technische universität
dortmund

Bachelor's Thesis

Scaling up the Equation-Encoder - Handling High Data Volume through the Efficient Use of Trainable Parameters

Jonathan Schill

March 2020

Reviewers:
Prof. Dr. Katharina Morik
M. Sc. Lukas Pfahler

TU Dortmund University
Department of Computer Science
Artificial Intelligence Group
http://ai-www.cs.tu-dortmund.de

Contents

1 Introduction **1**

1.1 Literature Search . . . 1
1.2 Math-based Literature Search . . . 2
1.3 Objective of this Thesis . . . 3
1.4 Structure of this Thesis . . . 4

2 Principles **5**

2.1 Machine Learning . . . 5
2.2 Deep Learning . . . 6
2.2.1 Deep Feedforward Networks . . . 6
2.2.2 Types of Layers . . . 7
2.2.3 Learning Distributed Representations . . . 10
2.2.4 Backpropagation . . . 11
2.2.5 Architectural Choices and Hyperparameters . . . 12

3 Related Work **17**

3.1 The Equation-Encoder . . . 17
3.1.1 Equation-Encoder Network Architecture . . . 19
3.2 SqueezeNet . . . 19
3.2.1 The Fire Module . . . 20
3.2.2 Overall Architecture . . . 22

4 Experiments **23**

4.1 Experimental Setup . . . 23
4.1.1 Dataset . . . 23
4.1.2 Performance Measures . . . 26
4.1.3 Baseline Equation-Encoder . . . 26
4.1.4 Squeezed Equation-Encoder . . . 27
4.1.5 Miscellaneous . . . 28
4.2 Experiments . . . 29
4.2.1 Experiment I - Vanishing Gradient . . . 29

4.2.2 Experiment II - Activation and Initialisation . . . 31
4.2.3 Experiment III - Learning Rate . . . 32
4.2.4 Experiment IV - Sampling via Citation Graph . . . 33
4.2.5 Experiment V - Margin . . . 34
4.2.6 Final Configuration . . . 35
4.2.7 Experiment VI - Full Dataset . . . 35

5 Evaluation **37**
5.1 Comparison . . . 37
5.1.1 Loss on Test Data . . . 37
5.1.2 Size and Speed . . . 38
5.1.3 User Study . . . 39
5.1.4 Conclusion . . . 41
5.2 Quality of the Search Engine . . . 41

6 Conclusion and Outlook **45**

A Further Information **47**
A.1 Proof: Why do we need nonlinear activation functions? . . . 47
A.2 Activation Functions . . . 48
A.3 Triples . . . 49
A.4 Search Results from User Study - SEE . . . 52
A.5 Search Results from User Study - BEE . . . 59

Notation **67**

List of Acronyms **68**

List of Figures **70**

List of Tables **72**

Bibliography **73**

Chapter 1

Introduction

This chapter should give a quick overview of the problem at hand. The following questions shall be answered: What problem do we want to solve? Why is this problem important? Why is it not a trivial problem? What will be the procedure in this thesis?

1.1 Literature Search

Literature search is a substantial part of the scientific process. Publishing papers and reading other people's papers is the way researchers all over the world communicate and cooperate with each other. If this exchange of knowledge does not work properly, the ability to benefit from research done by other people than yourself is strongly limited. For a working scientific system, it must be possible for researchers to find relevant publications. However, finding relevant papers is not a simple task. As the world gets more connected, the number of researchers cooperating with each other grows. This results in a rapidly growing amount of possibly relevant publications. In 2018 36,000 computer science related papers were uploaded on the pre-print service arXiv.org alone (See Figure 1.1). As Pfahler et al. [27] already stated in their work it is infeasible to filter, index or organise literature manually because of the sheer number of available documents.

Nowadays a big part of this work is done by search engines that are specialised for literature research, e.g. Google Scholar. These search engines make it possible to search for natural language keywords in all publications available in the search engine's database. A task that would take an unbelievable amount of person-hours if done manually. But even after this filter, the number of resulting papers maybe so high that reading all of them is impractical. To tackle this problem, most search engines do not only filter all available papers with a set of keywords, they also rank the results according to their relevance. The relevance may be influenced by different factors including number of matched keywords, number of citations or recency. Keyword-based search engines have simplified literature search and have become an indispensable part of the scientific process.

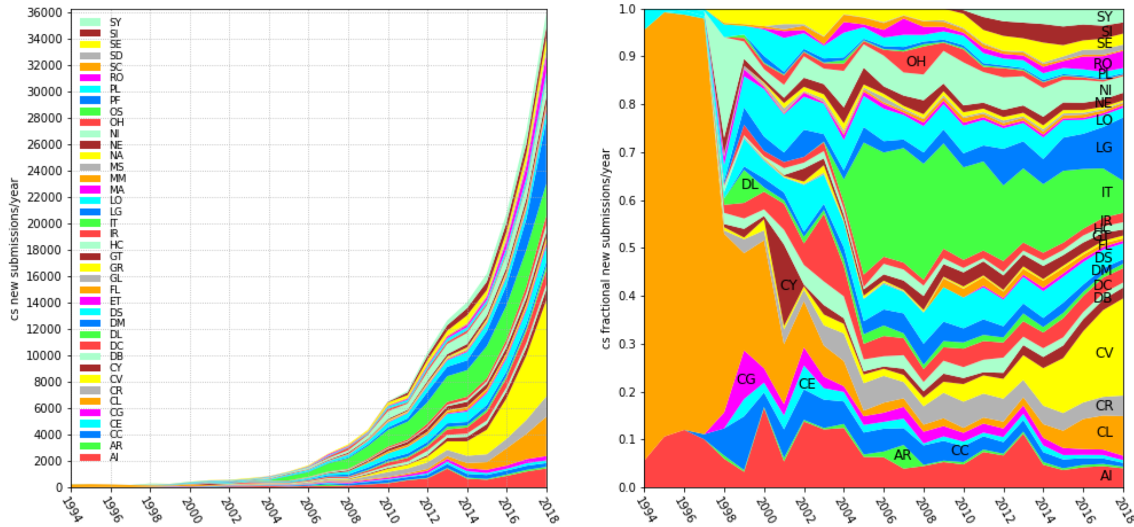


Figure 1.1: Yearly submission rates of computer science papers on the pre-print service arXiv.org. The letter combinations are acronyms for the different subject areas (e.g. LG stands for Machine Learning and CV stands for Computer Vision). A full list of the acronyms can be found here: here. Source: arXiv.org.

1.2 Math-based Literature Search

A limitation to this keyword-based approach is that in many research fields natural language is not the only component of a publication. In subjects like mathematics, computer science, physics etc. you will find a lot of mathematical expressions as well. These expressions are arguably as important as the natural language text surrounding them - if not more important. Some ideas presented in a paper are way easier represented by a formula or a collection of formulas than by natural language text. If natural language could express anything a formula can in a comparable compact way, formulas would not be used after all. Researchers in the respective fields often make use of this property of formulas when doing literature search. Instead of reading a possibly relevant paper completely they might just look at the formulas in it and make their decision concerning the relevance of this paper based on the formulas they have seen [27]. Given the role that mathematical expressions play in many scientific manuscripts, one may have the impression that for literature search using formulas has great potential.

There have been various efforts to make use of the potential of formula-based literature search [22, 35, 34]. But, these efforts have not lead to a widespread use of math-based search engines yet. This is probably the case because these search engines are simply not good enough. Evaluating similarity between mathematical expressions is hard. For the semantics of a formula the overall structure is very important. Singular symbols or short sequences of symbols can mean very different things since their interpretation is highly dependant on context. This property makes math-based search distinct from keyword

search. In natural language a single keyword or a short sequence of them is often expressive and their meaning is less dependant on context (see following Examples). For this reason techniques that work well for natural language, do usually not work well for mathematical expressions.

Example 1.2.1 $\sum_{x=1}^{n}(x-1)$ and $1/(x-1)$
Although both expressions contain the subexpression $(x-1)$ the expressions have very little in common on a semantic level.

Example 1.2.2 2^n and n^2
The semantic of both expressions is quite dissimilar although the expressions share the exact same symbols (Example taken from Kamali et al. [15]).

Example 1.2.3 Paper titles that contain the same word combination.
For example both of these titles contain the combbiniation *Embedding Learning*:
"Feature Selection via joint Embedding Learning and Sparse Regression" and
"Sampling Matters in Deep Embedding Learning ".
These two papers have arguably something in common on a semantic level.

Example 1.2.4 "This is our loss."
The semantic of this senetence can also change depending on its context. E.g. in context of machine learning or in the context of a funeral. This example shows that words can also change their semantics dependant on the context. However, it does not occure as often as with formulas.

1.3 Objective of this Thesis

In order to overcome some of the difficulties regarding math-based literature search, Pfahler et al. [27] proposed the Equation-Encoder (see Section 3.1): A Convolutional Neural Network (CNN) (see Definition 2.2.1) that evaluates similarities between bitmap representations of mathematical expressions. The Equation-Encoder seems to be able to solve the task at hand to a certain degree. Yet it is questionable whether it is powerful enough to work properly as a backend for a real-world search engine. There is much space for improvement and some of the possible improvements shall be investigated in this work. One obvious point of attack is the data that is used for training. The Equation-Encoder was trained with data that was curated from the pre-print service arXiv.org. This service gives "open access to 1,637,485 e-prints in the fields of physics, mathematics, computer science, quantitative biology, quantitative finance, statistics, electrical engineering and systems science, and economics."[1]. Pfahler et al. [27] used only a small subset with ∼44,000

[1] Source: arXiv.org, visited on 27.12.2019

publications from only one research area. The work of Sun et al. [31] suggests that training on a much larger and more diverse dataset might improve the performance of our model. But, as the volume of the dataset increases, the throughput of the Equation-Encoder becomes more important. Even on the small dataset used by Pfahler et al. [27] a single training run with 30 epochs takes about 24 hours on an Nvidia GTX-1080 GPU. Considering that we will have to deal with a dataset that is more than ten times larger, using the original Equation-Encoder is impracticle. In order to use a larger dataset, we need a new, more efficient model. This will be the main contribution of this work: Making experiments using a much larger dataset and designing a model that is more suitable for a dataset of this size.

