M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Language Of Molecules

Evaluation of Masked Language Modelling For Learning Structure Rules In Discrete Molecular Graphs

Jeppe Johan Waarkjær Olsen (s180213)

Supervisor and Co-supervisors:
Mikkel Nørgaard Schmidt (mnsc@dtu.dk)
Alexander Rosenberg Johansen (aler@dtu.dk)
Peter Bjørn Jørgensen (pbjo@dtu.dk)
Ole Winther (olwi@dtu.dk)

Kongens Lyngby 2020

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

The purpose of this thesis is to investigate how language modelling can be used to learn structure rules in molecules. We present language modelling and introduce several machine learning models from the field of natural language processing. This is then extended to the domain of graphs where we propose a new attention mechanism conditioned on all nodes and edges in the graph. The models are evaluated on two datasets of molecular graphs; a simple dataset defined by the octet rule and a more challenging dataset containing ions and hypervalent molecules. From our experiments we argue that our transformer models have the ability to learn not only the octet rule; existence of ions; and hypervalent molecules, but also to discriminate between elements – that would be considered equal under the octet rule – to some extent, even when no information is given about the bond order.

Abstract (Danish)

Formålet med denne afhandling er, at undersøge hvordan sprogmodeller kan bruges, til at lære regler omkring strukturer i molekyler. Vi præsenterer sprogmodellering og introducerer adskillige metoder fra sprogteknologi. Disse er derefter udvidet, til at virke på grafer, hvor vi foreslår en ny "attention" mekanisme, der tager højde for alle noder og forbindelser i grafen. Vi evaluerer vores modeller på to datasets. Et simpelt dataset, der er defineret af oktet reglen, og et mere udfordrende dataset, som indeholder ioner og hypervalente molekyler. Ud fra vores eksperimenter, argumenterer vi for, at vores transformer modeller har evnen til at lære både oktet reglen, eksistensen af ioner og hypervalente molekyler, men også til en hvis grad, at kunne diskriminere imellem grundstoffer, som burde være ens for oktet reglen. Vel og mærke uden, at give noget information om bindingstyperne i molekylet.

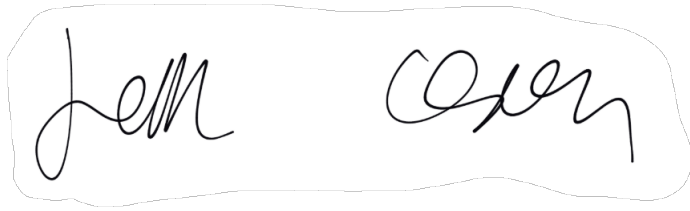
Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a M.Sc degree in Engineering.

The thesis consists of an introduction, theory sections on: basis of machine learning; natural language processing; and machine learning on molecular graphs, explanation of the experimental setup, a discussion of the experimental results and finally a conclusion.

The work done in this project has resulted in a paper submission – currently under review for the Journal of Chemical Information and Modeling. Due to the overlap in figures and tables, we will cite this as Olsen et al.[36]

Kongens Lyngby, January 1, 2020



Jeppe Johan Waarkjær Olsen (s180213)
Supervisor and Co-supervisors:
Mikkel Nørgaard Schmidt (mnsc@dtu.dk)
Alexander Rosenberg Johansen (aler@dtu.dk)
Peter Bjørn Jørgensen (pbjo@dtu.dk)
Ole Winther (olwi@dtu.dk)

Acknowledgements

I would like to thank my supervisors: Mikkel Nørgaard Schmidt, Peter Bjørn Jørgensen and Ole Winther. A special thank goes to Alexander Rosenberg Johansen and Jose Juan Almagro Armenteros for providing ideas and feedback; and creating a community for students interested in deep learning at DTU. Finally, i would also like to thank Martin Hangaard Hansen for giving insights into the field of material physics.

Contents

CHAPTER 1

Introduction

Many large industries are reliant on the properties and rules of molecules. A deeper understanding of these properties and rules may greatly benefit a variety of industries; from the medicinal and chemical production, to the creation of more efficient batteries.This can be estimated using quantum mechanics, but these calculations can be very computationally demanding. Since the amount of possible molecule configurations grows exponentially with the size of the molecule (estimated $10^{23} - 10^{60}$ potential drug-like molecules[10]), we are interested in finding computational efficient methods for either generating good candidate molecules, filter unstable molecules, or approximating the properties of molecules. In this thesis we will focus on estimating the stability of a molecule, i.e the probability of its existence, which can be used as a screening tool. Specifically we will use the recent developments in language modelling from Natural Language Processing (NLP) to learn structure rules of the molecules.

The thesis will first introduce the reader to relevant background theory in Chapter 2-4. Here, we will first define the basics of machine learning, then give an introduction to NLP. Finally, Chapter 4 is on the topic of machine learning on molecules, and how NLP can be used to solve some problems. Chapter 5 will describe the experimental setup, and tools used for the project. In Chapter 6, the experiments and results of these are presented and discussed. Finally in Chapter 7, we conclude on our findings, and discuss future work. The code produced in the project is publicly available [1]

1.1 Literature Review and Motivation

Machine learning has gained considerable attention in the field of computational chemistry, where it has been proven useful in several sub-fields. In *High Throughput Virtual Screening* (HTVS)[42] we are interested in using computationally efficient methods for reducing a large database of molecules to a smaller set of potential candidates. This is often done through a series of *computational funnels*, where we at each funnel remove molecules that are not deemed interesting. By using increasingly precise – but more computational demanding – methods at each funnel, we try to avoid wasting time. Machine learning has been used in this setting to give estimates on the properties of molecules, which is considerable faster than methods based on

[1] `https://github.com/jeppe742/language_of_molecules`

quantum mechanical calculations. A common dataset for this is the QM9 [43], where several methods have achieved very good approximations [48, 11, 24]

The databases used for HTVS are often generated based on heuristics designed by chemists. Instead, it would be beneficial if we could directly generate good candidate molecules based on some desired set of properties. This paradigm has seen interest, with the advances in generative machine learning models. Meta-heuristic methods like evolutionary algorithms have been used to generate novel molecules, biased towards certain properties [63, 45]. These methods, however, rely on expert crafted fitness functions and genotypes, which can be very non-trivial and problem specific. Instead, we can use advances in generative machine learning models, which are more data-driven. These methods include *Variational Auto Encoders* (VAE), *Generative Adversarial Networks* (GAN) and *Reinforcement Learning* (RL).[46]

Whereas state of the art property prediction relies on continuous 3D representations of the molecules, these representations are often not suited for generative models. First of all, 3D representations – e.g x,y,z coordinates of the atoms – require optimizing the structure of the molecule. Secondly, discrete representations are much more scalable, as a set of atoms now only has a finite number of configurations. A common representation is the SMILES string[56], which encodes the 2D molecular graph as a string. Gomez et al.[12] showed that using VAEs could encode SMILES strings into a latent space, and by using Gaussian processes, molecules could be sampled and optimized towards a desired property. Similarly, Guimaraes et al.[14] proposed using GANs to create a generative model using SMILES. The objective function could then be optimized using RL to bias the generation towards desired properties.

There has been a recent development towards replacing SMILES as the molecule representation. It is known that SMILES has several issues, like being non-unique, and not encoding the permutation invariance of molecules; several SMILES strings may represent the same molecule. Furthermore, slight changes in the molecule might lead to completely different SMILES, thus not encoding similarity well. Instead, we could use undirected graphs with discrete edges [10]. This can be seen in Junction-Tree VAE[23] and MolecularRNN[41], where a graph-based representation was used to generate molecules. These two works also address a known problem with molecular generation; the generated molecules might not fulfill the laws of physics, and thus cannot exists.

The solution to generating valid molecules has been to apply constraints to the molecules generated, such as constraints on the valence of atoms in MolecularRNN[41]. Another practical solution has been to simply check the validity of molecules using molecular toolkits like Rdkit[27] or OpenBabel[35], and discard invalid samples.[12] This however relies on handcrafted constraints. It is for example well known, that the valence constraint used in MolecularRNN does not hold for all molecules; there exists both electron deficient molecules like BF_3 and hypervalent molecules like SF_6.[29]

Similarly, molecular toolkits have certain assumptions when checking validity of molecules; they often fill in Hydrogen in the molecule, to fulfill their assumptions. Another limitation is that these assumption of validity do not say anything about how likely the molecule is to exist.

Inspired by NLP, we will extend language modelling to graphs to solve the problem of determining the probability of a molecule existing. Specifically, we will use masked language modelling, as in BERT [9], using an adaptation of the transformer architecture [52]. Language modelling has previously seen some uses on molecules. Segler et al.[49] proposed using a language model to generate molecules targeted at certain properties. This model did, however, use RNNs and SMILES, instead of modelling graphs directly. Language modelling has also shown promise as a pre-training task. Hu et al.[20] included masked language modelling – inspired by BERT [9] – as a set of pretraining methods for graphs. Other NLP pretraining tasks have also been introduced for molecules. Sabrina et al. introduced Mol2Vec [22], which is inspired by Word2Vec[33], to learn strong atom embeddings, used for downstream supervised learning. Little focus has, however, been put on investigating language modelling itself.

Language modelling can therefore be useful for several different purposes:

- For HTVS, a language model can be used as yet another screening method, to filter unstable molecules. Since we use a discrete 2D graph representation, this method is highly scalable.
- In generative models, we could either use a language model to discard unlikely molecules, generated from existing models, or as a generative model itself.
- When analyzing, optimizing, and/or generating molecules, many different formats and tools are used. It is not guaranteed that conversions between different formats with different tools result in the same molecular graph. For example, when generating QM9, Ramakrishnan et al. [43] used a consistency check to try to avoid this. Here, language modelling could be used to find graphs that might have artifacts.
- Finally, language modelling can be used as a pre-training method to improve downstream tasks.

1.2 Problem Statement

In this work we will investigate language modelling on molecular graphs, where the goal is to learn the underlying rules governing the structure of molecules. These rules are often based on valence theory and the octet rule, so we will gauge the ability of several models to learn this structure rule. We shall also investigate if language

modelling can be used to learn more complex structure rules than the octet rule. This could be learning rules related to hypervalent molecules, or ions.

1.3 Notation

Through out the thesis, we will use the following notation. Bold lower case letters are vectors $\boldsymbol{x} = (x_1, x_2, \ldots, x_N)^T$, where plain lower case x_1 are scalars. Matrices will be denoted by bold capital letters $\boldsymbol{X} = (\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_N})^T$.

CHAPTER 2
Machine Learning

2.1 Basics of Learning Algorithms

Machine learning is a multidisciplinary field that borrows concepts from other fields like optimization, statistics, probability theory, and computer science. In general, the purpose of machine learning is to extract patterns or information from data. Several subfields of machine learning exists, but two of the most common ones are *supervised learning* and *unsupervised learning*.

In *supervised learning*, you are interested in estimating a target variable y, given some input data x. The target could both be either a continuous or categorical variable. In practice this could be estimating the price of a house given basic information like number of floors and location, or classifying digits in images [28, 15].

More formally, the objective can be defined as given an observation of K input features $\mathbf{x} = (x_1, x_2, \ldots, x_K)^T$, we want to estimate the conditional probability of the target variable y, given the input:

$$P(y|\mathbf{x}). \tag{2.1}$$

The target variable y might, however, not always be available, but this does not mean that we cannot extract information from the data. *Unsupervised learning* deals with this case by trying to learn the underlying structure of the data. For example, this could include trying to cluster data into groups, or detecting outliers. The objective of *unsupervised learning* can often be formulated as estimating the joint probability distribution for the input data:

$$P(\mathbf{x}). \tag{2.2}$$

This distinction between supervised and unsupervised learning is, however, not always as strict, as unsupervised methods often employ elements of supervised learning. There even exists methods like *semi-supervised learning* and *active learning*, where a mix of supervised and unsupervised often is applied.

2.1.1 Models

Calculating the probability distributions from (2.1) and (2.2) are often impossible or practically infeasible, which is why we, in machine learning, apply models to approx-

imate the distributions. That is, we want to find a probability distribution function g such that, in the case of *supervised learning*:

$$P(y|\mathbf{x}) \approx G(y|\mathbf{x}). \tag{2.3}$$

In general, the models applied in machine learning can be classified as *parametric* or *non-parametric*. The *non-parametric* models are not defined by not having parameters, as the name might imply. Instead, they are defined by the approximation function not being fixed beforehand, but might depend on the size of the dataset. This could for example be taking the majority vote among the k nearest neighbours in feature space, which is the case of the KNN (K nearest neighbours) model [5]. Non-parametric models will not be discussed further in this work.

Parametric models, as opposed to *non-parametric* models, defines a fixed model structure with fixed number of parameters. This could be using a linear function, as in the case of linear and logistic regression for *supervised learning* or approximating the distribution of the data, e.g. using a Gaussian distribution for *unsupervised learning*. To indicate that the approximation function is described by some parameters $\boldsymbol{\theta}$, it is often denoted by $G(y|\mathbf{x}, \boldsymbol{\theta})$.

Since the parameters are continuous variables, we have to define how we select the best values. Following our probabilistic approach, one way of optimizing parameters $\boldsymbol{\theta}$ for some model is to maximize the probability of the parameters, given the data, called the *posterior distribution*. Using Bayes theorem, the posterior can be written as :

$$P(\theta|\mathbf{X}, \mathbf{y}) = \frac{P(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})P(\boldsymbol{\theta}|\mathbf{X})}{P(\mathbf{y}|\mathbf{X})}. \tag{2.4}$$

Since the denominator does not depend on $\boldsymbol{\theta}$, it is often ignored during optimization:

$$P(\theta|\mathbf{X}, \mathbf{y}) \propto P(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})P(\boldsymbol{\theta}|\mathbf{X}). \tag{2.5}$$

Parameters estimated using this method are called *Maximum A Posteriori* (MAP) estimates

$$\theta_{MAP} = \arg\max_{\theta} P(\theta|\mathbf{X}, \mathbf{y}). \tag{2.6}$$

The MAP estimate requires us to provide a probability distribution over the parameters $P(\boldsymbol{\theta}|\boldsymbol{X})$, called the *prior*. The prior corresponds to restricting what values we believe are probable for the parameters. Sometimes, however, we have no beliefs about the parameters, and thus all values are given the same probability, which corresponds to a infinitely wide uniform distribution. This corresponds to relaxing the problem by only maximizing the likelihood, called *Maximum Likelihood Estimation* (MLE). That is:

$$\theta_{MLE} = \arg\max_{\theta} P(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}). \tag{2.7}$$

In the next section, linear regression will be introduced as an example of how to estimate the parameters.

2.2 Regression with Gaussian approximation

Choosing a good approximation function $G(y|\mathbf{x}, \boldsymbol{\theta})$ can be very specific to the dataset in question. For regression, one very common approximation is, however, to assume, that the data is independent and identically distributed (iid), as a Gaussian distribution, with a mean described by some function, $f(\mathbf{x}, \boldsymbol{\theta})$, and variance β^{-1}:

$$G(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|f(\mathbf{x}, \boldsymbol{\theta}), \beta^{-1}). \tag{2.8}$$

The Gaussian distribution has several pleasant properties like being fully described by its mean and variance, and being simple enough that we can find analytical solutions to many problems. In practice, this assumption is often realistic, due to the Central Limit Theorem, which states that under certain assumptions, the sum of random variables tends to approach a Gaussian distribution as the number of variables tends to infinity. [5]

To use the MAP estimate, we can assume that the parameters $\boldsymbol{\theta}$ also are Gaussian distributed $P(\boldsymbol{\theta}|\mathbf{X}) = \mathcal{N}(\boldsymbol{\theta}|0, \alpha^{-1}\mathbf{I})$. Using (2.8), we can now calculate the MAP estimate by first multiplying the likelihood and prior,

$$P(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})P(\boldsymbol{\theta}|\mathbf{X}) = \prod_{i}^{n} \mathcal{N}(y_i|f(\mathbf{x}_i, \boldsymbol{\theta}), \beta^{-1})\mathcal{N}(\boldsymbol{\theta}|0, \alpha^{-1}\mathbf{I}). \tag{2.9}$$

For numerical purposes, it is beneficial to optimize the negative log, as the product of many small small numbers might lead to underflows or loss of precision. Taking the negative log of (2.9), and removing terms that do not depend on $\boldsymbol{\theta}$, gives

$$E(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i}^{n} \{f(\mathbf{x}_i, \boldsymbol{\theta}) - y_i\}^2 + \frac{\lambda}{2}\boldsymbol{\theta}^T\boldsymbol{\theta}, \tag{2.10}$$

where we have defined $\lambda = \alpha/\beta$. λ can be seen as hyperparameter that controls the complexity of the model, as a high λ will penalize large weights $\boldsymbol{\theta}$. The parameter is in practise not determined using the same data, as was used to find $\boldsymbol{\theta}$, as will be discussed in Section 2.4. If we had used the MLE by assuming a uniform prior for $\boldsymbol{\theta}$, we see that parameters found by MLE are equal to the ones found by minimizing the sum of squared errors (SSE), if you assume your data to be described by a Gaussian distribution as in (2.8). This also called the least squares method.

2.2.1 Linear regression

In Linear regression, we further assume that the function f is linear in the parameters, e.g. $f(\mathbf{x}, \boldsymbol{\theta}) = x_1\theta_1 + \exp(x_2)\theta_2$ is still a linear function. In general, these functions can be written as

$$f(\mathbf{x}, \boldsymbol{\theta}) = \theta_0 + \sum_{i=1}^{K} \phi_i(\mathbf{x})\theta_i, \tag{2.11}$$

where $\phi_i(\mathbf{x})$ is a basis function. Most often this is defined as $\phi_i(\mathbf{x}) = x_i$. This can also be written as a vector product if we redefine $\phi_0(\mathbf{x}) = 1$,

$$f(\mathbf{x}, \boldsymbol{\theta}) = \boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x}). \tag{2.12}$$

Inserting (2.12) into (2.10), we can find a closed-form solution, for the parameters $\boldsymbol{\theta}$

$$\boldsymbol{\theta}_{MAP} = \left(\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \lambda\mathbf{I}\right)^{-1}\boldsymbol{\Phi}^T\mathbf{y}, \tag{2.13}$$

where we have defined $\boldsymbol{\Phi} = (\boldsymbol{\phi}(\mathbf{x}_0), \boldsymbol{\phi}(\mathbf{x}_1), \ldots, \boldsymbol{\phi}(\mathbf{x}_K))^T$

2.3 Classification with categorical distribution

In classification, your target variable is confined to a set of discrete classes $y \in \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_M\}$. This means that we have to use a discrete probability distribution, like the categorical distribution

$$G(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{C}at(y|\mathbf{p}), \tag{2.14}$$

where $\mathbf{p} = (p_1, p_2, \ldots, p_M)$ is the vector of corresponding probabilities for each category, i.e. $P(y = \mathcal{C}_i) = p_i$. This means that we now have to estimate the probabilities $\mathbf{p}$ with some function $f(\mathbf{x}, \boldsymbol{\theta}) : \mathbb{R}^K \to \mathbb{R}^M$. Note that the function should map into the simplex to fulfill that $\sum_i p_i = 1$. Similar to regression, we can try to find the MAP estimate for $\boldsymbol{\theta}$. If we assume the same Gaussian prior for the weight $\boldsymbol{\theta}$, this results in

$$P(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})P(\boldsymbol{\theta}|\mathbf{X}) = \prod_{i}^{N} \mathcal{C}at(y_i|f(\mathbf{x}_i, \boldsymbol{\theta}))\mathcal{N}(\boldsymbol{\theta}|0, \alpha^{-1}\mathbf{I}). \tag{2.15}$$

Again taking the negative log, we find

$$E(\boldsymbol{\theta}) = -\sum_{i}^{N} \log f(\mathbf{x}_i, \boldsymbol{\theta})_{y_i} + \frac{\alpha}{2}\boldsymbol{\theta}^T\boldsymbol{\theta}, \tag{2.16}$$

where $f(\mathbf{x}_i, \boldsymbol{\theta})_{y_i}$ corresponds to taking the predicted probability for the class of y_i. The first term is known as *cross-entropy*, and is a central element from information theory. α is a hyperparameter, similar to what we saw in regression, which controls the model complexity.

