CHALMERS
UNIVERSITY OF TECHNOLOGY

Deep Learning for Drug Discovery

Property Prediction with Neural Networks on Raw Molecular Graphs

Master's thesis in Complex Adaptive Systems

EDVARD LINDELÖF

Department of Biology and Biological Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis

Deep Learning for Drug Discovery

Property Prediction with Neural Networks on Raw Molecular Graphs

EDVARD LINDELÖF

Department of Biology and Biological Engineering
Division of Systems and Synthetic Biology
Chalmers University of Technology
Gothenburg, Sweden 2019

Deep Learning for Drug Discovery
Property Prediction with Neural Networks on Raw Molecular Graphs
EDVARD LINDELÖF

Collaborator:	Michael Withnall, AstraZeneca
Supervisor:	Hongming Chen, AstraZeneca
	Ola Engkvist, AstraZeneca
Examiner:	Christer Larsson, Department of Biology and Biological Engineering

Master's Thesis
Department of Biology and Biological Engineering
Division of Systems and Synthetic Biology
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in $\LaTeX$
Gothenburg, Sweden 2019

Deep Learning for Drug Discovery
Property Prediction with Neural Networks on Raw Molecular Graphs
EDVARD LINDELÖF
Department of Biology and Biological Engineering
Chalmers University of Technology

Abstract

The lengthy and expensive process of developing new medicines is a driving force in the development of machine learning on molecules. Classical approaches involve extensive work to select the right chemical descriptors to use as input data. The scope of this thesis is neural network architectures learning directly on raw molecular graphs, thereby eliminating the feature engineering step. The starting point of experimentation is a reimplementation of the previously proposed message passing neural networks framework for learning on graphs, analogous to convolutional neural networks in how it updates node hidden states through aggregation of neighbourhoods. Three modifications of models in this framework are proposed and evaluated: employment of a recently introduced activation function, a neighbourhood aggregation step involving weighted averaging and a message passing model incorporating hidden states in the graph's directed edges instead of its nodes. The resulting models are hyperparameter optimized using a parallelized variant of Bayesian optimization. Comparison to literature benchmarks for machine learning on molecules shows that the new models are competitive with state-of-the-art, outperforming it on some datasets.

Keywords: drug discovery, cheminformatics, graph neural networks, deep learning, Bayesian optimization.

Acknowledgements

This project would not have been possible without the continuous whiteboard discussions with Michael Withnall, whose input at some points saved me weeks of implementation headaches. Nor would it have been without the initial direction and useful insights from my supervisor Hongming Chen. My thanks go also to Ola Engkvist, for his fantastic work to curate a group of creative individuals and to provide an atmosphere of friendliness and enthusiasm. And to Christer Larsson, for being my supervisor and examiner at Chalmers.

I have thoroughly enjoyed my six months together with Ola's group at AstraZeneca. I want to thank Noé Sturm for the great dinner parties, Josep Arús for the Catalan wine, Oliver Laufkötter for the great grilling, Laurianne David for her observant kindness, and all other easy-going individuals I have been given the chance to meet during my stay. Finally, I want to thank Rocío Mercado for having been a fantastic company during weekend writing.

Edvard Lindelöf, Gothenburg, March 2019

Contents

1 Introduction

Drug discovery – the work carried out by pharmaceutical companies to find candidates for new medicines – is a slow and expensive process. The relevant chemical space is so large that only a tiny fraction of it can be tested through chemical experiments, even if fully automatized robotic systems are used. Molecules to consider for these screening experiments do not have to be chosen entirely at random, since there are relationships that can be identified between the structure of a molecule and its physiological effect. These structure-activity relationships are not easily analyzable, however, and there is no general method for modelling them. Machine learning approaches therefore become a natural part of the toolbox of medicinal chemists and cheminformaticians working to make the drug discovery process more effective. A challenge when applying classical machine learning algorithms is that they require data of a rigid format, e.g. fixed length vectors containing decimal numbers. When working with molecules, this limitation manifests itself in tedious feature engineering to select chemical descriptors. The emerging field of neural networks taking graphs as input data – in this document referred to as graph neural networks (GNNs) – presents great potential for overcoming the problem by learning directly on the compounds' raw graph representations.

1.1 The drug discovery process

The process leading to a new medicine can be seen as an iterative screening. The starting point is an extremely large set of compounds in chemical space. The set is then shrunk in many steps, where each one is designed to only let through compounds that possess desired properties.

In *The Practice of Medicinal Chemistry*, a breakdown of the whole process into intermediate steps is described [1]. First and foremost a biochemical *target* is chosen. This can be for example an enzyme, a protein or a receptor, thought to in some way be related to a medical condition one wishes to treat. The first objective of the drug discovery process is then to find chemical compounds that interact with said target. Chemical experiments are carried out to this end and compounds that appear to interact, according to some measured signal, are called *hits*. Merely knowing that a compound is a hit is very far from having a new medicine however. The next step is to test for activity again under various experimental conditions to see that it's robust. Compounds that stand this test are labelled *validated hits*. The subset of validated hits that are judged interesting with regards to parameters such as patentability and synthesizability are called *leads*. A lead needs to be further examined and perhaps slightly varied to assure that it possesses physico-chemical properties such that it can be used as a drug in practice, for example the human body must be able to absorb and distribute it. For a resulting *optimized lead* to become a *clinical candidate*, it must first be tested for toxicity in animal or cell models. The described steps are summarized in a flowchart in Figure 1.1.

When considering the low success rate and large cost of some steps of drug discovery, the potential for applying machine learning methods becomes clear. The hit screening phase, for example, is typically carried out at very large scale. *High-throughput screening* (HTS) with industrial robots may be utilized to conduct a vast number of experiments in a relatively short amount of time. A prediction algorithm can classify molecules as probable hits before any experiments have started, and thereby aid medicinal chemists in selecting a subset of chemical space to search through by HTS.

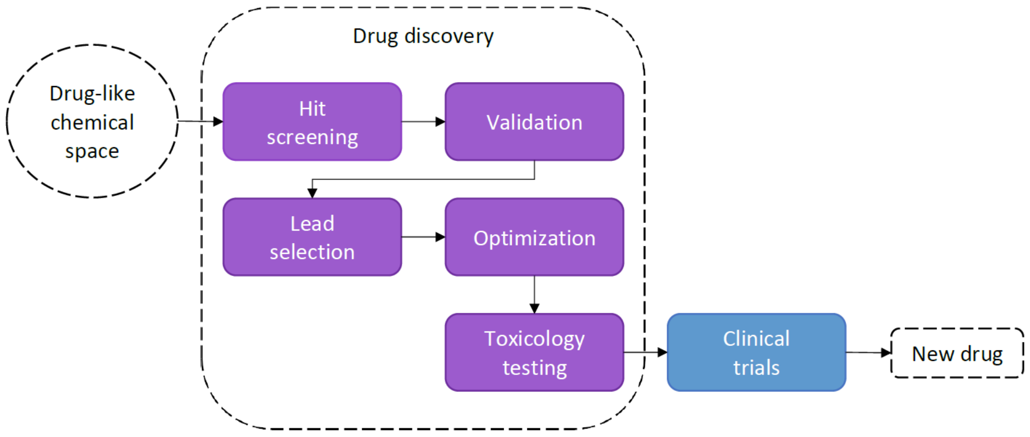

Figure 1.1: Overview of the drug development process.

1.2 Machine learning on molecules

The concept of using chemical information about a compound to predict their biological properties with a mathematical model is more than 50 years old [2]. The typical approach is called quantitative structure-activity relationship (QSAR) modelling. The use of such a model to generate a predicted value $\hat{y}$ of a biological property using a function f can be described as

$$\hat{y} = f(\text{chemical descriptors}) + \varepsilon \ ,$$

where ε is the error. Since f may be anything from a linear function to some more complicated model, the QSAR approach has naturally employed various classical machine learning algorithms such as support-vector machine (SVM) and random forest [3], methods that became refined enough for practical use in the 1990's [4] and early 2000's [5], respectively.

A challenging step in designing a QSAR model is the selection of chemical descriptors. Since it is not certain that an applied machine learning algorithm is able to handle high-dimensional input, the choice of descriptors should be small, but made carefully enough to still be able to characterize the relevant behaviour. There exists no generally effective method to make this selection. Approaches to systematize it to avoid relying on chemical experience and 'intuition' include stepwise-selection procedures and genetic algorithms, among other things [6].

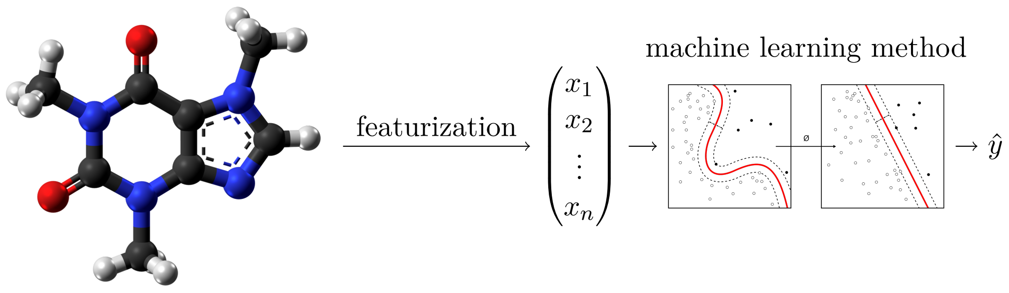

Figure 1.2: Steps needed to construct a prediction algorithm for molecules. Typically the molecule must somehow be featurized into a fixed length vector $(x_1, \cdots, x_n)^\intercal$ before it can be passed through a classical machine learning algorithm such as SVM, that generates a prediction $\hat{y}$. Molecule graphics and SVM illustration from Wikimedia Commons [7, 8].

Figure 1.2 illustrates the needed steps to construct a prediction algorithm to work on molecules. First, the molecule needs to somehow be transformed into a fixed length vector. Once a method of doing this has been chosen, any of a large range of popular machine learning methods can be applied but will not work well if the molecule-to-vector step was designed poorly. The methods that this work focuses on take the approach of inputting raw graph representations of molecules, with basic chemical information about atoms and bonds, into a GNN architecture. The GNN takes care of all steps from molecule to prediction.

Given the tediousness of the QSAR modelling mentioned above, the development of methods that do not need explicit feature engineering is well motivated. Since graphs are such a general data structure and since so much feature engineering gets avoided when applying graph neural networks, they can truly be considered instances of deep learning.

1.3 Objective

The purpose of this thesis work is to explore how deep learning methods for drug discovery, specifically property prediction algorithms taking molecular graphs as input data, can be improved. The approach entails replacement of all steps shown in Figure 1.2 with a neural network architecture whose building blocks are trained jointly end-to-end. It is a true instance of deep learning in that it generalizes and automates the molecule-to-feature-vector step and eliminates the need for application specific knowledge.

1.3.1 Delimitations

All implementation work of this project focuses on neural networks for graphs. As will be understood in Chapter 3, GNNs are a novel field with a large number of possible directions of advancement. Therefore, a specific class of GNNs known as message passing neural networks (MPNNs) is given major focus. Novel contributions are implemented and described in terms of extensions to and modifications of this framework.

Some non-graph-based machine learning methods are interesting to discuss for benchmarking purposes. This is done by referencing the literature, not by implementation.

The data used for training and evaluation of implemented models consists of publicly available datasets relevant to various steps of the drug discovery process. The implemented models can in principle be used on any molecular datasets, or in fact on any dataset consisting of pairs of graphs and output values. While there are some interesting available datasets related to, for example, quantum chemistry, anything that is not drug discovery related is considered out of scope. Furthermore, there is much to say about how to curate chemical data appropriately for machine learning, as well as what metrics to use for evaluations. Mainly for the sake of enabling proper benchmarking, most data aggregation is done in an identical manner as in closely related literature, as is the selection of all performance metrics used.

1.4 Thesis outline

An overview of the drug discovery process has been given and the motivation of deep learning models for molecules has become clear. The remainder of the document aims to build up an understanding of state-of-the-art GNNs and then describe the experimental work to improve them that constitutes this project.