1.4 Structure of this Thesis

This thesis is structured as follows: We will begin with a brief introduction to the theoretical foundations of the methods that are important for this work - namely machine learning and deep learning. With these basics covered, we will take a look at two deep learning architectures that are crucial for this work: the Equation-Encoder itself and an architecture called SqueezeNet [11] that will be the main inspiration for the new model. In the following chapter, we will derive the exact architecture and configuration of this new model by combining both architectures that were presented earlier and by conducting experiments regarding the architecture of the network and the configuration of the learning algorithm. The resulting model and a baseline model[2] will then be trained on a dataset that is substantially larger than the dataset from Pfahler et al. [27]. This allows us to evaluate the performance of the proposed model in relation to the baseline's performance. Amongst other evaluation methods, there will be a small user study which aims at assessing the real-word applicability of both models. This thesis will end with a summary of the obtained results and an outlook on future work.

[2]The Equation-Encoder [27] with a few adjustments.

Chapter 2

Principles

This chapter shall give the reader a basic understanding of the underlying principles of the methods used in this work. We will start with a very brief introduction into the field of machine learning using the example of our math-similarity problem and continue with a more comprehensive introduction into the subfield of deep learning. My remarks regarding these topics will focus on the aspects that are particularly important for the scope of this thesis.

2.1 Machine Learning

Mitchell et al. [23] defines machine learning as follows: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.".

There are many variants of machine learning methods and a wide range of tasks that we can solve with machine learning. However, we will focus on how machine learning is applicable to our problem. The question is therefore: What does machine learning mean regarding our problem of math-based literature search?

Our task is to evaluate semantic similarity between mathematical expressions. A broader and more precise overview of different possible performance measures will be given in Chapter 5, but in principle we just need some mathematical expressions with some true annotation about their semantic relation. We may then just let our model (computer program) calculate its estimate of the semantic similarity between some of the given mathematical expressions and compare them to the relations that are given by the annotations. The closer our model's predictions are to the annotations, the better we consider the performance of our model. The experience E, that shall help our model to improve its performance regarding P, is of the same form as the data that we need for our

performance measure. Again we need mathematical expressions and annotations about their semantic relation[1].

The method that we will use for learning from E to get better at similarity evaluation between mathematical expressions is called Deep Feedforward Network (DFN). Its functionality will be described in the next section.

2.2 Deep Learning

Deep learning is a subfield of machine learning and a powerful method for certain tasks. It caused some major breakthroughs for different machine learning related tasks e.g. image classification [18] or playing the complex board game *Go* at human-level [30]. The topic is particularly important for this work because the Equation-Encoder [27] is a deep learning method. The following section shall give an overview of some basic deep learning aspects and some more advanced aspects that are relevant for reconstructing the Equation-Encoder in a more efficient way.

2.2.1 Deep Feedforward Networks

The following definition and explanation of DFNs is based on the popular textbook of Goodfellow et al. [6]. A DFN[2] defines some function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$. It consists of a set of several parametrized subfunctions $\mathbb{L} = \{l_1, l_2, ..., l_n\}$ that are consecutively applied to the input $\vec{x}$ and a directed acyclic graph $\mathcal{G}$ that defines the order in which the subfunctions are applied. A subfunction from $\mathbb{L}$ is also called *layer*. We refer to l_k as the kth layer of our network. The first layer l_1 is called input layer and l_n is called output layer. All other layers are called hidden layers. The depth of a network is defined as $|L|$. This is where the term *deep learning* comes from. Together with their order, the layers from L determine the network's function f. The output of this function $f(\vec{x}) \in \mathbb{R}^m$ for a specific input $\vec{x} \in \mathbb{R}^n$ is given by

$$f(\vec{x}) = l_n(l_{n-1}(...(l_1(\vec{x})))).$$

The purpose of a DFN is to approximate a function $f^* : \mathbb{R}^n \rightarrow \mathbb{R}^m$. To this end, the network learns all the parameters of its layers θ. Ordinarily the desired function f^* is given by a dataset $\mathbb{D}$ that contains approximate examples of f^*. These examples consist of a feature vector $\vec{x}$ and a label $\vec{y} \approx f^*(\vec{x})$. For the learning process, an *objective function* is defined in order to measure how good our current network is at approximating f^*. Since this objective function is often measuring some error, it is also called *loss function* or

[1] The data that we use for evaluation and the data that we provide to the computer program in order to learn (or train) should be disjunct. Otherwise the program could achieve an optimal performance by just storing the exact examples from the training set.

[2] These models are originally inspired by neural activity in the brain. Hence there are also called Neural Networks (NNs) in literature.

cost function. A simple example for such a loss function on a single training example $(\vec{x}, \vec{y})$ with $f^*(\vec{x}) \approx \vec{y}$ is the squared error: $J(x) = (f(\vec{x};\theta) - \vec{y})^2$.

Of course, this loss function is usually not computed on a single example, but on a set of examples. During the learning process, we alter the network's parameters in a way that minimises the cost function. We want to achieve that for as many examples $(\vec{x}, \vec{y})$ as possible $f(\vec{x}) \approx \vec{y} \approx f^*(\vec{x})$. The optimistation process is usually done with some gradient descent method [3]. Because it would be infeasible for large datasets to compute the loss and the respective gradient for the whole dataset at once, we divide the dataset into mini-batches, compute the cost and the respective gradient for every one of these batches and then perform a gradient descent step for every gradient [6, pp.274-276].

2.2.2 Types of Layers

Fully-connected layers (cf. [6, p. 168]) A fully-connected layer has a weight matrix $W_k \in \mathbb{R}^{m \times n}$, a bias vector $\vec{b}_k \in \mathbb{R}^m$ and an activation function $a : \mathbb{R} \to \mathbb{R}$. The output $\vec{y} \in \mathbb{R}^m$ of a fully-connected layer for a specific input $\vec{x} \in \mathbb{R}^n$ is given by

$$\vec{y} = l_k(\vec{x}) = a(W_k\vec{x} + \vec{b}_k).$$

The activation function is applied elementwise to the vector that is resulting from $(W_k\vec{x} + \vec{b}_k)$. Because of this activation function, the output of a layer itself is often called *activation*. The output values of the layer before a is applied are called *pre-activations*.

If the respective layer is the output layer of a DFN, a might be the identity function: $i(x) = x$. In all other cases, the activation function should be nonlinear. Otherwise, the transformation performed by two following fully-connected layers could also be done by one layer (Explanation in the Appendix A.1). Hence, activation functions are often called *nonlinearities*. A commonly used activation function is *ReLU* [6, p. 189] (see Appendix A.2).

Convolutional layers (cf. [6, chapter 9]) Although convolutional layers may be used with input tensors of any dimensionality, we will focus on the special case of a convolutional layer for which the input has three dimensions, since this work deals with 3-dimensional tensors that represent images. The input could also be seen as a set of matrices that are concatenated in a third dimension. A single matrix corresponds to a color channel of e.g. an RGB-image. We will refer to a single matrix as a channel or a feature map.

A convolutional layer consists of a number of *kernels* (also called *filters*), a bias vector $\vec{b} \in \mathbb{R}^{c_{out}}$ and an activation function $a : \mathbb{R} \to \mathbb{R}$. Each of these kernels is a 3D-Tensor of the form $K \in \mathbb{R}^{c_{in} \times w \times h}$. It defines an operation $* : \mathbb{R}^{c_{in} \times m \times n} \to \mathbb{R}^{m' \times n'}$. The output $(X * K) \in \mathbb{R}^{m' \times n'}$ of such a kernel operation is a single feature map. For the overall output of the convolutional layer, all resulting feature maps are combined by concatenating them in a third dimension. So, the function defined by a convolutional layer is of the form

$l_{conv} : \mathbb{R}^{c_{in} \times m \times n} \rightarrow \mathbb{R}^{c_{out} \times m' \times n'}$, where c_{out} equals the number of kernels of the respective layer.

In order to understand how the convolution operation $*$ of a kernel K works, it may be helpful to think of the K as a window that is gliding over the input [6, p. 330] (see Figure 2.1). We start by placing the kernel in the top-left corner of the input X. At this point for every value $k \in K$ there is a value $x \in X$ that is in the same place. Now, for

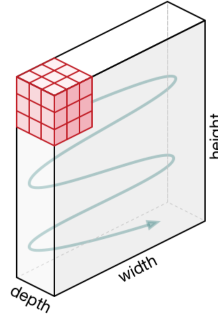


Figure 2.1: Illustration of how the kernel (red cube) is placed over the input tensor and of the movement of the kernel. Source: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

every value of k and x that are in the same place, we multiply these values and add the products together. For our output matrix $Z \in \mathbb{R}^{m' \times n'}$ the resulting sum defines the value in the top-left corner $z_{0,0}$. To compute the value next to it $z_{1,0}$, we move the kernel one step to the right. Then we can again multiply the values in the same place and add them together. We do this for all values in the first row until we reach $z_{m,0}$. The next step is to move the kernel down and to move successively to the left from thereon. This gives us the values in the row $\vec{z}_{\bullet,1}$. We continue this movement of the kernel until we shifted it along the whole input. The mathematical formulation of this operation[3] is:

$$z_{i,j} = (X * K)_{i,j} = \sum_d \sum_w \sum_h x_{d,i+w,j+h} k_{d,w,h} \tag{2.1}$$

$$Z = (X * K) \tag{2.2}$$

With a well defined convolutional operation we are now also able to define the function of a convolutional layer l_{conv}. For an input tensor X, a set of filters $\mathbb{K} = \{K_0, ..., K_n\}$, a bias vector $\vec{b}$ and the activation function a, a singular feature map Y_j is given by:

$$Y_j = a(b_j + (X * K_j)) \tag{2.3}$$

Note that b_j is a scalar value. It is added elementwise to each value of the matrix resulting from $(X * K_j)$.

[3] Technically this is cross-correlation and not a convolution. However, these operations are used synonymously in literature [6, p. 329]. The implementation of convolutional layers used in this work [26] implements cross-correlation.

In our example we shift the kernel always by one while gliding over the input. It is also possible that it is shifted by more than one. The value by that we shift a kernel is called *stride* [6, p. 343]. For kernels with height or width greater than one, the output dimensions are smaller than the input dimensions. But, we may want to maintain the dimensions. To achieve this, we increase the effective height and width of the input tensor by adding new scalars with value zero to its margin. This extra added values are called *padding* [6, p. 343]. A DFN that uses at least one convolutional layer is called a CNN (see Definition 2.2.1).

