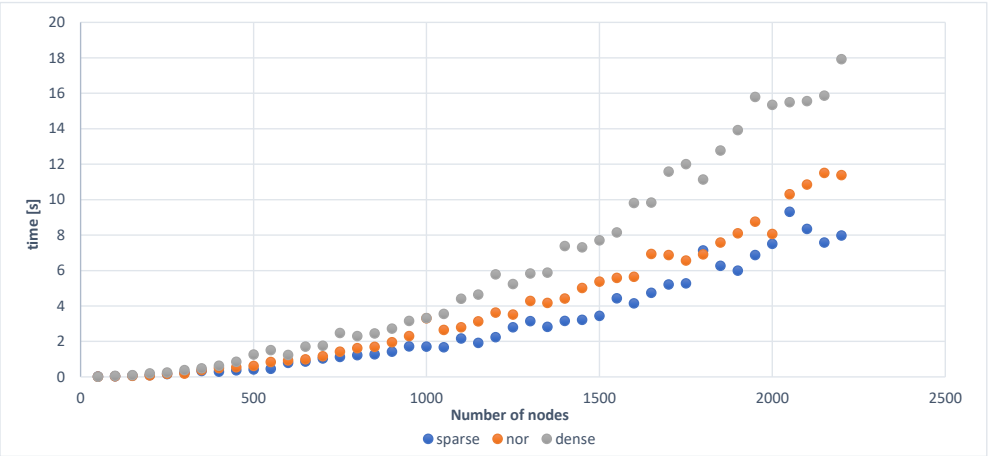


Figure 6.10: Graph approach using dynet

The figure 6.11 shows the tensorflow using the graph approach. The graph approach is surprisingly fast compared to the other frameworks using this approach. The tensorflow optimizes all of the layers into one computation graph and creates matrices out of the neurons the same way as the matrix approach does. We can see that tensorflow has a good time for the computation but is surprisingly slow in the creation of the computation graph. This means it has inner algorithms to compute and optimize the computation to be fast. The problem of tensorflow is that it is hard to share weights and values through multiple graphs making it not suitable. The other problem is that optimizer for calculation is probably really

universal so it will take increasingly more time to create the graph, which negates the fast computation times compared to the other frameworks.

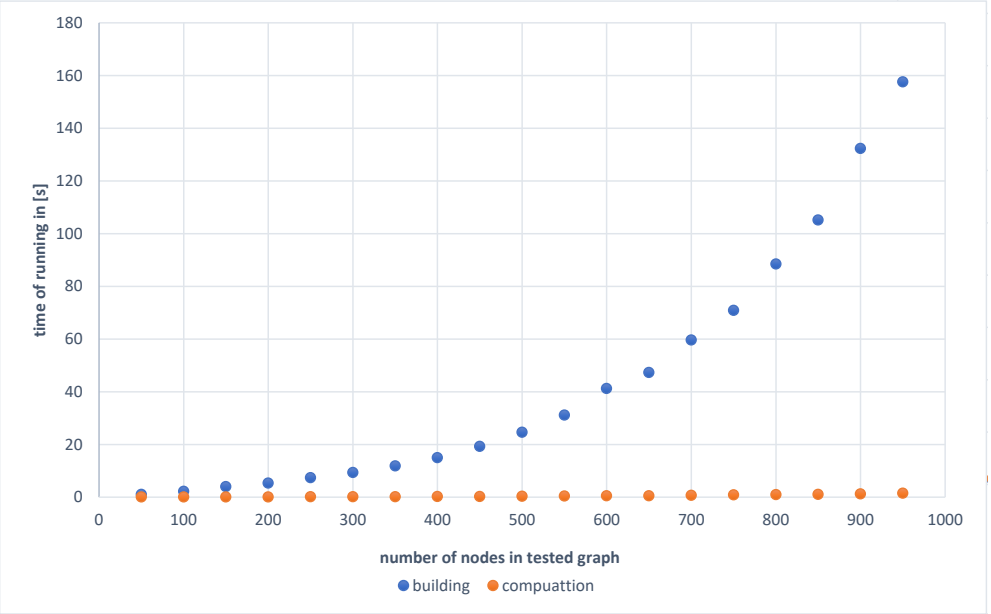


Figure 6.11: Tensorflow graph showcasing the stagering increase in time to build compared to time to execute one hundred iterations