Chapter 2 covers a range of neural network architectures and techniques, selected based on their relevancy to the class of implemented models. All the concepts are however well-known in the machine learning community. Depending on the reader's familiarity with neural networks and machine learning, a large part of it may be skipped. Chapter 3 contains a description of state-of-the-art GNNs, some useful terminology regarding them and how they relate to two earlier approaches – one from cheminformatics and one from image analysis. The proposed improvements to MPNNs that constitute the principal novelty of this work are described in Chapter 4. Aspects that need to be laid out clearly before training the models, such as data, programming frameworks and performance metrics, are covered in Chapter 5. A conceptually heavy part of it is on the Bayesian optimization used for hyperparameter optimization which, albeit not containing any parts that are scientifically novel, makes up an important component of this project engineering-wise. Chapters 6-8 describe a scientific process that demonstrates the performance improvement from implementing the new extensions: the experimental setup, a presentation of results and a discussion. The document is wrapped up with the concluding Chapter 9.

2

Supervised machine learning and neural networks

The main scope of this work is GNNs, a specific class of neural network architectures covered in depth in Chapter 3. The use of neural networks for prediction is a part of the larger machine learning field. A few of the aspects covered in this section are general to machine learning while the remainder are specific to neural networks. All included concepts are relevant either to understand how to design GNNs, how to train them, or how to tune them for increased performance.

2.1 Machine learning concepts

Machine learning is the study of algorithms that utilize data to carry out certain tasks. The subfield of machine learning concerned with datasets consisting of input/output pairs and whose objective is to learn how to predict the output of a certain input, is called supervised learning. The process of feeding the algorithm with input/output pairs in the hope that it learns the relevant patterns is called training. For some algorithms it is useful to feed the data more than once, each full traversal of data during training is then called an *epoch*. A notorious problem with trained algorithms is that despite fitting the training data well, the prediction performance may not generalize to new input. To be able to critically assess whether a model can generalize, the dataset is typically *split* into two parts, referred to as *training set* and *test set*. The test set is kept out of reach of the algorithm during training but is used afterwards to see how well the algorithm's predictions match the previously unseen test output. It is common to further divide the dataset by removing a part from the training set to use as *validation set*. The validation set may turn out useful for example when several models have been trained, and the best one is to be chosen. If this selection is made based on performance on the test set, the test set is in principle not a test set, so the selection is instead made using the validation set. In summary, the division into three is useful to be able to carry out the following scheme for training, choosing and evaluating models.

1. Train several models to the input/output pairs of the training set

2. Choose the trained model that has the best *validation score* (that best predicts the outputs of the validation set)

3. Assess generalized performance of the chosen model by computing the *test score* when predicting the outputs of the test set

Step 1 is typically a *hyperparameter optimization* – a search for good values of model parameters that affect performance but cannot be adjusted as part of training. A very common thing is that the *training score* of a model, the score achieved when predicting outputs of the to-the-model already known training set, is lower than the validation score. This is called *overfitting.* There exist certain *regularization* techniques that can be incorporated into models for the specific purpose of minimizing overfitting. These usually have parameters that are tuned in the first and second steps of the scheme above, rendering the training-validation-test split especially useful.

2.2 Neural networks

The term *neural network* is in a machine learning context used to refer to various parameterized functions with the important property of being differentiable with respect to each parameter. The differentiability assures that gradient-based methods may be used to attempt to train them to minimize a differentiable loss function given a set of input data. Different neural networks can be combined in quite flexible ways to form new, bigger ("deeper" or "wider") neural networks that are differentiable with respect to all the parameters of the smaller ones. For example, if $f : X \longrightarrow Z$ and $g : Z \longrightarrow Y$ are neural networks, $l : Y \longrightarrow \mathbb{R}$ a loss function and $x \in X$ some data, then differentiating the loss $l(g(f(x))$ with respect to the parameters of both f and g is simply a matter of applying the chain rule. The same generalizes to any number of nested neural networks and is then known as the *backpropagation* [9] principle. Because of the possibility to build larger architectures from smaller ones, it is often convenient to use terms like *layers* and *blocks* when discussing larger networks.

2.2.1 Feed forward neural network

The most appropriate architecture to give as a concrete example of a neural network is the fully connected feed forward network (FFNN), sometimes also known as the multilayer perceptron [10]. Each layer of this network consists of an affine transformation followed by application of a so called activation function. The layer maps a vector $\boldsymbol{z}_{n-1}$ to a new vector $\boldsymbol{z}_n$ according to

$$\boldsymbol{z}_n = \sigma(\boldsymbol{W}_n \boldsymbol{z}_{n-1} + \boldsymbol{b}_n) \ , \tag{2.1}$$

where $\boldsymbol{W}_n$ and $\boldsymbol{b}_n$ are a matrix and a vector parameterizing the layer, and σ is known as an activation function, that may in practice be chosen to be for example the hyperbolic tangent. N nested operations of this form constitutes an N-layer feed forward network. If applied to data $\boldsymbol{x}$ on its own, $\boldsymbol{z}_0 = \boldsymbol{x}$ is the input and $\boldsymbol{z}_N$ is the output, whose number of elements is decided by properly choosing dimensions of $\boldsymbol{W}_N$ and $\boldsymbol{b}_N$. $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_{N-1}$ are called hidden states, and the layers computing them hidden layers. A common way to illustrate a feed forward neural network is seen in Figure 2.1, here with one hidden layer.

It was shown in 1989-1991 [11, 12] that under mild assumptions on the activation function, a feed forward network with one hidden layer can approximate any reasonable function to arbitrary precision with a finite number of elements in $\boldsymbol{z}_1$. One may thus believe that further work to

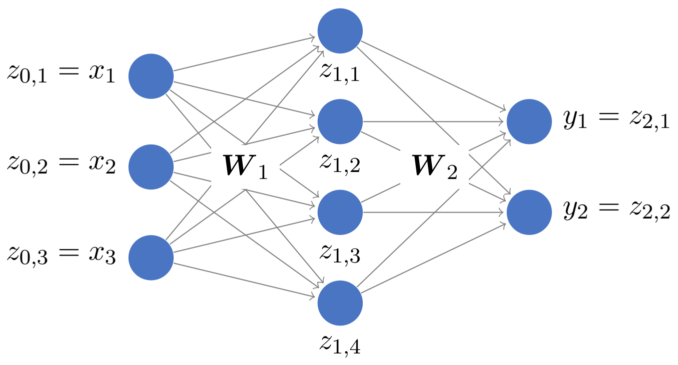

Figure 2.1: Common illustration of a feed forward neural network. This particular one has one hidden layer and takes the input $\boldsymbol{x} \in \mathbb{R}^3$ to generate the output $\boldsymbol{y} \in \mathbb{R}^2$.

create neural network architectures is unnecessary. However, the theorem merely states the existence of parameters $\boldsymbol{W}_1, \boldsymbol{b}_1, \ldots, \boldsymbol{W}_N, \boldsymbol{b}_N$ that give arbitrary precision, but nothing about how to find them. In the typical practical case, the optimal parameters are impossible to find because the optimization problem is too hard. The deep learning field however demonstrates that application-tailored neural network architectures, designed in ways heuristically thought to handle particular tasks well, are in practice possible to tune to greater performance.

2.2.2 Recurrent neural network

Recurrent neural networks [9] (RNNs) are particularly suitable for learning on data that is sequential in nature. Suppose the data is the sequence of vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$. To describe the idea of a recurrent neural network, two equations are necessary,

$$\begin{aligned} \boldsymbol{h}_t &= f_h(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) \\ \boldsymbol{y}_t &= f_o(\boldsymbol{h}_t) \ . \end{aligned}$$

The parameterized functions f_h and f_o may take on various forms. In a “classical” recurrent neural network, they may be set to something similar to the layer (2.1). The hidden state $\boldsymbol{h}_t$ may be thought of as a memory where information about the so-far-encountered inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{t-1}$ is stored. $\boldsymbol{h}_0$ may be instantiated to for example $\boldsymbol{0}$. Depending on the application, the output of the network may be taken as the whole sequence $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T$, or simply the last vector $\boldsymbol{y}_T$. A common way of illustrating a recurrent neural network is seen in Figure 2.2.

It has turned out that for good performance, a somewhat more engineered layer than (2.1) is highly useful [13, 14]. One such layer is the gated recurrent unit [15] (GRU), whose f_h is defined by the equations

$$\begin{aligned} \boldsymbol{r}_t &= \text{sigmoid}(\boldsymbol{W}_r \boldsymbol{x}_t + \boldsymbol{U}_r \boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \\ \tilde{\boldsymbol{h}}_t &= \tanh(\boldsymbol{W}_h \boldsymbol{x}_t + \boldsymbol{U}_h (\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \\ \boldsymbol{z}_t &= \text{sigmoid}(\boldsymbol{W}_z \boldsymbol{x}_t + \boldsymbol{U}_z \boldsymbol{h}_{t-1} + \boldsymbol{b}_z) \\ \boldsymbol{h}_t &= \boldsymbol{z}_t \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{z}_t) \odot \tilde{\boldsymbol{h}}_t \ , \end{aligned}$$

where $\boldsymbol{W}_r, \boldsymbol{W}_h, \boldsymbol{W}_z, \boldsymbol{U}_r, \boldsymbol{U}_h$ and $\boldsymbol{U}_z$ are parameter matrices, $\boldsymbol{b}_r, \boldsymbol{b}_h$ and $\boldsymbol{h}_z$ are parameter vectors, and $\odot$ denotes element-wise multiplication. To get an understanding of the intuitions that

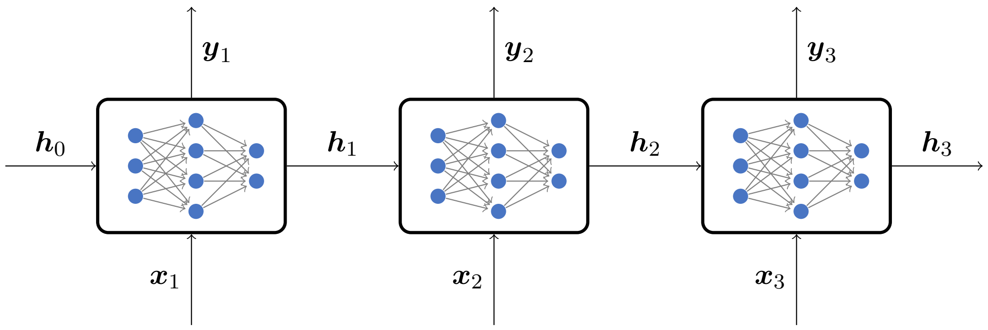

Figure 2.2: One way to illustrate the use of a recurrent neural network. Each rectangle represents application of the same neural network block. The block itself may take on various architectures.

lead to this procedure for updating the hidden state, consider one of its elements $h_{t,j}$ and examine the equations one by one. First, a *reset gate* $r_{t,j}$ is computed based on the latest input $\boldsymbol{x}_t$ and the current memory $\boldsymbol{h}_{t-1}$. Since $r_{t,j} \in (0,1)$ while $h_{t-1} \in (-1,1)$ (having been computed with the sigmoid function and the hyperbolic tangent function, respectively) the element-wise multiplication in the computation of the new candidate state $\tilde{\boldsymbol{h}}_t$ can be thought of as a means of letting the former turn off the impact of the latter. Aside from the element-wise multiplication, the computation of the candidate state is a simple affine transformation followed by an activation function, much like a normal feed forward layer. In the third and fourth equations, the *update gate* $z_{t,j} \in (0,1)$ is computed to decide how much influence the candidate state element $\tilde{h}_{t,j}$ should be allowed on the hidden state element $h_{t,j}$.

An important layer which is similar to GRU, and a predecessor of it, is the long-short term memory unit (LSTM). Only GRU is described in detail because it is built on similar concepts as LSTM but is slightly simpler. Empirical evaluation [16] shows that employing either a GRU or a LSTM vastly improves performance compared to a classical RNN, but which one is the better of the two is unclear. The networks implemented in this work may contain both GRUs and LSTM units as part of certain blocks. The impact of choosing one over the other is considered out of scope.

2.2.3 Activation functions

The choice of activation function can significantly impact performance of networks. The mentioned sigmoid and hyperbolic tangent alternatives are simple and have easily computable derivatives. In the last few years many alternatives have been proposed however. One that is important to mention as it is used in GNNs from literature discussed later in this document, is the rectified linear unit activation function [17]

$$\mathrm{ReLU}(x) = \begin{cases} x \text{ if } x > 0 \\ 0 \text{ otherwise} \end{cases} . \tag{2.2}$$

One of the ideas behind ReLU is to encourage a network to "turn off" some hidden units, yielding a sparser signal. The non-differentiability at 0 appears to not matter in practice.