2.3.1 Logistic regression

In the previous section, we assumed some function f that produced probabilities for the classes. Now we shall see an example of such a function. As in section 2.2.1, we will use linear functions. Linear functions, as defined in (2.11) might, however, produce values that do not fulfill the properties of probabilities; $\sum_i p_i = 1$, and $0 \leq p_i \leq 1$. To ensure these properties, we often use the *softmax* function, defined as

$$\text{softmax}(x)_j = \frac{e^{x_j}}{\sum_i^M e^{x_i}}, \quad \forall j \in \{1, \ldots, M\}. \tag{2.17}$$

To estimate our probabilities, we now just have to put the pieces together. First, we define a linear function for each class, and then take the softmax function of the output. In matrix notation this can be written as $\boldsymbol{\Theta}\boldsymbol{\phi}(\mathbf{x})$, where $\boldsymbol{\Theta} \in \mathbb{R}^{M \times K}$ is now a matrix. This gives us

$$P(y = \mathcal{C}_i | \mathbf{x}, \boldsymbol{\Theta}) = f(\mathbf{x}, \boldsymbol{\Theta})_i = \text{softmax}(\boldsymbol{\Theta}\boldsymbol{\phi}(\mathbf{x}))_i. \tag{2.18}$$

Using (2.16), we can now find the optimal parameters for $\boldsymbol{\Theta}$. The softmax function, however, causes the optimization to no longer having a closed form solution. Instead, the problem can be solved using iterative optimization methods, based on the gradient, which will be discussed in section 2.5.

So far we have tried to model the posterior class distribution $P(y = \mathcal{C}|\mathbf{x})$ directly. These model are called *discriminative* models. Using Bayes rule, we can, however, decompose the problem by instead modeling the likelihood and prior distribution

$$P(y = \mathcal{C}|\mathbf{x}) = \frac{P(\mathbf{x}|y = \mathcal{C})P(\mathcal{C})}{P(\mathbf{x})}. \tag{2.19}$$

These models are called *generative* models, since they describe how the data is generated given the class.

2.4 Generalization

Thus far, we have only looked at the performance of our models on the data we used to estimate the parameters $\mathbf{X}$. When creating predictive models, we are, however, interested in how well our model performs on unseen data, called the *generalization* of the model. If we assume that the data is generated by $\mathcal{N}(y|h(\mathbf{x}), \sigma^2)$ where $h(\mathbf{x})$ is the true underlying function of the data, the generalization error can be defined as

$$\mathbb{E}\left[f(\mathbf{x}, \boldsymbol{\theta}) - h(\mathbf{x})\right]^2, \tag{2.20}$$

where $f(\mathbf{x}, \boldsymbol{\theta})$ is our model, that tries to approximate the true function $h(\mathbf{x})$, and the expectation is taking with respect to different datasets. The generalization error can also be written as

$$\mathbb{E}\left[f(\mathbf{x},\boldsymbol{\theta}) - h(\mathbf{x})\right]^2 = \sigma^2 + \left(\mathbb{E}\left[f(\mathbf{x},\boldsymbol{\theta})\right] - h(\mathbf{x})\right)^2 + \mathrm{Var}\left[f(\mathbf{x},\boldsymbol{\theta})\right], \tag{2.21}$$

where σ^2 is the irreducible error from the noise in the data. The second term is the *bias* of the model, which indicates how far off our predictions are on average. The final term is the *variance* of the model, which describes how sensitive the model is to differences in the dataset used for estimating the parameters [16]. Bias and variance are opposing properties, meaning that as you lower the bias of your model, you most likely increase the variance and vice versa, hence the term *bias-variance-trade off*. In Figure 2.1, we see an example of linear regression, where we have generated 10 different datasets, by adding Gaussian noise with different seeds. In Figure 2.1a we see that our model is too simple, and cannot describe the data, hence the average prediction over many datasets will be off the true value, and we say that the model has high bias. The predictions are, however, fairly constant across the different datasets, so the model has low variance. Figure 2.1b shows the opposite case, where the model is too complex. The average predictions are close to the true value, but the model depends highly on the dataset, thereby having low bias, but high variance.

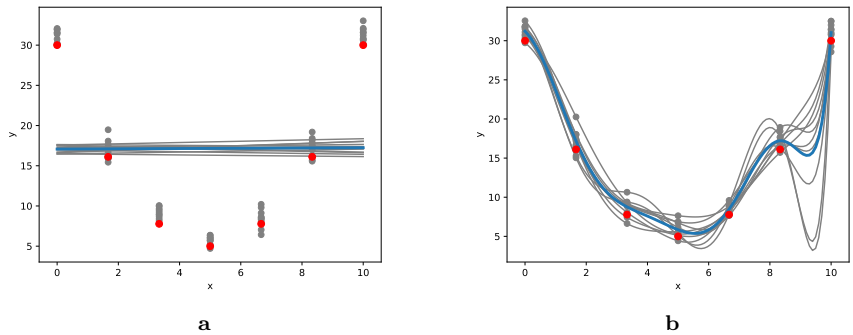


Figure 2.1: Grey lines corresponds to a linear regression model, trained on 10 datasets with random Gaussian noise. The blue line is the average prediction of the 10 models, and the red points are the true value. (a) is a first order polynomial model, where (b) is a 7th order polynomial

In Sections 2.2.1 and 2.3.1, we have seen that both Linear- and Logistic regression introduced a hyperparameter to control the complexity of the model. When using the MAP or MLE method, the hyperparameters of a model cannot be estimated on the same data that is used to estimate the parameters $\boldsymbol{\theta}$, as a sufficiently complex model should be able approximate all the datapoints, including the noise and possible outliers. This is also called *overfitting*. It is possible to estimate the hyperparameters and parameters on the same data if you calculate the full posterior distribution, instead

of just calculating a point estimate of the mode. This is called *Bayesian inference* and will not be discussed further.

Several methods exist for getting good estimates of the hyperparameters, but the simplest is to split your data into a training and validation dataset. By only estimating the parameters on the training set, any overfitting would result in poor performance on the validation set, since the model does not generalize to unseen data.

Besides changing hyperparameters, other methods exists to lower the variance of models, like increasing the number of datapoints, injecting more noise into the data, or creating an ensemble of multiple models [16].

2.5 Gradient descent

As discovered in Section 2.3.1, it often is not possible to find closed-form solutions, even for simple models. Instead, we must resort to using iterative methods to approximate the optimal parameters. The most simple methods are based on the gradient. The gradient with respect to the parameters, $\nabla E(\boldsymbol{\theta})$, points in the direction of highest rate of increase of the function; hence if we take a small step in the opposite direction, we should expect the objective function to decrease. The update to the parameters at iteration i can be written as

$$\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i - \nabla E(\boldsymbol{\theta}^i), \tag{2.22}$$

where we have chosen some initial values of the parameters $\boldsymbol{\theta}^0$. The naive gradient step often does not work that well, since the gradient is only well defined within a small region of $\boldsymbol{\theta}^i$, so if gradient is large, we might end up taking a step that is too large. This can be alleviated simply by introducing a step size hyperparameter η, also called the *learning rate*:

$$\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i - \eta \nabla E(\boldsymbol{\theta}^i). \tag{2.23}$$

2.5.1 Mini-batch gradient descent

The parameter update in Equation (2.23) requires calculating the gradient over all the training data $\mathbf{X}$. This has several downsides, e.g. being computational inefficient for large datasets. A common solution to this is just to divide your training data into batches of size n_{batch}, called the *batch-size*, and then perform a gradient step after each batch. Naturally, the gradient estimates will not be as good, as fewer datapoints are used, but this might actually be a benefit to us. Gradient descent is prone to getting stuck in local minimas, but by only estimating the gradient of a subset of the data, the specific subset might give rise to a slightly different minima, giving us a better change of escaping local minima. Furthermore, noisy gradient steps

can introduce more noise into our model, which can have regularizing effects, hence reducing the variance of the model [13].

2.6 Artificial Neural Networks

So far, we have only discussed simple models that uses a linear functions, as described in Equation (2.11). In principle, by choosing different basis functions $\phi(\mathbf{x})$, the models can have very high modeling capabilities. This, however, requires determining the functions beforehand, called *feature-engineering*, which can be non-trivial and requires trial and error. Instead, we can parameterize the basis functions, so that they can be optimized during training. In artificial neural networks, also called *feed-forward neural networks*, we use a series of non-linear transformations called *layers*. Each layer consists of a linear transformation, followed by a non-linear *activation function*. This can be written as

$$z_j = h\left(a_j\right), \tag{2.24}$$

$$a_j = \theta_{j0} + \sum_{i=1}^{N} \theta_{ji} x_i, \tag{2.25}$$

where h is a non-linear activation function, and j subscript indicates that the output size of the layer is a vector $\mathbf{z} \in \mathbb{R}^{n_h}$. This can also be written in matrix notation

$$\mathbf{z} = h\left(\mathbf{\Theta}^{(l)}\mathbf{x}\right), \tag{2.26}$$

where we have defined $x_0 = 1$, and the (l) superscript indicates the number of the layer.

If we only consider one layer in a classification scenario, the resulting approximation function can be written as

$$P(y = \mathcal{C}_i|\mathbf{x}, \mathbf{\Theta}) = \text{softmax}\left(\mathbf{\Theta}^{(2)}\mathbf{z}\right)_i = \text{softmax}\left(\mathbf{\Theta}^{(2)} h\left(\mathbf{\Theta}^{(1)}\mathbf{x}\right)\right)_i, \tag{2.27}$$

where we have defined $z_0 = 1$, to include a constant bias term in all layers. A visual representation of the network can be seen in Figure 2.2

One of the powers of the neural network is its flexibility. To increase the complexity of your model, you can simply either increase the dimension of $\mathbf{z}$, or stack multiple layers together as

$$\mathbf{z}^{(l)} = h^{(l)}\left(\mathbf{\Theta}^{(l)}\mathbf{z}^{(l-1)}\right). \tag{2.28}$$

Choosing a good activation function is its own research topic, but currently one of the most popular ones is the Rectified Linear Unit (ReLU):

$$\text{ReLU}(x) = \max(0, x). \tag{2.29}$$

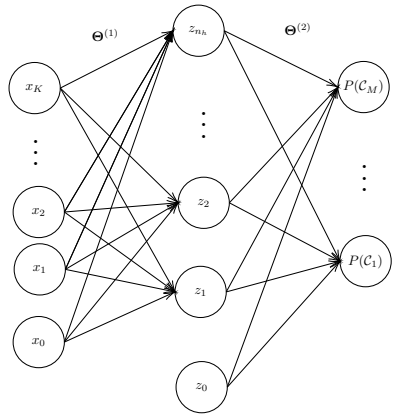


Figure 2.2: Diagram of a neural network with one hidden layer. Each connection, indicates a corresponding weight parameter

The ReLU has several convenient properties like being extremely computational efficient, and having a constant derivative of either 0 or 1, which can help stabilize the gradients [13].

2.6.1 Backpropagation algorithm

With our model defined, we need to optimize the parameters. Since the layers of the neural network contain non-linear functions, we must resort to iterative methods, of which gradient descent as described in Section 2.5 is popular. To use gradient descent, we need to find the gradients of the parameters with respect to our objective function $E(\boldsymbol{\Theta})$. To generalize the formulation, we first notice that our one layer model from Equation (2.27) can be written using only the terms in Equation (2.28), if we let $\mathbf{z}^{(0)} = \mathbf{x}$ and $h^{(2)} = \text{Softmax}$. The partial derivative of any parameter in the network can now be written using the chain rule:

$$\frac{\partial E}{\partial \Theta_{ji}^{(l)}} = \frac{\partial E}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial \Theta_{ji}^{(l)}}. \tag{2.30}$$

The third term can easily be found

$$\frac{\partial a_j^{(l)}}{\partial \Theta_{ji}^{(l)}} = \frac{\partial}{\partial \Theta_{ji}^{(l)}} \left(\sum_i \Theta_{ji}^{(l)} z_i^{(l-1)} \right) = z_j^{(l-1)}. \tag{2.31}$$

Similar, the second term is simply the derivative of the activation function

$$\frac{\partial z_j^{(l)}}{\partial a_j^{(l)}} = \frac{\partial}{\partial a_j^{(l)}} h^{(l)}\left(a_j^{(l)}\right) \tag{2.32}$$

For the first term, there are two possibilities. If $z^{(l)}$ is the final output layer of the model, then we can simply calculate the partial derivative of the objective function. If the layer is one of the intermediate, called *hidden* layers, then the derivative depends on the derivative of the following layers. To accommodate this, we can simply apply the chain rule once again

$$\frac{\partial E}{\partial z_j^{(l)}} = \sum_k^g \frac{\partial E}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k^g \frac{\partial E}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k^g \frac{\partial E}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial a_k^{(l+1)}} \Theta_{kj}^{(l+1)}, \tag{2.33}$$

where the sum over k corresponds to the g neurons that use $z_j^{(l)}$ as input. The partial derivative with respect to the parameters can now be written as

$$\frac{\partial E}{\partial \Theta_{ji}^{(l)}} = \frac{\partial E}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial a_j^{(l)}} z_j^{(l-1)} \tag{2.34}$$

$$= \delta^{(l)} z_j^{(l-1)}, \tag{2.35}$$

where we have introduced

$$\begin{aligned} \delta^{(l)} &= \frac{\partial E}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial a_j^{(l)}} \\ &= \begin{cases} \frac{\partial E}{\partial z_j^{(l)}} \frac{\partial}{\partial a_j^{(l)}} h^{(l)}\left(a_j^{(l)}\right) & \text{if } (l) \text{ is an output layer} \\ \sum_k^g \Theta_{kj}^{(l+1)} \delta^{(l+1)} \frac{\partial}{\partial a_j^{(l)}} h^{(l)}\left(a_j^{(l)}\right) & \text{if } (l) \text{ is a hidden layer} \end{cases} \end{aligned} \tag{2.36}$$

$\delta^{(l)}$ is often called the *error* [5].

We see that for hidden layers we must recursively apply Equation (2.36) until we reach the output layer. In practise, however, it is much more efficient to calculate the error of the output layer, and traverse backwards in the network, keeping all the $\delta^{(l)}$ in memory. This is also where the algorithm gets its name, *backpropagation*, from [13]. Backpropagation, together with gradient descent, allows one to take advantage of the flexibility of neural networks. We can alter topology of the network by changing activation function, the size or number of layers, and easily get the gradients. In Chapter 3, we shall introduce more complex network architectures, which still can be optimized using backpropagation and gradient descent.

CHAPTER 3

Natural Language Processing

Natural language processing (NLP) is the study of how to make computers process, understand and interact with natural language. In this chapter, we will first define the topic of language modelling and then give an introduction to some of the contemporary machine learning models used in NLP.

3.1 Language Modelling

Languages can often be classified either as a *formal language* or a *natural language*. Whereas formal languages – like programming languages – have a very well defined structure, natural language can be hard to model. For most natural languages, like English, there is some grammar rules and heuristics but the languages are often too complex to be fully described by these due to things like ambiguity, and the fact that languages are continuously evolving. For example, the following sentences are grammatically and semantically correct, but contains extreme ambiguity.[1]

"Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo"

"James while John had had had had had had had had had had had a better effect on the teacher"

"Can can can can can can can can can can"

Even though above sentences are linguistically correct, most would agree, that such sentences should be considered linguistic puzzles, and would never be used in a normal conversation.

Instead of creating linguistic models we can use a data-driven approach to model the languages. Specifically, for a statistical language model we are interested in modeling the probability of certain sequences of words. Given a sequence of N tokens $w_1, w_2, \ldots, w_N$, we are interested in modeling the joint probability of the sequence

[1] `https://en.wikipedia.org/wiki/List_of_linguistic_example_sentences`

$$P(w_1, w_2, \ldots, w_N). \tag{3.1}$$

3.2 N-gram Models

Estimating the joint probability in Equation (3.1) is often not possible, but using the chain rule, we can decompose the problem as

$$P(w_1, w_2, \ldots, w_N) = P(w_1)P(w_2|w_1)P(w_3|w_1^2)\ldots P(w_N|w_1^{N-1}), \tag{3.2}$$

where w_i^j defines the sequence of words from token i to j: $w_i, w_{i+1}, \ldots, w_{j-1}, w_j$. [25] It's worth noting that here we have assumed the sequence to have a directional dependence; e.g. w_2 depends on w_1 but w_1 does not depend on w_2.

Equation (3.2) can be extended to the probability of an entire text corpus, by including a product over all sentences. This form is similar to what we saw in Section 2.3. If we first do not assume a prior the maximum likelihood estimate just corresponds to minimizing the cross-entropy. The conditional probabilities of each word – given its history – estimated using maximum likelihood turns out to simply be the relative frequencies,

$$P(w_i|w_1^{i-1}) = \frac{\text{count}(w_1^{i-1}, w_i)}{\text{count}(w_1^{i-1})}. \tag{3.3}$$

The possible number of combinations of words does, however, mean that it is infeasible to simply estimate the probabilities by counting. Even with all text available on the web some specific sentences might have no records, even though they are perfectly reasonable. In a *n-gram* model we solve this problem by approximating the history to only include the last n words [25].

$$P(w_i|w_1^{i-1}) \approx P(w_i|w_{i-n+1}^{i-1}). \tag{3.4}$$

E.g given the sentence "The quick brown fox jumps over the lazy dog", a *bigram* model (n=2) would estimate the probability of "dog" given the history as

$$P(\text{dog}|\text{The quick brown fox jumps over the lazy}) \approx P(\text{dog}|\text{lazy}). \tag{3.5}$$

3.2.1 Evaluation

The formulation of the language model is very similar to a classification model. Languages does however have a distinct difference compared to most classification problems. As mentioned, languages have a lot of ambiguity; the same input could give rise to many outcomes. E.g given the sentence "The house is", the next word could be "large","red" or "cold", all likely candidates. When training and evaluating the

model we only have access to the target of specific sentence realizations, which penalizes the model if it does not give 100% to the specific target. However, by averaging over the entire dataset our model should learn probabilities that corresponds to the likeliness of the targets in the data.

Since we are predicting probabilities, it is common to use a different metric than accuracy, to access the model performance. Perplexity is a common language modelling metrics. Like cross-entropy, it has roots in information theory, and is defined as the exponential of the average sample cross-entropy [25]

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i}^{N}\log P(w_i)\right), \tag{3.6}$$

where $P(w_i)$ is probability of the ith sample word in the dataset, given some model, and N is the total number of samples.

3.2.2 Smoothing

By reducing the effective history with a n-gram model, we have a trade off between using longer history for more contextual information, versus getting better estimates of the probabilities due to more examples with shorter history. Even with short history, due to limited training data, we are however not guaranteed to have examples of all possible n-grams. This causes the issue that the probabilities for n-grams not in the training dataset are estimated to be 0, and we might end up getting infinite perplexities.