This architecture is particularly useful for high-dimensional input, where some spatial relation between the different features (scalars in the input) plays an important role. E.g. we want to process some image and we represent the image as a Tensor $X \in [0,1]^{3\times 32\times 32}$. We could use fully-connected layers by transforming our input to a vector with $3 * 32^2 = 3072$ dimensions, but there would be two problems: First, this would lead to a huge amount of trainable parameters in the layer's weight matrix. Even if we scaled the dimensionality of the output of the first layer down to 1024, we would have $3072 * 1024 = 3\,145\,728$ parameters for our weight matrix. In contrast, a convolutional layer with 128 kernels of dimensionality $3 \times 3 \times 3$ would require only $3^3 * 128 = 3456$ parameters. This allows us to deal with high-dimensional input with a relatively small amount of trainable parameters [6, p. 330]. Second, important patterns may be found in different places. If our images showed animals and we wanted to train our DFN to recognize what animals are depicted, it would be useful for the network to detect certain shapes that are characteristic for certain animals e.g. the shape of a beak for detecting a duck. Unfortunately, for different images the beak is most certainly at different places. A single weight of a fully-connected layer does only correspond to one single value (or pixel) of the input. If we succesfully train several weights to detect a beak at one place, but for the next image the beak is at some other place, these weights will not help to detect this second beak. Convolutional kernels do not have this problem because they are moved over the input image and every weight corresponds to multiple values in the input [6, pp. 332-333].

Pooling layers (cf. [6, pp. 335-339]) Pooling layers are layers without trainable parameters. It is their purpose to reduce the dimensionality of the tensors passed through the network. A pooling layer defines a function $l_p : \mathbb{R}^{c\times m\times n} \rightarrow \mathbb{R}^{c\times m'\times n'}$ with $m > m', n > n'$. Another property of pooling layers is that their output is approximately invariant to small changes of the input. I.e. if some values in the input change by a small margin, in most cases the output does not change. The reduction of dimensionality is done by gliding a rectangular window over the singular channels of the input and computing some summary statistics of the values within this window at every position (See Figure 2.2).

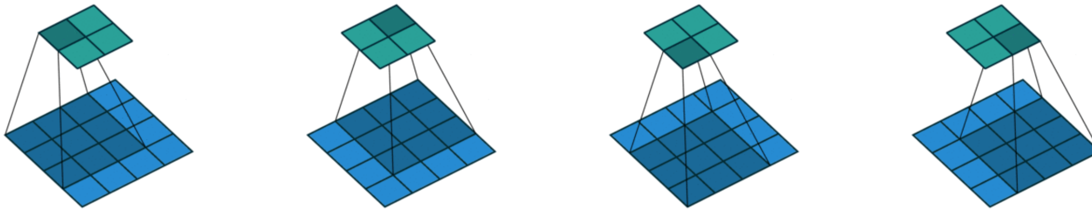


Figure 2.2: Illustration of how a 3×3 pooling window is placed over the input (blue/bottom grid) to produce the output (cyan/top grid). Source [4].

There are two types of pooling layers that we will use in this work:

1. *Max pooling* For max pooling the summary statistic is computed by returning the highest value within the window at the current position.

2. *Average pooling* For average pooling the summary statistic is computed by returning the mean value of all values within the window at the current postion.

Definition 2.2.1 Convolutional Neural Networks
CNNs are DFNs with at least one convolutional layer. Usually they consist of some convolutional layers, some pooling layers and optionally some fully-connected layers at the end of the network.

Batch-Normalisation layers Batch-Normalisation [13] is a technique for stabilising the training process. This is done by ensuring that the input for a layer always comes from a normal distribution with a mean of zero and a variance of one. To this end, each element of the input is normalised batchwise. Let $X \in \mathbb{R}^{N \times n}$ be a matrix with a mini-batch of N feature vectors with dimensionality n. With $\vec{x}^{(k)}$ we denote a vector containing the kth value of each vector in our mini-batch. For each dimension we normalise $\vec{x}^{(k)}$:

$$\vec{x'}^{(k)} = \frac{\vec{x}^{(k)} - \mathbb{E}[\vec{x}^{(k)}]}{\sqrt{\mathrm{Var}[\vec{x}^{(k)}]}} \tag{2.4}$$

This might constrain the model expressibility. Most pre-activation would be in the interval $[-1, 1]$. Hence, we introduce a scale parameter $\gamma^{(k)}$ and a shift parameter $\beta^{(k)}$ for each activation $\vec{x}^{(k)}$. These parameters are trainable and learned along with all other parameters of our network. The transformation with γ and β is of the form:

$$\vec{y}^{(k)} = \gamma^{(k)} \vec{x'}^{(k)} + \beta^{(k)} \tag{2.5}$$

2.2.3 Learning Distributed Representations

For our application we want our DFN to evaluate similarities between mathematical expressions. For doing so, we want the model to approximate a function $e : \mathbb{M} \to \mathbb{R}^n$ with the following property:

$$\begin{gathered}\text{Let } a,b,c \in \mathbb{M} \text{ be some representations of mathematical expressions.}\\ \text{If and only if } a \text{ is semantically more similar to } b \text{ than to } c,\\ \text{the following equation holds: } \langle e(a), e(b)\rangle > \langle e(a), e(c)\rangle\end{gathered} \tag{2.6}$$

With $\mathbb{M}$ we denote a set of numerical representations of mathematical expressions. The results of function e are called *distributed representations* [6, pp. 544-550] or *embeddings* of the mathematical expressions. Each dimension of a resulting embedding $e(x) \in \mathbb{R}^n$ can be seen as a particular feature of the respective formula x. With a properly learned function, it is easy to evaluate the similarity between two images of mathematical expressions. We can just compute both distributed representations and calculate the similarity i.e. the dot product of the two resulting vectors. A suitable objective function for learning a function like e is presented in Section 3.1.

2.2.4 Backpropagation

As already stated the parameters θ are optimised with gradient descent. For gradient descent we need the partial derivative of each parameter and backpropagation [28] is an efficient algorithm for computing all of them. A DFN is a composition of the functions that are defined by its layers. To compute the derivatives of all these composed functions backpropagation makes use of the chain rule of calculus (see Definition 2.2.2).

Definition 2.2.2 Chain rule of calculus
The chain rule of calculus defines the derivative of some function composition. For $y = f(g(x))$ the derivative of y with respect to (w. r. t.) x is given by [6, p.203]:

$$\frac{\partial y}{\partial x} = \frac{\partial g(x)}{\partial x} \frac{\partial y}{\partial g(x)} \tag{2.7}$$

The properties of the chain rule of calculus can be exploited for the efficient computing of the gradient in the context of function compositions. Let $J(x) = f(g(h(x)))$ with $z_1 = h(x)$, $z_2 = g(z_1)$ and $z_3 = f(z_2)$, . We want to compute the partial derivatives of $J(x)$ w. r. t. z_2, z_1 and x. We could simply derive the respective formulas and compute all partial derivatives with these formulas:

$$\begin{aligned}\frac{\partial J(x)}{\partial x} &= \frac{\partial z_1}{\partial x}\frac{\partial z_2}{\partial z_1}\frac{\partial z_3}{\partial z_2}\\ \frac{\partial J(x)}{\partial z_1} &= \frac{\partial z_2}{\partial z_1}\frac{\partial z_3}{\partial z_2}\\ \frac{\partial J(x)}{\partial z_2} &= \frac{\partial z_3}{\partial z_2}\end{aligned} \tag{2.8}$$

As we go back from z_2 to x the formula for the derivative gets longer. If x was also a result of some composed functions and we wanted to get the derivative of $J(x)$ w. r. t. the inputs

of these functions, the formula would get even longer. And while the formulas would get longer the computational cost would rise. For this reason, this naïve approach does not seem to be very efficient. For a more efficient way, we can exploit the relations between the partial derivatives of z_2, z_1 and x. A careful reader might have noticed some repetitions in Equation (2.8). We could use the definition of $\frac{\partial J(x)}{\partial z_1}$ to write the derivative of $J(x)$ w. r. t. x like this:

$$\frac{\partial J(x)}{\partial x} = \frac{\partial z_1}{\partial x}\frac{\partial J(x)}{\partial z_1} \tag{2.9}$$

Actually, for one step more into the chain of functions, we only get one *new* partial derivative. The rest of the partial derivatives are computed already if we compute the derivatives in the order z_2, z_1, x. We can reuse $\frac{\partial J(x)}{\partial z_1}$ to compute $\frac{\partial J(x)}{\partial x}$ more efficient. This means that with this procedure we only need to compute one derivative per parameter, while the number of derivatives to be calculated is linearly growing for the naïve procedure.

Because of the layered structure of a DFN, the partial derivative of a single weight w in layer l_i depends on the partial derivative of certain other weights $\mathbb{W}$ in layer l_{i+1}. By computing the partial derivatives in the right order, we can reuse the partial derivatives of the parameters from l_{i+1} for our computations in layer l_i. This is why the algorithm is called backpropagation. The gradient, which is some kind of error signal, is propagated through the network in reverse order - from the output layer to the input layer.

2.2.5 Architectural Choices and Hyperparameters

The design space for deep models is huge. For some given machine learning task there is an infinite amount of DFNs that we could deploy for solving the task. We need to choose how deep we want our model to be, what nonlinearity we want to deploy and which types of layers we want to use. We also need to define the order of our layers, the kernel size and many things more (cf. [6, pp.193-199]). Additionally, we need to set hyperparameters (see Definition 2.2.3) for our learning process - e.g. the learning rate or the size of the mini-batches.

> **Definition 2.2.3** Hyperparameters
> Hyperparameters configurate a machine learning algorithm and determine its behaviour. However, they are not learned by the algorithm itself. They have to be chosen by the user (cf. [6, p. 118]).

An exhaustive search for all these opportunities is infeasible. There are guidelines for designing a model, but these guidelines are in most cases heuristic and may change over time while research on this field continues. Theoretical foundations with performance guarantees for such guidelines are rare. In the end, we simply have to try out a set of possible configurations for a given task in order to know what will work best. Configurations that lead to better performance may exist but may be not found [6, chapter 11].