Another important function that may be considered an activation function is the softmax σ :

$\mathbb{R}^K \longrightarrow \mathbb{R}^K$,

$$\sigma(\boldsymbol{z})_j = \frac{\exp(z_j)}{\sum_k \exp(z_k)} ,$$

which can be said to normalize the input vector in the sense of rescaling it so that its elements sum up to 1. It is therefore a natural choice for modelling probabilities. In this work it is mainly used in a set aggregation context to generate weights for doing weighted summation.

2.2.4 Training a neural network

In the common approach to finding the weights of a neural network that is used in this work, differentiability with respect to every parameter of the network is vital. By the backpropagation principle and careful making sure that every neural network block and layer is fully differentiable, the gradient with respect to the vector $\boldsymbol{w}_n$ of all network parameters can be computed. This can be used to update the weights iteratively according to some gradient descent-based method [18], of which the simplest form is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \nabla_{\boldsymbol{w}} f_t \ . \tag{2.3}$$

Simple gradient descent effectively takes steps in the direction where the differentiated function is the steepest, with a step length determined by the so-called learning rate η. Under some assumptions on the function to be optimized, for example involving convexity, the convergence to the global minimum of some gradient descent methods can be guaranteed. For all but the most trivial neural network architectures however, this assurance cannot be had. It has been empirically established that simple gradient descent is insufficient for reaching satisfactory performance, but that various extensions can work well. These extensions usually incorporate some form of stochasticity, typically by feeding the training input/output pairs in small randomly chosen *batches* [10]. The gradient descent then takes a step based on the gradient of the loss given a single batch of training data, rather than all of it.

A popular such alternative is the Adam optimizer [19], which is used in this work. This method replaces the simple gradient descent update rule with

$$\begin{aligned}
\boldsymbol{m}_t &= \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\nabla_{\boldsymbol{w}} f_t \\
\boldsymbol{v}_t &= \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)(\nabla_{\boldsymbol{w}} f_t)^2 \\
\hat{\boldsymbol{m}}_t &= \frac{\boldsymbol{m}_t}{1-\beta_1^t} \\
\hat{\boldsymbol{v}}_t &= \frac{\boldsymbol{v}_t}{1-\beta_2^t} \\
\boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \eta \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \varepsilon} ,
\end{aligned}$$

where all operations applied to the vectors are done elementwise. ε is a small constant just to avoid division by 0. $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$ are exponentially smoothed moving averages of the square of the gradient. $\hat{\boldsymbol{m}}_t$ and $\hat{\boldsymbol{v}}_t$ are versions corrected for the bias present in early iterations because of the initialization to 0. The last equation comprises a clever choice of effective step size. When the element of the gradient fluctuates over iterations, its average will be small while the average of its square will be large, and thus so the corresponding element in $\hat{\boldsymbol{m}}_t/\sqrt{\hat{\boldsymbol{v}}_t}$. The optimizer thus adaptively modifies the effective learning rate for each parameter between iterations. The hyperparameters $\beta_1, \beta_2 \in [0, 1)$ can be said to determine the time scale of the effective learning rate adaptation.

2.2.5 Regularization techniques

Since overfitting is a notorious problem for neural networks, the techniques developed to counteract it are numerous. The primary ones used in this work are weight decay and dropout. Of these two, the former is the simplest and most straight forward to implement [20]. Adding a simple term in the gradient descent rule is sufficient. In the simplest case, (2.3) becomes

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta(\nabla_{\boldsymbol{w}} f_t + \lambda \boldsymbol{w}_t) \ ,$$

where λ is a constant determining the severity of the decay. Applying weight decay to one neural network parameter w_i is equivalent to adding $\lambda \frac{w_i^2}{2}$ to the network's loss function, since $\frac{\partial}{\partial w_i}(\lambda \frac{w_i^2}{2}) = \lambda w_i$. Weight decay can therefore be understood as a penalty on network parameters, favoring smaller values on those that are not important, which sometimes leads to better generalizability.

Dropout is a more recent idea which compared to weight decay is less mathematically elegant but has turned out effective in practice [21]. The application of dropout at a given layer (of input units or hidden units) is simple to describe: let each unit have probability p of being set to 0. This is done during training only, and the units are rescaled with a factor $\frac{1}{1-p}$ to compensate the weakening of the signal. The idea behind dropout is to make neurons rely on robust and simple patterns rather than intricate ones involving co-adaptation to each other.

2.2.6 Techniques to deal with large depth

There are some challenges to training neural networks that are particularly troublesome for architectures that are deep (that have many hidden layers). Notably one may run into the problems of *vanishing or exploding gradients*. How the vanishing problem can appear is described in the following. Consider the computation of the derivative of the loss function with respect to a weight in an early layer of a deep feed forward neural network. Since this computation is done through backpropagation (iterative application of the chain rule) the derivative is a sum of products, each having among the factors one derivative per activation function in later layers. Common choices of activation functions, for example the hyperbolic tangent and sigmoid functions, have derivatives close to 0 for large inputs. The products then become even smaller and thus partial derivatives may vanish.

Various techniques can be employed to counteract these depth-related problems. For example, well-engineered RNNs like the GRU and the LSTM unit possess intrinsic properties that deal with it [15]. Another popular way of solving the problem is to use the ReLU activation function. The property of ReLU that is beneficial with respect to the vanishing gradient problem is that its derivative is 1 for all positive inputs.

An alternative that can be used regardless of chosen activation functions and building blocks is explicit normalization between layers. The technique is then to rescale hidden units in some intermediate step of the architecture, by subtracting with a mean and dividing by a standard deviation. The statistics may be computed per-element, using the values that element takes on for the different samples in a batch, and is then known as batch normalization [22]. Or they may be computed per-hidden state vector, in which case it's called layer normalization [23]. Assuring mean 0 and variance 1 in this way solves both the problem of vanishing and of exploding gradient.

This work uses a normalization scheme that entails using an activation that implicitly leads to normalized hidden units. This is a very recent idea and will be covered further in Chapter 4.

3

Previous work on graph neural networks

This chapter aims to give an overview of GNNs, detailed enough to enable good understanding of the new contributions presented the subsequent chapter. First the (molecular) graph data representation is described clearly, together with some corresponding definitions and notation. For historical curiosity but also for understanding, two techniques that may be viewed as predecessors to GNNs on molecules are then briefly mentioned before moving on to GNNs. The subclass of GNNs that will be the most thoroughly covered is the message passing neural networks (MPNN) framework.

3.1 Graphs – definitions

A *graph* is an ordered pair $G = (V, E)$ of a set V of *nodes* together with a set E of *edges*, which are pairs of elements of V. It is said that the node v is part of the graph G, or that $v \in G$, if $v \in V$. The *neighbourhood* of v is the set $N(v) = \{w \mid (w, v) \in E\}$. If the pairs in E are ordered, the graph is said to be *directed*, otherwise it is said to be *undirected*.

The representation of a molecule as an undirected graph is intuitive. Atoms are simply represented as nodes while chemical bonds between atoms are represented as edges. To be able to represent information about atoms and bonds, a graph comes together with two corresponding sets $\{\boldsymbol{n}_v \mid v \in V\}$ of node feature vectors and $\{\boldsymbol{e}_{vw} \mid (v, w) \in E\}$ of edge feature vectors. Throughout this document, the term "graph" is used to implicitly refer to these two sets, together with the graph itself. An illustration of the data structure is seen in Figure 3.1.

3.2 Two predecessors to graph neural networks

Convolutional neural networks (CNNs) are well known for their high performance in image classification. A simple convolutional layer (a layer that in CNN terminology has stride size 1, one input channel and one output channel) can be written

$$(f * g)[i, j] = \sum_{k=-K}^{K} \sum_{l=-K}^{K} f(i-k, j-l) g(k, l) \ , \tag{3.1}$$

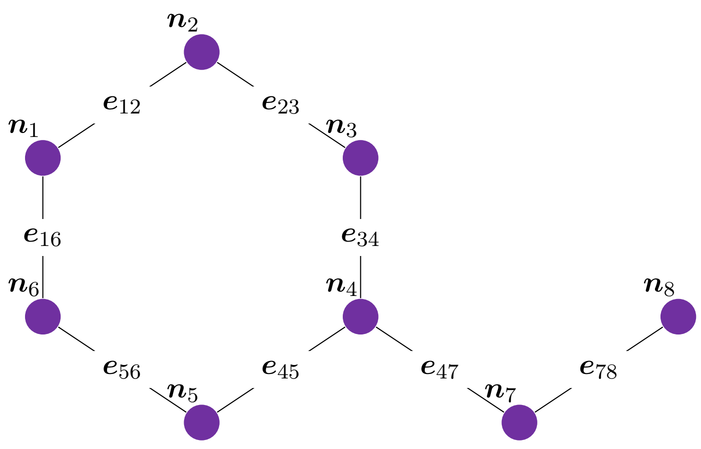

Figure 3.1: Illustration of the data structure used to describe a molecule: an undirected graph, a feature vector for each node and a feature vector for each edge.

where $f(i, j)$ is the image intensity of the pixel at (i, j). The matrix $(g(i, j))_{i,j=-K}^{K}$ is adjusted during training. $(f * g)[i, j]$ is a weighted sum of intensities of a neighbourhood of pixels. Since images are special cases of graphs, a natural question is whether (3.1) can be generalized. This turns out to require quite severe approximations. GNNs can be viewed as analogous to convolutional neural networks, however, in that they contain blocks that like (3.1) aggregates information from neighbourhoods.

A second technique that can be viewed as a predecessor to GNNs for molecules is that of extended-connectivity circular fingerprints [24] (ECFP). An outline of the algorithm, that turns a molecular graph into a fixed length “fingerprint” vector $\boldsymbol{f}$, is given in Figure 3.2. As can

```
1. $h_v^{(0)} \leftarrow g(\boldsymbol{n}_v)$ for all $v$ in $G$    (compute an integer representing each atom)
2. $s \leftarrow \{h_v^{(0)} \mid v \in G\}$
3. for $t = 0, \ldots, K$
     for each $v$ in $G$
       (a) $m \leftarrow \{h_v^{(t)} \mid v \in N(v)\}$    (take neighbourhood)
       (b) $h_v^{(t+1)} \leftarrow \text{hash}(m)$    (update atom integer based on it)
       (c) add $h_v^{(t+1)}$ to $s$
4. $f_i = 1$ if $s$ has an element $h$ such that $h \bmod L = i$, otherwise $f_i = 0$
```

Figure 3.2: Outline of how an ECFP vector $\boldsymbol{f}$ of length L is generated.

be seen, ECFP has a neighbourhood aggregation step. This makes the technique analogous to GNNs in the same way that CNNs are. An important difference is that the step is not learned during training. Instead it is engineered in a way thought to be generally useful for QSAR modelling.

3.3 Graph neural networks

An important paper in bringing *graph convolutions* into the context of machine learning on molecules is *Convolutional Networks on Graphs for Learning Molecular Fingerprints* [25]. The method is designed with ECFP as a starting point. The main difference is that no hash function is used, but that the neighbourhood aggregation step involves matrix multiplication of each neighbour with a learned weight matrix. The authors demonstrate that their model performs similarly to ECFP if the weights are large and random, and better than ECFP if the weights are then adjusted with training. *Semi-Supervised Classification with Graph Convolutional Networks* [26] proposes another locally applied matrix operation and describes more precisely in what sense it approximates a convolution. It has been pointed out [27] that the use of the term “convolution” to describe the layers of these papers is a harsh approximation. Analogous to convolutional layers of CNNs, they are applied locally to each node of the graph. They are however not true to the mathematical concept of convolution as an integral that can be approximated by a weighted sum. The summation of neighbours is not weighted.

Aside from the above-mentioned papers, there are several others [28] proposing various layers to be used in a similar fashion. It is only a fraction of them that focus on molecular data. Others may for example study large citation networks, where the task is typically to make one prediction per node rather than one for the whole graph. Ideas that have shown to be effective for node-level predictions can potentially be employed for graph-level prediction. An example of this is in how the present work takes inspiration from *Graph Attention Networks* [29], covered in Section 4.2. The commonality between all mentioned methods is that they use some form of operation to locally aggregate information from neighbourhoods of nodes, and that the operation’s learned parameters are kept the same regardless of where in the graph it is applied. The main source of inspiration for this work, *Neural Message Passing for Quantum Chemistry* [30], highlights this and defines a framework of which eight previously proposed models can be seen as special cases (at least with small modifications of it). It is covered in detail in the following section.

3.4 The message passing neural networks framework

An MPNN consists of two steps. The first is called the message passing phase. In several iterations, node states are updated using information from their intermediate neighbourhoods. After the message passing follows the readout step, which transforms the set of final node states into a fixed length vector in a way that is invariant to node ordering. All functions used are constructed with neural network building blocks, frequently FFNNs, described in Section 2.2.1. If the MPNN is to be used for making graph-level predictions, as in all experiments of this work, this graph embedding vector is then fed through a FFNN to generate output appropriately formatted for the task.