The problem of zeros can be solved by introducing a smoothing factor in our frequency estimate. By adding k occurrences to all words we are guaranteed that probability of any word is always nonzero

$$P(w_i|w_1^{i-1})_{k-smooth} = \frac{\text{count}(w_{i-n+1}^{i-1}, w_i) + k}{\text{count}(w_{i-n+1}^{i-1}) + V}, \tag{3.7}$$

where V is the size of our vocabulary, i.e the total number of distinct words. This smoothing corresponds to using a Dirichlet prior over the probabilities, and using the MAP estimate [25]. k is hyperparameter that controls the bias and variance of the model. As we increase k we go towards a uniform distribution with high bias and low variance, and for $k = 0$ we saw that the probabilities for rare words could be very dependent on the dataset, hence having high variance.

3.3 Embeddings

Instead of relying solely on frequency counts, another way of modeling the word probabilities is to introduce a model with tunable parameters. In order to do this we first

have to find a way of representing the words, as our methods only works on numerical input. A naive approach would be to just encode a given word as the number it appears in a dictionary. This however does not encode any information about the relations between words.

Instead we should try encode words as vectors. The simplest way of doing this is to have a vector with all zeros except for the index of the given word. This is called *one-hot representation* [25]. The English language does however contain a large amount of words so the dimension of the vector could be very large and the effectiveness limited by the curse of dimensionality [5]. Instead we should stride to use dense vectors.

To encode words as dense vectors, called *word embeddings*, we use an *embedding* function. By using trainable embedding functions we can optimize the location of words in the embedding space. Examples of embedding functions range from shallow neural networks [34, 39] to large networks that also encodes contextual information[40, 9], but shares that language modelling and various training techniques are used to encode relationships between words.

3.4 Bag of Words

One problem with using parameterized models on languages is that the input sequences are not necessarily the same length. This means that we cannot simply use logistic regression since we would need a different number of parameters depending on the length of the sequence. A simple approach, is to use some aggregation function to convert the input into a fixed length before passing it to our model. This is called a *Bag of Words* (BoW) model. This can be seen in Figure 3.1 where each word first is encoded as $\boldsymbol{x_i}$ using some embedding function and then all the word embeddings are aggregated into a single fixed size variable $\boldsymbol{z}$

$$\boldsymbol{z} = f(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_N}), \tag{3.8}$$

where $f : \mathbb{R}^{k,N} \to \mathbb{R}^k$ is some aggregation function.

The encoding $\boldsymbol{z}$ can be used as input to additional models or we can simply take the softmax and get the class probabilities directly. $\boldsymbol{z}$ can be seen as an encoding of the sentence where we only have information about the words that make up the sentence but not their order. Common aggregation functions include the mean and sum where the later also can include information about the length of the sentence. The BoW model may seem very naive but the temporal dependence might not always be necessary. In the sentence in Figure 3.1 one could imagine that simply knowing that the sentence included the words ”sky” and ”colour”, the word ”blue” could be a reasonable guess for the next word in the sentence.

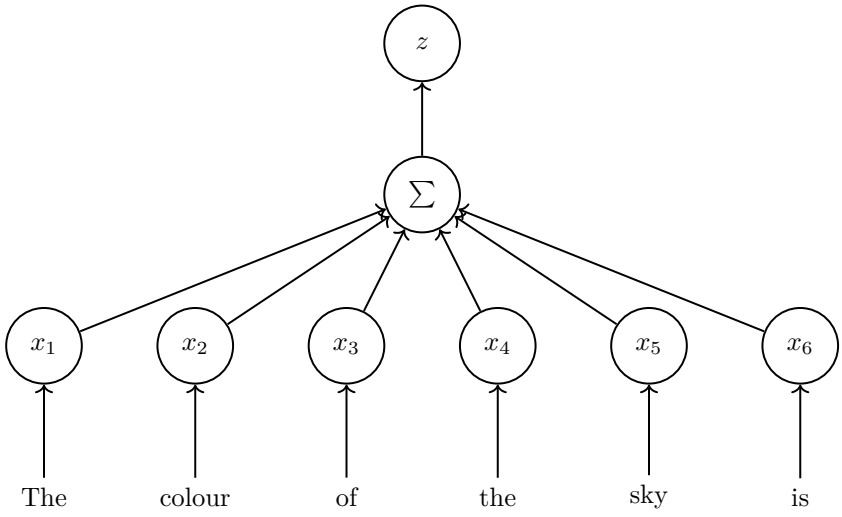


Figure 3.1: Bag of Words (BoW) model

3.5 Recurrent Neural Networks

The assumption of the Bag of Words model – that we can represent a sentence without considering the order of the words – might work in some cases but if we want to create a truly expressive representation we need to include the sequential information. For example, the following two sentences have the same bag of words representation but arguably very different interpretation.

"The boy broke the stick"
"The stick broke the boy"

To include the temporal information we need to find a way of inputting variable length sequences into a model with fixed number of parameters. This can be done by noting that the encoding of a word, called the *hidden representation*, given its context can be written as a auto-regressive function

$$h_i = f(w_i, h_{i-1}), \tag{3.9}$$

where h_i is the hidden representation of the ith word. This can be seen by applying the function recursively

$$h_3 = f(w_3, h_2) = f(w_3, f(w_2, f(w_1, h_0)), \tag{3.10}$$

where h_0 often is initialized as some default value [13]. Here we see that the value of h_3 depends on all the previous words in the sequence. In Recurrent Neural Networks

(RNN) we use a shallow neural network as the encoding function f. Such a simple RNN can be described by the following equations[13]

$$\boldsymbol{h_i} = \tanh(\boldsymbol{\Theta_x x_i} + \boldsymbol{\Theta_h h_{i-1}} + \boldsymbol{\theta_0}) \tag{3.11}$$

$$\boldsymbol{z_i} = \boldsymbol{\Theta_z h_i} + \boldsymbol{\theta_1} \tag{3.12}$$

We note that the parameters are shared between each time step giving us a model with fixed number of parameters. This can also be seen in Figure 3.2 for an example sentence.

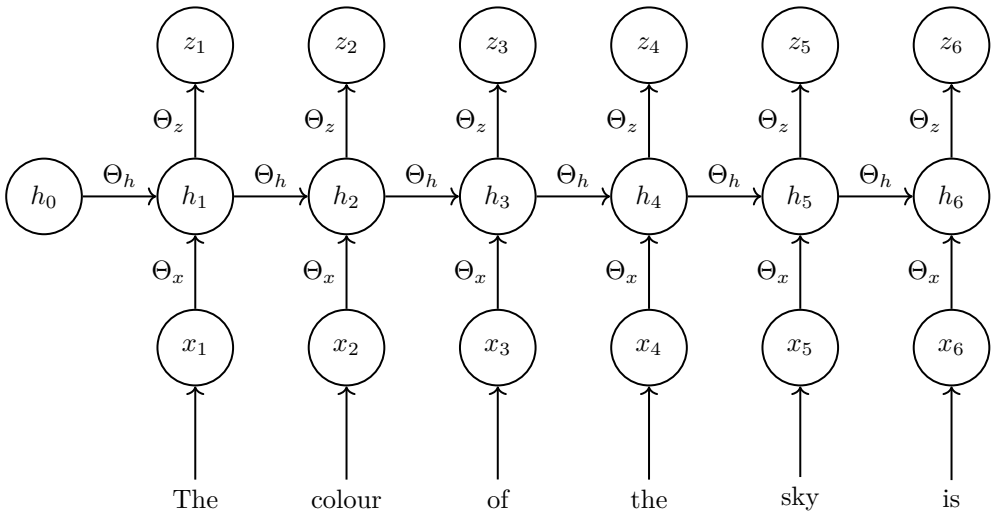


Figure 3.2: Recurrent Neural Network (RNN) model

Finally, $\boldsymbol{z_i}$ can be used for further transformations, like stacking another layer on top, or turned into class probabilities by using the softmax function. In our case of language modelling we would use the final hidden state as the representation of the context of the sentence.

$$P(w_i = \mathcal{C}_j | w_1^{i-1}) = \text{softmax}(\boldsymbol{z_i})_j \tag{3.13}$$

Recurrent neural networks can be trained with gradient descent and backpropagation. Since the parameters are shared between all time steps, we need to aggregate the contributions for each time step. This extension is called *backpropagation through time* (BPTT) [13].

3.6 Attention Mechanism

While recurrent neural networks have the capability to handle sequences of arbitrary length they have several weaknesses. Their auto-regressive nature means that as the sequences gets longer, modeling long term dependencies becomes harder as the start of the sequence are many time steps in the past [8]. Many proposed solutions have been created, like using Gated Recurrent Units (GRU) or Long-Short-Term-Memory (LSTM) cells [19, 7]. These solutions does however not address the fundamental problem; that information still have to flow through the entire network. This path in the network can be seen in Figure 3.3a

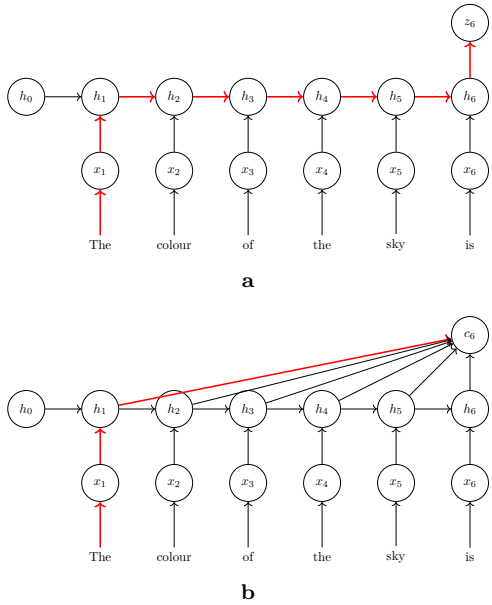


Figure 3.3: Path of information flow in recurrent neural networks. The red path is the shortest path of the information from the first word to the final sentence encoding. (a) is classic RNN, and (b) is a RNN extended with an attention mechanism over the hidden states

A solution to this was proposed by Bahdanau et al. called the *attention* mechanism [3]. Instead of only using the last hidden state of the network h_N as the encoding of the sentence the attention mechanism introduces a *context vector*, $\boldsymbol{c_i}$, which is a weighted sum of all the intermediate states $h_1, \ldots, h_{i-1}$.

$$\boldsymbol{c_i} = \sum_{j}^{i} \alpha_{ij} \boldsymbol{h_j}, \tag{3.14}$$

where the attention weights are row-wise normalized $\sum_j \alpha_{ij} = 1$. In general they can be calculated as

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_k^N \exp(e_{ik})} \tag{3.15}$$

$$e_{ij} = a(\boldsymbol{s_{i-1}}, \boldsymbol{h_j}), \tag{3.16}$$

where $a(\cdot)$ is some attention function. As seen in Figure 3.3b this allows us to recall information in previous states much easier. Attention was originally developed for sequence to sequence models, where you would use the state of the previous prediction s_{i-1} to condition the new prediction [3]. This shows another advantage of the attention mechanism; whereas the classic RNN created a global sentence encoding $\boldsymbol{z_i}$, the sentence encoding of the attention mechanism $\boldsymbol{c_i}$ can be conditioned on some additional information, which provides a more dynamic encoding. Depending on what we want to condition on, different things in the sentences might be important. Since the attention weights are normalized they can also give insights into model interpretability by visualizing which tokens gives the highest contribution.

In our case of language modelling we are not generating a new sequence so we do not have the previous state s_{i-1}. Instead we could use *self-attention* where we instead condition on the current hidden state h_i [6]:

$$e_{ij} = a(\boldsymbol{h_i}, \boldsymbol{h_j}). \tag{3.17}$$

There exists many attention mechanisms for calculating the attention weights, which also depends on the problem at hand [31]. The next section will discuss one of the current state of the art mechanisms.

3.7 Transformer Models

So far we have discussed solutions to the problem of long term dependencies in recurrent neural networks. Their auto-regressive nature does however propose several more flaws. First of all they are computationally inefficient since we have to run the model sequentially over each token, which cannot be parallelized. Secondly, we have assumed that the contextual dependence only runs from left to right. This is not always the case as can seen in the following two sentences. Here the meaning of "lie" depends on the words that appears later in the sentence:

"He would often lie on the couch in the afternoon."
"He would often lie which is why he had no friends."

A solution to the directional dependence was proposed by creating bidirectional RNNs. These consisted of one RNN running from left to right, and one from right to

left [47]. This however only provides a crude solution and still does not address the problem of parallelization. Given the success of attention [31, 54, 37, 30, 61], Vaswani et al.[52] proposed to discard RNNs altogether in favour of using self attention. This model architecture is called a *transformer*.

The transformer addresses three of the flaws of RNNs mentioned earlier; long term dependence, sequence directionalty and computational efficiency. By using self attention instead of recurrence we calculate the interaction between all words, not relying on a specific sequence direction, as can be seen in Figure 3.4. This also means that we do not have the problem of long term dependency. Finally, since we do not have any sequential operations, this has the added benefit of being easily parallelized. The network may seem identical to feed-forward neural network discussed in Section 2.6, but have the main difference that it uses a fixed set of shared parameters to calculate the hidden state, instead of one for each input token.

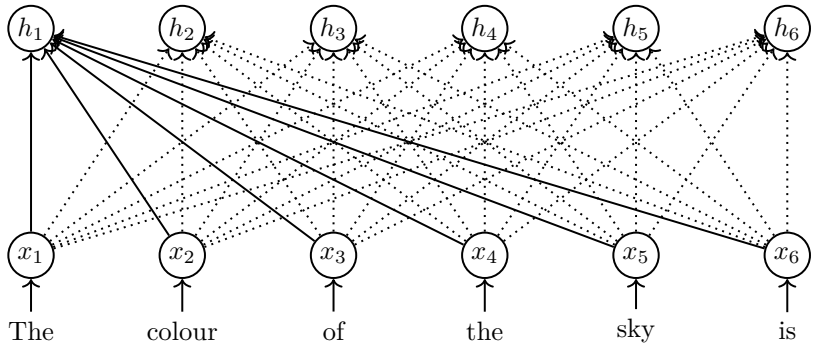


Figure 3.4: Transformer architechture, based on self attention

Following Vaswani et al.[52], the hidden representation of each layer is calculated using self attention, layer normalization [2], skip-connections[17] and a feed forward neural network, as follows

$$\boldsymbol{h}_{\boldsymbol{i}}^{(l)} = \text{Layernorm}\left(\boldsymbol{z}_{\boldsymbol{i}}^{(l)} + \text{FNN}(\boldsymbol{z}_{\boldsymbol{i}}^{(l)})\right) \tag{3.18}$$

$$\boldsymbol{z}_{\boldsymbol{i}}^{(l)} = \text{Layernorm}\left(\boldsymbol{h}_{\boldsymbol{i}}^{(l-1)} + \text{Attention}(\boldsymbol{h}_{\boldsymbol{i}}^{(l-1)}, \boldsymbol{h}^{(l-1)})\right), \tag{3.19}$$

where $\boldsymbol{h}^{(l-1)}$ indicates all the hidden states of the previous layer $\boldsymbol{h}_{\mathbf{1}}^{(l-1)}, \ldots, \boldsymbol{h}_{\boldsymbol{N}}^{(l-1)}$. $\text{FNN}(\boldsymbol{z}_{\boldsymbol{i}})$ is a shallow feed forward neural network

$$\text{FNN}(\boldsymbol{z}_{\boldsymbol{i}}) = \boldsymbol{\Theta}_2 \text{ReLU}\left(\boldsymbol{\Theta}_1 \boldsymbol{z}_{\boldsymbol{i}} + \boldsymbol{\theta}_1\right) + \boldsymbol{\theta}_2. \tag{3.20}$$

$\text{Layernorm}(\boldsymbol{z}_{\boldsymbol{i}})$ is layer normalization

$$\text{Layernorm}(\boldsymbol{z}_{\boldsymbol{i}}) = \frac{\boldsymbol{z}_{\boldsymbol{i}} - \mathbb{E}[\boldsymbol{z}_{\boldsymbol{i}}]}{\sqrt{\text{Var}[\boldsymbol{z}_{\boldsymbol{i}}] + \epsilon}} \cdot \theta_\gamma + \theta_\beta, \tag{3.21}$$

where $\theta_\gamma, \theta_\beta$ are trainable parameters, ϵ is a hyperparameter and the expectation and variance are calculated over the mini-batch [2].

3.7.1 Multi-Head Attention

At the core of the transformer model, is self attention. The definition of attention in the transformer follows (3.17) and (3.14), but introduces trainable parameters

$$\boldsymbol{c_i^{(l)}} = \sum_j^N \alpha_{ij} \left(\boldsymbol{\Theta}_V \boldsymbol{h_j^{(l-1)}} \right) \tag{3.22}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_k^N \exp(e_{ik})} \tag{3.23}$$

$$e_{ij} = a(\boldsymbol{h_i^{(l-1)}}, \boldsymbol{h_j^{(l-1)}}) \tag{3.24}$$

In the transformer model we use *scaled dot-product attention* as our attention function.

$$a\left(\boldsymbol{h_i^{(l)}}, \boldsymbol{h_j^{(l)}}\right) = \frac{\left(\boldsymbol{\Theta_Q} \boldsymbol{h_i^{(l)}}\right)^T \left(\boldsymbol{\Theta_K} \boldsymbol{h_j^{(l)}}\right)}{\sqrt{d_k}}, \tag{3.25}$$

where $\boldsymbol{\Theta_V}, \boldsymbol{\Theta_Q}, \boldsymbol{\Theta_K} \in \mathbb{R}^{d_k \times d_k}$ are trainable parameters, and d_k the size of the hidden dimension. The transformer also introduces a new scheme of using multiple context vectors for each layer. This is called *multi-head* attention, and simply concatenates n_{heads} context vectors followed by a linear projection. This gives us the full attention function from (3.19)

$$\text{Attention}\left(\boldsymbol{h_i^{(l)}}, \boldsymbol{h^{(l)}}\right) = \left[\boldsymbol{c_{i,1}^{(l)}}, \boldsymbol{c_{i,2}^{(l)}}, \ldots, \boldsymbol{c_{i,n_{heads}}^{(l)}}\right] \boldsymbol{\Theta_O} \tag{3.26}$$

Each context vector $\boldsymbol{c_{i,j}}$ uses separate parameters $\boldsymbol{\Theta_V}, \boldsymbol{\Theta_Q}, \boldsymbol{\Theta_K}$. [52]

3.7.2 Positional Encoding

By abandoning the recurrence of RNNs, in favor of self attention, we have however also removed any information about the sequence of which the tokens in our sentence appears. This can be solved by introducing a positional encoding that is added to the embedding of the words

$$\text{PE}(pos, i) = \begin{cases} \sin\left(pos / 10000^{2i/d}\right) & \text{if } i \text{ is odd} \\ \cos\left(pos / 10000^{2i/d}\right) & \text{if } i \text{ is even} \end{cases} \tag{3.27}$$

where pos is the position of the word in the sequence, i is the ith dimension of the embedding, and d is the total dimension of the embedding [52].

3.8 Masked language modelling

In Section 3.2 we defined the goal of our language model to be estimating the conditional probability of each word, given the previous words $P(w_i|w_1^{i-1})$. Here we assumed a unidirectional dependence, from left to right, between words. This we now know is not strictly correct as words can be dependent on later parts of the sentence.