6.4 Parameter sharing

The experiments before this don't use sharing variables only use one graph as input. The real data used for testing the problem of shared values are graph created from molecules from the mutagenesis dataset [4]. The attributes of those graphs are that they have only one output variable and are quite dense with average density near 78%. In this setting, we need to run either one big graph 186 times or 186 smaller ones. The biggest difference is in the size. Both of the approaches take about the same memory, with the multiple graphs taking more by a few percents. The time spent sharing values is always smaller than the time for the evaluation of a neural network. The testing shows that using small graphs are more effective in most parts, because of the flexibility of evaluating a small part of the neural network. The other finding concerns that with increasing number of neural networks the efficiency of the one big neural network compared to the approach of many small ones are decreasing. Meaning it is preferable to use many small neural networks compared to the one enormous graph. On the other hand for a small number of graphs is ideal to use only one big computation graph. This is because the amount of unused weights dramatically increases the computation time of the neural network.

	custom framework sparse	custom framework dense	pytorch	dynet
time for one run	2.99925	1.116	1.187176	1.099453

Table 6.1: Sharing variables on the mutagenesis dataset [4]

6.5 Discussion

In the experiment, we show that the more potent representation is the matrix one. We show that it can be used for relatively big data using gpu acceleration. We proposed two solutions programmed in c++ using eigen. The experiments show that the sparse matrix representation using eigen [12] is less optimized in comparison to the dense matrix. The sparse matrices aren't available to pytorch or dynet, this is explained in section 6.2.3. Both of them have sparse variables in experimental phases. This means in the near future we can expect to use sparse matrix declarations of the neural networks.

The graph approach was outclassed in comparison to the matrix approach. Probably the main reason behind this epic failure was the fact the deltas are counted for the whole layer, allowing faster computation of the backpropagation. This hold stronger for bigger neural network. This can be observable form the result of the experiments in section 6.3 which we can see that computation time compared to the number of nodes increased much more than for the matrix approach. The one plus of the graph approach is that it doesn't care about number of layers, while the matrix approach is restricted by using more layers than necessary.

Most important observations are that the matrix approach can work well using both cpu and gpu, where the deciding factor between which to use is the average number of nodes. The experiments shows that the optimal switching value from cpu to gpu is around 800 nodes in the Figure 6.6.

The testing of parameter sharing shows that the limit to the number of graphs simulated at one point is problematic for memory consumption. It is required to hold all the graphs and input data for each of the graphs and also to hold the table for sharing values indicators. The problem of frameworks is to deal with a large number of coexisting variables at once, while the backpropagation update only small part of the variables. This means that the advantage of the custom solution is to have predefined graphs and updating the only one graph part at the time. The main difference between having multiple small graphs or combining them into one is the number of sharing variables between them. The increasing number of shared variables seem to tip the scales for the one big graph and running only parts. By running only parts is meant to zero-padd all the unused inputs and to use backpropagation according to the output of each graph. The other approach have all the parameters in the memory and build a computation graph on fly. It depends how much the overall structures of the neural network are interconnected for determining the better solution.

Chapter 7

Conclusions

The thesis was about the integration of the deep learning and relational learning, with a particular focus on a general approach called templating. The templating approach creates templates for an unfolding of dynamically structured neural networks out of the relational data. The big problem and struggle with this approach are how to efficiently implement and train the varying neural networks. In this thesis, we proposed two approaches. The "neuron" approach is based on computing individual neurons. The "matrix" approach is about precomputing matrices, representing the layers of the neural network weights, and to use these matrices for evaluation of the neural network.