A formal description of how an MPNN turns a graph into an embedding is given in the following. Node hidden states are initialized with the node features: $\boldsymbol{h}_v^{(0)} = \boldsymbol{n}_v$ for all v in G. The message passing phase then follows the update rule

$$\begin{cases} \boldsymbol{m}_v^{(t)} = \sum\limits_{w \in N(v)} M_t(\boldsymbol{h}_v^{(t)}, \boldsymbol{h}_w^{(t)}, \boldsymbol{e}_{vw}) \\ \boldsymbol{h}_v^{(t+1)} = U_t(\boldsymbol{h}_v^{(t)}, \boldsymbol{m}_v^{(t)}) \end{cases} , \tag{3.2}$$

where M_t and U_t are neural network blocks. The readout step following K iterations of message passing is written

$$\boldsymbol{r} = R\Big(\{(\boldsymbol{h}_v^{(K)}, \boldsymbol{h}_v^{(0)}) \mid v \in G\}\Big) , \tag{3.3}$$

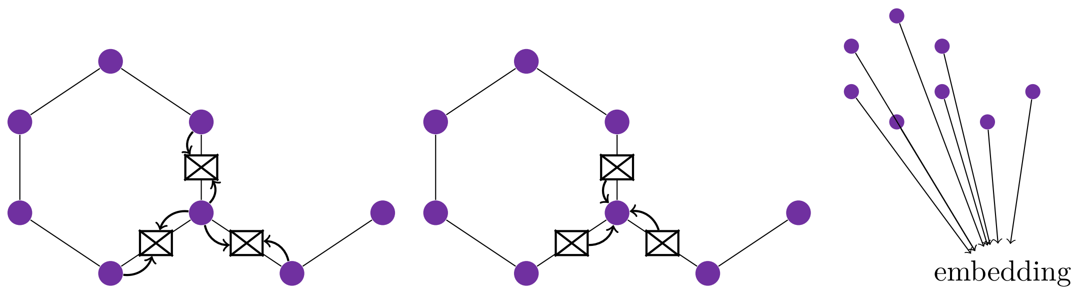

Figure 3.3: MPNN steps. The first two panels illustrate the update of one node. The drawn messages correspond to terms in the sum in equations (3.2). The right panel illustrates the readout step, which aggregates the set of final node hidden states.

where R should be constructed from neural network blocks in such a way that it is invariant to node ordering. An illustration of the whole procedure is provided in Figure 3.3.

In a powerful MPNN of the MPNN paper [30] , M_t, U_t and R consist of an *edge network*, a GRU unit and a *Set2Vec* [31] block, respectively. The GRU unit has been described in Section 2.2.2. This architecture is denoted ENNS2V throughout the remainder of this document. The edge network is a FFNN taking the edge feature vector as input to generate a matrix A_{vw} to be multiplied with the neighbour hidden state. In other words, letting f_{NN} denote the FFNN,

$$M_t(\boldsymbol{h}_v^{(t)}, \boldsymbol{h}_w^{(t)}, \boldsymbol{e}_{vw}) = A_{vw}\boldsymbol{h}_w^{(t)} = \Big(f_{\text{NN}}(\boldsymbol{e}_{vw})\Big)\boldsymbol{h}_w^{(t)} \ . \tag{3.4}$$

The GRU layer treats $\boldsymbol{h}_v^{(t)}$ as hidden state and $\boldsymbol{m}_v^{(t)}$ as input. Set2Vec uses an LSTM unit, mentioned in Section 2.2.2 to sequentially take weighted averages of all the vectors of a set. The weights are generated by computing one score per vector then normalizing them with softmax. The final embedding depends on the whole sequence of averages.

Another MPNN showing good performance was originally proposed in *Gated Graph Sequence Neural Networks* [32], though the version of it with hidden layers in the message function is more recent [33]. Throughout this document it is denoted GGNN. The model assumes categorical edge features and uses one FFNN per edge type. The message function can thus be written

$$M_t(\boldsymbol{h}_v^{(t)}, \boldsymbol{h}_w^{(t)}, \boldsymbol{e}_{vw}) = f_{\text{NN}}^{(\boldsymbol{e}_{vw})}(\boldsymbol{h}_w^{(t)}) \ ,$$

where $f_{\text{NN}}^{(\boldsymbol{e}_{vw})}$ denotes the FFNN corresponding to the edge type indicated by $\boldsymbol{e}_{vw}$. The update function is a GRU just as for ENNS2V. The readout function is a *graph gather* step using two FFNNs. More precisely,

$$R\Big(\{(\boldsymbol{h}_v^{(K)}, \boldsymbol{h}_v^{(0)})\}\Big) = \sum_{v \in G} f_{\text{NN}}(\boldsymbol{h}_v^{(K)}) \odot \sigma\Big(g_{\text{NN}}((\boldsymbol{h}_v^{(K)}, \boldsymbol{h}_v^{(0)}))\Big) \ , \tag{3.5}$$

where f_{NN} and g_{NN} are FFNNs, $\odot$ denotes elementwise multiplication, σ is the sigmoid function and the (,) of the right hand side denotes concatenation.

4 New contributions

The starting point of the experimentation of this work is reimplementation of the ENNS2V and GGNN models. This chapter covers the three main modifications of them. The first, employment of the very recent SELU activation function, is minor in terms of novelty and trivial when it comes to implementation. Nevertheless, it is a modification important enough to be highlighted. The second is a method to make the MPNN message passing step more expressive through weighted summation. The third idea is to remove the hidden states from the nodes and instead put a hidden state in each directed edge.

4.1 The SELU activation function

A first and implementation-wise simple modification of the models copied from the literature is to employ the scaled exponential linear unit (SELU) activation function [34]. This replaces Eq. (2.2) with

$$\mathrm{SELU}(x) = \begin{cases} \lambda x \text{ if } x > 0 \\ \lambda\alpha(e^x - 1) \text{ otherwise} \end{cases},$$

where λ and α are set to constant values slightly larger than 1. Applying SELU gives neural networks a “self-normalizing” property. The authors take a dynamical systems approach to analyze how the mean and variance of activations are affected when iteratively applying large layers with this activation function. They show that under reasonable assumptions the system has an attractive fixed point, and analytically compute λ and α to set this point to mean 0 and variance 1.

Because GNN architecture designs easily become deep, the employment of some sort of intermediate normalization is likely beneficial. The popular batch- and layer-normalization techniques come with potential problems. For the former it is that the batch dimension for a batch of graphs is hard to define; should it be normalized over the graphs, over the nodes, over the edges, or maybe over all nodes or all edges of the whole batch? Layer-normalization may be problematic as well because many layers of the architecture are small. SELU becomes the natural choice of normalization scheme for GNNs.

4.2 A more expressive message passing step

Defining the new message passing step requires considering a somewhat more general version of the MPNN framework. The non-weighted summation in Eq. (3.2) constitutes a limitation. The generalization of the first row in Eq. (3.2) that is instead considered is

$$\boldsymbol{m}_v^{(t)} = A_t\Big(\boldsymbol{h}_v^{(t)}, \{(\boldsymbol{h}_w^{(t)}, \boldsymbol{e}_{vw}) \mid w \in N(v)\}\Big) , \tag{4.1}$$

where the aggregation function A_t, like the readout function, is invariant to the order of set members.

A challenge in designing an aggregation function lies in eliminating the dimension corresponding to the cardinality i.e. the unknown number of set members. The previously mentioned Set2Vec block does this by generating weights with help of softmax to be able to take weighted averages, as does the *graph attentional layer* of Graph Attention Networks [29]. The latter can be expressed as the aggregation function

$$A_t\Big(\boldsymbol{h}_v^{(t)}, \{(\boldsymbol{h}_w^{(t)}, \boldsymbol{e}_{vw})\}\Big) = \sum_{w\in N(v)} \boldsymbol{W}\boldsymbol{h}_w^{(t)} \frac{\exp\left(g_{\text{NN}}(\boldsymbol{W}\boldsymbol{h}_v^{(t)}, \boldsymbol{W}\boldsymbol{h}_w^{(t)})\right)}{\sum\limits_{w'\in N(v)} \exp\left(g_{\text{NN}}(\boldsymbol{W}\boldsymbol{h}_v^{(t)}, \boldsymbol{W}\boldsymbol{h}_{w'}^{(t)})\right)} ,$$

where $\boldsymbol{W}$ is a learned matrix and g_{NN} is a neural network that takes two vectors and outputs a scalar. The layer is computationally efficient, especially if keeping g_{NN} small, and therefore suitable for node classification in very large graphs. The layer is furthermore employed in a *multi-head* manner, meaning that several instances of it, with matrices $\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(L)}$ and neural networks $g_{\text{NN}}^{(1)}, \ldots, g_{\text{NN}}^{(L)}$, are applied in parallel and the results then concatenated.

Inspired by the graph attentional layer, the aggregation function used in the present work is computationally heavier but potentially more expressive. It also takes edge features into account. Two FFNNs $f_{\text{NN}}^{(\boldsymbol{e}_{vw})}$ and $g_{\text{NN}}^{(\boldsymbol{e}_{vw})}$ per edge type are used, that all output vectors of length d_m. It can be written

$$A_t\Big(\boldsymbol{h}_v^{(t)}, \{(\boldsymbol{h}_w^{(t)}, \boldsymbol{e}_{vw})\}\Big) = \sum_{w\in N(v)} f_{\text{NN}}^{(\boldsymbol{e}_{vw})}(\boldsymbol{h}_w^{(t)}) \odot \frac{\exp\left(g_{\text{NN}}^{(\boldsymbol{e}vw)}(\boldsymbol{h}_w^{(t)})\right)}{\sum\limits_{w'\in N(v)} \exp\left(g_{\text{NN}}^{(\boldsymbol{e}vw')}(\boldsymbol{h}_{w'}^{(t)})\right)} , \tag{4.2}$$

where $\odot$ and the fraction bar represent element-wise multiplication and element-wise division, respectively. The cardinality is eliminated by taking d_m weighted averages. One weight for every element of every vector in the neighbourhood set is generated, that depends on every other element of the set; along one dimension via softmax and along the other via $g_{\text{NN}}^{(\boldsymbol{e}_{vw})}$. The same is in principle true for a d_m-headed graph attentional layer with $\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(d_m)} \in \mathbb{R}^{1\times d_n}$ (where d_n is the length of the node vectors). The latter however potentially mitigates the power of the neural networks $g_{\text{NN}}^{(1)}, \ldots, g_{\text{NN}}^{(L)}$, as they each only take two scalars as input, which in turn are generated with matrices that also must be able to do the work of $f_{\text{NN}}^{(\boldsymbol{e}_{vw})}$.

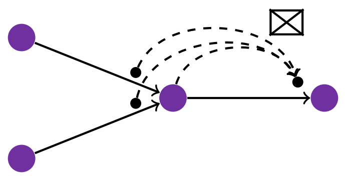

Figure 4.1: Illustration of the information used to update the hidden state of a directed edge.

4.3 Edge hidden states

A potential limitation of the MPNN framework is that every message pass entails aggregation of information from a symmetric neighbourhood. This work investigates the idea of putting a hidden state in each directed edge and aggregating hidden states from directed edges that are neighbours in an edge sense. In the resulting message passing step, the update of a hidden state has a corresponding direction. Note that even if the input graph is undirected, as is the case for a molecule, it is considered directed during the message passing step.

A directed edge is considered a neighbour of another directed edge if it bites the latter's tail. What information is used to update one hidden state is illustrated in Figure 4.1. Formally, the set of neighbours to an edge (v, w) in a graph $G = (V, E)$ is the set $N(v, w) = \{(w, v') \in E \mid v' \neq v\}$. The edge hidden state $\boldsymbol{h}_{vw}^{(t)}$ of edge (v, w) at time t is updated according to

$$\begin{cases} \boldsymbol{m}_{vw}^{(t)} = A_t\Big(\boldsymbol{e}_{vw}, \{\boldsymbol{h}_{wv'} \mid (w, v') \in N(v, w)\}\Big) \\ \boldsymbol{h}_{vw}^{(t+1)} = U_t(\boldsymbol{h}_{vw}^{(t)}, \boldsymbol{m}_{vw}^{(t)}) \end{cases} , \tag{4.3}$$

where the aggregation function A_t is invariant to the order of set members.