The Role of Depth

In theory, any DFN with at least three layers is able to approximate any desired function that has the following property to an arbitrary degree. The function has to be either continuous on a closed and bounded subset of $\mathbb{R}^n$ or it has to map from and into a finite dimensional discrete space. This property of DFNs is described by the universal approximation theorem[4] [10]. However, this does not mean that a DFN is always able to learn such a desired function f^*. Or in other words: There is a configuration for the parameters θ with which the DFN would represent f^*, but it is uncertain whether this configuration will be found during the gradient-based learning process. Additionally, the universal approximation theorem gives no bound for the size of the hidden layer. Although there might be a DFN with three layers that is able to approximate some desired function, this DFN might require an infeasibly large hidden layer (cf. [6, p. 195]). For this second problem increasing the depth of a network might help. The work of Montufar et al. [24] suggests

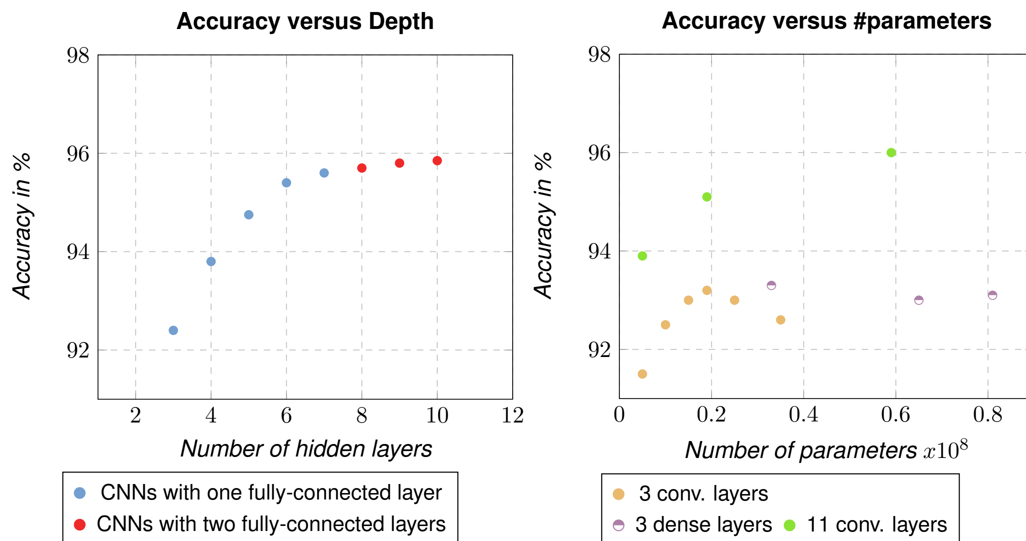


Figure 2.3: As can be seen from the left graph a sufficiently deep model showed to be an important factor for achieving good accuracy in the work of Goodfellow et al. [7]. The right graph shows that this effect is not reducible on the number of trainable parameters in deeper models. It showed that shallower models with about the same number of weights and biases performed inferior. Data from: [7].

that functions which would require a huge amount of trainable parameters with only three layers might be representable with a much lower number of parameters if the network is deeper. Experiments from Goodfellow et al. [7] on a dataset for house-number recognition

[4] In fact the universal approximation theorem applies to all borel measurable functions. But a complete definition of borel measurability would lead too far and at this point it should be enough to say that any function that is continuous and on a closed and bounded subset of $\mathbb{R}^n$ or any function that maps from and to a finite dimensional discrete space is borel measurable.

[25] showed that depth is an important factor for achieving a good accuracy whereas increasing the sheer number of trainable parameters does not necessarily help (see Figure 2.3).

Montufar et al. [24] give the following explanation for the effectiveness of depth: Each layer is capable of mapping different input regions to a common output region. This can be seen as some kind of abstraction. The abstraction done by one layer is not arbitrarily complex. But, because of the compositional structure of a DFN, higher layers can make use of abstraction that was done in lower layers. The different input regions that some layer l_k maps to the same output region are again the result of several region summarising mappings in the lower layers $\forall l_i,\ 0 < i < k$. This means that the mapping in layer l_k implicitly affects all input regions from the lower layers that have already been summarised to the respective input regions for layer l_k (see Figure 2.4). The number of input regions that can be summarised to common output regions grows exponentially in depth. For this reason, deep models are in general superior in terms of expressivity to shallow models. But they are also harder to train as we will see later in this section.

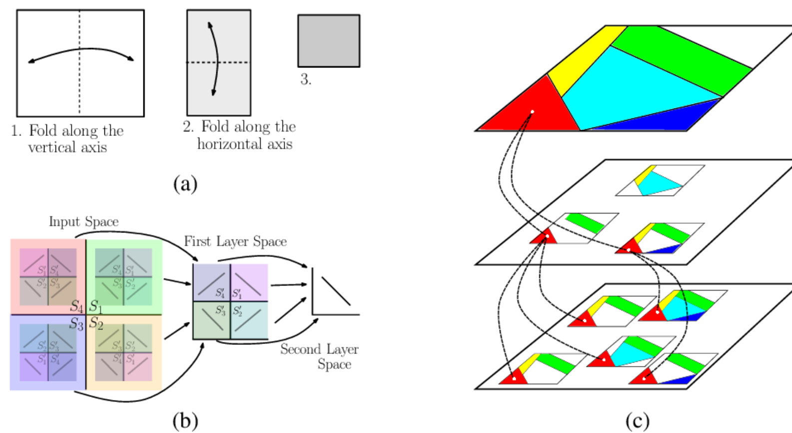


Figure 2.4: Subfigure (a) and (b) illustrate the abstraction performed by single layer as a folding of the input space and show how a successive folding can make regions linearly seperable. Subfigure (c) visualises how the abstraction from lower layers (bottom to middle) can be reused in the higher layers (middle to top) to summarise different input regions. In this case the first layer maps its five red input regions to two red output regions. Finally, this two red regions are then mapped to a common output region by the second layer. Source: [24].

Initialisation

Gradient descent is an iterative method that does some kind of local search for minimas on the given objective function. If the given objective function is sufficiently complex (which will be the case for interesting deep learning problems), the outcome of the optimisation process is highly dependant on the initial starting point. An unfortunate starting point might cause that our training process shows no convergence at all or that we get stuck in a

suboptimal local minimum. In contrast, a good starting point that is already near the desired minimum will lead to short training and good performance [6, p.297]. Unfortunately, there is no initialisation strategy with a guarantee for good performance. There are only heuristics for initialising trainable parameters which have proven themselves empirically. These strategies aim at ensuring certain properties at the beginning of the training procedure. Nevertheless, the optimisation of deep models is not yet sufficiently understood to explain whether these properties remain unchanged during the training process [6, p.297].

One example of such heuristics is the *kaiming initialisation* (see Definition 2.2.4) proposed by He et al. [8]. This approach aims at keeping the variance of the activations constant throughout the network. This means that the initial configuration of the weights should ensure that at initialisation time the variance of the input and the output is equal at each layer. He et al. [8] consider this property to be useful because it prevents the magnitude of the signal propagated through the network (i.e. the magnitude of the activations) to be scaled by a certain factor after each layer. A scaling by a constant factor β in each layer would lead to an exponential scaling by β^L over all L layers. The authors argue that this scaling of the signal's magnitude will bring a DFN to stall or to diverge dependant on whether β is smaller or greater than one. They derive the following initialisation scheme for networks with convolutional or fully-connected layers and the ReLU activation function that keeps this scaling factor close to one and thereby keeps the signal's magnitude constant throughout the network.

> **Definition 2.2.4** Kaiming Initialisation
> All weights are drawn from a normal distribution with zero mean and fixed variance: $w \sim \mathcal{N}(0, var)$ with $var = 2/fan_{in}$. Biases are initialised with zero.
> For some convolutional layer: $fan_{in} = ch_{in} * k_{width} * k_{height}$ and $fan_{in} = ch_{out} * k_{width} * k_{height}$

Gradient Propagation

Intuitively, the gradient can be seen as some error signal for all the trainable parameters. It controls how the weights and biases are altered. So, for a smooth training the gradient should be propagated properly throughout our network. But this is not always the case. Gradients in DFNs have a tendency to get too small or too big [9]. Both of these extremes are unwanted. As can be seen in Section 2.2.4 the gradient in some layer l_i depends on the gradient in the following layer l_{i+1}. This gradient is again dependant on the gradient in the next layer l_{i+2} and so on. All these values are multiplied (see Equation (2.9)). If the respective partial derivatives are under one but to close to zero, the gradient gets smaller as nearer you get to the input layer. This is called the *vanishing gradient* problem. If the respective partial derivatives are far bigger than one, the gradient gets bigger as nearer you get to the input. This is called the *exploding gradient* problem. The deeper a network gets,

the stronger this gradient effect is. For this reason, deeper models are in general harder to train.

Dynamics of the Forwarded Signals

Yang et al. [33] rank the dynamics within a DFN on a scale between *stable* and *chaotic*. The ranking on this scale is determined by the network's tendency to align activations for different inputs throughout the network or its tendency to produce very different activations for similar inputs. This can be measured by forwarding two different inputs through the network and computing the cosine similarity between the activations for the respective inputs layer-wise. For a cosine similarity near one, the activations are very similar. Thus, the network's behaviour is called rather stable. For a cosine similarity near zero, the activations are very different. Thus, the network's behaviour is called rather chaotic. Yang et al. [33] state that efficient learning is only possible on the edge of chaos.

If the network is too stable, the activations for very different inputs get more similar, the deeper they get forwarded through our network. In that case, the gradient will be close to zero - especially for the parameters in the early layers. If the activations in layer l_k are very similar regardless of whether the inputs are similar or not, the activations of the input layer do not have much effect on the activations of layer k either. Hence, the effect of the parameters in the input layer is marginal which causes the gradient to vanish.

A network that is too chaotic will produce very different activations even for similar inputs. If so, there will be a contrary effect. The parameters of the network - especially in the early layers - will have a huge impact on the evolution of activations throughout the network. The respective gradient tends to explode i.e. become huge.

Chapter 3

Related Work

Now that we are familiar with deep learning, we will focus on two deep learning models that are crucial for this work. First we will take a look at the Equation-Encoder [27], which is the method that we try to improve in terms of efficiency. We will then take a look at SqueezeNet [11]: A CNN architecture that is designed for achieving satisfactory performance with a small number of trainable parameters. The construction of SqueezeNet gives us some insights about the effects that certain architectural choices have on the size and the performance of a model. We will use these insights to construct a new, more efficient Equation-Encoder in the next chapter.

3.1 The Equation-Encoder

In contribution to the ultimate goal of creating a search engine for scientific publications that works with mathematical formulas as queries instead of natural language keywords, Pfahler et al. [27] proposed a method for evaluating similarity between bitmap representations of mathematical expressions: the Equation-Encoder. The Equation-Encoder is a CNN that embeds bitmap representations of mathematical expressions into a low dimensional vector-space in a similarity preserving fashion. The function e that the Equation-Encoder tries to approximate has the property presented in Section 2.2.3. For the Equation-Encoder, the numerical representations of the input are of the form $X \in [0,1]^{32x333}$.