The proposed approaches were implemented in deep learning frameworks as well as with a custom solution. The "neuron" approach was tested on generated graphs with varying density and number of graph components. Changing the density shows the correlation between the number of edges and computation time. The testing of the number of nodes in a graph shows that the time scales with the second power to the number of nodes in the graph. The tests show varying efficiency across existing frameworks, with pytorch being the slowest one and dynet the fastest one when using cpu. The problem of tensorflow was not computation speed but the time it takes for a building of the computation graph. The graph approach was not tested on the gpu because it requires many small branching computations rather than batch processing. The gpu is slow to load data from the cpu but scales very well with the amount of data sent to it.

The "matrix" approach seems to be better than the "neuron" by a large margin. Meaning the only approach considered using is "matrix" approach. In subsection 6.2.2 we explained that by using gpu we can apply this method to bigger neural networks. Changing the size of the neural networks allows for using more relational rules and more training data. This improvement should allow templating to be used on a more wide range of problems and to be used on problems with higher complexity.

If we had more time we could write own library for the sparse matrices which will probably make them faster in comparison to classic matrices.

Bibliography

[1] Wikimedia Commons. Colored neural network, 2018.

[2] tensorflow fold github. `https://github.com/tensorflow/fold`, Oct 2017.

[3] Marian Briedon. Custom framework. `https://github.com/Briedon/NeuralNetwork`, March 2018.

[4] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity. *Journal of medicinal chemistry*, 34(2):786–797, 1991.

[5] Luc De Raedt. *Logical and relational learning*. Springer Science & Business Media, 2008.

[6] Lise Getoor and Ben Taskar. *Introduction to statistical relational learning*. MIT press, 2007.

[7] Adam Santoro, David Raposo, David GT Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. A simple neural network module for relational reasoning. *arXiv preprint arXiv:1706.01427*, 2017.

[8] [1609.04747] an overview of gradient descent optimization algorithms. `https://arxiv.org/abs/1609.04747`. (Accessed on 04/30/2018).

[9] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[10] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[11] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[12] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[13] u39kun. u39kun/deep-learning-benchmark, Feb 2018.

[14] Robert J Schalkoff. *Artificial neural networks*, volume 1. McGraw-Hill New York, 1997.

[15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[16] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[17] Bekir Karlik and A Vehbi Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122, 2011.

[18] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[19] George Bebis and Michael Georgiopoulos. Feed-forward neural networks. *IEEE Potentials*, 13(4):27–31, 1994.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[21] Pedro HO Pinheiro and Ronan Collobert. Recurrent convolutional neural networks for scene labeling. In *31st International Conference on Machine Learning (ICML)*, number EPFL-CONF-199822, 2014.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[23] Training an artificial neural network - intro | solver. `https://www.solver.com/training-artificial-neural-network-intro`. (Accessed on 04/30/2018).

[24] Brian Dolhansky. Artificial neural networks: Mathematics of backpropagation (part 4).

[25] Gustav Sourek, Vojtech Aschenbrenner, Filip Zelezny, Steven Schockaert, and Ondrej Kuzelka. Lifted relational neural networks: Efficient learning of latent relational structures. *Journal of Artificial Intelligence Research*, 62:69–100, 2018.

[26] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[27] Nikhil Ketkar. Introduction to pytorch. In *Deep Learning with Python*, pages 195–208. Springer, 2017.

[28] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[29] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017.

[30] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.", 2012.

[31] Michael Himsolt. Gml: A portable graph file format. *Html page under http://www. fmi. uni-passau. de/graphlet/gml/gml-tr. html, Universität Passau*, 1997.

[32] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, page 4. Citeseer, 2011.

[33] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel avx: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 19:20, 2008.

[34] Pytorch. Sparse autograd. `https://github.com/pytorch/pytorch/issues/2389`, 2018.

[35] Dynet. Sparse autograd. `http://dynet.readthedocs.io/en/latest/operations.html`, 2018.

Appendix A

Contents of the CD

Cd contains:

- graphs of mutagenesis in the gml format
- python sources
- c++ folder with custom solution
- thesis.pdf