One hypothetical advantage compared to MPNN is explained in the following. Consider a small graph of three nodes A, B and C connected as A–B–C, as illustrated in Figure 4.2. Suppose information passage from A to C is relevant for the task. For information to travel this path, two message passes are necessary. In the first pass, information is passed from A to B, as desired. However, information is also passed from C to B, so that part of B's memory is being occupied with information that the final destination C already has. This back-and-forth passing of information happening in an MPNN hypothetically dilutes the hidden state of node B. When hidden states instead reside in the directed edges, it cannot happen. The closest thing corresponding to a hidden state in B is the hidden states in the edges (directed towards B) AB and CB. The update of C (more precisely the edge BC) uses information from AB, but not from CB.

For the experiments of this work, node features are embedded into edge features by feeding the concatenation of the raw edge and node feature vectors through a FFNN $f_{\mathrm{NN}}^{\mathrm{emb}}$,

$$\boldsymbol{e}'_{vw} = f_{\mathrm{NN}}^{\mathrm{emb}}((\boldsymbol{e}_{vw}, \boldsymbol{n}_v, \boldsymbol{n}_w)) \ ,$$

before the message step is carried with $\boldsymbol{e}'_{vw}$ in place of $\boldsymbol{e}_{vw}$ in (4.3). An aggregation function

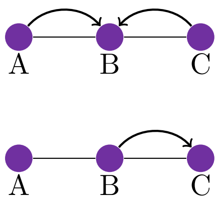

Figure 4.2: The path for information to travel to a second order neighbour via two message passing iterations. For information to get from A to C, it must get to B in an intermediate step. Since the message passing iterations has no concepts of direction, some information will be passed from C to B then back to C. Hypothetically this dilutes the information storage of each node.

similar to (4.2) is used, namely

$$A_t\Big(\boldsymbol{e}'_{vw}, \{\boldsymbol{h}_{wv'}\}\Big) = \sum_{\boldsymbol{x}\in S^{(t)}_{vw}} f_{\text{NN}}(\boldsymbol{x}) \odot \frac{\exp\left(g_{\text{NN}}(\boldsymbol{x})\right)}{\sum_{\boldsymbol{x}'\in S^{(t)}_{vw}} \exp\left(g_{\text{NN}}(\boldsymbol{x}')\right)} \ ,$$

where $S^{(t)}_{vw} = \{\boldsymbol{e}'_{vw}\} \cup \{\boldsymbol{h}_{wv'} \mid (w, v') \in N(v, w)\}$. After an edge message passing step of K iterations, a node hidden state for each node is taken as the sum of the edge hidden state of edges that the node is end of,

$$\boldsymbol{h}^{(K)}_v = \sum_{w\in N(v)} \boldsymbol{h}^{(K)}_{vw} \ .$$

This is done to be able to utilize the same readout functions as seen effective for the MPNNs.

5

Training and implementation details

The following covers details that need to be defined before setting up any experiment involving the described models. The five datasets considered are described, as well as the format they are stored as and what performance metrics are appropriate to use. What chemical information is used for atom feature vectors and edge feature vectors is then covered, before loss functions appropriate for the different types of datasets are described. The heaviest part of the chapter is a description of Bayesian optimization, which is used for tuning the hyperparameters of the models. Finally, a range of used technologies in the form of programming frameworks and hardware is listed.

5.1 Data

Five different datasets are used for evaluation of models. All of them are public datasets provided by and described in *MoleculeNet: A Benchmark for Molecular Machine Learning* [35], which also recommends a performance metric as well as splitting method for each. All molecules in all datasets are stored as strings formatted according to the simplified molecular-input line-entry system (SMILES) specification [36] but are readily converted into corresponding graph representations using a software package. The output of a molecule is a vector of one or more values, which are binary labels for classification and real numbers for regression. Some of the multitask datasets are sparse in the sense that aside from containing known positive and negative labels, they also contain missing ones.

The estimated solubility dataset **ESOL** is a regression dataset with. The prediction target for each compound is the solubility in water. Being able to predict this is useful for the lead optimization step of the drug discovery process. The recommended performance metric for this dataset is root-mean-square-error. The blood-brain barrier penetration dataset **BBBP** contains one label per compound, telling whether it is able to enter the brain or not. The blood-brain barrier of the human body is very effective at locking out foreign substances, including many drugs. Hence the challenge is highly relevant in developing drugs targeting the central nervous system. This is the only dataset that is not split randomly but instead according to a scaffold method. This assigns molecules into clusters of high structural similarity, and its use is considered to yield results that are more practically meaningful. The recommended performance metric for this dataset, as well as for subsequently mentioned SIDER and Tox21, is area under the curve of the receiver operating characteristic (ROC-AUC). The area refers to an integral over decision thresholds, computed to eliminate a trade-off problem between true positive and false positive rates. The Side Effect Resource Database **SIDER** contains information about adverse drug reactions. The version of it used by MoleculeNet groups side effects into 27 groups. Two examples of categories are eye disorders and metabolic disorders. The version of the **Tox21** dataset used contains 8014 compounds. The literature contains benchmarks on various subsets of Tox21, whose original version contains 12707 compounds [37]. The output labels represent outcome of various experiments indicating toxicity. The maximum unbiased validation dataset **MUV** is the largest dataset used in this work. The 17 tasks are challenging to predict biochemical target activities and the MoleculeNet benchmark is reported in terms of area under the precision-recall curve (PR-AUC). PR-AUC is similar to ROC-AUC but considered more appropriate for highly imbalanced datasets. MUV has very few actives. A summary of all datasets used is seen in Table 5.1.

Table 5.1: Dataset details.

Dataset	Number of compounds	Number of tasks	Task type	Splitting method	Performance metric
ESOL	1128	1	regression	random	RMSE
BBBP	2039	1	classification	scaffold	ROC-AUC
SIDER	1427	27	classification	random	ROC-AUC
Tox21	7831	12	classification	random	ROC-AUC
MUV	93087	17	classification	random	PR-AUC

5.2 Atom features and bond features

In the experiments, each atom feature vector has 75 binary valued elements and is generated with a readily available function in a utilized software package. It is a concatenation of one-hot encoded vectors of various relatively low-level features. Specifically, it contains information about chemical element, number of directly bonded neighbours, formal charge, whether the atom has an unpaired valence electron, hybridization type, whether the atom is aromatic and number of hydrogens. Each edge feature vector is a 4-element one-hot encoding of the bond types single, double, triple and aromatic.

5.3 Loss functions

For ESOL, which is the only regression dataset, the mean squared error loss function is used. Given a batch of predictions $\hat{\boldsymbol{y}} \in \mathbb{R}^N$ and target values $\boldsymbol{y} \in \mathbb{R}^n$, the loss is then

$$l(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \ .$$

The remaining four datasets are all classification datasets, and three contain data about more than one task. Furthermore, Tox21 and MUV contain missing labels; a compound may not have values indicating activity or non-activity for all tasks. The main loss function for classification in this work is a form of multitask cross entropy that ignores missing data points [37]. Let $\boldsymbol{Y} \in \{0,1\}^{n \times p}$ and $\boldsymbol{M} \in \{0,1\}^{n \times p}$ represent a batch of target values in a multitask classification context, where $Y_{ij} = 1$ means compound i is known positive for task j, and $M_{ij} = 1$ means the datapoint is not missing. Given a matrix of rescaled network outputs $\hat{\boldsymbol{Y}} \in [0,1]^{n \times p}$, the loss function is

$$l(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = -\frac{1}{m} \sum_{i=1}^{n} \sum_{j=1}^{p} M_{ij} \Big(Y_{ij} \log(\hat{Y}_{ij}) + (1 - Y_{ij}) \log(1 - \hat{Y}_{ij}) \Big) \ ,$$

where $m = \sum_{i=1}^{n} \sum_{j=1}^{p} M_{ij}$ is the number of present datapoints in the batch. The rescaling of neural network outputs to $[0,1]$ gives numbers that can be interpreted as probabilities and is done by using the sigmoid function.

Because all classification datasets considered have unbalanced positive/negative class ratios, a variant of the aforementioned loss that incorporates task weighting is also investigated:

$$l(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = -\frac{1}{m} \sum_{i=1}^{n} \sum_{j=1}^{p} M_{ij} \Big(w_j Y_{ij} \log(\hat{Y}_{ij}) + (1 - Y_{ij}) \log(1 - \hat{Y}_{ij}) \Big) \ , \tag{5.1}$$

where w_j is the the number of negatives for task j in the training set divided by the number of positives for task j in the training set. Given a training set with few positives, this potentially improves performance by making the training focus more on the few positives that are to be found.

5.4 Parallelized Bayesian optimization of hyperparameters

Selection of hyperparameters for neural network architectures, especially more intricate ones, has great impact on performance [38]. As a next step after guessing values based on experience and intuition, the simplest method is a grid search, meaning an exhaustive evaluation of every point of a selected subset of hyperparameter space. This approach is trivially parallelizable but suffers greatly from the curse of dimensionality. With d hyperparameters, and a_i the number of included values along dimension i for $i = (1, \dots, d)$, the number of evaluations is $\prod_{i=1}^{d} a_i$. Since one evaluation is a whole training run of a neural network, grid search easily becomes infeasible. One way to allow for a large dimensionality is to do a random search, i.e. to sample points to evaluate from a uniform distribution.

Bayesian optimization [39] (BO) is a strategy that in practice works for a relatively broad range of functions. It is claimed to work for problems of dimensionality up to about $d = 20$ [39]. The idea is to model the observations of the objective function as a stochastic process and use this to give a hint on where it is best to evaluate the function next. After some initial number of evaluations are made, a sequential iterative procedure starts, in which the stochastic process is fit with all so-far observed evaluations then used to compute the next point to evaluate. The assumptions on the objective function are loose; for example, it does not need to be differentiable and its observations may be noisy.

The aforementioned properties render BO useful for the hyperparameter tuning of many neural networks. The method has for example been applied to an at the time state-of-the-art convolutional neural network with 9 hyperparameters [40]. The result was a drop in test error from 18% to 15% on image classification. GNNs tend to have a large number of hyperparameters and thus a BO strategy to tune them is justified. The particular kind of setup used in this work is a parallelized version, that can for practical purpose be seen as a trade-off between a trivially parallelizable random search and the sequential BO. A description is given in the following.

5.4.1 Description

In Bayesian optimization of a noisy objective function f, the observations made so far are modelled as a stochastic process, in practice usually as a Gaussian process. In the context of tuning a neural network, $f(\boldsymbol{x})$ is the validation score resulting from a training run as a function of the vector of hyperparameters $\boldsymbol{x}$. Suppose the points that have been evaluated so far are $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$ and that $\boldsymbol{y}_n = (y_1, \ldots, y_n)^\intercal$ is the vector of observations of f. According to the Gaussian process model, the observation at a new point $\boldsymbol{x}$ is then normally distributed with mean

$$\mu_n(\boldsymbol{x}) = \boldsymbol{k}_n(\boldsymbol{x})^\intercal \left(\boldsymbol{K}_n + \sigma_n^2 \boldsymbol{I}\right)^{-1} \boldsymbol{y}_n \tag{5.2}$$

and variance

$$\sigma_n^2(\boldsymbol{x}) = k(\boldsymbol{x}, \boldsymbol{x}) - \boldsymbol{k}_n(\boldsymbol{x})^\intercal \left(\boldsymbol{K}_n + \sigma_n^2 \boldsymbol{I}\right)^{-1} \boldsymbol{k}_n(\boldsymbol{x}) \ , \tag{5.3}$$

where k is a chosen *kernel function*, $\boldsymbol{k}_n(\boldsymbol{x}) = (k(\boldsymbol{x}_1, \boldsymbol{x}), \ldots, k(\boldsymbol{x}_n, \boldsymbol{x}))^\intercal$, $\boldsymbol{K}_n$ is the matrix such that $(\boldsymbol{K}_n)_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ and σ_n^2 is the estimated variance of the prior.

Known properties of the posterior distribution defined by (5.2) and (5.3) are exploited to decide where to next evaluate f. After collecting the new observation, the kernel parameters are refitted and a new posterior is had, and the procedure continues iteratively. The point at which to make the next evaluation is found through maximizing an *acquisition function*, which ideally takes on larger values where it is better to make the next evaluation of f. A typical acquisition function to use is expected improvement, which gives the maximization problem

$$\boldsymbol{x}_{n+1} = \underset{\boldsymbol{x}}{\operatorname{argmax}} E[f(\boldsymbol{x}) - f_n^* \,|\, (\boldsymbol{x}_1, y_1, \ldots, \boldsymbol{x}_n, y_n)] \ , \tag{5.4}$$

where f_n^* is the largest of the previous observations $y_1, \ldots, y_n$.