Since the transformer model does not have any inherit assumptions about directionality, Devlin et al.[9] proposed using *masked language modelling*. Here we mask random tokens in the input sequence, and try to recover the original token. This means that we now try to model $P(w_i|\widetilde{w}_1^N)$, where $\widetilde{w}_1^N$ is the whole sentence, where some of the tokens have been replaced with a <mask> token. By masking multiple words we force the model to create more robust representations. The joint probability of a sentence can then be calculated by masking each token individually

$$P(w_1, w_2, \ldots, w_N) = \prod_i^N P(w_i|\widetilde{w}_1^N), \tag{3.28}$$

where $\widetilde{w}_1^N$ is the input sentence with the ith token masked. In practice we only use the predictions of the masked token during training, as it is trivial to predict the non-masked word due to it being given as input. The training objective thus becomes to try and recover the original sentence, which corresponds to the following objective function

$$P\left(w_1^N|\widetilde{w}_1^N\right) = \prod_{\widetilde{w}_i=<mask>}^{i} P(w_i|\widetilde{w}_1^N) \tag{3.29}$$

In Chapter 4 we will introduce how language modelling can be used to learn structure rules in molecules.

CHAPTER 4

Machine Learning On Molecules

In this chapter we will introduce how machine learning and NLP can be used on molecules. First we will discuss the problem of choosing a good representation of a molecule. Then we will discuss some basic chemistry, which forms the basis for the molecular structure rules we are trying to learn. Next we will introduce how our NLP models from Chapter 3 can be adapted to graphs, and how they compare to contemporary models used for molecular graphs.

4.1 Representing Molecules

In order to use our machine learning models we need to find a way of representing the molecules. When choosing a representation we are generally interested in the following properties: [10, 32]

- **Permutation invariance**: Since the atoms in a molecule have no inherent order, changing the order should give the same results.
- **Translation invariance**: Translating the molecule in 3D space does not change its properties
- **Rotation invariance**: Rotating the molecule also does not change its properties
- **Uniqueness** There should only exists one unique representation for each molecule
- **Invertibility** Given a representation it should be possible to recover the original molecule.
- **Similarity**: Similar molecules should give similar representations

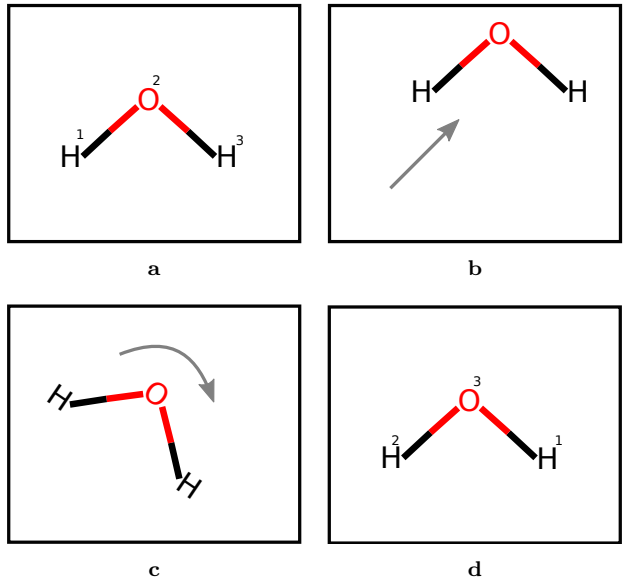


Figure 4.1: Visualization of (b) translation invariance, (c) rotation invariance and (d) permutation invariance

4.1.1 3D Coordinates

The properties of a molecule can be described by solving the Schrödinger equation. Here the molecule is represented as a set of charges in 3D space so this would seem the preferred representation. That is, the input molecule is a list

$$(\boldsymbol{a_1}, \boldsymbol{a_2}, \ldots, \boldsymbol{a_N}) \tag{4.1}$$

where $\boldsymbol{a_i} = (c_i, x_i, y_i, z_i)$ are the corresponding charge and x,y,z coordinates of the ith atom.

This does, however, have several downsides, like being neither permutation, translation or rotation invariant.

4.1.2 Molecular Graphs

From chemistry, we know that we can represent a molecule as a molecular graph where the nodes are the atoms, and the edges are bonds between atoms [29]. In this case the input molecule is represented as a graph G

$$G = (\boldsymbol{v}, \boldsymbol{E}), \tag{4.2}$$

where $\boldsymbol{v} = (v_1, v_2, \ldots, v_N)$ are the element type for each atom, and $\boldsymbol{E}_{ij}$ is the weight associated with the edge between the ith and jth atom.

If we want to use information about the 3D geometry of the molecule we can use a weighted graphs where the weight of each edge corresponds to the distance between those two atoms, which is correlated with the strength of a bond. As mentioned in Chapter 1, using continuous representations has its limitations for generative models. So instead we use a discrete representation, where we assume that each connection between two atoms can be of four types; none, single, double or triple bond. In this case $\boldsymbol{E}_{ij} \in \{0, 1, 2, 3\}$. This formulation is based on valance theory, which will be discussed in Section 4.2.

Using molecular graphs has the benefit of being translation and rotation invariant, invertible and also encoding similar molecules with a similar representation. They are, however, neither unique or permutation invariant by nature. As we shall see in Section 4.3, if we use models designed for graphs the problem of permutation and uniqueness can be ignored.

4.1.3 1D Strings

A very common way of representing molecular graphs is the Simplified Molecular-Input Line-Entry System (SMILES) string[57]. Here, the the molecular graph is traversed in a depth-first manner and then converted to a string using a specific syntax. An example of this can be seen in Figure 4.2. SMILES strings have been used due to their simplicity; since the representation is just a sequence of tokens we can use all the NLP models from Chapter 3 out of the box.

In theory, SMILES strings contains the same information as the discrete molecular graph but have several downsides; small changes in the molecule, might lead to large differences in the SMILES string, thus it does not encode similarity well. Like the molecular graph, SMILES strings are not permutation invariant or unique, but this can be solved by canonicalization [58]. Using SMILES strings does, however, introduce a layer of abstraction. If we want model the structure rules of the molecule we also have to learn the grammar of the SMILES string, and possible the canonicalization, which introduces unnecessary complexity.

Similar to SMILES, another string representation is the International Chemical Identifier (InChI) [18]. This is however not used often in literature as some studies suggest that they perform worse, due to introducing even more overhead from a more complex syntax [12].

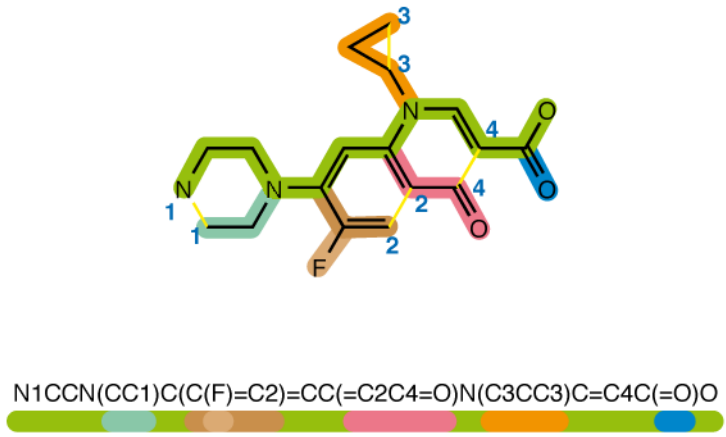


Figure 4.2: Visualization of the SMILES syntax. Figure taken from [1]

One final 1D representation is molecular fingerprints, like the Extended Connectivity FingerPrint (ECFP) [44]. These representation encode neighbourhood information about each atom, which is then converted to a fixed size vector by a hashing function. While these representations encode graph information; are permutation, translation and rotation invariant; and unique they are, however, not invertible. This makes them unsuited for generative models as we cannot recover the molecule from a generated representation directly.

In the remaining of this work we shall use the discrete molecular graph representation as we argue it is superior for our purposes; by choosing machine learning models developed for graphs we can achieve all the desired invariances and properties from our list. Furthermore, since we are trying to learn structure rules, based on atoms and their bonds, the molecular graph allows this to be modelled directly without any syntax overhead from e.g SMILES strings.

In the next section we shall describe the theoretical structure rules of molecules that we are trying to learn.

4.2 Molecular Structure Rules

A molecule is made up of a set of atoms. These atoms are in turn made up of neutrons, protons and electrons. The number of neutrons and protons determine the

[1] Simplified molecular-input line-entry system. In Wikipedia, The Free Encyclopedia. Retrieved 22:19, December 11, 2019, from `https://en.wikipedia.org/w/index.php?title=Simplified_molecular-input_line-entry_system&oldid=920714417`

element of the atom, while the electrons mostly are responsible for the interactions between atoms. We are therefore interested in a theory of how the electrons interact in a molecule. The most common and simple theory is the octet rule based on the Lewis structures. Here we assume that the electrons of an atom is located in electron shells, as seen in Figure 4.3. Each shell has a capacity of 8 electrons, except the first which only can have 2. The octet rule states that atoms strides to fill their outermost shell, called the *valence shell*, such that it has 8 electrons since this is the most stable configuration [29]. The electron shells corresponds to different energy levels in the atom, and are thus filled one at a time. This means that only the electrons in the valence shell, called *valence electrons*, contribute to the interactions between atoms. Elements are generally neutral charged meaning that the number of electrons, and hence valence electrons, are determined by the number of protons in the element.

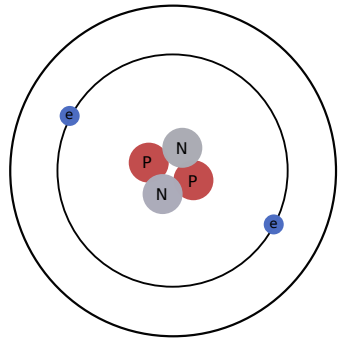


Figure 4.3: Electron shell model of helium

Elements are organized in the periodic table where the main-groups corresponds to the number of valence electrons in the element. There are several ways for atoms to fill their valence shell like forming ionic or covalent bonds with other atoms. In ionic bonds, an electron is donated by an atom to another atom, and in a covalent bond two atoms shares a pair of atom. In this work we will only focus on covalent bonds as these are the only ones present in our dataset.

Each atom can share up to three of its valence electrons corresponding to a single, double or triple bond. To determine which atomic configurations are possible in a molecule we now just have to match atoms according to their number of valence electrons, such that they all end up with 8. An example of this is H_2O where Oxygen (O) has 6 valence electrons, thus need 2 more, and Hydrogen (H) having 1 valence electron, and thus needing 1 more. As can be seen in Figure 4.4, by creating a covalent bond between each hydrogen and oxygen, the oxygen now has gained 2 electrons and each of the hydrogen has gained 1. This example can be extended to arbitrary number of atoms, and include double and triple bonds.

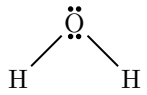


Figure 4.4: Lewis structure for H_2O. Dots corresponds to electron pair and line to a covalent bond

In principle, the above description of the octet rule also allows for certain ions in a molecule. For example, in O_3 as seen in Figure 4.5. As mentioned above, oxygen wants 2 more electrons by creating two covalent bonds. We can, however, see only one of the oxygen has managed to do so; one has only one covalent bond, and one has three. Seen at a local scale this would not satisfy the octet rule but it turns out that if we account for all the electrons in the molecule the octet rule is satisfied. This is seen by adding the charges to the elements [29].

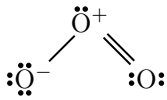


Figure 4.5: Lewis structure for O_3. Dots corresponds to electron pair and line to a covalent bond

In general, formal charges on atoms are unfavourable as molecules with zero charges are more stable. An example can be seen in Figure 4.6, where both structures are correct under the octet rule but (b) is more stable due to not having any charges [29].

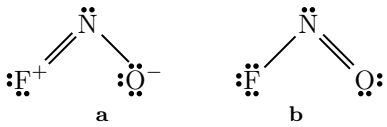


Figure 4.6: Lewis structure for ONF. (a) has atoms with formal charge, whereas (b) does not

In the above example, the total charge of ONF is still zero but this is not always the case. This can be seen with H_3O^+ in Figure 4.7. Here we again have a charge on one of the atoms, and all atoms satisfy the octet rule. Notable is however, that the total number of valence electrons from each atom (1 from each hydrogen and 6 from oxygen) is one more than what we have in the molecule. This results in a overall charge on the molecule since it has dumped one of its electrons.

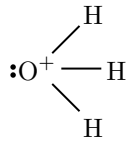


Figure 4.7: Lewis structure for H_3O^+. Dots corresponds to electron pair and line to a covalent bond

As has been seen, describing the behavior of charged atoms in molecules can be a tricky task as the octet rule allow for many molecule structures that does not exist. Another downside of the octet rule is that there are cases where it is not satisfied by molecules. Two of these exceptions are electron deficient molecules – where one or more atoms has fewer than 8 valence electrons – and hypervalent molecules, where an atom has more than 8 valence electrons. Two examples of this can be seen in Figure 4.8, where (a) is a electron deficient molecule since B only has a total of 6 electrons, and (b) is hypervalent, since P has a total of 10 electrons.

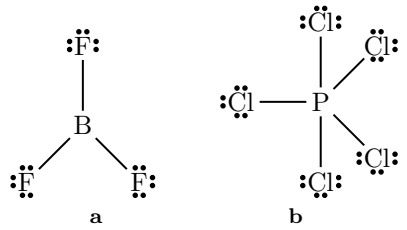


Figure 4.8: Lewis structure for (a) the electron deficient BF_3 molecule (b), the hypervalent PCl_5 molecule

There has been some work to try and explain electron deficient and hypervalent molecules using the octet rule. This can be done with the concept of resonance but like with ions it does not give a clear explanation why some compounds experience this when others do not [29]. It is clear that the octet rule only is a simplified theory and more modern theories based on quantum mechanics, like orbital theory, have therefore been developed. This is however significantly more complex, and much harder to use as a heuristic for stability.

From the presented examples we note that

- For almost all molecules without any charged atoms the stable molecule configurations can be determined by the octet rule.
- The octet rule offers some heuristic for explaining molecules that contains charged atoms, but cannot describe the full picture.

- There exists examples like electron deficient and hypervalent molecules that does not satisfy the octet rule.

The goal of our language model is therefore to hopefully address all three points thus providing a better estimate of the validity of a molecule.

4.3 Graph Language Modelling

In Section 4.1, we argued that discrete molecular graphs was the optimal molecule representation for our purpose. As the models we have discussed so far in Chapter 2 and 3, are not obviously defined for graphs, we need to define how to adapt our models to use them as input. Let's recall that the molecular graph is represented as a list of atoms $\boldsymbol{v}$, and the associated adjacency matrix $\boldsymbol{E}$. Using same formalism as in Chapter 3, the atoms $\boldsymbol{v}$ are our tokens, which corresponds to words, and $\boldsymbol{E}$ describes their connectivity. In fact, a sentence from Chapter 3 can be seen as a directed graph with adjacency matrix that is all zeros, except for $j = i + 1$, if you assume a unidirectional dependence as we did with RNNs.

The extension of masked language modelling therefore corresponds to masking atoms while keeping the bonds. The conditional probability we are trying to model can therefore be written as

$$P\left(v_i|\widetilde{\boldsymbol{v}}, \boldsymbol{E}\right). \tag{4.3}$$

Again we only try to predict the masked atoms, during training, which leads to the following objective function, similar to the one in Section 3.8, where we try to recover the original molecular graph

$$P(G|\widetilde{G}) = \prod_{\substack{i \\ v_i=<mask>}} P(v_i|\widetilde{\boldsymbol{v}}, \boldsymbol{E}). \tag{4.4}$$

4.4 Bag of Words

The first model we can look at is the BoW model. Here we ignored the order of the tokens in the sequence, which corresponds to ignoring the adjacency matrix. This means that we can use the model directly by simply applying an embedding function to all the atoms, and aggregating them. This model we will call *Bag of Atoms*, and corresponds to just modelling $P(v_i|\widetilde{\boldsymbol{v}})$. If we use the sum as aggregation function this corresponds to

$$\boldsymbol{z_i} = \sum_j \boldsymbol{x_j}, \tag{4.5}$$

where $\boldsymbol{x_i}$ is the embedding of the ith atom.

This model does however have the problem that if we mask multiple atoms all of them will have the same representation, and the model will thus predict the same for all of them.

One simple way of including information about the topology of the graph is to only condition the BoW on the neighbours of the atom, as can be seen in Figure 4.9. This model we call *Bag of Neighbours*, and corresponds to

$$\boldsymbol{z_i} = \sum_{\substack{j \\ \boldsymbol{E}_{ij} \neq 0}} \boldsymbol{x_j}. \tag{4.6}$$

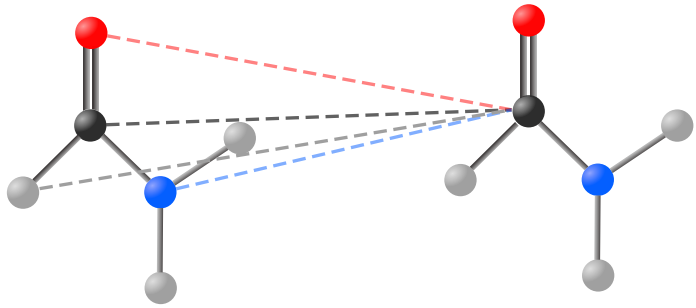


Figure 4.9: Diagram of how only neighbourhood information is used in Bag of Neighbours, and Graph Attention

4.5 Graph Transformer

When we defined the transformer in Section 3.7 we had to use a positional encoding due to the model not encoding any information about the sequence of tokens. This means that if we remove the positional encoding the transformer can directly be used on graphs, as it relies on correlations between nodes. Just doing this does, however, not include any information about the connections in the graph. Similar to the Bag of Neighbours this can be solved by only conditioning the attention mechanism on the neighbours of each atom. This is called *Graph Attention* [53].

Using our formulation of the transformer this can be implemented by only calculating self attention to the neighbours, when calculating the context vector,

$$\boldsymbol{c}_{\boldsymbol{i}}^{(l)} = \sum_{\substack{j \\ \boldsymbol{E}_{ij} \neq 0}} \alpha_{ij} \left(\boldsymbol{\Theta}_v \boldsymbol{h}_{\boldsymbol{j}}^{(l-1)} \right) \tag{4.7}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{\substack{k \\ \boldsymbol{E}_{ik} \neq 0}} \exp(e_{ik})} \tag{4.8}$$

This formulation does have some limitations; it does for example not include edge information. In our case, this could be information like bond type between atoms. We can solve this by redefining the attention mechanism as follows [50]:

$$\boldsymbol{c_i^{(l)}} = \sum_{\substack{j \\ \boldsymbol{E}_{ij} \neq 0}} \alpha_{ij} \left(\boldsymbol{\Theta}_v \boldsymbol{h_j^{(l-1)}} + o_{ij}^V \right) \tag{4.9}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{\substack{k \\ \boldsymbol{E}_{ik} \neq 0}} \exp(e_{ik})} \tag{4.10}$$

$$e_{ij} = \frac{\left(\boldsymbol{\Theta_Q} \boldsymbol{h_i^{(l)}} \right)^T \left(\boldsymbol{\Theta_K} \boldsymbol{h_j^{(l)}} + o_{ij}^K \right)}{\sqrt{d_k}} \tag{4.11}$$

where o_{ij}^V,o_{ij}^K are the ouput of two different embedding functions applied to the corresponding entry in the adjacency matrix $\boldsymbol{E}_{ij}$. This means that the context vector now is conditioned on edge information of the neighbours as well, as seen in Figure 4.10. This we will call *Edge Graph Attention*.