Wang et al. [32] propose a loss function that makes it possible for a DFN to learn the desired similarity preserving function. The loss function (see Equation (3.1)) requires three samples (or formulas): the anchor sample x, the positive sample x_+ and the negative sample x_-. The semantic similarity between the anchor and the positive sample should be higher than the similarity between the anchor and the negative sample. These three samples form a *triple*. After the network computed the embeddings for all three samples, we can compute the loss with some kind of similarity measure between the vector representations

e.g. the dot product. The margin value Δ determines by how much x and x_+ should be more similar than x and x_-.

$$\ell_{tri}(x, x_+, x_-) = \max\{0, \Delta - \langle e(x), e(x_+)\rangle + \langle e(x), e(x_-)\rangle\} \tag{3.1}$$

For this loss function to work well the input embeddings need to be normalised to unit length ($|x|_2 = 1$). Otherwise the Equation-Encoder may satisfy an absolute margin just by returning embeddings that contain big values, which would result in high values for the dot product of two embeddings. Therefore, Pfahler et al. [27] normalise the embeddings computed by the Equation-Encoder with $e'(x) = \frac{e(x)}{||e(x)||_2 + \varepsilon}$ (ε is a small constant that ensures numerical stability). We will adapt this technique for all models used in this work.

Balntas et al. [2] suggest a technique that they call the *anchor swap* in order to make the triple loss more strict. This technique exlpoits that the relation between x and x_+ is symmetric. The symmetry allows us two swap the roles of the anchor sample and the positive sample if this yields a higher loss. With an anchor swap the triple loss function looks like this:

$$\ell_{tri}(x, x_+, x_-) = \max \begin{cases} 0 \\ \Delta - \langle e(x), e(x_+)\rangle + \langle e(x), & e(x_-)\rangle \\ \Delta - \langle e(x_+), e(x)\rangle + \langle e(x_+), & e(x_-)\rangle \end{cases} \tag{3.2}$$

The data used by Pfahler et al. [27] comes from two publicly available crawls of arXiv-ids. The formulas of the respective publications were extracted and compiled to bitmaps. In order to learn e by using this data, some kind of supervision is needed. Unfortunately, the dataset comes with no labels that define a clear relation of similarity between formulas. To address this problem, Pfahler et al. [27] defined some weak-labels based on the information available. This weak-label approach is based on a simple assumption: Two equations are likely to be similar if they are taken from the same document, and if two equations are taken from two different random documents, they are less likely to be similar. With these weak-labels, the Equation-Encoder can be trained with the loss function (3.1). Pfahler et al. [27] sample the anchor formula and the positive formula from the same paper and the negative formula from a different paper.

Once the Equation-Encoder has been trained, all formulas from all papers listed in the search engine's database are embedded. The obtained embeddings can then be inserted into an efficient index structure which allows us to perform a query concerning a mathematical expression by embedding the query formula and performing an approximate nearest neighbour search [12].

3.1.1 Equation-Encoder Network Architecture

Pfahler et al. [27] propose two Equation-Encoder architectures, a small one and a large one. In this work we will only deal with the large one. The Equation-Encoder consists of six convolutional layers and two fully-connected layers. There is a max pooling layer after every other convolutional layer and a batch-normalisation between the fully-connected layers. The resulting architecture can be seen in Table 3.1. Pfahler et al. [27] used ReLU as the nonlinearity for all layers and intialised the weights with a version of the kaiming initialisation (see Definition 2.2.4) that draws the weights from a uniform distribution instead of drawing them from a normal distribution[1].

layer type	output size	filter size / stride	#parameter
Convolutional	64x30x331	3x3 / 1	640
Convolutional	64x28x329	3x3 / 1	36,928
Max Pooling	64x14x82	2x4 / 2x4	0
Convolutional	64x12x78	3x5 / 1	61,504
Convolutional	64x10x74	3x5 / 1	61,504
Max Pooling	64x5x18	2x4 / 2x4	0
Convolutional	64x3x16	3x3 / 1	36,928
Convolutional	64x1x14	3x3 / 1	36,928
Max Pooling	64x1x4 → 256	1x3 / 1x3	0
Fully-connected	64	-	16,448
Batch-Normalisation	64	-	128
Fully-connected	64	-	4,160
Total			255,168

Figure 3.1: Architecture of the large Equation-Encoder used by Pfahler et al. [27]. Input size is 1x32x333.

3.2 SqueezeNet

SqueezeNet is a special CNN architecture that is designed for using as few trainable parameters as possible while maintaining the network's ability to approximate the desired function. Originally it was designed for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [29]. It performed as good as the winner from 2012 [18], but has 50 times less trainable parameters. There are three different strategies in the SqueezeNet architecture to save trainable parameters:

[1] This initialisation scheme is the default for the used deep learning framework [26]

1. **Use 1x1 convolutional filters** The height and width of convolutional filters are important factors when it comes to reducing the amount of trainable parameters. A 3x3 filter for example has 9 times as much parameters as a 1x1 filter.

2. **Reduce input channels** Despite their economic use of parameters, 1x1 filters are a bit constrained in their ability to detect certain features. We may also want to use filters with a higher dimensionality. To keep the amount of parameters consumed by this more powerful filters small, we can reduce the number of input channels for the respective layer. Lets say we have a layer with 64 3x3 filters. For each input channel we need $64 * 9 = 576$ parameters. So for example, if we halved the number of input channels from 64 to 32, we would save $32 * 576 = 18\,432$ parameters.

3. **Pool as late as possible** We discard some information when we are using pooling layers. Convolutional layers are more effective if they have access to more information. As a consequence we give them more information by not pooling too early.

3.2.1 The Fire Module

Strategies one and two result in a new module that is the main component of SqueezeNet. Iandola et al. [11] call this the *fire module*. A fire module consists of two consecutive convolutional layers (see Figure 3.2). We refer to the first one as the *squeeze layer*. The purpose of the squeeze layer is to reduce the number of channels that are forwarded to the next layer as a consequence of strategy two. To achieve this the layer has less kernels than it has input channels. This reduction layer would be useless if it consumed the amount of parameters that it saves for the following layer. Thus, a squeeze layer uses only 1x1 kernels. The second layer in a fire module is called the *expand layer*. As the name implies this layer annuls the reduction of the channels. Therefore, it has more kernels than the squeeze layer. The expand layer uses 1x1 as well as 3x3 kernels. With a standard convolutional operation this would result in feature maps with different dimensions. Iandola et al. [11] solve this mismatch by using padding for the convolutions with the 3x3 kernels.

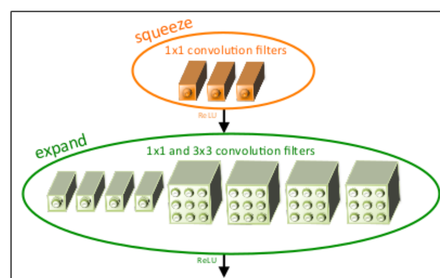


Figure 3.2: A visualisation of a fire module. Source: [11].

For the design of a fire module Iandola et al. [11] introduce two hyperparameters: The first is the percentage of 3x3 filters in an expand layer pct_{3x3}. The second is the squeeze

ratio, which is the proportion between the number of filters in the squeeze layer and the number of filters in the expand layer. Both hyperparameters control the model's size (number of trainable parameters). It is clear that a model with higher squeeze ratio or higher pct_{3x3} is at least as powerful as versions with lower values for these hyperparameters. There is a tradeoff between the model's size and performance. Iandola et al. [11] studied the relation between those properties with experiments on the ILSVRC dataset. Their models showed no improvements for pct_{3x3} greater than 0.5 or squeeze ratios greater than 0.75. For their final model (SqueezeNet) they used $pct_{3x3} = 0.5$ and a squeeze ratio of 0.125. Since it was not the goal to design a model as accurate as possible, but a model that is much smaller than AlexNet [18] while maintaining performance, they did not choose the squeeze ratio with the best performance (86.0% Top-5 accuracy). Instead they chose the squeeze ratio to be 0.125, which lead to a performance that is a bit lower (80.3% Top-5 accuracy).

There are three additional hyperparameters that control the number of channels throughout the network: With $freq$ and $incr_e$ Iandola et al. [11] control when and by how much the number of channels is increased i.e. after $freq$ fire modules they increase the number of output channels of the expand layer by $incr_e$. The value of $base_e$ determines the number of output channels from the expand layer of the first fire module.

SqueezeNet does not have any fully-connected layers at the end of the network because these layers need a rather large amount of trainable parameters and the work of Lin et al. [20] suggests that a global average pooling layer at the end of a CNN works at least as well as a fully-connected layer.

3.2.2 Overall Architecture

All nonlinearities in SqueezeNet are ReLU activations and the weights are initialised with a method proposed by Glorot et al. [5]. The overall architecture of SqueezeNet can be seen in Table 3.3. It might be irritating to see that the total amount of trainable parameters is much higher for SqueezeNet than for the Equation-Encoder (1,248,424 vs. 255,168), but we have to keep in mind that SqueezeNet has much more channels. This is necessary because SqueezeNet needs an output layer with a dimensionality of 1000 for the task that it should solve. Already the first layer in SqueezeNet has more channels than the Equation-Encoder and in contrast to the Equation-Encoder the number of channels is increased every other layer for SqueezeNet. With fewer channels the SqueezeNet architecture would also require a much lower amount of weights and biases. We will see this when we design a new Equation-Encoder by combining the architectures of SqueezNet and the original Equation-Encoder (see Section 4.1.4).

layer type	output size	filter size / stride	squeeze ratio	pct_{3x3}	#parameter
Convolutional	96x111x111	7x7 / 2x2	-	-	14,208
Max Pooling	96x55x55	3x3 / 2x2	-	-	0
Fire	128x55x55	-	0.125	0.5	11,920
Fire	128x55x55	-	0.125	0.5	12,432
Fire	256x55x55	-	0.125	0.5	45,344
Max Pooling	256x27x27	3x3 / 2x2	-	-	0
Fire	256x27x27	-	0.125	0.5	49,440
Fire	384x27x27	-	0.125	0.5	104,880
Fire	384x27x27	-	0.125	0.5	111,024
Fire	512x27x27	-	0.125	0.5	188,992
Max Pooling	512x13x13	3x3 / 2x2	-	-	0
Fire	512x13x13	-	0.125	0.5	197,184
Convolutional	1000x13x13	1x1 / 1x1	-	-	513,000
Average Pooling	64x1x1	13x13 / 13x13	-	-	0
Total					1,248,424

Figure 3.3: Architecture of SqueezeNet. Input size is 3x224x224.

Chapter 4

Experiments

This chapter describes the experiments that we conduct for deriving a new Equation-Encoder architecture which is suitable for high data volume. First, we will take a look at the experimental setup, then we will move on to the actual experiments and finally we will close with a short summary of the obtained results and the derived architecture/hyperparameters. A comprehensive evaluation of the results will be done in the next chapter.

4.1 Experimental Setup

In this section, we will introduce the data, the performance measures and the architecture types that will be used for the experiments.