The parallelized BO used in the present work entails a modification of the acquisition function optimization step. The problem with (5.4) is that it returns only one point. The suggestion $\boldsymbol{x}_{n+2}$ is acquired only after the costly objective evaluation of $\boldsymbol{x}_{n+1}$ is done, rendering the method entirely sequential with respect to objective function evaluations. Local penalization [41] is one method to instead acquire a batch of K points $\boldsymbol{x}_{n+k}, \ldots, \boldsymbol{x}_{n+K}$ to evaluate in parallel, by replacing (5.4) with

$$\boldsymbol{x}_{n+k} = \underset{\boldsymbol{x}}{\operatorname{argmax}} \Big(E[f(\boldsymbol{x}) - f_n^* \,|\, (\boldsymbol{x}_1, y_1, \ldots, \boldsymbol{x}_n, y_n)] \prod_{j=1}^{k-1} \varphi(\boldsymbol{x}, \boldsymbol{x}_{n+j}) \Big) \text{ for } k = 1, \ldots, K \ . \tag{5.5}$$

The local penalty factor φ is a chosen function that is non-decreasing in $|\boldsymbol{x} - \boldsymbol{x}_{n+k-1}|$ and takes values in $[0, 1]$. It penalizes the points $\boldsymbol{x}_{n+1}, \ldots, \boldsymbol{x}_{n+k-1}$, making the maximization in (5.5) favour unevaluated points.

The kernel function models the spatial dependence of covariance between observations. A choice that has been proposed suitable for hyperparameter optimization of machine learning models [40] is the Matern 5/2 kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \theta_0 \Big(1 + \sqrt{5 r^2(\boldsymbol{x}, \boldsymbol{x}')} + \frac{5}{3} r^2(\boldsymbol{x}, \boldsymbol{x}')\Big) \exp(-\sqrt{5 r^2(\boldsymbol{x}, \boldsymbol{x}')}) \ ,$$

where
$$r^2(\boldsymbol{x}, \boldsymbol{x}') = \sum_{i=1}^{d} \frac{(x_i - x_i^{'})^2}{\theta_i^2} \ .$$

The above description leaves out a number of details that are beyond the scope of this work, namely how to solve (5.4) and (5.5), how to choose φ and how to fit φ and k. These things are internally handled by the utilized software.

5.5 Software and hardware

All model implementations are in the **Python** programming language. The most important python library used is the deep learning framework **PyTorch** [42]. It essentially consists of a collection of neural network building blocks that keep track of their own gradients, a collection of gradient descent-based optimizers and support for GPU acceleration. Furthermore the chem-informatics library **RDKit** [43] is used to turn SMILES strings into graphs and to generate edge feature vectors. The **DeepChem** [44] library provides the function for generating atom feature vectors. **scikit-learn** [45] is used to compute ROC-AUC and PR-AUC. The Bayesian optimization implementation is that of **GPyOpt** [46].

The training of models is done on a GPU cluster managed through the job scheduling software **Slurm**. Training of a model utilizes one GPU, which is either an NVIDIA GK210GL Tesla K80 or an NVIDIA GV100GL Tesla V100 PCIe.

6

Experimental setup

The main experimental work is roughly organized into two stages. The first is to use models from the literature as a starting point, then add various extensions thought to possibly increase performance. The models that appear most powerful are then selected for comparison to benchmarks, to thoroughly evaluate their performance. After the benchmarking, one dataset/model combination is evaluated once more, but with a modified data preprocessing step.

All evaluations of dataset/model combinations except those for the MUV dataset entail the running of one hyperparameter optimization. Hyperparameter optimization of MUV falls outside of the time plan for this project work, but standalone training runs for three models are carried out. For every training run, part of the hyperparameter optimization or otherwise, the validation score and test score considered are those of the model state of the epoch after which it showed the highest validation score.

6.1 Model experimentation

All experiments in this section except that involving a modified loss function are done on the BBBP dataset only. BBBP is conveniently small for this purpose. Furthermore, it is a classification dataset and the results might therefore carry over better to SIDER, Tox21 and MUV.

The reimplemented models from literature, ENNS2V and GGNN described in Section 3.4, are first evaluated without modification. The replacement of the ReLU activation function with SELU in all FFNNs is then evaluated. Two more modifications of GGNN are investigated: an attention mechanism in the message function and a slight change of the attention mechanism of the readout function. The former is that described in detail in Section 4.2 and the resulting model is denoted AttentionGGNN. The latter is done by replacing the sigmoid function in Eq. (3.5) with softmax, so that the weights of an element in a term, corresponding to a node, in the weighted sum depends on the weights of all other elements.

The model implemented with edge hidden states is denoted EdgeModel. It borrows building blocks seen to work for the above models. The update function of the edge message passing step is thus the GRU. As described in Section 4.3, the edge hidden states are aggregated into their corresponding nodes after the message passing to be able to use the same readout function as for the previous models. The readout function is the graph gather of Eq. (3.5), with the only difference being that $\boldsymbol{h}_v^{(0)}$ is replaced with $\boldsymbol{h}_v^{(K)}$.

The final modification investigated that could be considered part of the model experimentation phase is the modified loss function of Eq. (5.1). This is evaluated for one model but for the Tox21 dataset in addition to BBBP. The modified loss function generates different weights for different tasks, thus Tox21 is an appropriate dataset to evaluate it on as it contains multiple tasks.

6.2 Benchmarking

Papers on deep learning methods [26, 30, 35] generally present results as a table that compares the performance of new methods to that of previously known ones on several datasets. A proper benchmark comprises a dataset preprocessed in a specific way, a prediction algorithm and a performance metric. To be able to compare a new algorithm to previous ones, factors that are not part of the algorithm itself need to be kept fixed.

To be able to thoroughly compare the implemented models to state-of-the-art from literature, the benchmarking evaluations of this work are made according to the same scheme as used for the molecular machine learning benchmark collection MoleculeNet [35]. The code provided by the authors is used to generate three different training-validation-testing splits (meaning nine subsets in total) in a 80/10/10 ratio, with the correct random seeds. The performance evaluation of one model on one dataset is done by first hyperparameter optimizing the validation score of one split, then rerunning training on each of the three splits using the found hyperparameter setting. Mean and standard deviation over the reruns are then computed. BBBP is an exception to the scheme with three splits. Since scaffold (not random) splitting is prescribed for this dataset, only one split is used.

The number of epochs per training run is set so that the individual training runs appear to converge. This is done for each dataset through guesswork followed by a small number (≤ 3) of preliminary training runs. The chosen values are included in Appendix A. The values are doubled before starting the reruns, as this gives at least some likelihood for improvement at a small computational cost.

6.3 Modified data preprocessing

The data aggregated with MoleculeNet code has two traits that it would not have if using a modified preprocessing step, which is arguably more appropriate. The first is that missing labels are treated as negatives. This means that the graceful handling of missing labels described in Section 5.3 is not used, and that the computation performance metrics includes negative labels that really represent missing information. The second is that some dataset samples contain small counterions together with the druglike molecule, the presence of which ought to be irrelevant when the substance is dissolved in the blood.

To investigate the impact of the factors above, the evaluation of AttentionGGNN on Tox21 is run again, but using a preprocessing procedure that includes the information about missing labels and that removes fragments not covalently bonded to the druglike molecules. Tox21 is chosen because it contains information about missing labels.

6.4 Hyperparameter search

The setup of the parallel Bayesian optimization (BO) comprises many choices, notably a search space need to be designed. The tuning is largely a matter of trial-and-error and a trade-off of computational resources and time, not the result of a rigorous analysis. Two principal BO parameters are batch size (of the parallel BO, i.e. not related to the batch size when training a neural network) and number of iterations. For the model experimentation phase, iterations are set to 20 and batch size to 10. This results in a total of 210 training runs, of which 10 are of the initial batch, which evaluates randomly sampled hyperparameter settings. From the model experimentation logs it appears large improvements rarely happen towards the end of an optimization. For the benchmark optimizations the number of iterations is therefore lowered to 9, for a total of 100 training runs. The total optimization time varies between models and datasets. The fastest ones take approximately two days while the slowest take eight.

In the following are some rules of thumb on how to set up the search space to render the method effective with the above settings. Table 6.1 shows the domain setting used for GGNN. Domains for three other models are found in Appendix B.

6.4.1 Practical setup of Bayesian optimization

To find settings and hyperparameter domains of the BO routine, a phase of preliminary experimentation turned out necessary. Some practical takeaways from this tuning, that may be useful for successful BO in general, are the following:

- Including 10 well-chosen hyperparameters in the optimization and setting the rest to fixed values yields good results. Increasing the dimensionality may lead to worse results in a practical context.
- Hyperparameters commonly reported in powers of 10, e.g. learning rate and weight decay, need rescaling before being passed to the BO. This can be achieved either by simply taking the logarithm or by using a general *input warping* scheme.
- When working with a high dimensional domain, it is better to keep some integer-valued ranges sparse rather than dense. As an example, it might be better to let a hidden layer size take on values in $\{25, 50, 100\}$ than in $\{25, 26, \dots, 100\}$.
- It is vital to define the domain so that all of its values lead to at least reasonable performance. This is especially true for hyperparameters that abruptly break the learning if set too big or too small such as the log-rescaled learning rate.

Some further practical observations, more specific to hyperparameter optimization of GNNs, are:

- Leaving the number of hidden layers of a given FFNN fixed and varying the hidden layer size is more effective than varying both of them.
- When including hyperparameters for explicit regularization in the form of weight decay

on all parameters or dropout ratios for the various FFNNs, all appeared to be set to very small values by the BO. Setting all of them to 0 up front yields better results.

- The message size and hidden layer sizes of FFNNs of the message passing step should be set relatively small, i.e. ~ 35.
- The gather width and hidden layer sizes of the output FFNN should be set relatively big, i.e. ~ 350.

The BO domain for GGNN shown in Table 6.1 is an example of a domain defined according to the above specified rules of thumb. A choice remaining to explain is the fixed value 50 for the batch size, which is chosen more for practical reasons than anything else. 50 appears to be a good trade-off on the given hardware, allowing for headspace memory-wise when tuning e.g. the FFNN sizes while letting the training run at close to maximum speed. The fact that each evaluation in a batch finishes in the same amount of time also renders the optimization faster, as the routine will not start a new BO batch of evaluations before the whole previous one has finished. Varying the batch size may theoretically yield better performance as it affects the degree of stochasticity of the gradient descent, but the aforementioned advantages outweigh this.

Table 6.1: Domain for hyperparameter optimization of GGNN. The model specific neural network blocks are defined in Section 3.4.

	Hyperparameter	Domain/value
Varied	Learning rate	$[10^{-5.5}, 10^{-4}]$[a]
	Message passes	$\{1, 2, \ldots, 10\}$
	Message size	$\{10, 16, 25, 40\}$
	Hidden dimension of $f_{\text{NN}}^{(e_{vw})}$	$\{50, 85, 150\}$
	Hidden dimension of f_{NN}	$\{15, 26, 45, 80\}$
	Hidden dimension of g_{NN}	$\{15, 26, 45, 80\}$
	Readout dimension of f_{NN} and g_{NN}	$\{30, 45, 70, 100\}$
	First hidden dimension of output FFNN	$\{360, 450, 560\}$
	Second hidden dimension of output FFNN	$[0.2, 0.6]$[b]
	Dropout p of output FFNN	$[0.0, 0.1]$
Fixed	Batch size	50
	Weight decay	0
	Hidden layers in $f_{\text{NN}}^{(e_{vw})}$, f_{NN} and g_{NN}	2
	Hidden layers in output FFNN	2
	Dropout p of $f_{\text{NN}}^{(e_{vw})}$, f_{NN} and g_{NN}	0

[a] Log-rescaled
[b] Relative to the first hidden dimension

7

Results

This chapter is divided into sections that roughly appear in the chronology of the experimental work. First, results from the model extension phase are reported. This includes scores on one dataset for literature models and their extended versions, as well as an evaluation of performance impact from employing the modified classification loss function together with one model. Then, the thorough benchmarking process is covered. Scores for three new models are compared to literature values, for four different datasets. After that, performance impact from refining the data preprocessing is shown, to give a hint on how important data curation is when evaluating machine learning models on molecular data. Finally, scores for some standalone runs on the MUV dataset are reported.