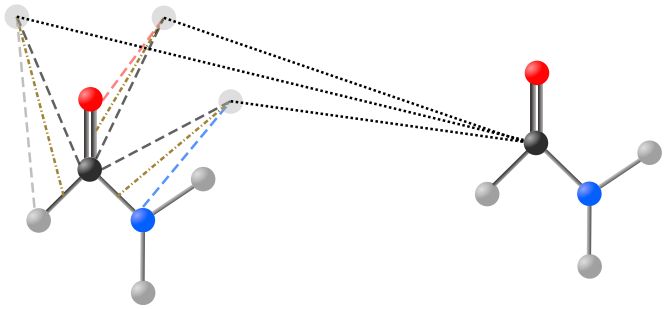


Figure 4.10: Diagram of how neighbourhood and edge information is used in Edge Graph Attention

In theory, by applying multiple layers of edge graph attention, we should obtain all information in the graph. It does however force information from nodes far away to have a long dependence as we saw with RNNs. We therefore propose a final alteration to the transformer model. Since we are including information about the connectivity directly in the attention mechanism we can relax the assumption about only conditioning on the neighbours. This makes it possible for out model to get information about all the nodes in the graph directly, which is seen in Figure 4.11This results in the slightly simplified attention mechanism, which we call *Full Graph Attention*

$$\boldsymbol{c_i^{(l)}} = \sum_j \alpha_{ij} \left(\boldsymbol{\Theta}_v \boldsymbol{h_j^{(l-1)}} + o_{ij}^V \right) \tag{4.12}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})} \tag{4.13}$$

$$e_{ij} = \frac{\left(\boldsymbol{\Theta_Q} \boldsymbol{h_i^{(l)}} \right)^T \left(\boldsymbol{\Theta_K} \boldsymbol{h_j^{(l)}} + o_{ij}^K \right)}{\sqrt{d_k}} \tag{4.14}$$

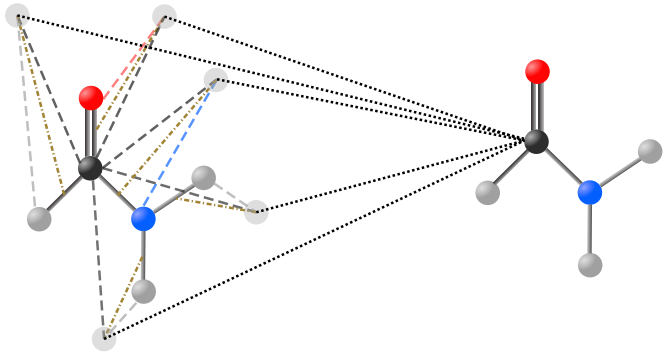


Figure 4.11: Diagram of how all node and edge information is used in Full Graph Attention

4.6 Graph Neural Networks

There already exists a lot of machine learning models developed for graphs in the literature [64, 60, 62]. By following the formulation of Gilmer et al.[11], most of these graph models can be described as so called *Message Passing Neural Networks* (MPNN). This class of models are described by updating the nodes in the graph by using a message function $M^{(l)}$ and an update function $U^{(l)}$,

$$\boldsymbol{m_i^{(l+1)}} = \sum_{\substack{j \\ \boldsymbol{E}_{ij} \neq 0}} M^{(l)} \left(\boldsymbol{h_i^{(l)}}, \boldsymbol{h_j^{(l)}}, \boldsymbol{E}_{ij} \right) \tag{4.15}$$

$$\boldsymbol{h_i^{(l+1)}} = U^{(l)} \left(\boldsymbol{h_i^{(l)}}, \boldsymbol{m_i^{(l+1)}} \right). \tag{4.16}$$

While the models we described above was inspired by NLP most of them turns out to be special cases of these message passing networks. In Bag of Neighbours, if we

denote $\boldsymbol{h_i^{(0)}} = \boldsymbol{x_i}$, then we see that the message function is $M^{(0)}\left(\boldsymbol{h_i^{(0)}}, \boldsymbol{h_j^{(0)}}, \boldsymbol{E}_{ij}\right) = \boldsymbol{h_j^{(0)}}$ and the update function is just the identity $U^{(0)}\left(\boldsymbol{h_i^{(0)}}, \boldsymbol{m_i^{(1)}}\right) = \boldsymbol{m_i^{(1)}}$. Similar, for our transformer if we consider graph attention, and edge graph attention this corresponds to using our attention mechanism as message function, and again the identity update function. With our final full graph attention we do however deviate from this framework as we allow the message function to look at nodes that are not neighbours. Calculating the interaction between all nodes in the graph can be very expensive for larger graphs, which is one of the reasons why MPNN are restricted to only looking at their neighbours. Due to the fact that the molecular graphs in our case are reasonably small (<100 nodes), and that these interaction calculations can be parallelized thanks to GPUS, this is not considered a problem for this project.

CHAPTER 5
Experimental Setup

In this chapter we shall describe the experimental setup and tools used in this project. First we will introduce the software tools used to design and train our models, then we will discuss the machine learning pipeline used – from data loading to training the models – and finally how we organized the post hoc analysis.

5.1 Deep Learning Frameworks

As discussed in Section 2.6, one of the strengths of neural networks are their flexibility. In order to train an arbitrary neural network we just need to keep track of the derivative of each function that we apply in order to use backpropagation. This has allowed for the development of neural network frameworks that automatically handles the backpropagation once you have specified the model architecture. Two of the most popular frameworks are Tensorflow[1] and PyTorch[38]. Since there has been a tendency in the field of creating larger and deeper models, and using larger and larger datasets, the computational efficiency also becomes a concern that these frameworks can tackle by using high performing libraries for designed for this. Most of the computations required in a neural network consists of matrix operations, which is highly parallelizable. This is why the use of *Graphical Processing Units* (GPUs) have seen widespread use in the field. The deep learning frameworks automatically handles distributing the calculates to one or more GPUs.

In this work we use PyTorch as it is the framework familiar to the author. PyTorch also provides easier debugging and an subjectively easier syntax, compared to Tensorflow. All the models are implemented from scratch only using the building blocks available in PyTorch.

Since we are potentially doing many experiments – running multiple models on several datasets – it can be good to also use a tool for keeping track of your experiments. The most common tool for this is TensorBoard (part of Tensorflow) but in this project we will use Weights & Biases [55]. Besides allowing you to monitor metrics during training, like TensorBoard, this tools also allows for easier post hoc analysis by filtering and grouping results based on parameters. Another benefit is the fact

that your results are stored in their cloud solution giving you the option to monitor your experiments from any device.

5.2 Data Loader

The first part of any machine learning project is the data. To allow for multiple datasets we designed our data pipeline as flexible as possible. The input from the datasets are SMILES strings, which are converted to molecular graphs using RdKit[27]. These molecular graphs are then split into training, validation and test set, and saved as pickle files. Instead of using a traditional random split we split the data based on the Bemis-Murcko scaffold [4]. This emphasises that the overall structure of molecules in the different splits are different, giving a more realistic estimate of the generalization to unseen molecules [59].

5.2.1 Padding

When using a deep learning framework like PyTorch, even though the linear algebra calculations are highly optimized there is an overhead cost each time you call an operation from the python API. Since the forward pass of our network can be calculated for all datapoints independently, it is common to collect the input in one large matrix of size $\mathbb{R}^{B\times N}$, where B is the batch size, and N is the sequence length. This matrix then allows us to do calculations for the entire batch in one call per operation. The input sequences are however not the same length. A simple solution to this is to just add a special *padding token* to make the sequences the same length, which can be seen in Figure 5.1. This does, however, require us to make sure that the padding token does not have an effect on the predictions. We solve this by forcing the attention weight to be zero for the padding tokens.

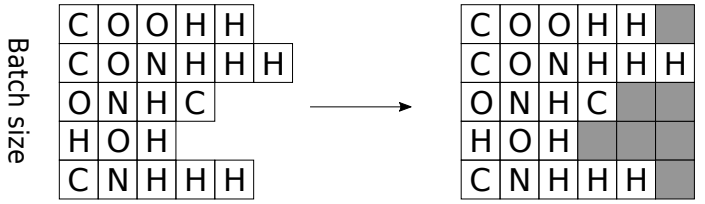


Figure 5.1: Visualization of how padding is added to make input fixed size. Grey squares corresponds to padding

5.2.2 Masking Strategy

Instead of creating a dataset where the masked atoms are fixed, we create a dynamic masking scheme. This allows us to change the number of masked atoms per molecule

during training and testing. Secondly it can serve as a sort of data augmentation; instead of predefining the masked atoms in each molecule beforehand we randomly select the atoms to be masked during training. When running multiple epochs this is an approximation to having trained on all possible configurations of masks, which may not be computational feasible due to its combinatorial nature.

We also introduce another source of data augmentation. Instead of only masking a predefined number of atoms per molecule during training we employ a epsilon-greedy scheme [51]. Whenever we mask a molecule, we mask the chosen number of atoms, denoted n_{mask}, with probability $1 - \varepsilon$. With the remaining ε probability, we select a random number of masks, drawn from a uniform distribution. This allows our model to generalize better to any number of masks as it is not trained on a specific number of masks.

5.3 Training

As mentioned, the use of a deep learning framework like PyTorch allows one to automate the calculation of gradients. This means that by building your models as modules its possible to create a generic training script. By supplying this training script you can easily launch experiments by changing the parameters. The training script used in this project has command line arguments like model architecture, dataset, number of hidden layers etc. This script also automatically logs metrics to Weights & Biases, and saves the parameters of the model. To try and avoid overfitting, we keep track of the best validation error for the model being trained, and only save the model if the current validation error is lower than this.

5.3.1 Hardware

Training large models like the transformer model requires a certain amount of computational power. In this project we used the resources available at DTU HPC[1], namely their GPU queue that consists of several Nvidia V100 GPUs. The HPC uses the LSF batch scheduling system, which allows us to schedule and run multiple experiments at the same time. Using our training script, an experiment can be launched by simply creating a job on the queue that calls the training script with the desired parameters.

5.4 Post Hoc Analysis

After training our models we are interested in evaluating their performance. Instead of having to regenerate predictions for all models when we either change a model or decide on a new way of analysing the performance, we will save the predictions of

[1] `https://www.hpc.dtu.dk/`

each model in a SQL database. As we can have potentially many models with many predictions we have to design the database schema with some care. The schema used in this project can be seen in Figure 5.2.

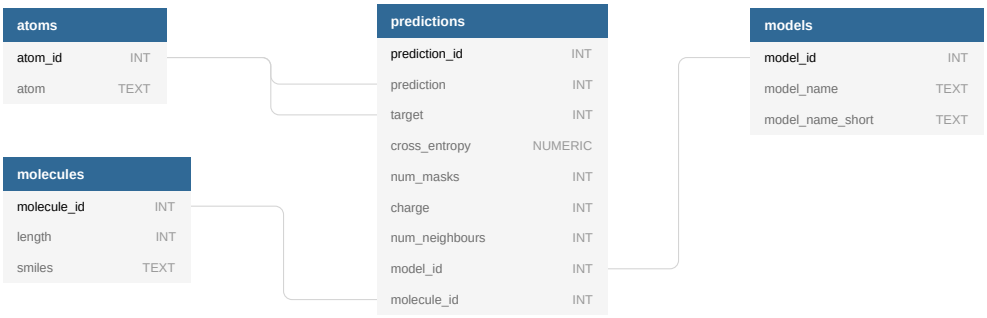


Figure 5.2: Database schema used for storing predictions

The atoms, molecules and models tables are used to reduce the duplication of data. Each row in the predictions table, corresponds to the prediction of one atom, and thus information about the model or molecule, might be identical in many rows. By creating these auxiliary tables we only have to store one integer linking the row to the correct value, reducing the space used significantly. This is called database normalization. It is possible to optimize the database schema further, if your goal is to reduce space consumption. This can be done by noting that num_masks, charge and num_neighbours are values that are shared across models. The database schema is however designed for data ingestion and analysis performance, and moving the mentioned columns to their own table, would require joining this new table with the molecules table during data ingestion. The performance overhead of this was deemed too high, and since the HPC environment provides large amounts of storage, the schema was kept as is.

In order to provide uncertainty estimates of our models each model is trained 10 times with different start seed. In our schema this was implemented by the model_name containing the number of the model while model_name_short did not. This meant that we could simply group by the model_name_short. While SQL databases are optimized for performing calculations directly in the database using SQL queries, not all calculations could be done easily in SQL. It was therefore needed to extract the data, and perform the calculation using python. Due to the amount of data in our database (over 800.000.000 predictions stored at the end of the project), it would not be possible to load all predictions into memory. Instead the data was processed for each model separately, loading data into memory, performing calculations, and clearing memory.

CHAPTER 6

Experiments

This chapter is dedicated to the experiments performed in the project. First we will introduce the datasets used, then define the models used and finally present the results of the experiments.

6.1 Datasets

In this project, we will use two datasets: QM9[43], which is used as a simple benchmark dataset; and a subset of the ZINC database[21], which is used as a more challenging dataset. The two datasets will be introduced in the following.

6.1.1 QM9

The QM9 dataset is a common dataset used for property prediction. Is consists of 130.000 organic molecules made up of the 5 elements: hydrogen, oxygen, nitrogen, carbon and fluorine. This dataset will be used as a benchmark due to its simplicity; it does not contain any electron deficient or hypervalent molecules, and to simplify it further we remove the 1808 molecules containing atoms with charges. The dataset can therefore be fully described by only looking at the number of valence electrons of the neighbour atoms, and using the octet rule. Fluorine and hydrogen have the same number of valence electrons; thus this is the only source of ambiguity in the dataset. The purpose of this dataset is therefore to see if our models are able to learn the octet rule.

From Figures 6.1a and 6.1b we see the distribution of elements for different sizes of molecules for the training and test dataset. It is clear that the dataset is heavily biased, with hydrogen and carbon making up the majority. This is especially the case as the number of atoms in the molecule increases. The atom distribution for different molecule sizes are different for the training and test dataset. This might be due to our scaffold split or simply by chance. Finally, we note that the size of the molecules are not uniform; we have fewer small and large molecules.

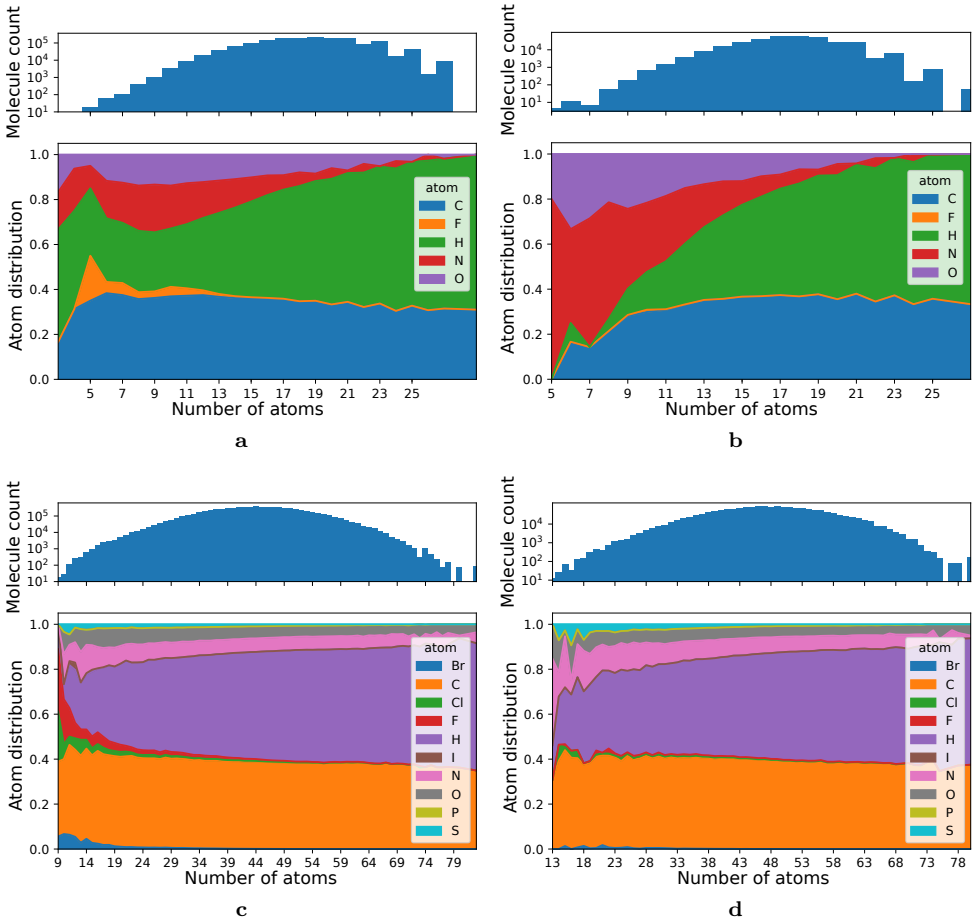


Figure 6.1: Count (top) and distribution of elements per molecule size (bottom) for (a) QM9 training dataset, (b) QM9 test dataset, (c) ZINC training dataset, and (d) ZINC test dataset. Figure taken from Olsen et al [36]

6.1.2 ZINC

As our second dataset we – similar to Gomez et al.[12] – use a subset of 250.000 molecules from the ZINC database[21] consisting of the 12 elements: hydrogen, oxygen, nitrogen, carbon, fluorine, phosphor, iodine, sulphur, bromine and chlorine. The dataset therefore has more ambiguity than QM9. Together with the fact the dataset has molecules that contain atoms with charges, and also several hypervalent molecules, this dataset is used to test our models ability to learn structure rules beyond the octet rule.

Similar to QM9 – as seen in Figures 6.1c and 6.1d – the size of the molecules is non-uniform. The distribution of elements does also depend slightly on the number of atoms in the molecule, however, mostly for small molecules. Furthermore, we note that the molecules in ZINC are significantly larger.

6.2 Model Implementations

Our experiments will include four of the models previously discussed: unigram, bag of atoms, bag of neighbours and graph transformer. Additionally, we will compare the models to a model based on the octet rule. All models with trainable parameters are optimized on the cross entropy loss function using the Adam optimizer[26].

6.2.1 Unigram

As the unigram model does not use any context it can be used by simply finding the frequency of each element in the dataset; $P(v_i = \mathcal{C}_j|\widetilde{\boldsymbol{v}}, \boldsymbol{E}) = P(\mathcal{C}_j)$. The resulting unigram probabilities for each element can be seen in Table 6.1 for our two datasets. As our prediction is done by predicting the most likely atom we see that we will always predict hydrogen.

		Dataset	
		QM9	**ZINC**
Elements	P(**H**)	0.519	0.47407
	P(**C**)	0.347	0.38691
	P(**O**)	0.078	0.05416
	P(**N**)	0.054	0.06109
	P(**F**)	0.002	0.00856
	P(**S**)	0	0.00913
	P(**Cl**)	0	0.00452
	P(**Br**)	0	0.00144
	P(**I**)	0	0.00011
	P(**P**)	0	0.00001

Table 6.1: Unigram probabilities for QM9 and ZINC training dataset. The unigram probabilities corresponds to the frequency of elements in the dataset. Figure adapted from Olsen et al[36]

6.2.2 Bag of Words

The unigram model might be a too naive model so to have a stronger baseline we also use the bag of atoms, and bag of neighbours introduced in Section 4.4. The

implementation embeds the input atoms – using an embedding function – sum them together, apply several layers of feed forward neural network and finally converts the output to probabilities using the softmax function:

$$P(v_i = \mathcal{C}_k|\widetilde{\boldsymbol{v}}, \boldsymbol{E}) = \text{Softmax}\left(\boldsymbol{h}_{\boldsymbol{i}}^{(l)}\right)_k \tag{6.1}$$

$$\boldsymbol{h}_{\boldsymbol{i}}^{(l)} = \text{ReLU}\left(\boldsymbol{\Theta}^{(l)}\boldsymbol{h}_{\boldsymbol{i}}^{(l-1)} + \boldsymbol{\theta}^{(l)}\right) \tag{6.2}$$

$$\boldsymbol{h}_{\boldsymbol{i}}^{(0)} = \boldsymbol{z_i} \tag{6.3}$$

$$\boldsymbol{z_i} = \sum_j \boldsymbol{x_j} \tag{6.4}$$

$$\boldsymbol{x_i} = \text{Embedding}(v_i), \tag{6.5}$$

where the (l) superscript corresponds to the number of hidden layers in the feed forward neural network, and the sum in (6.4) either is over all atoms or just the neighbours, corresponding to bag of atoms, or bag of neighbours.