4.1.1 Dataset

As already stated the available data was taken from arXiv.org. Preliminary to this work raw LaTeX sources of all papers that were published on arXiv.org before March 2019 were downloaded. The formulas in the respective papers were extracted by searching for regular expressions that represent different LaTeX math-environments. The resulting LaTeX-expressions were compiled to bitmaps that show the respective formulas. Not all of the extracted formulas compile properly to a png-file and some of the sources downloaded from arXiv.org are empty. Thus, the formulas in our dataset are still only a part of the set of all formulas that are possibly available on arXiv.org

Meta data about the formulas is available as well. For each formula we store the paper and the section in which the formula appears. For each paper we also store the author, the subject, the abstract, the year in which the paper was published and outgoing citations. All in all, this results in a dataset with 651,414 papers, 32,661,225 formulas and 748,719 citations between papers within the dataset. From this formula collection three datasets for different purposes were curated:

Tuning Dataset We will use this dataset for hyperparameter tuning. Hyperparameter tuning with all data would have been computationally expensive and would have consumed a lot of time. Therefore, a dataset with a tenth of the original size was curated. This dataset contains all available publications from the subjects *General Relativity and Quantum Cosmology (Physics)*, *Information Theory (Computer Science)* and *Probability (Mathmatics)*. All three subjects contribute roughly the same amount of formulas. The tuning dataset is already bigger and contains more different subjects than the dataset used by Pfahler et al. [27].

Qualitative Evaluation Dataset We will use this dataset for qualitative evaluation (see Section 5.1.3). In order to test the suitability as a real-world search engine, it would be insightful to perform queries on an index structure that contains embeddings from the trained model. However, to be able to evaluate the quality of the query result domain specific knowledge is needed. Due to a lack of experts for all subjects in our data, the dataset for qualitative evaluation dataset contains only publications from the field of *Machine Learning (Computer Science)*.

It would probably be hard for our model to evaluate similarities between machine learning specific formulas if it had never encountered formulas from this subject. Therefore, this dataset contains solely papers published after 2017. Machine learning papers/formulas published before 2017 can be used for training.

Full Dataset This is the dataset on which we will train our final model. It contains all available publications except for the papers contained in the qualitative evaluation dataset.

Dataset	#triples	#papers	#formulas	#citations
Tuning	1,000,000/250,000	36,750/9,189	2,985,193/740,100	37,993/2,207
Qual. Eval.	-	4,631	149,083	2,027
Full	10,000,000/2,500,000	517,442/129,341	26,029,067/6,483,075	702,056/44,636

Table 4.1: A Description of the Datasets in numbers. The value left from the slash refers to the train split of a dataset the other refers to the test split.

The tuning dataset and the full dataset are both divided into two parts: one that contains roughly 80% of the data and is used for training models and another one containing the rest of the data that is used for comparing the performance of different models. We will refer to these subdatasets as the train data and the evaluation/test data.

Sampling of Triples

For a triple, we need an anchor formula first. This anchor is sampled randomly from a paper which is again randomly chosen from all possible papers[1]. With a given anchor we can sample the negative and the positive formula. The negative sample is chosen in the same way as the anchor but with the constraint that the resulting equation cannot be from the same paper as the anchor. The positive sample may be chosen in three different ways:

1. *Same Section* The positive sample is uniformly drawn from all equations that appear in the same section as the anchor.

2. *Same Paper* The positive sample is uniformly drawn from all equations that appear in the same paper (not necessarily the same section) as the anchor.

3. *Along Citation (optional)*[2] This is only possible if the paper of the respective anchor has outgoing citations to other papers within the dataset. If this is the case, first a cited paper is uniformly drawn and then a random formula from this cited paper is chosen as the positive sample.

Quality of the Sampled Triples

To give a quick overview of the quality of the sampled triples, 50 triples were drawn randomly from the full dataset. From this 50 triples, I chose eight examples manually (see Section A.3) to discuss their quality[3].

The first observation is that the negative sampling works quite well. The semantic and syntactic similarity between anchor and negative sample seems to be very low. There are few or no shared symbols and there is arguably a great difference in the formula's structure.

The quality of the positive sample is much more diverse. Most of the positive pairs in A.3 share at least a few symbols, but there are also triples where the anchor and the positive formula do not have a single symbol in common (Triple 1,3 and 7). Triple 1 is a special case because for the positive formula it is not the sampling that went wrong but the extraction of the formula. This is clearly not a mathematical expression and should have not been extracted from the LaTeX-Source in the first place.

In Triple 6 the positive formula contains a sub-expression that appears in almost the same manner in the anchor formula ($|\Delta(A_i + A_j)|^2$) and for Triple 4 the number of shared symbols is very small, but therefore the overall structure is very similar ($\Delta\Box^{\gamma}_{s,t} = \Box^0 - \Box^{\gamma}_{s,t}$). There are positive pairs where the anchor and the positive formula seem to describe similar

[1] This ensures that all papers are equally represented and that papers with a huge amout of formulas are not overrepresented

[2] This sampling method is optional and its value shall be determined in an experiment (see Section 4.2.4).

[3] All 50 formulas can be found in the supplementary material and here.

properties. E.g. In Triple 2 both formulas describe the relation of some values O_i and B_i and in Triple 8 both formulas indicate that some λ_L should be maximised. An important lesson that we can learn from this small dataset investigation is that we are dealing with a very loose definition of semantic similarity. With our positive pairs we tell our model when to consider two formulas rather similar. We see that similarity between anchor and positive formula can look very different (shared symbols but different structure, same structure but few shared symbols, similar sub expressions etc.). For this reason, we can expect our model to learn a rather loose perception of similarity.

In conclusion, the negative and the positive sampling method seem to match quite well. In most cases, the semantic relation between the anchor and the positive formula is higher. Nevertheless, this quick view into the data also implicates that the sampling and the formula extraction is far from perfect and we have several triples that may be counter-productive for our learning process.

4.1.2 Performance Measures

To be able to find adequate hyperparameters and a good architecture we need some kind of quantitative performance measure. This allows us to compare different configurations. The two basic performance measures for the following experiments are computed on a set of triples and are defined as follows:

1. Ranking Loss: $P(x, x_+, x_-) = \begin{cases} 1, & \text{if } \langle e(x), e(x_+)\rangle \leq \langle e(x), e(x_-)\rangle \\ 0, & \text{if } \langle e(x), e(x_+)\rangle > \langle e(x), e(x_-)\rangle \end{cases}$
2. Constant Margin Loss: $P(x, x_+, x_-) = \max\{0, \Delta - \langle e(x), e(x_+)\rangle + \langle e(x), e(x_-)\rangle\}$

Of course both performance measures are computed on a test set, which has not been used for training. The constant margin loss is our objective function (see Eq. (3.1)) on that our model is trained and the ranking loss is some discrete version of this objective function. While the ranking loss simply counts how many of the triples are classified incorrect (i.e. the negative sample is more similar to the anchor as the positive sample: $\langle e(x), e(x_+)\rangle \leq \langle e(x), e(x_-)\rangle$), the constant margin loss also cares about how incorrect a false classification is (i.e. by how much the margin is violated). For both losses, we will swap the anchor and the positive sample if this yields a higher loss.

4.1.3 Baseline Equation-Encoder

The baseline model against which we will compare our new Equation-Encoder is the larger model from Pfahler et al. [27] with some small adjustments. We will refer to this model as the Baseline Equation-Encoder (BEE). The smaller model from Pfahler et al. [27] performed substantially worse than the BEE on the tuning dataset. Thus, it will be not

considered for the following experiments. Pfahler et al. [27] turned Batch-Normalisation off after the first training epoch. Since the Squeezed Equation-Encoder (SEE) (see Section 4.1.4) uses Batch-Normalisation throughout the whole training procedure, we will also do this for the BEE. This will give us a fair comparison between both models. For the same reason, the baseline model will use the nonlinearity, the initialisation scheme and the margin that worked best for the SEE (see Configuration 4.2.1). The overall structure of the BEE is depicted in Table 3.1.

4.1.4 Squeezed Equation-Encoder

Since our new architecture is heavily influenced by SqueezeNet [11], we will name it the Squeezed Equation-Encoder. The SEE is based on the torchvision[4] [26] implementation of SqueezeNet. This is not exactly the same architecture as the one presented in the original

layer type	output size	filter size / stride	squeeze ratio	pct_{3x3}	#parameter
Convolutional	64x15x166	3x3 / 2x2	-	-	640
Max Pooling	64x7x82	3x3 / 2x2	-	-	0
Batch-Normalisation	64x7x82	-	-	-	128
Fire	64x7x82	-	0.5	0.5	12,384
Fire	64x7x82	-	0.5	0.5	12,384
Max Pooling	64x3x40	3x3 / 2x2	-	-	0
Batch-Normalisation	64x3x40	-	-	-	128
Fire	64x3x40	-	0.5	0.5	12,384
Fire	64x3x40	-	0.5	0.5	12,384
Max Pooling	64x1x19	3x3 / 2x2	-	-	0
Batch-Normalisation	64x1x19	-	-	-	128
Fire	64x1x19	-	0.5	0.5	12,384
Fire	64x1x19	-	0.5	0.5	12,384
Fire	64x1x19	-	0.5	0.5	12,384
Fire	64x1x19	-	0.5	0.5	12,384
Convolutional	64x1x8	1x5 / 2x2	-	-	20,544
Batch-Normalisation	64x1x8	-	-	-	128
Convolutional	64x1x3	1x3 / 2x2	-	-	12,352
Average Pooling	64x1x1	1x3 / 1x3	-	-	0
Total					133,120

Figure 4.1: Architecture of the SEE. Input size is 1x32x333.

paper [11], because the authors continued to develop their model and ended up with a slightly different architecture that achieves the same accuracy with a smaller amount of necessary computations[5]. This architecture will be the starting point for the SEE. From

[4] https://github.com/pytorch/vision

[5] The repository can be found here.

thereon we will make some adjustments that are either necessary for our specific task or improve the performance. SqueezeNet is designed for quadratic inputs and output vectors with a dimensionality of 1000, but we have rectangular inputs and output vectors with a dimensionality of 64. Therefore, the last two convolutional layers have rectangular kernels instead of quadratic ones and one more convolutional layer was added. Because of the lower output dimensionality, it was also possible to reduce the number of channels. For all plain convolutional layers and for all fire modules, we use 64 in- and output channels[6]. This means that in our case $base_e$ is 64. The values of $freq$ and $incr_e$ do not play any role because we do not increase the number of channels at all. For all fire modules we use a fixed value for $pct_{3x3} = 0.5$ and the squeeze ratio (also 0.5). The SEE also has some Batch-Normalisation layers for better convergence. The overall structure of the SEE is depicted in Table 4.1.