7.1 Model extension phase

The results of the model extension phase are summarized in Table 7.1. Out of the two models reimplemented as described in literature, ENNS2V outperforms GGNN. Upon switching activation function from ReLU to SELU however, the GGNN is at an advantage. While the addition of softmax to the GGNN with ReLU leads to an improvement, the same thing reduces performance of the GGNN SELU. Addition of attention to the message passing phase of the GGNN SELU is neutral or slightly negative. EdgeModel appears to be on par with the best performing GGNN. The three models showing best performance at this stage, and that are thus selected for further and more careful evaluation, are one instance each of GGNN, AttentionGGNN and EdgeModel. All use the SELU activation function.

7.1.1 Comparison of loss functions for classification

According to the result of two hyperparameter optimizations on two different classification datasets, the modified weighted loss function yields no apparent performance benefit. The best validation score of the best training run of the hyperparameter optimizations is seen in Table 7.2.

Table 7.1: Performances in the form of ROC-AUC of different models on the BBBP dataset. The validation scores reported are from the best model of the best training run of a parallel BO of 20 iterations. The total number of training runs per model is 210 and each training run was allowed 100 epochs. The models chosen for benchmarking are highlighted in bold.

Model	Extension	Validation score
ENNS2V	–	0.963995
	SELU	0.968168
GGNN	–	0.949631
	gather softmax	0.959627
	SELU	**0.976514**
	SELU+gather softmax	0.964771
AttentionGGNN	**SELU**	**0.972147**
	SELU+gather softmax	0.948175
EdgeModel[a]	**SELU**	**0.976029**

[a] Only run for 17 iterations as opposed to 20, because too little GPU-time was allocated at the cluster and a rerun suspected to have small impact on result

Table 7.2: Performances in the form of ROC-AUC using different classification loss functions. The validation scores reported are from the best model of the best training run of a parallel BO. For BBBP, BO iterations were set to 20 and epochs to 100. For Tox21 they were set to 10 and 200.

Dataset	Validation score Normal loss function	Validation score Modified loss function
BBBP	0.972147	0.971370
Tox21	0.852888	0.842929

7.2 Comparison to benchmarks

The results of the new experiments are summarized together with MoleculeNet results in Table 7.3. All three of GGNN, AttentionGGNN and EdgeModel use the SELU activation function, as seen beneficial in the previous section. The measured improvements of the new models vary between datasets. On ESOL, all three new models beat the best MoleculeNet model with regards to both validation score and test score. On BBBP and SIDER, MoleculeNets best graph model test scores are beaten by AttentionGGNN and EdgeModel, respectively. On Tox21, all new models beat the validation scores of the best MoleculeNet model, but none matches the test scores. Of the new models, EdgeModel consistently yields the best average validation score, while the test scores do not exhibit the same pattern.

7.3 Impact of preprocessing

Evaluation of the impact of using a more refined preprocessing shows that this is indeed an important factor in evaluating machine learning methods on molecular data. The scores resulting from two different preprocessing procedures are seen in Table 7.4.

Table 7.3: Comparison of three implemented models to benchmarks from MoleculeNet. The results of the implemented models were generated according to the same pattern as used in MoleculeNet. The best models of MoleculeNet are chosen based on test score. The best validation scores and test scores of implemented models (i.e. the best of the non-MoleculeNet scores) are highlighted in bold.

Benchmark	Model	Validation score	Test score
ESOL RMSE	Best of MoleculeNet	0.55 ± 0.02	0.58 ± 0.03
	Best graph model of MoleculeNet	0.55 ± 0.02	0.58 ± 0.03
	GGNN	$\mathbf{0.247 \pm 0.008}$	0.279 ± 0.029
	AttentionGGNN	0.258 ± 0.008	$\mathbf{0.276 \pm 0.048}$
	EdgeModel	0.255 ± 0.019	0.295 ± 0.024
BBBP ROC-AUC	Best of MoleculeNet	0.964 ± 0.000	0.729 ± 0.000
	Best graph model of MoleculeNet	0.943 ± 0.002	0.690 ± 0.009
	GGNN	0.965 ± 0.004	0.656 ± 0.040
	AttentionGGNN	0.955 ± 0.004	$\mathbf{0.713 \pm 0.010}$
	EdgeModel	$\mathbf{0.965 \pm 0.001}$	0.689 ± 0.014
SIDER ROC-AUC	Best of MoleculeNet	0.650 ± 0.013	0.684 ± 0.009
	Best graph model of MoleculeNet	0.609 ± 0.021	0.638 ± 0.012
	GGNN	0.649 ± 0.021	0.633 ± 0.018
	AttentionGGNN	0.654 ± 0.013	0.632 ± 0.012
	EdgeModel[a]	$\mathbf{0.662 \pm 0.006}$	$\mathbf{0.642 \pm 0.025}$
Tox21 ROC-AUC	Best of MoleculeNet	0.825 ± 0.013	0.829 ± 0.006
	Best graph model of MoleculeNet	0.825 ± 0.013	0.829 ± 0.006
	GGNN	0.834 ± 0.004	0.821 ± 0.013
	AttentionGGNN	0.840 ± 0.007	$\mathbf{0.823 \pm 0.004}$
	EdgeModel	$\mathbf{0.841 \pm 0.010}$	0.820 ± 0.004

[a] Only includes 40 training runs instead of 100 because too little GPU-time was allocated at the cluster and a restart suspected to have small impact on result

Table 7.4: Performance in the form of ROC-AUC on Tox21, using different preprocessing procedures. The BOs were run in batches of 10 for a total of 100 training runs. Epochs of each hyperparameter optimization training run were set to 250.

Preprocessing	Validation score	Test score
MoleculeNet	0.840 ± 0.007	0.823 ± 0.004
Modified	0.859 ± 0.004	0.847 ± 0.005

7.4 MUV performance

The result of standalone training runs on MUV for different models and under different preprocessing schemes is shown in Table 7.5. Hyperparameter settings are taken from hyperparameter optimization results for Tox21. AttentionGGNN with modified data preprocessing yields a higher reported test score than for any other case. The importance of preprocessing and graceful handling of missing labels is increasingly clear.

Table 7.5: Results of standalone training runs on MUV PR-AUC, with the two different preprocessing schemes. Each training run ran for 300 epochs.

Preprocessing	**Model**	**Validation score**	**Test score**
MoleculeNet	Best of MoleculeNet	0.202 ± 0.032	0.184 ± 0.020
	Best graph model of MoleculeNet	0.127 ± 0.028	0.109 ± 0.028
	Graph Convolution of MoleculeNet	0.049 ± 0.023	0.046 ± 0.031
	AttentionGGNN	0.118	0.053
Modified	GGNN	0.190	0.175
	AttentionGGNN	0.177	0.189
	EdgeModel	0.212	0.178

8

Discussion

This chapter begins with an analysis of the experimental results, which appear strong. The experimental work has however unravelled that there is a great need for skepticism towards the kind of numerical scores reported, motivating the critical discussion that then follows. Finally, the discussion is wrapped up with a description of some identified advantages and disadvantages of the class of implemented models.

8.1 Performance

In the model extension phase, the clearest of all improvements is that achieved by employing SELU in the literature GGNN. Doing the same modification to the literature ENNS2V model seems to give an improvement too, but it's not as clear. A hypothetical explanation could be that the original models are dissimilar with regards to depth. The SELU authors demonstrate clearly that the activation function leads to the greatest improvement for deep networks. The given GGNN implementation is effectively deeper than the ENNS2V, having two hidden layers in the message function and two hidden layers in the readout function. The latter effectively has a linear transformation in the message function and no concept of depth in the Set2Vec readout function. When it comes to the switching from simple sigmoid to softmax in the GGNN readout function, it interestingly seems to lead to improvement when SELU is not involved, but not if it is. Hypothetically this could be attributed to a need of normalization in the GGNN, as both SELU and softmax can be viewed as normalizing methods. Adding the attention aggregate function in the message function of the GGNN with SELU does not lead to an improvement, but neither is the decrease in performance significant. The same is true for the comparison between GGNN with SELU and the EdgeModel.

When comparing models to MoleculeNet benchmarks using the prescribed dataset splits, metrics and evaluation pattern, the emerging picture is that all three new models are competitive with state-of-the-art. Looking more closely, one observes variation between datasets. The clearest improvements are seen for ESOL, where all the three new models outperform all MoleculeNet models with good margin. For BBBP, AttentionGGNN outperforms the best graph-based MoleculeNet model with a relatively good margin, but not the overall best model. Despite not beating all competition, this represents progress because the graph-based models require no feature engineering. The winning MoleculeNet model is a kernel SVM on molecular fingerprints, a featurization hand-engineered specifically for molecules.

There seems to be a pattern in that the newly run experiments have a bigger difference between validation score and testing score than MoleculeNet experiments. One thing that could in principle affect this is the hyperparameter optimization, as it adapts the hyperparameters to the validation set provided to it. This effect should however be small, since the reported score is the average of three reruns, of which only one uses the same validation set as the hyperparameter optimization. The validation set is in the context of a rerun a tool to decide which model to use; the reported testing score is that which the model performs at the point during training that it achieves the highest validation score. It is possible that the corresponding method used by MoleculeNet is not exactly the same. Nevertheless, a high validation score indicates a potentially powerful model because there are things that can increase the generalizability beyond the validation set, for example the use of a bigger validation set.

The uncertainty in impact of the use of validation score during training is only one reason why benchmarking machine learning models is inherently difficult.

8.2 Challenges in interpreting results

There is a clear increase in measured scores when preprocessing the Tox21 and MUV data differently, by excluding missing labels instead of treating them as negative and cleaning the samples of counterions. This is on one hand an illustration of the importance of taking meticulous care to gracefully curate data, on the other it is one more example of something that may distort benchmark comparisons.

Further reasons to be skeptical about the numerical results lie in the stochasticity of several steps in the evaluation scheme. The hyperparameter optimization is one of them. Being notoriously expensive, running it more than once to be able to compute standard deviations is practically infeasible. The initialization and training of each model are other steps involving stochasticity. It is difficult to assess whether results are a product of luck in finding good hyperparameters. Assuming this step is repeatable, there may still be luck involved in the three reruns.

A last potential confounding factor that should be mentioned is the impact of the different Bayesian optimization setup. MoleculeNet does 20 hyperparameter evaluations in sequence while the present work uses 10 iterations of 10 evaluations each. How much impact this has would be expensive to assess. In one view, the use of a more sophisticated method is justified in the case of the new models, as their number of hyperparameters is inherently large. MoleculeNet optimizes a notably smaller number of hyperparameters.

8.3 Advantages and disadvantages of the approach

Aside from the demonstrated high performance on certain tasks, the most immediate advantage of using GNNs for molecular property prediction is the elimination of feature engineering. Competing methods may use for example the fixed ECFP featurization. This is relatively “deep”, as it works on molecular graphs. The design details of ECFP is nevertheless a case of feature engineering to achieve something which works well for cheminformatics tasks. A GNN is “deeper” since an input graph does not necessarily have to be a molecule.

Another advantage to GNNs is that, as any neural network, they are flexible. Taking any of the GNNs described in this document, but removing the output FFNN, what one obtains is a neural network building block that generates a potentially powerful graph embedding. A new field with great potential that uses GNNs this way is that of *de novo* molecular generation [47, 48]. These use neural networks to represent a distribution of possible building steps, conditioned on the current molecular graph. A building step can for example be to add a node at a certain place in the graph. Starting from a smaller molecular graph or just one atom, building steps are sampled iteratively to generate larger molecular graphs one step at a time.

The above advantages come with caveats. When designing a GNN architecture, one easily ends up with a large number of hyperparameters. This necessitates the use of well-engineered methods to optimize them, such as a properly set up Bayesian optimization. The automation of domain specific feature engineering comes at the cost of increasing the size of this extra layer of parameters in need of tuning. Furthermore, GNNs require a long time to train, as does any large neural network. This is despite the use of powerful GPU acceleration. During some quick sanity-check experiments during the project work a known to be fast random forest classifier has been seen to train to convergence in less than one minute on hardware that needs an hour to train a GNN. The long training time becomes especially problematic given the necessity to train many models to find good hyperparameters.

Using MPNN architectures is computationally expensive, and the new modifications make them more so. While in principle the methods work on any graphs, the cost makes them infeasible for graphs that are large and/or dense. The molecular graphs of the datasets considered in this work largely fulfill the following requirements:

- small size (5-50 nodes rather than thousands)
- relatively few edges
- low maximum node degree (an atom having more than three non-hydrogen neighbours is very rare)

It can be argued that for the specific domain of drug discovery, large computational cost is not an issue in light of the unavoidable large cost of developing any medicine. For other domains and applications, this might not be the case.