6.2.3 Graph Transformer

As our most complex model we use the transformer, which we recall is defined as

$$P(v_i = \mathcal{C}_k|\widetilde{\boldsymbol{v}}, \boldsymbol{E}) = \text{Softmax}\left(\boldsymbol{h}_{\boldsymbol{i}}^{(l)}\right)_k \tag{6.6}$$

$$\boldsymbol{h}_{\boldsymbol{i}}^{(l)} = \text{Layernorm}\left(\boldsymbol{z}_{\boldsymbol{i}}^{(l)} + \text{FNN}(\boldsymbol{z}_{\boldsymbol{i}}^{(l)})\right) \tag{6.7}$$

$$\boldsymbol{z}_{\boldsymbol{i}}^{(l)} = \text{Layernorm}\left(\boldsymbol{h}_{\boldsymbol{i}}^{(l-1)} + \text{Attention}(\boldsymbol{h}_{\boldsymbol{i}}^{(l-1)}, \boldsymbol{h}^{(l-1)})\right) \tag{6.8}$$

$$\boldsymbol{h}_{\boldsymbol{i}}^{(0)} = \boldsymbol{x_i} \tag{6.9}$$

$$\boldsymbol{x_i} = \text{Embedding}(v_i), \tag{6.10}$$

where the attention mechanism in (6.8) is one of the three types introduced in Section 4.5. We propose two variants of this model; one where we feed information about the bond order, i.e $\boldsymbol{E}_i j \in \{0, 1, 2, 3\}$, which we call *bond transformer*; and one where we make the adjacency matrix binary, thus only encoding information about which atoms are neighbours called *binary transformer*. This is done first of all to investigate how much information we gain by including the bond order. Furthermore, the notion of bond order is not always trivial as molecules in practice can have bond orders that are not discrete. For example in a benzene ring there are two equally good configurations of the bonds as seen in Figure 6.2a. According to valence theory the molecule will resonate between these two and the bond order will be an average of the two configurations – 1.5 in this case – as shown in Figure 6.2b [29]. For carbon rings this is called an *aromatic bond* but many other cases exists.

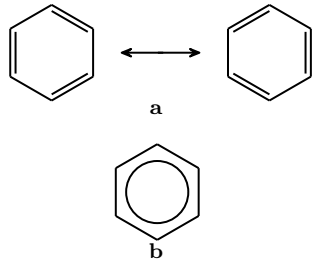


Figure 6.2: Structure diagram of benzene C_6H_6. A line corresponds to a carbon carbon covalent bond, and the hydrogen atoms are implicit. (a) The two resonance configurations of bonds. (b) The mean of the resonance structures, corresponding to 1.5 order bonds.

6.2.4 Octet Rule Unigram

To compare our models with some of the heuristics currently used to determine valid molecules we introduce a model based on the octet rule. As discussed in Section 4.2, the octet rule can be used – to some extent – to also describe molecules with charged atoms. It is however too liberal as it allows many molecules that would not exists. The model introduced is therefore a slight simplification; we ignore the possibility of charged atoms, and only allow elements that match the number of covalent bonds. I.e. if the masked atom has three covalent bonds the possible elements are nitrogen and phosphor. Since we need to give probabilities for the predicted elements we use the unigram probabilities which can be seen in Table 6.2 for the QM9 dataset.

		Number of covalent bonds			
		1	2	3	4
Elements	P(**H**)	0.9965	0	0	0
	P(**F**)	0.0035	0	0	0
	P(**O**)	0	1	0	0
	P(**N**)	0	0	1	0
	P(**C**)	0	0	0	1

Table 6.2: Probabilities for each element given the number of covalent bonds – for QM9 – using the octet rule unigram model

The ZINC dataset contains molecules with charged atoms and hypervalent molecules. If we only use the simplified octet rule and unigram probabilities from above, this will result in giving zero probability to certain samples, as our model for example not allows for more than 4 covalent bonds. This will in turn cause us to get perplexities of infinity. To solve this we use k-smoothing from Section 3.2.2. The value of k is

optimized on the validation dataset and, as can be seen in Figure 6.3, the optimal value is found to be $k = 1842$. The resulting probabilities can be see in Table 6.3, where we also have extended the model to allow for 5 or 6 covalent bonds. In this case we just give uniform probabilities as this should not be possible, given the elements in the dataset.

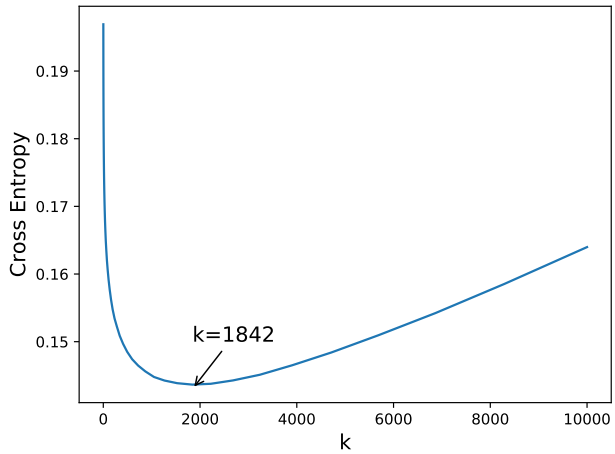


Figure 6.3: Cross entropy as a function of k-smoothing evaluated on the ZINC validation dataset.

		Number of covalent bonds					
		1	2	3	4	5	6
Elements	P(**H**)	0.9658	0.0038	0.0039	0.0006	0.1	0.1
	P(**F**)	0.0179	0.0038	0.0039	0.0006	0.1	0.1
	P(**Cl**)	0.0097	0.0038	0.0039	0.0006	0.1	0.1
	P(**Br**)	0.0034	0.0038	0.0039	0.0006	0.1	0.1
	P(**I**)	0.0007	0.0038	0.0039	0.0006	0.1	0.1
	P(**O**)	0.0005	0.8274	0.0039	0.0006	0.1	0.1
	P(**S**)	0.0005	0.1422	0.0039	0.0006	0.1	0.1
	P(**N**)	0.0005	0.0038	0.9647	0.0006	0.1	0.1
	P(**P**)	0.0005	0.0038	0.0041	0.0006	0.1	0.1
	P(**C**)	0.0005	0.0038	0.0039	0.9946	0.1	0.1

Table 6.3: Unigram probabilities for octet rule model on the ZINC training dataset.

6.3 Comparing Graph Attentions

In Section 4.5 we introduced three different attention mechanisms. We start off our experiments with comparing these. This is done by training three binary and bond transformers – one for each attention mechanism – on the QM9 and ZINC datasets. The transformers have 4 layers, with 3 number of attention heads and an embedding dimension of 64.

In Figure 6.4 we see the evolution of the validation perplexity as the models are trained. Here we see that for both the bond and binary transformer – across both datasets – the full graph attention performs best; it converges to a much lower and more stable minimum. Since this is the only attention mechanism that is conditioned on the entire graphs – and not just the neighbouring atoms – this makes sense.

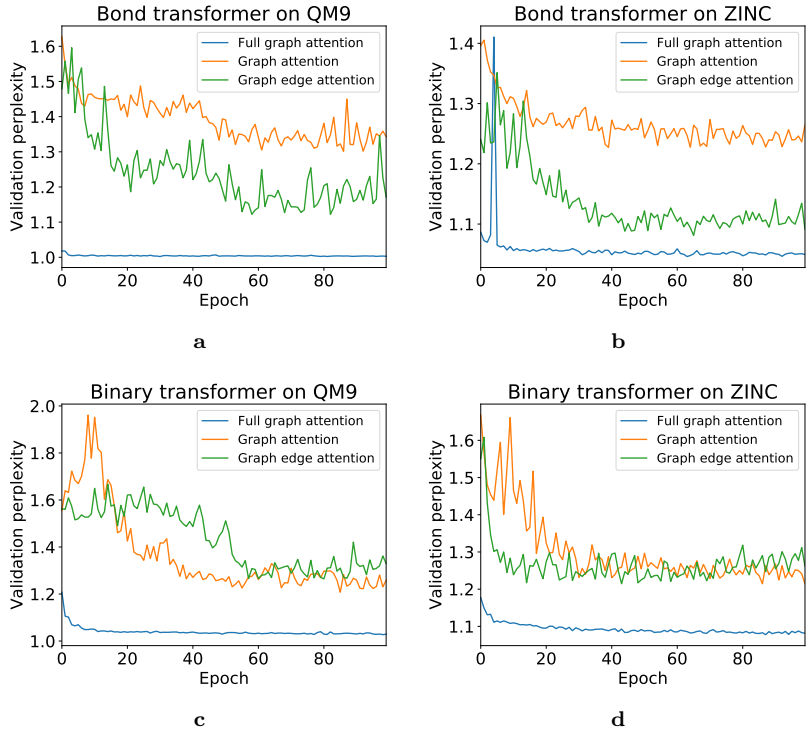


Figure 6.4: Perplexity on validation dataset – with one atom masked per atom – for each epoch of training. (a) is a bond transformer trained on QM9, (b) is a bond transformer trained on ZINC, (c) is a binary transformer trained on QM9 and (d) is a binary transformer trained on ZINC.

The difference between the bond and binary transformer indicates another result; for the bond transformer, the graph edge attention performs better than the graph attention on both datasets. This indicates that the additional information about bond order, given to the graph edge attention, provides useful information to the model. As the bond order gives information about how many valence electrons are available, this is no surprise. For the binary transformer, we do not give any bond order information. This causes both graph attention and graph edge attention to get the same information, and we see that these models converges to very similar performance. As the full graph attention seems to be superior for our problem we shall only consider this from now on.

6.4 Epsilon-greedy

In Section 5.2.2 we introduced the ε-greedy strategy used for masking atoms during training. The reasoning was that we expected it to help create more robust models. To test this we trained two bond and binary transformers – with and without the ε-greedy strategy – on the QM9 and ZINC datasets. Similar to previous section, we used 4 layers with 3 attention heads, embedding dimension of 64 and used $n_{masks} = 1$ during training. In Figure 6.5 we see the resulting perplexity on the validation dataset, where we have masked different number of atoms per molecule. As all models are trained with $n_{masks} = 1$, they all perform well for one masked atom. However, as soon as we start masking more atoms per molecule the two models trained without ε-greedy ($\varepsilon = 0$) performs increasingly worse. The two models trained with $\varepsilon = 0.2$ does however seem to be much more robust so therefore all models will be trained with this, going forward.

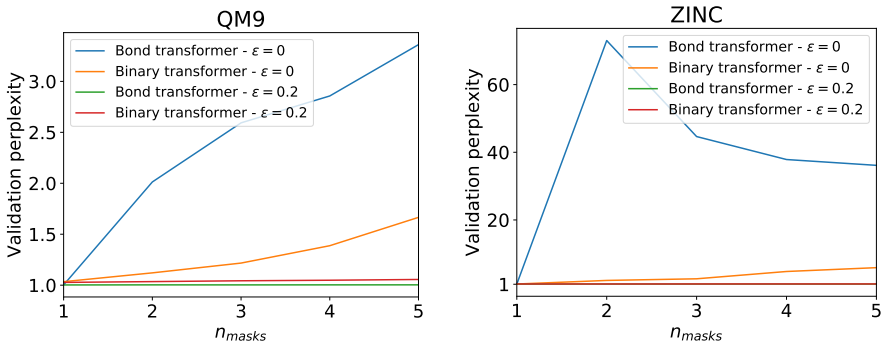


Figure 6.5: Validation perplexity of binary and bond transformer – with and without ε-greedy masking strategy – with different number of masked atoms. (a) is on the QM9 dataset and (b) is on the ZINC dataset

6.5 Results

In this section we will present our main results on the QM9 and ZINC datasets. First we will discuss how to evaluate our models, and then we will introduce the quantitative and qualitative results.

6.5.1 Evaluation metrics

So far we have considered perplexity the main evaluation metric of language modelling. As we are doing classification we might also be interested in how often we predict the correct atom instead of only considering probabilities. Accuracy is a very easy to interpret metric but in our case it might be misleading, as the data is very biased. In this case it is common to use the F1 metric which is a harmonic average of the precision and recall [25]:

$$\text{F1} = \frac{2}{\text{precision}^{-1} + \text{recall}^{-1}} \tag{6.11}$$

The F1 metric has the problem that it is only defined for binary classification. There does however exists multiple ways of using it for multi-class classification. Two of these are the F1-micro and F1-macro. In the F1-micro we pool all the true positives, false negatives and false positives before calculating the F1 score as in (6.11). For the F1-macro we calculate the F1 for each class and then takes the unweighted average. The F1-macro therefore puts more emphasis on underrepresented classes as all classes have the same weight regardless of occurrence in the dataset, compared to F1-micro [25].

As discussed in Section 3.2.1 we often just ignore the ambiguity of languages in NLP and use the sample average of our metric. We are however in a situation where we have a more well defined description of when and how ambiguity in the target class happens, based on the octet rule. For QM9 we know that the only ambiguity is between hydrogen and fluoride, which both has one valence electron and are considered equally correct. Since we are interested in comparing our models to a heuristic based on the octet rule, we introduce an *octet F1* metric, which considers the prediction as correct if the prediction and target has the same number of valence electrons (e.g hydrogen and fluoride). We can therefore use the metric to determine how good our model is – compared to model based on the octet rule – as our octet rule unigram model will get 100% octet F1 on the QM9 dataset. The sample F1 might, however, still be useful – especially for the ZINC dataset – as the octet rule only is a heuristic which assumptions are not always satisfied.

6.5.2 Learning Octet Rule on QM9

We train our transformer models with 8 layers, 4 attention heads and an embedding dimension of 64. The Bag of Words models only has 4 layers as we found more layers would cause the models to become unstable and overfit. The results of our models on the QM9 dataset can be seen in Table 6.4 where we present the octet F1 micro, octet F1 macro and sample perplexity with one masked atom per molecule. Here we see that our bond transformer achieves close to a perfect score for all three metric. Since the dataset is fully explainable by the octet rule and we provide the model bond order information, the problem is, however, easy to solve by simply counting the number of covalent bonds of the masked atom. As we are using the octet F1 score this would give 100%. It is therefore expected that the bond transformer should be able to solve this simple task.

More impressive is the performance of the binary transformer; even without any bond order information it is able to get more than 99% octet F1 micro. As we only are masking one atom – and the dataset is defined by the octet rule – we should still be able to recover the masked atom deterministically, even without bond order information; only elements with a certain number of valence electrons will allow all atoms in the molecule to satisfy the octet rule. This problem, however, requires you to potentially consider all the atoms in the molecule in order to be able to deduce the correct answer.

Model	Octet F1 micro	Octet F1 macro	Perplexity
`bond-transformer`	**99.99**±0.01	**99.99**±0.01	**1.002**±0.001
`binary-transformer`	99.73 ±0.01	93.44 ±4.20	1.009 ±0.002
`bag-of-neighbors`	90.67 ±0.01	77.18 ±0.01	1.281 ±0.004
`bag-of-atoms`	65.77 ±4.48	44.30 ±4.92	3.310 ±0.478
`Unigram`	47.32	32.85	3.104
`octet-rule-unigram`	100	100	1.002

Table 6.4: Performance of our models for 1 masked atoms per molecule on the QM9 test dataset. The uncertainty corresponds to the standard deviation of ten models, trained with different start seed. Figure adapted from Olsen et al.[36]

The bag of neighbours provides a strong baseline achieving over 90% octet F1 micro. This indicates that a lot of information is available just in the neighbours of the masked atoms. This makes sense as only considering the number of neighbours can provide information about which elements to exclude; e.g if the masked atom has four neighbours the only possible element is carbon. The bag of atoms model outperforms the unigram model, but both models have rather weak performance.

From this experiment, we argue that the bond transformer – and to some extent the binary transformer – is able to learn the simple structure rule that defines the

QM9 dataset.

As the transformer models has several hyper parameters, we also experimented with different model sizes. This can be seen in Table 6.5 where we vary the number of layers, heads and embedding dimension to test very small all the way up to a large model sizes for the binary transformer. Here we see that we are able to get comparable performance to the bag of neighbours model, even with a very small model of only 265 parameters. As we increase the embedding size – and hence number of parameters – we see a large improvement in performance. We choose to use 8 layers, 6 attention heads and an embedding size of 64 for all our experiments, as we consider this a reasonable compromise between performance and computational requirements.

Model	Octet F1 micro	Perplexity	t (min)	#Params
`l=1, heads=1,` d_{emb}`=4`	86.0	1.426	60	199
`l=2, heads=1,` d_{emb}`=4`	89.9	1.261	63	265
`l=2, heads=3,` d_{emb}`=64`	96.3	1.089	77	118149
`l=4, heads=3,` d_{emb}`=64`	98.4	1.031	82	234885
`l=8, heads=6,` d_{emb}`=64`	99.7	1.008	110	866181

Table 6.5: Performance of `binary-transformer` models with different number of trainable parameters, for one masked atoms per molecule. `l` is the number of layers, `heads` the number of attention heads, d_{emb} the embedding dimension and t the training time. Figure adapted from Olsen et al[36]

Finally, we considered if the length of the molecule had an influence on the performance. These results we did, however, not find that interesting; the models performed the worst on very small molecules – which is most likely due to very few occurrences in the dataset – and gets increasingly better as the molecules becomes larger – which is might be due to larger molecules mainly consisting of hydrogen and carbon. See Figure A.2

6.5.3 ZINC

Our goal for the ZINC dataset is to see if our models are able to learn structure rules that exceeds the octet rule. The results can be seen in Table 6.6 for one masked atom per molecule. Here, we first of all notice that the octet rule unigram model does not get 100% octet F1, which indicates that the dataset cannot be fully explained by the octet rule. We see that both the bond and binary transformer performs similar or better than the octet rule unigram model. Since the dataset has more ambiguity than QM9 it is interesting to consider the sample F1 scores where both transformer models outperforms the octet rule unigram. This is especially the case for sample F1 macro, which indicates that the models has learned – to some extent – to discriminate between some elements that would be considered equally likely using the octet rule.

Model	Octet F1 (micro/macro)	Sample F1 (micro/macro)	Perplexity
`bond-transformer`	**99.52**±0.04 / **97.97**±3.17	**98.64**±0.03 / **62.67**±3.19	**1.047**±0.001
`binary-transformer`	99.13 ±0.05 / 91.38 ±4.94	98.18 ±0.06 / 55.76 ±4.89	1.063 ±0.002
`octet-rule-unigram`	99.17 / 88.65	97.22 / 38.48	1.164
`bag-of-neighbors`	90.73 ±0.03 / 76.50 ±0.45	89.00 ±0.03 / 29.47 ±0.37	1.412 ±0.004
`bag-of-atoms`	50.84 ±0.30 / 56.75 ±0.58	49.06 ±0.32 / 9.84 ±0.53	3.135 ±0.073
`Unigram`	48.05 / 56.40	46.10 / 6.31	3.221

Table 6.6: Performance of our models for 1 masked atoms per molecule on the ZINC test dataset. The uncertainty corresponds to the standard deviation of ten models, trained with different start seed. Figure adapted from Olsen et al.[36]

As our models can handle masking multiple atoms per molecule it might be interesting to investigate how the performance is affected by increasing number of masks. This can be seen in Figure 6.6, which shows the perplexity of molecules with masked up to 80 atoms – corresponding to masking all atoms. See Figure B.1 for similar plot of the remaining metrics. Here we see that the bond transformer is almost unaffected by the increasing number of masks; even without any information about the atoms in the molecule it still performs very well. This indicates that the model mostly uses the structural and bond order information. As the number of valence electrons – and therefore which group of elements – most of the time can be determined by the bond order alone, this might be expected.