4.1.5 Miscellaneous

Objective function The objective function for the training process will be the constant margin loss from Section 4.1.2. As suggested by Janocha et al. [14] we will square the training loss for faster convergence.

Hardware The experiments for this entire work were done on a machine with the following technical specifications:

CPUs: 56x Intel(R) Xeon(R) CPU E5-2690 v4 2.60GHz
GPU: Nvidia GeForce GTX 1080
RAM: 503 GB

Optimiser For all experiments we will use the Adam-optimiser [16].

[6]The first convolutional layer is an exception it has only one input channel.

4.2 Experiments

As already mentioned in Section 2.2.5 the design space for our model and our algorithm is huge. Even with the type and order of our layers fixed, there are a lot of choices to be made. For this thesis, we conduct experiments regarding the activation function, the initialisation scheme, the learning rate and the margin of the loss function. All experiments except for the last one (see Section 4.2.7) use the Tuning Dataset. The starting point are the choices from Pfahler et al. [27] and we will try to improve performance from thereon. The architecture/hyperparameter choices we begin with are:

Configuration 4.2.1 Initial Configuration for the SEE

Activation:	ReLU
Initialisation:	Kaiming initialisation (with uniform distribution)
Learning Rate:	0.0025
Margin:	1.0

4.2.1 Experiment I - Vanishing Gradient

With this initial setup the SEE performs poorly. For the first five epochs (five million triples forwarded), the training loss hardly changes. The optimisation seems to be stuck. After five million triples that have been forwarded there is some minimal improvement in the loss and the gradient's norm starts to have non-zero values, but the result is still very unsatisfying (see Figure 4.2). As we can see in the right plot of Figure 4.2, the reason for this is the vanishing gradient problem. The y-axis shows the norm of the weight's gradient in the first layer to give an overview of how big the partial derivatives are in general. For the first five epochs, the values are very close to zero. This means that the gradient i.e. the error signal is not properly propagated through the network, which causes the optimisation process to stall.

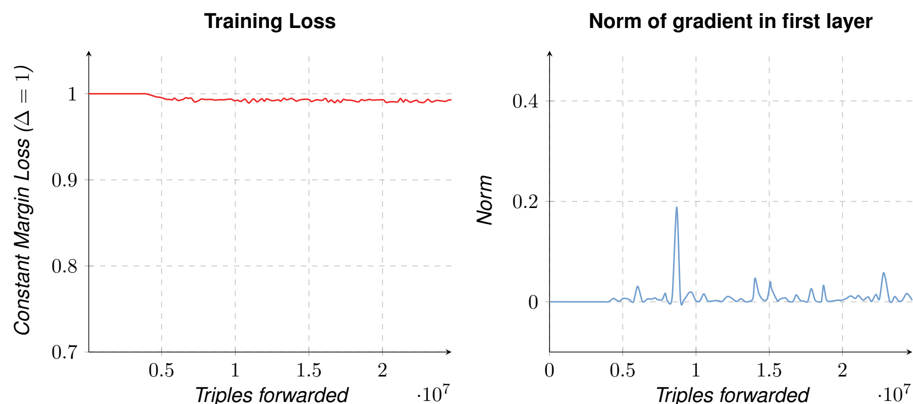


Figure 4.2: A training run of the SEE with the initial setup. The optimisation process seems to suffer from the vanishing gradient problem.

For a better understanding of the dynamics within the SEE with standard setup, we will take a look at the variance[7] and the cosine similarity[8] of the activations at initialisation time (see Figure 4.3, top left plot). The curve of the cosine similarity of the activations

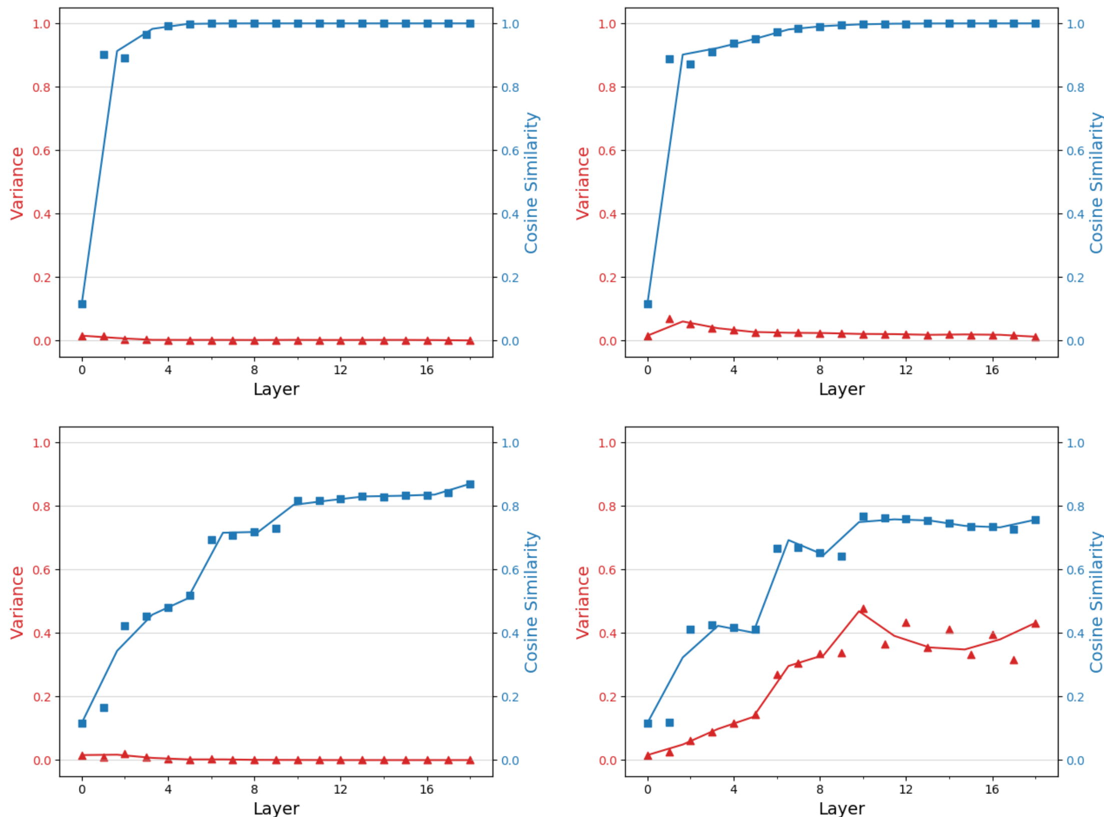


Figure 4.3: The cosine similarity (blue squares) and the variance (red triangles) of the feature maps at a given layer within the SEE. The data points for layer zero show the variance and the cosine similarity for the input. Top Left: ReLU/Kaiming, Top Right: SELU/Kaiming, Bottom Left: ReLU/LeCun, Bottom Right: SELU/LeCun.

within the network converges very quickly to one, meaning that the activations get really similar really fast. In the terminology of Yang et al. [33] the dynamic of this network is too stable. The low variance emphasizes this dynamic. But where does this low variance come from? A quick look into the input data shows that the input itself has only a variance of approximately 0.015. Actually, this is not surprising having in mind that our input images mainly consist of white pixels (numerically represented as zero). But, together with the variance preserving property of our initialisation scheme it inhibits training. If the variance is already low for the input layer and we initialise our weights in a way that aims at preserving that variance, we will have low variance throughout all the network. My

[7]Following the strategy of [8] (see Section 2.2.5 - Initialisation)

[8]Following the strategy of [33] (see Section 2.2.5 - Dynamics of the Forwarded Signals)

hypothesis is that this causes the stable behaviour of the SEE and thereby the vanishing gradient.

4.2.2 Experiment II - Activation and Initialisation

This first experiment aims at determining an activation function and an initialisation scheme with which we can evade the vanishing gradient problem. In order to achieve stable gradient propagation, we will try an alternative activation function. The activation function of our choice is SELU [17] (see Appendix A.2). SELU dampens the variance of the output if the variance of the input is high and increases the variance of the output for input with low variance. This property seems to be quite useful for our network since we suffer from too low variance.

In fact SELU manages to increase the variance of the activations by a small margin (see Figure 4.3, top right plot). Nevertheless, the variance is still very low and the cosine similarity still converges very fast to one. SELU alone will probably not solve our vanishing gradient problem. Thus, we will expand our approach: Klambauer et al. [17] suggest that

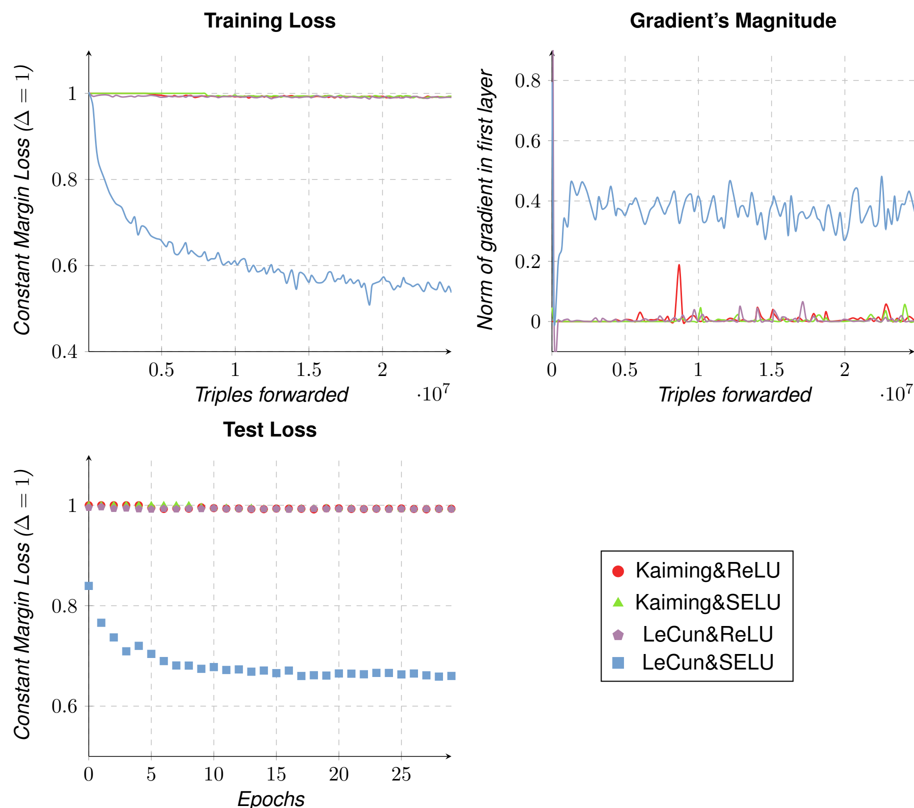


Figure 4.4: Experiments for determining the activation function and the initialisation scheme.

a certain initialisation scheme (see Definition 4.2.1)[9] should be used in combination with SELU.