9

Conclusion

This thesis provides a detailed description of a class of neural network architectures for graphs, as well as three newly proposed modifications that could be made to its previous instances in the literature. Furthermore, it describes how to use an appropriate parallel Bayesian optimization to tune their hyperparameters.

The comparison to benchmarks shows that the new implementations are at least on par with state-of-the-art machine learning techniques for molecular property prediction. For certain benchmarks, all previous methods are outperformed. The cases in which the new models outperform previous graph-based models but are still beaten by classical algorithms on fixed featurizations, can also be seen as representing progress. This is because graph-based models are truer examples of deep learning in the sense that they require less feature engineering. The comparison of graph-based models with older techniques such as SVM is between something in its infancy and something mature. That the former shows comparable performance at all indicates that there is more to come if refining the approach. A key learning from the benchmarking process is that future work on deep learning on molecules needs to be carried out with a great deal of caution in terms of preprocessing datasets appropriately.

The non-need of feature engineering is a characteristic of graph neural networks that stands out. It is debatable, however, how truly general the class of implemented methods is; the computational cost of the architectures restricts them to practical use on relatively small graphs. Another advantage is the flexibility of the architectures, rendering them useful in for example *de novo* molecular generation.

Bibliography

[1] Peter Imming. “Medicinal Chemistry: Definitions and Objectives, Drug Activity Phases, Drug Classification Systems”. In: *The Practice of Medicinal Chemistry*. Elsevier Science, 2015. Chap. 1, pp. 5–10.

[2] Corwin Hansch. “A Quantitative Approach to Biochemical Structure-Activity Relationships”. In: *Accounts of Chemical Research* 2.8 (1969), pp. 232–239.

[3] Arkadiusz Dudek, Tomasz Arodz, and Jorge Galvez. “Computational Methods in Developing Quantitative Structure-Activity Relationships (QSAR): A Review”. In: *Combinatorial Chemistry & High Throughput Screening* 9.3 (2006), pp. 213–228.

[4] Corinna Cortes and Vladimir Vapnik. “Support-vector networks”. In: *Machine learning* 20.3 (1995), pp. 273–297.

[5] Leo Breiman. “Random Forests”. In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32.

[6] Mohsen Shahlaei. “Descriptor selection methods in quantitative structure-activity relationship studies: A review study”. In: *Chemical Reviews* 113.10 (2013), pp. 8093–8103.

[7] Jynto. *Caffeine molecule ball from xtal*. Jan. 2015. URL: `https://commons.wikimedia.org/wiki/File:Caffeine_molecule_ball_from_xtal_(1).png` (visited on 03/08/2019). License: Creative Commons CC0 1.0.

[8] Alisneaky and Zirguezi. *Kernel Machine*. Apr. 2011. URL: `https://commons.wikimedia.org/wiki/File:Kernel_Machine.svg` (visited on 03/08/2019). License: Creative Commons CC BY-SA 4.0.

[9] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *Cognitive modeling* 5.3 (1988), p. 1.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: `http://www.deeplearningbook.org` (visited on 03/08/2019).

[11] George Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[12] Kurt Hornik. “Approximation capabilities of multilayer feedforward networks”. In: *Neural networks* 4.2 (1991), pp. 251–257.

[13] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. “Speech recognition with deep recurrent neural networks”. In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649.

[14] Alex Graves et al. “A novel connectionist system for unconstrained handwriting recognition”. In: *IEEE transactions on pattern analysis and machine intelligence* 31.5 (2009), pp. 855–868.

[15] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[16] Junyoung Chung et al. “Empirical evaluation of gated recurrent neural networks on sequence modeling”. In: *arXiv preprint arXiv:1412.3555* (2014).

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

[18] Andrew Ng. *CS229 Lecture notes.* 2018. URL: `http://cs229.stanford.edu/notes/cs229-notes1.pdf` (visited on 03/03/2019).

[19] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[20] Anders Krogh and John A Hertz. “A Simple Weight Decay Can Improve Generalization”. In: *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 4.* Morgan Kaufmann, 1992, pp. 950–957.

[21] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (June 2014), pp. 1929–1958.

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

[23] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint*
arXiv:1607.06450 (2016).

[24] David Rogers and Mathew Hahn. “Extended-Connectivity Fingerprints”. In: *Journal of Chemical Information and Modeling* 50.5 (2010), pp. 742–754.

[25] David K Duvenaud et al. “Convolutional networks on graphs for learning molecular fingerprints”. In: *Advances in neural information processing systems.* 2015, pp. 2224–2232.

[26] Thomas N Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *International Conference on Learning Representations (ICLR).* 2017.

[27] Ferenc Huszar. *How powerful are Graph Convolutions?* Blog. 2016. URL: `https://www.inference.vc/how-powerful-are-graph-convolutions-review-of-kipf-welling-2016-2` (visited on 02/24/2019).

[28] Jie Zhou et al. “Graph Neural Networks: A Review of Methods and Applications”. In: *arXiv preprint arXiv:1812.08434* (2018).

[29] Petar Veličković et al. “Graph attention networks”. In: *arXiv preprint arXiv:1710.10903* (2017).

[30] Justin Gilmer et al. “Neural message passing for quantum chemistry”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70.* JMLR. org. 2017, pp. 1263–1272.

[31] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. “Order matters: Sequence to sequence for sets”. In: *International Conference on Learning Representations (ICLR).* 2016.

[32] Yujia Li et al. “Gated graph sequence neural networks”. In: *arXiv preprint arXiv:1511.05493* (2015).

[33] Evan N Feinberg et al. “PotentialNet for Molecular Property Prediction”. In: *ACS central science* 4.11 (2018), pp. 1520–1530.

[34] Günter Klambauer et al. “Self-normalizing neural networks”. In: *Advances in neural information processing systems.* 2017, pp. 971–980.

[35] Zhenqin Wu et al. “MoleculeNet: a benchmark for molecular machine learning”. In: *Chemical science* 9.2 (2018), pp. 513–530.

[36] David Weininger. “SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules”. In: *Journal of chemical information and computer sciences* 28.1 (1988), pp. 31–36.

[37] Andreas Mayr et al. “DeepTox: toxicity prediction using deep learning”. In: *Frontiers in Environmental Science* 3 (2016), p. 80.

[38] James S Bergstra et al. “Algorithms for hyper-parameter optimization”. In: *Advances in neural information processing systems*. 2011, pp. 2546–2554.

[39] Peter I. Frazier. “A Tutorial on Bayesian Optimization”. In: (July 2018). URL: `https://www.researchgate.net/publication/326290370_A_Tutorial_on_Bayesian_Optimization` (visited on 03/08/2019).

[40] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Advances in Neural Information Processing Systems* 4 (June 2012).

[41] Javier González et al. “Batch bayesian optimization via local penalization”. In: *Artificial Intelligence and Statistics*. 2016, pp. 648–657.

[42] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: *NIPS-W*. 2017.

[43] *RDKit: Open-source cheminformatics*. URL: `http://www.rdkit.org` (visited on 03/08/2019).

[44] Bharath Ramsundar et al. *Deep Learning for the Life Sciences*. O’Reilly Media, 2019.

[45] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python ”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[46] The GPyOpt authors. *GPyOpt: A Bayesian Optimization framework in python*. 2016. URL: `http://github.com/SheffieldML/GPyOpt` (visited on 03/08/2019).

[47] Jiaxuan You et al. “Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation”. In: *NeurIPS*. 2018.

[48] Yibo Li, Liangren Zhang, and Zhenming Liu. “Multi-Objective De Novo Drug Design with Conditional Graph Generative Model”. In: *Journal of Cheminformatics* 10 (Jan. 2018).

A

Number of epochs

Table A.1: Number of epochs used in hyperparameter optimizations.

Dataset	Number of epochs
ESOL	1200
BBBP	600
SIDER	1000
Tox21	250

B

Hyperparameter domains

Table B.1: Domain for hyperparameter optimization of ENNS2V. f_{NN} is defined in Section 3.4. Set2Vec is defined in the corresponding paper[31].

	Hyperparameter	Domain/value
Varied	Learning rate	$[10^{-5.5}, 10^{-4}]$[a]
	Message passes	$\{1, 2, \ldots, 10\}$
	Message size	$\{10, 16, 25, 40\}$
	Hidden dimension of f_{NN}	$\{50, 85, 150\}$
	Set2Vec LSTM iterations	$\{5, 6, \ldots, 12\}$[b]
	Set2Vec memory size	$\{15, 23, 35, 50\}$[b]
	First hidden dimension of output FFNN	$\{360, 450, 560\}$
	Second hidden dimension of output FFNN	$[0.2, 0.6]$[c]
	Dropout p of output FFNN	$[0.0, 0.1]$
Fixed	Batch size	50
	Weight decay	0
	Hidden layers in f_{NN}	3
	Hidden layers in output FFNN	2
	Dropout p of f_{NN}	0

[a] Log-rescaled
[b] Analogous to half GGNNs readout dimension
[c] Relative to the first hidden dimension

Table B.2: Domain for hyperparameter optimization of AttentionGGNN. The model specific neural network blocks are defined in Section 3.4 or 4.2.

	Hyperparameter	Domain/value
Varied	Learning rate	$[10^{-5.5}, 10^{-4}]$[a]
	Message passes	$\{1, 2, \ldots, 10\}$
	Message size	$\{10, 16, 25, 40\}$
	Hidden dimension of $f_{\mathrm{NN}}^{(e_{vw})}$	$\{50, 85, 150\}$
	Hidden dimension of $g_{\mathrm{NN}}^{(e_{vw})}$	$\{50, 85, 150\}$
	Hidden dimension of f_{NN}	$\{15, 26, 45, 80\}$
	Hidden dimension of g_{NN}	$\{15, 26, 45, 80\}$
	Readout dimension of f_{NN} and g_{NN}	$\{30, 45, 70, 100\}$
	First hidden dimension of output FFNN	$\{360, 450, 560\}$
	Second hidden dimension of output FFNN	$[0.2, 0.6]$[b]
	Dropout p of output FFNN	$[0.0, 0.1]$
Fixed	Batch size	50
	Weight decay	0
	Hidden layers in $f_{\mathrm{NN}}^{(e_{vw})}$, $g_{\mathrm{NN}}^{(e_{vw})}$, f_{NN} and g_{NN}	2
	Hidden layers in output FFNN	2
	Dropout p of $f_{\mathrm{NN}}^{(e_{vw})}$, $g_{\mathrm{NN}}^{(e_{vw})}$, f_{NN} and g_{NN}	0

[a] Log-rescaled

[b] Relative to the first hidden dimension

Table B.3: Domain for hyperparameter optimization of EdgeModel. Message f_{NN} and g_{NN}, and $f_{\mathrm{NN}}^{\mathrm{emb}}$, are defined in Section 4.3. Gather f_{NN} and g_{NN} are defined in Section 3.4.

	Hyperparameter	Domain/value
Varied	Learning rate	$[10^{-5.5}, 10^{-4}]$[a]
	Message passes	$\{1, 2, \ldots, 8\}$
	Hidden dimension of $f_{\mathrm{NN}}^{\mathrm{emb}}$	$\{60, 105, 180\}$
	Edge embedding size	$\{30, 50, 80, 130\}$
	Hidden dimension of message f_{NN}	$\{50, 85, 150\}$
	Hidden dimension of message g_{NN}	$\{50, 85, 150\}$
	Hidden dimension of gather f_{NN}	$\{15, 26, 45, 80\}$
	Hidden dimension of gather g_{NN}	$\{15, 26, 45, 80\}$
	Readout dimension of gather f_{NN} and g_{NN}	$\{30, 45, 70, 100\}$
	First hidden dimension of output FFNN	$\{360, 450, 560\}$
	Second hidden dimension of output FFNN	$[0.2, 0.6]$[b]
	Dropout p of output FFNN	$[0.0, 0.1]$
Fixed	Batch size	50
	Weight decay	0
	Hidden layers in $f_{\mathrm{NN}}^{\mathrm{emb}}$	2
	Hidden layers in message f_{NN} and g_{NN}	2
	Hidden layers in gather f_{NN} and g_{NN}	2
	Depth of output FFNN	2
	Dropout p of $f_{\mathrm{NN}}^{\mathrm{emb}}$	0
	Dropout p of message f_{NN} and g_{NN}	0
	Dropout p of gather f_{NN} and g_{NN}	0

[a] Log-rescaled

[b] Relative to the first hidden dimension