For the binary transformer we see that the perplexity does increase, as we increase the number of masked atoms. The perplexity plateau around 50 masked atoms per molecule, just above the perplexity of the octet rule unigram model. A similar thing is seen for the bag of neighbours model. With all masks the bag of neighbours model has essentially only information about the number of neighbours. The fact that the model still has around half the perplexity of the unigram and bag of atoms, indicate that just the number of neighbours provides a lot of information. As mentioned previously this is no surprise.

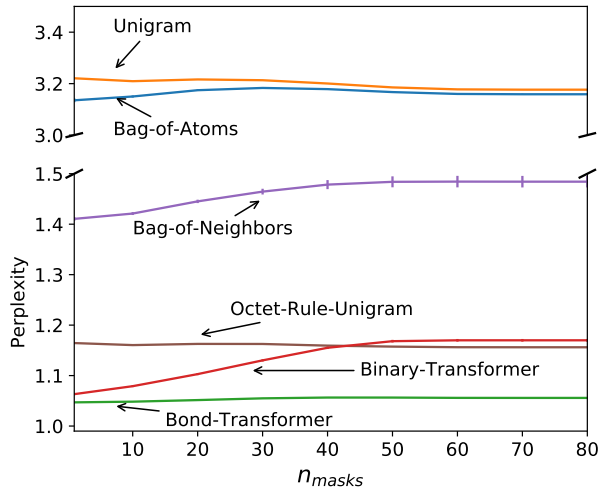


Figure 6.6: Sample perplexity evaluated by different number of masked atoms. Error bar corresponds to standard deviation of 10 models trained with different start seed. Figure taken from Olsen et al.[36]

To investigate to what degree our models have learned to discriminate between ions, hypervalent elements and ambiguous elements, we generate confusion matrices based on the number of covalent bonds the masked atom has. The confusion matrix for atoms with four covalent bonds can be seen in Figure 6.7. In this case the possible targets for the ZINC dataset are C, N^+ ions and a hypervalent S. We see that while the octet rule unigram model – by design – only predicts C, both the bond and binary transformer has learned to correctly discriminate between the three elements to some extent. A similar result can be seen for two covalent bonds in Figure B.3 from the appendix.

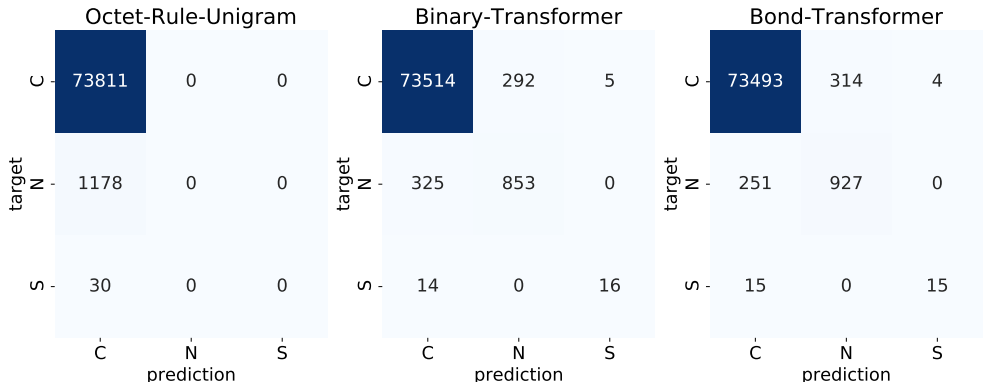


Figure 6.7: Confusion matrix for the ZINC test dataset, with $n_{masks} = 1$, where the masked atom has four covalent bonds. Figure taken from Olsen et al.[36]

From Figure 6.8 we have the confusion matrix for atoms with six covalent bonds. This consists exclusively of hypervalent S. Both the binary and bond transformer correctly predicts all the atoms. As sulfur is the only element with six covalent bonds in the dataset, this should be easy for the model to learn. Furthermore we found that the hypervalent S often occurs in the dataset in a specific configuration with two double bonded oxygen, a nitrogen and an oxygen (similar to Figures 6.10a and 6.10b). Nevertheless, this is still an phenomenon that is not explained using the octet rule.

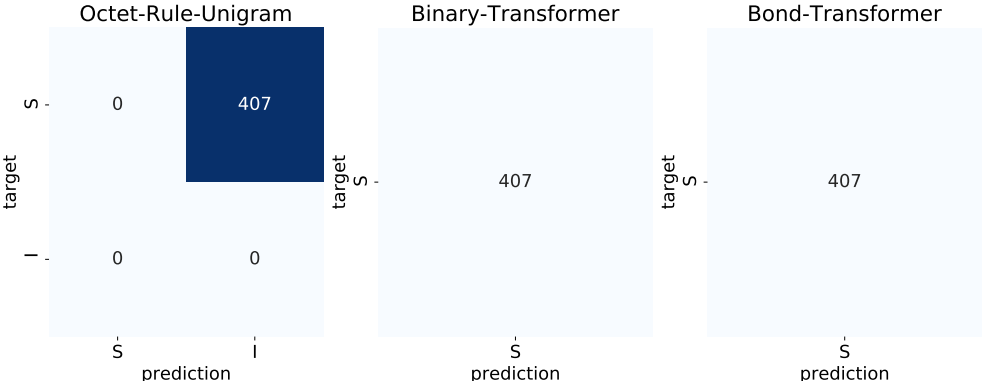


Figure 6.8: Confusion matrix for the ZINC test dataset, with $n_{masks} = 1$, where the masked atom has six covalent bonds. Figure taken from Olsen et al.[36]

Finally, we look at atoms with only one covalent bond. In this case our dataset contains a lot of ambiguity since H, F, Cl, Br and I all have 7 valence electrons. Furthermore, the ZINC dataset has O^- and S^- ions. From Figure 6.9 we see that both the binary and bond transformer has learned to distinguish hydrogen and fluoride –

and even oxygen ions – to some extent. The remaining elements are, however, mostly not predicted correctly. This might be due to the elements being very underrepresented in the training data.

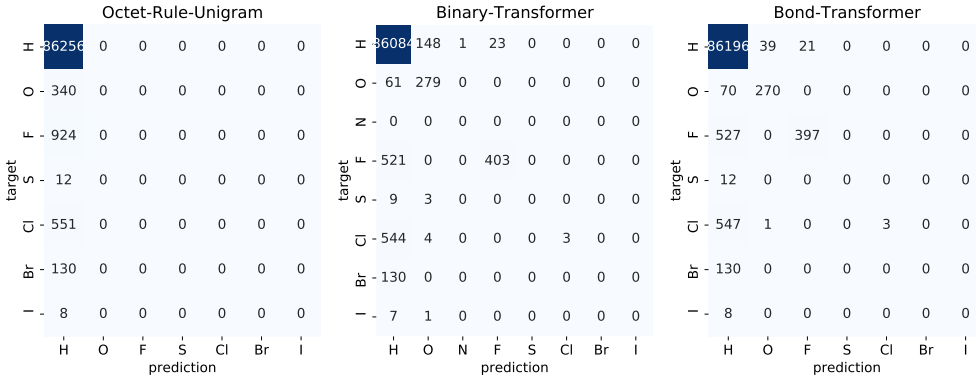


Figure 6.9: Confusion matrix for the ZINC test dataset, with $n_{masks} = 1$, where the masked atom has one covalent bonds. Figure taken from Olsen et al.[36]

As our final results we inspect a few predictions on the ZINC dataset to qualitatively evaluate our model. We choose to use the binary transformer as we deem it the most interesting due to the very good performance even without bond order information. Six predictions are chosen – three correct and three incorrect – and their predicted probabilities over elements are shown in Figure 6.10. In Figure 6.10a we see that the model is able to correctly predict nitrogen even though it is a negative nitrogen ion which is very rare in the dataset. The model also gives a reasonable high probability to oxygen, which is very reasonable from the perspective of the octet rule. The specific configuration of a hypervalent sulphur double bonded to two oxygen, a carbon and a nitrogen is common in the dataset. This might offer an explanation why the binary transformer is able to correctly predict the nitrogen. This mentioned configuration can also be seen in Figure 6.10b where our model is very certain that the correct atom is sulphur.

The example in Figure 6.10c is also predicted correct with very high model certainty. In this case, however, we do not have an immediate explanation why the model is so sure. This indicates that the model has learned some structure in the molecule that makes sulphur more likely.

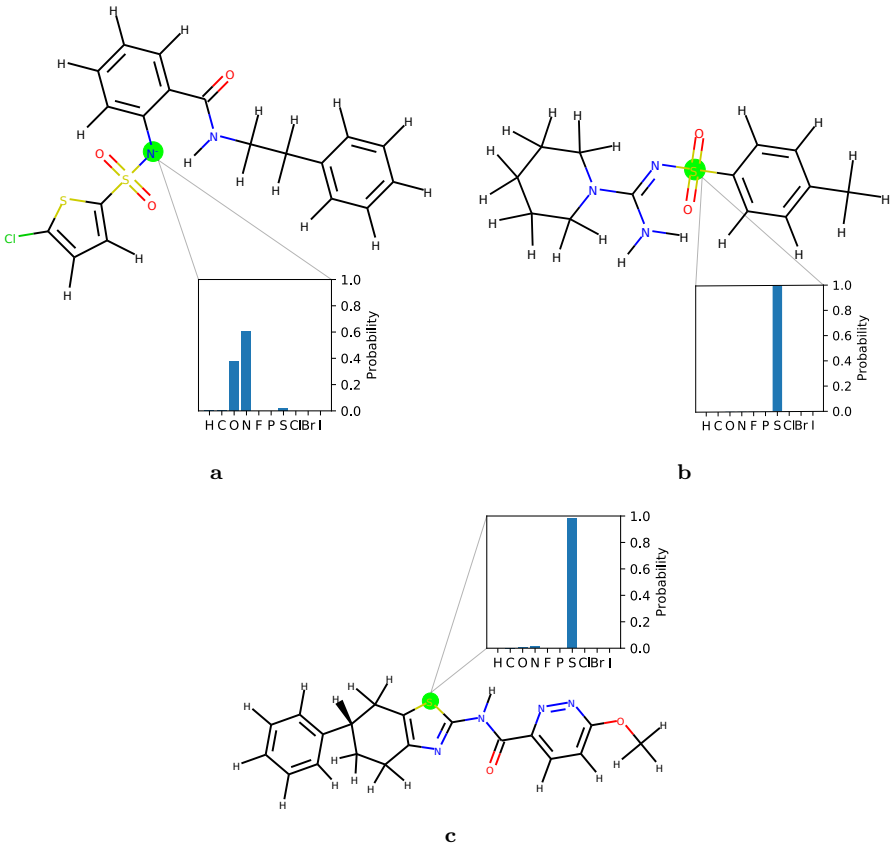


Figure 6.10: Predicted atom probabilities. The molecule corresponds to the true molecule, where the colored atom is the target we want to predict. Green corresponds to correct predictions. Figure adapted from Olsen et al[36]

Next, we consider some incorrect predictions. As we have many ambiguous elements with one covalent bonds with a very bias distribution towards hydrogen it is worth checking if the predicted probabilities have the same bias. In Figure 6.11a the model incorrectly predicts hydrogen. The model is, however, in doubt and makes the correct element the second most likely, even though chloride only appears half as often as fluoride. This suggests that even though hydrogen is incorrectly predicted the model has learned some structure that makes chloride more likely than fluoride for this specific molecule. Furthermore we see that all the elements with non-zero probability corresponds to elements with 7 valence electrons. This is similar to what we seen in Figure 6.11b. Here we again predicts the wrong element, but the model is in doubt and puts second most probability on the correct element. The predicted

sulphur would also be considered a valid prediction under the octet rule. Finally, in Figure 6.11c we have an example where the model makes a wrong prediction but is very certain. One could argue that without bond order information, guessing hydrogen would make sense. We, however, know that the model should be able to deduce that the masked element should have two covalent bonds.

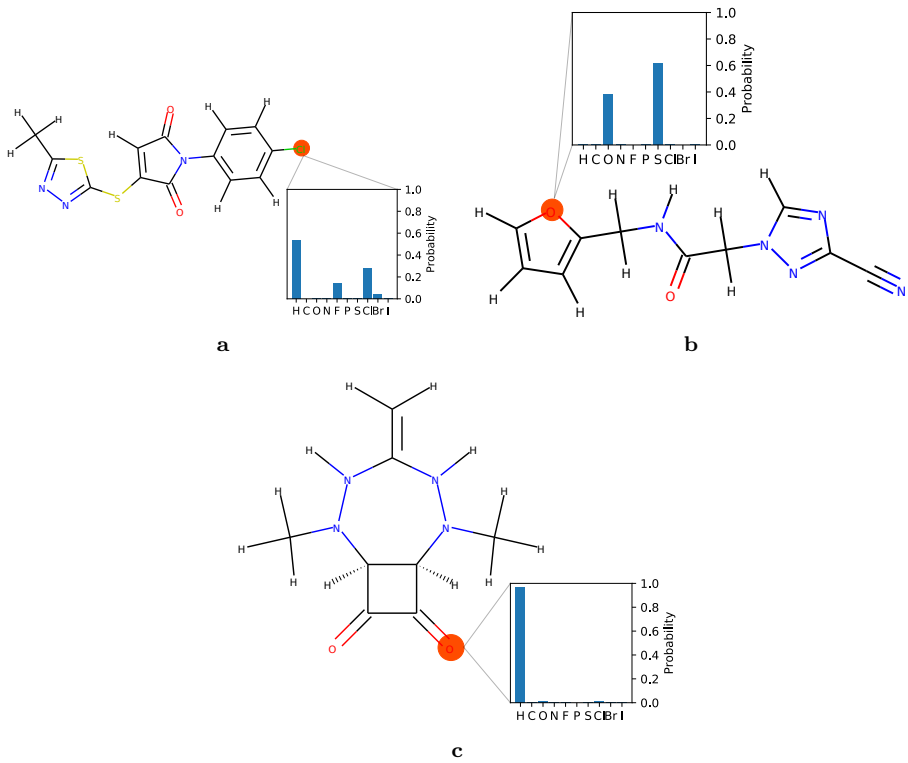


Figure 6.11: Predicted atom probabilities. The molecule corresponds to the true molecule, where the colored atom is the target we want to predict. Red corresponds to wrong predictions. Figure adapted from Olsen et al[36]

CHAPTER 7

Conclusion

In this thesis we have introduced the task of masked language modelling to molecular graphs. This is done to try and model the underlying structure rules in molecules which often are based on – but not limited to – the octet rule.

We extend several models from the NLP literature – like unigram and Bag of Words models – to work on graphs, including proposing a new attention mechanism for graphs that is conditioned on all atoms and bonds in the molecule. This attention mechanism is compared to two other mechanisms – one conditioned on neighbouring atoms and bonds and one only conditioned on the neighbour atoms. From our experiments we conclude that our new attention mechanism performs significantly better for our problem.

This new attention mechanism is used to propose two variations of the current state of the art architecture in NLP – the transformer – that works on graphs; one that is given bond order information and one that only has information about connectivity.

We evaluate our models on the QM9 and ZINC datasets. From the QM9 dataset we see that the binary and bond transformer achieves (99.73% / 93.44%) and (99.99% / 99.99%) octet F1 (micro / macro). We argue that this shows that the bond transformer – and to some extent the binary transformer – has learned the octet rule which is the underlying structure rule of the dataset. When evaluated on the ZINC dataset both transformer models are able to outperform the octet rule unigram model across all metrics. Most interesting is perhaps the fact that the binary and bond transformer achieves (98.18% / 55.76%) and (98.64% / 62.67%) sample F1 (micro / macro). Together with the analysis of confusion matrices we argue that our transformer models not only have learned the existence of ions and hypervalent molecules, but also to discriminate between elements – that would be considered equally likely under the octet rule – to some extent.

7.1 Future Work

The main limitation of our models are the fact that we only can mask atoms and not bonds. Instead of assuming the bonds are given and only predicting the atoms it could be interesting to investigate if our models also would be able to learn how

the bonds are structured in a molecule. By being able to predict bonds this would also allow us to create a generative model; by calling the model repeatedly we would be able to generate new molecules. This is similar to the MolecularRNN model of Popova et al.[41], but our model would be based on the transformer architecture.

From our experiments we found that the full graph attention we proposed outperforms the attention mechanisms only conditioned on neighbourhood information. As most graph neural networks follows the message parsing scheme of repeatedly updating the state of a node – based on the neighbours – it would be interesting to investigate if our full graph attention transformer model also could be usefull in other graph based applications.

APPENDIX A

QM9

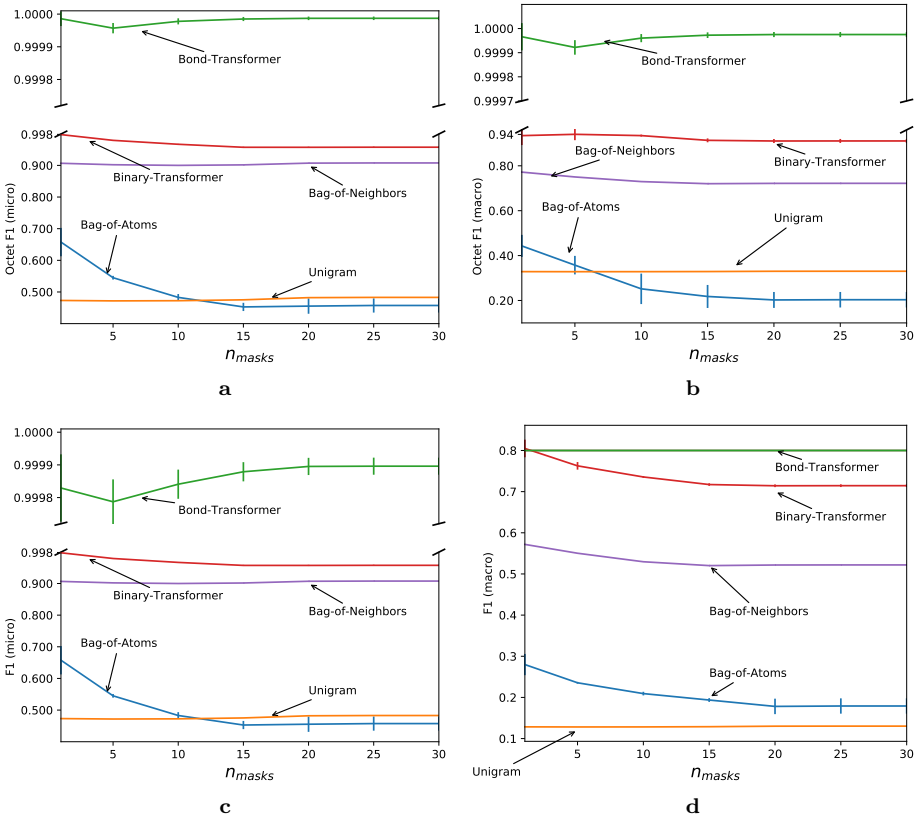


Figure A.1: Octet F1 micro (a), octet F1 macro (b), sample F1 micro (c) and sample F1 macro (d) evaluated by different number of masked atoms on the QM9 test dataset. Error bar corresponds to standard deviation of 10 models trained with different start seed. Figure taken from Olsen et al.[36]

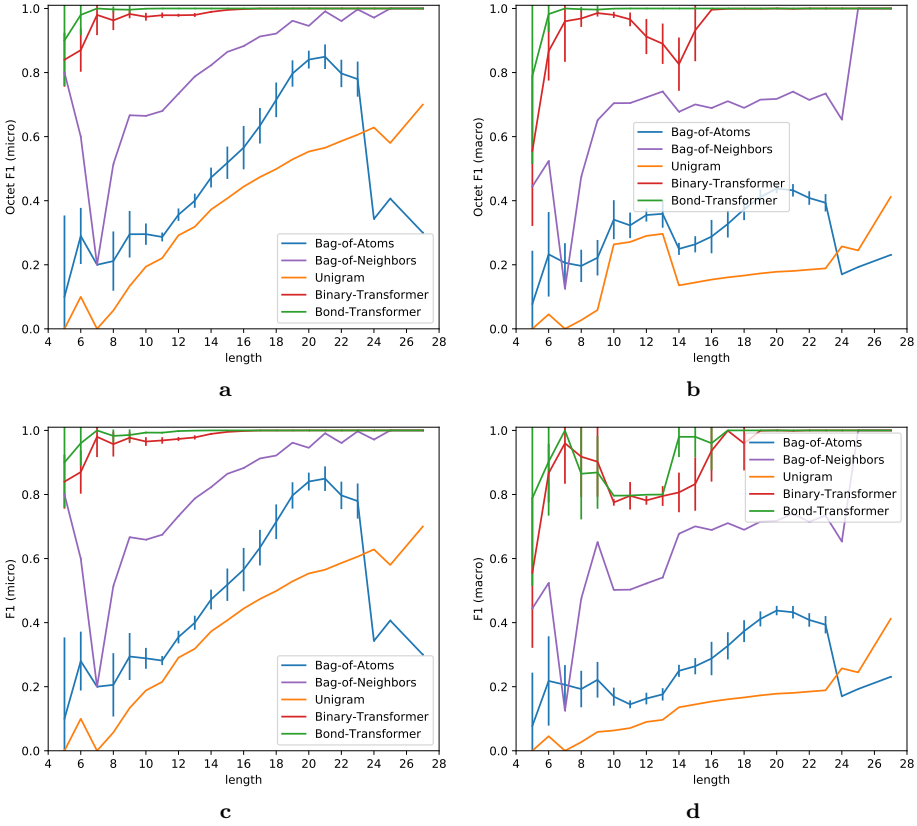


Figure A.2: Octet F1 micro (a), octet F1 macro (b), sample F1 micro (c) and sample F1 macro (d) evaluated on molecules of varying size with 1 atom masked on the QM9 test dataset. Error bar corresponds to standard deviation of 10 models trained with different start seed. Figure taken from Olsen et al.[36]

APPENDIX B

Zinc

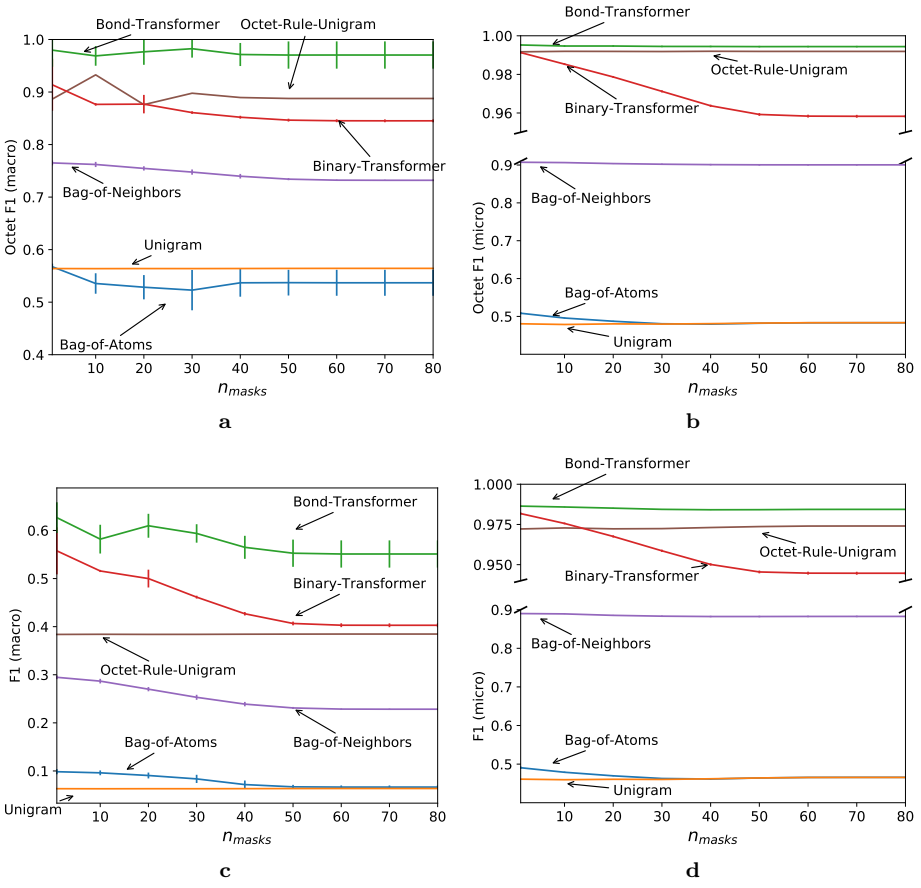


Figure B.1: Octet F1 micro (a), octet F1 macro (b), sample F1 micro (c) and sample F1 macro (d) evaluated by different number of masked atoms on the ZINC test dataset. Error bar corresponds to standard deviation of 10 models trained with different start seed. Figure taken from Olsen et al.[36]

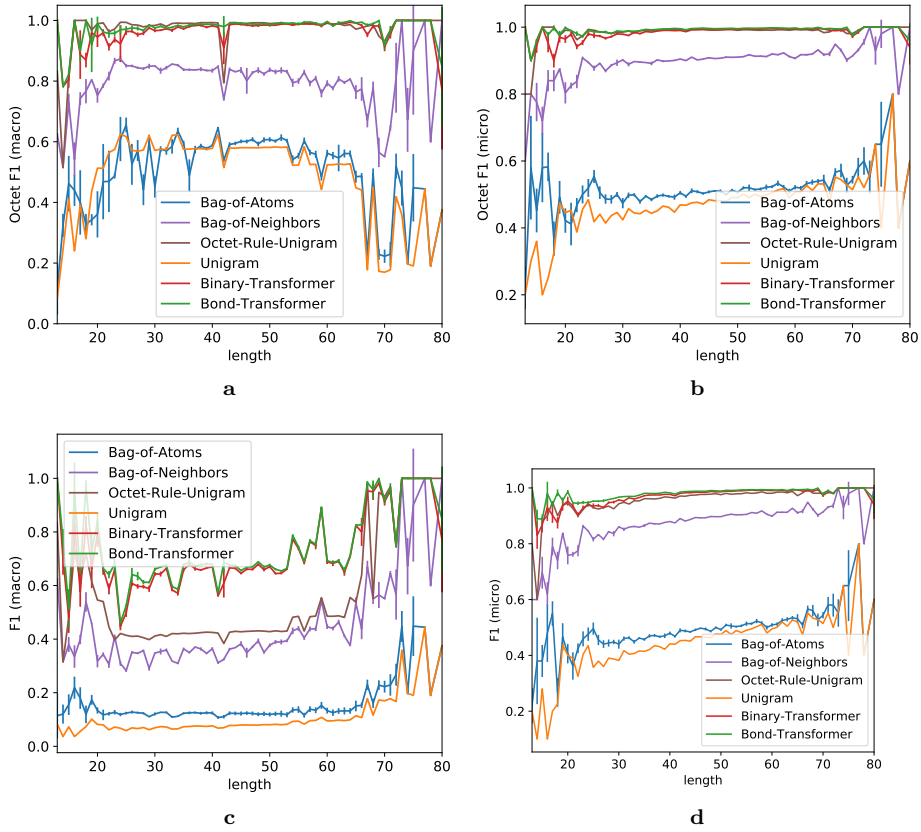


Figure B.2: Octet F1 micro (a), octet F1 macro (b), sample F1 micro (c) and sample F1 macro (d) evaluated on molecules of varying size with 1 atom masked on the QM9 test dataset. Error bar corresponds to standard deviation of 10 models trained with different start seed. Figure taken from Olsen et al.[36]

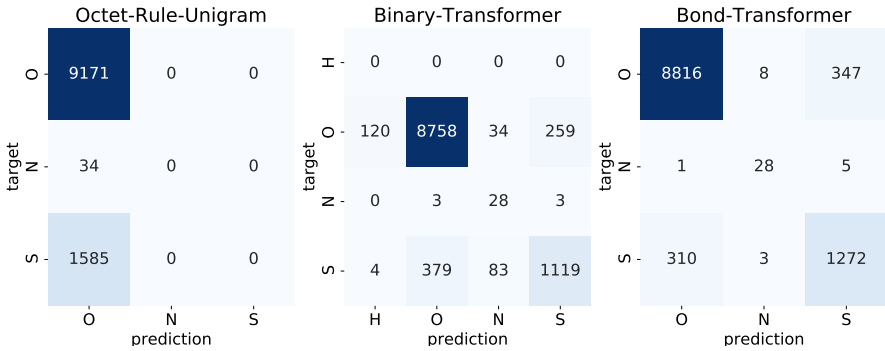


Figure B.3: Confusion matrix for the ZINC test dataset, with $n_{masks} = 1$, where the masked atom has two covalent bonds. Figure taken from Olsen et al.[36]

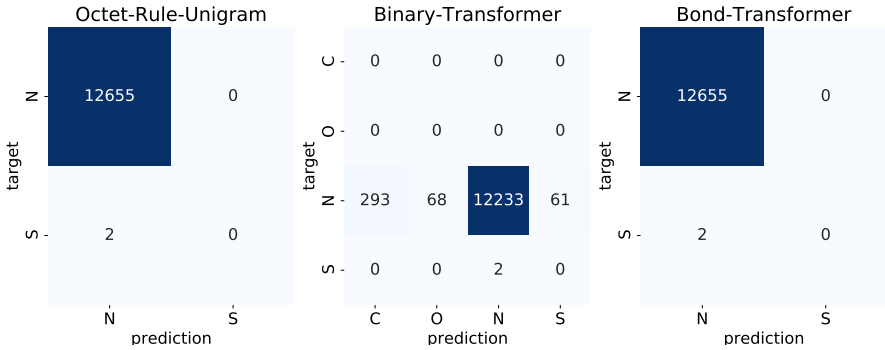


Figure B.4: Confusion matrix for the ZINC test dataset, with $n_{masks} = 1$, where the masked atom has three covalent bonds. Figure taken from Olsen et al.[36]

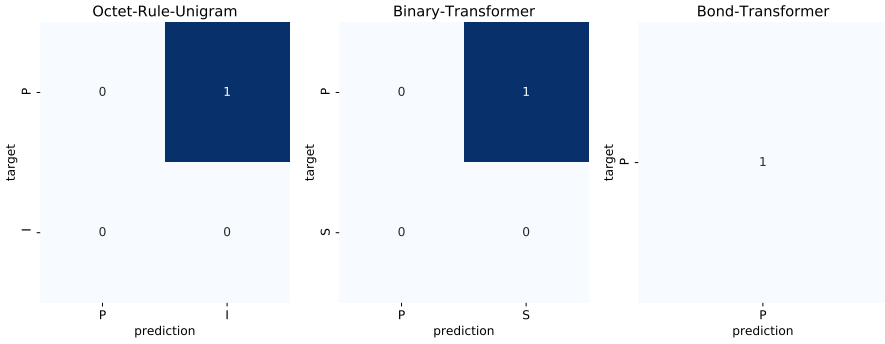


Figure B.5: Confusion matrix for the ZINC test dataset, with $n_{masks} = 1$, where the masked atom has five covalent bonds. Figure taken from Olsen et al.[36]

Bibliography

[1] Martín Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

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

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[4] Guy W Bemis and Mark A Murcko. "The properties of known drugs. 1. Molecular frameworks". In: *Journal of medicinal chemistry* 39.15 (1996), pages 2887–2893.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[6] Jianpeng Cheng, Li Dong, and Mirella Lapata. "Long short-term memory-networks for machine reading". In: *arXiv preprint arXiv:1601.06733* (2016).

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

[8] Kyunghyun Cho et al. "On the properties of neural machine translation: Encoder-decoder approaches". In: *arXiv preprint arXiv:1409.1259* (2014).

[9] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[10] Daniel C Elton et al. "Deep learning for molecular design-a review of the state of the art". In: *Molecular Systems Design & Engineering* (2019).

[11] 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, pages 1263–1272.

[12] Rafael Gómez-Bombarelli et al. "Automatic chemical design using a data-driven continuous representation of molecules". In: *ACS central science* 4.2 (2018), pages 268–276.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[14] Gabriel Lima Guimaraes et al. "Objective-reinforced generative adversarial networks (ORGAN) for sequence generation models". In: *arXiv preprint arXiv:1705.10843* (2017).

[15] David Harrison Jr and Daniel L Rubinfeld. "Hedonic housing prices and the demand for clean air". In: *Journal of environmental economics and management* 5.1 (1978), pages 81–102.

[16] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[17] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pages 770–778.

[18] Stephen Heller et al. "InChI-the worldwide chemical structure identifier standard". In: *Journal of cheminformatics* 5.1 (2013), page 7.

[19] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pages 1735–1780.

[20] Weihua Hu et al. *Pre-training Graph Neural Networks*. 2019. arXiv: `1905.12265 [cs.LG]`.

[21] John J Irwin et al. "ZINC: a free tool to discover chemistry for biology". In: *Journal of chemical information and modeling* 52.7 (2012), pages 1757–1768.

[22] Sabrina Jaeger, Simone Fulle, and Samo Turk. "Mol2vec: unsupervised machine learning approach with chemical intuition". In: *Journal of chemical information and modeling* 58.1 (2018), pages 27–35.

[23] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. "Junction tree variational autoencoder for molecular graph generation". In: *arXiv preprint arXiv:1802.04364* (2018).

[24] Peter Bjørn Jørgensen, Karsten Wedel Jacobsen, and Mikkel N Schmidt. "Neural message passing with edge updates for predicting properties of molecules and materials". In: *arXiv preprint arXiv:1806.03146* (2018).

[25] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2009. ISBN: 0131873210.

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

[27] Greg Landrum. *RDKit: Open-source cheminformatics*. URL: `http://www.rdkit.org`.

[28] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pages 2278–2324.

[29] Libretexts. *Introduction to Inorganic Chemistry*. September 2019. URL: https://chem.libretexts.org/Bookshelves/Inorganic_Chemistry/Book%3A_Introduction_to_Inorganic_Chemistry.

[30] Zhouhan Lin et al. "A structured self-attentive sentence embedding". In: *arXiv preprint arXiv:1703.03130* (2017).

[31] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. "Effective approaches to attention-based neural machine translation". In: *arXiv preprint arXiv:1508.04025* (2015).

[32] Adam C Mater and Michelle L Coote. "Deep learning in chemistry". In: *Journal of chemical information and modeling* 59.6 (2019), pages 2545–2559.

[33] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[34] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[35] N. M. O'Boyle et al. "Open Babel: An open chemical toolbox". In: *J. Cheminf* (2011).

[36] Jeppe Johan Waarkjær Olsen et al. "Autoencoding undirected molecular graphswith neural networks". In: *preprint* (2019).

[37] Ankur P Parikh et al. "A decomposable attention model for natural language inference". In: *arXiv preprint arXiv:1606.01933* (2016).

[38] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Edited by H. Wallach et al. Curran Associates, Inc., 2019, pages 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[39] Jeffrey Pennington, Richard Socher, and Christopher Manning. "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pages 1532–1543.

[40] Matthew E Peters et al. "Deep contextualized word representations". In: *arXiv preprint arXiv:1802.05365* (2018).

[41] Mariya Popova et al. "MolecularRNN: Generating realistic molecular graphs with optimized properties". In: *arXiv preprint arXiv:1905.13372* (2019).

[42] Edward O Pyzer-Knapp et al. "What is high-throughput virtual screening? A perspective from organic materials discovery". In: *Annual Review of Materials Research* 45 (2015), pages 195–216.

[43] Raghunathan Ramakrishnan et al. "Quantum chemistry structures and properties of 134 kilo molecules". In: *Scientific data* 1 (2014), page 140022.

[44] David Rogers and Mathew Hahn. “Extended-connectivity fingerprints”. In: *Journal of chemical information and modeling* 50.5 (2010), pages 742–754.

[45] Chetan Rupakheti et al. “Strategy to discover diverse optimal molecules in the small molecule universe”. In: *Journal of chemical information and modeling* 55.3 (2015), pages 529–537.

[46] Benjamin Sanchez-Lengeling and Alán Aspuru-Guzik. “Inverse molecular design using machine learning: Generative models for matter engineering”. In: *Science* 361.6400 (2018), pages 360–365.

[47] Mike Schuster and Kuldip K Paliwal. “Bidirectional recurrent neural networks”. In: *IEEE Transactions on Signal Processing* 45.11 (1997), pages 2673–2681.

[48] Kristof Schütt et al. “Schnet: A continuous-filter convolutional neural network for modeling quantum interactions”. In: *Advances in Neural Information Processing Systems*. 2017, pages 991–1001.

[49] Marwin HS Segler et al. “Generating focused molecule libraries for drug discovery with recurrent neural networks”. In: *ACS central science* 4.1 (2017), pages 120–131.

[50] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. “Self-attention with relative position representations”. In: *arXiv preprint arXiv:1803.02155* (2018).

[51] Richard S Sutton and Andrew G Barto. “Reinforcement Learning: An Introduction”. In: ().

[52] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pages 5998–6008.

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

[54] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. “Pointer networks”. In: *Advances in Neural Information Processing Systems*. 2015, pages 2692–2700.

[55] *Weights & Biases*. Version 0.8.9. September 2019. URL: `https://www.wandb.com/`.

[56] 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), pages 31–36.

[57] 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), pages 31–36.

[58] David Weininger, Arthur Weininger, and Joseph L Weininger. “SMILES. 2. Algorithm for generation of unique SMILES notation”. In: *Journal of chemical information and computer sciences* 29.2 (1989), pages 97–101.

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

[60] Zonghan Wu et al. “A comprehensive survey on graph neural networks”. In: *arXiv preprint arXiv:1901.00596* (2019).

[61] Kelvin Xu et al. “Show, attend and tell: Neural image caption generation with visual attention”. In: *International conference on machine learning*. 2015, pages 2048–2057.

[62] Keyulu Xu et al. “How powerful are graph neural networks?” In: *arXiv preprint arXiv:1810.00826* (2018).

[63] Naruki Yoshikawa et al. “Population-based de novo molecule generation, using grammatical evolution”. In: *Chemistry Letters* 47.11 (2018), pages 1431–1434.

[64] Jie Zhou et al. “Graph neural networks: A review of methods and applications”. In: *arXiv preprint arXiv:1812.08434* (2018).

{2.7182818284}ϑφερτυθιοπσδφγηξκλ