Definition 4.2.1 LeCun Initialisation
All weights are drawn from a normal distribution with zero mean and fixed variance: $w \sim \mathcal{N}(0, var)$ with $var = 1/fan_{in}$, biases are initialised with zero.

Together with the LeCun initialisation the variance and the cosine similarity look a lot better throughout the SEE (see Figure 4.3, bottom right plot). Variance increases while the input gets forwarded, but it does not ecxeed a value of 0.5. The cosine similarity increases too, but it starts a lot lower and it does not converge towards one. Note that only the LeCun initialisation with ReLU and without SELU manages to obtain a good curve for the cosine similarity (see Figure 4.3, bottom left plot). However, this configuration has lower variance than the configurations with SELU.

This small preliminary experiments might give an intuition of what works better for training our SEE and what is worth trying out, but they cannot replace an experimental evaluation of different setups. Thus, we still need to run four experiments for all possible combinations of ReLU&SELU and Kaiming&LeCun initialisation.

For the combination of ReLU and Kaiming, we already know the outcome. The optimisation process is stuck. The setups with Kaiming&SELU and LeCun&ReLU perform equally bad. They are stuck for all 30 epochs and hardly show any improvements. Fortunately, the fourth combination (LeCun&SELU) works pretty well. For this combination the loss starts converging at the very beginning and the gradient in the first layer seems to be sufficiently large most of the time (see Figure 4.4).

An evaluation with the constant margin loss of all four models emphasizes this result. LeCun initialisation together with the SELU nonlinearity outperforms all other setups (see Figure 4.4). Hence, for the rest of our experiments we will work with SELU nonlinearities and LeCun initialisation.

4.2.3 Experiment III - Learning Rate

The learning rate has a big impact on the behaviour of our optimisation algorithm and thereby on the model's performance. Hence, it is important to figure out an adequate learning rate for out experimental setup. In this experiment, we test six different learning rates in the range from 0.1 to 0.000001 for the configuration derived from Experiment II. The training and the test loss can be seen in Figure 4.5.

The clear winner of this experiment is the model trained with a learning rate of 0.001, which is not that far from our initial learning rate 0.0025 that was used by Pfahler et al. [27].

[9]The initialisation scheme is very similar to the one proposed by LeCun et al. [19]. Therefore we will refer to it as LeCun initialisation

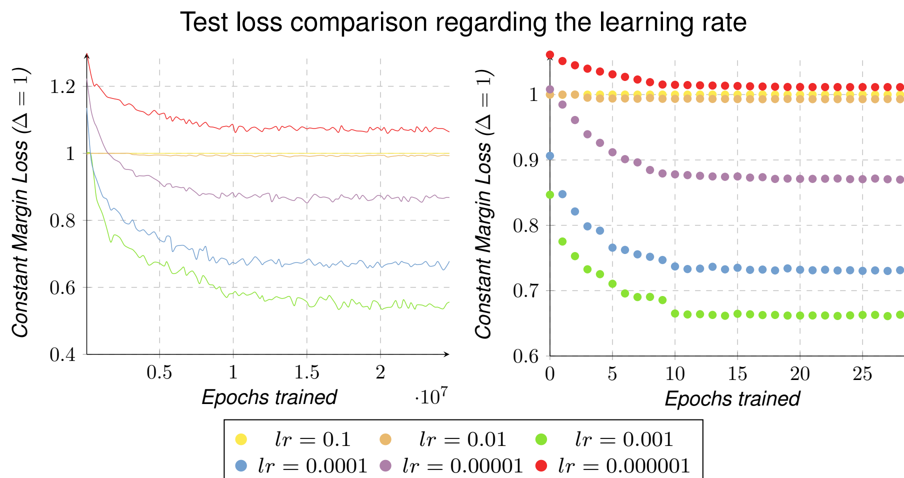


Figure 4.5: Experiments for determining an adequate learning rate.

4.2.4 Experiment IV - Sampling via Citation Graph

With this experiment, we want to find out whether it is profitable (in terms of performance) to sample the positive formula for a triple also via the citation graph. To do this, we compare the model which performed best in the learning rate experiment[10] against a model that was trained on the same data but with triples that are partially sampled with the help of the citation graph. For both models we will compute our performance measures on two test datasets: One that makes use of citations and one that does not.

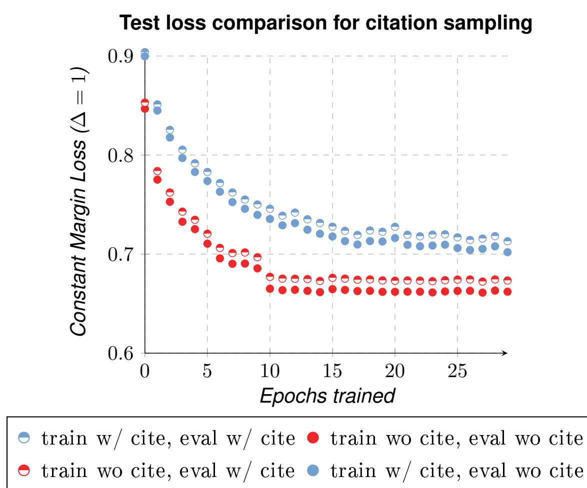


Figure 4.6: Results of the experiments regarding citation sampling.

Both datasets come from the test split of the tuning dataset. This means that the second one is the same that was used for the previous Experiments I, II & III. On the dataset that contains triples with citation sampling the optimiser with the learning rate from

[10] performed on triples sampled without citation graph

the previous experiment performed poorly. Thus, a new learning rate was determined in an experiment similar to Experiment III. The learning rate that performed best and will therefore be used for the second model in this experiment is 0.0001. The experiment showed that using the citation graph for positive sampling has at best no positive effect. It rather has a small negative effect on the performance. The model trained on triples with no citation sampling outperforms the other model for both evaluation datasets (see Figure 4.6). As a consequence of this result, sampling via citation will not be used for the following experiments and for the rest of this work.

4.2.5 Experiment V - Margin

Another hyperparameter that might influence the performance of our model is the margin Δ that is used in our objective function. Pfahler et al. [27] chose a value of one for the margin, but they did not experiment with other margins in order to justify this choice. Since the dot product between two normalised vectors is bound to the interval $[-1, 1]$, the difference between two such dot products can be two at most. Hence, for a margin of one

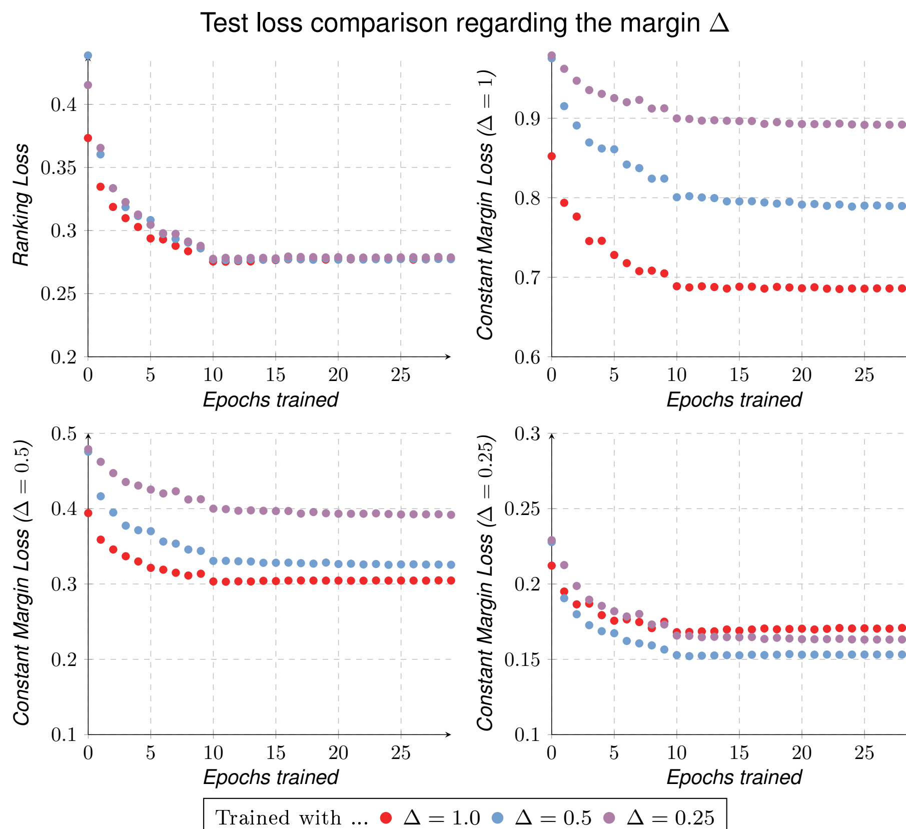


Figure 4.7: The four performance measures for the margin experiment. Only for the constant margin loss with a margin of 0.25 the version of the model trained with a margin of one does not perfom best.

most triples will violate that margin and the weights will be updated for most of the triples. This can be an advantage because our model learns something from nearly every triple, but it might be also a drawback because the weights are updated although the difference between $\langle e(x), e(x_+)\rangle$ and $\langle e(x), e(x_-)\rangle$ is already sufficiently large. A high margin might force the model to push two examples further apart although their dot product is already a good representation of their semantic relation. With this experiment we want to find out whether the advantages or the drawbacks of a high margin prevail. Three different margin are considered: 0.25, 0.5 and 1.0. We will determine the best working margin on the basis of four performance measures:

- 1. The ranking loss
- 2. - 4. The constant margin loss with different margins (0.25, 0.5, 1.0)

The experiment shows that a margin of one is the best choice. It outperforms the other margins in three out of four performance measures, although the difference between results achieved with different margins are not that big. For example with the ranking loss measure all three margins achieve approximately the same performance. The advantage of the high margin ($\Delta = 1$) is that it a) converges faster and b) that it pushes the samples further apart i.e. it computes embeddings that are more different or more diverse. It performs much better for the higher constant margin losses (1.0, 0.5). This property can be considered useful for our application. Nearest neighbor search will probably work better if the vectors are more distinguishable.

4.2.6 Final Configuration

These experiments lead to the following configuration for our networks and our algorithm:

Configuration 4.2.2 Final Configuration

Activation:	SELU
Initialisation:	LeCun initialisation
Learning Rate:	0.001
Margin:	1.0

4.2.7 Experiment VI - Full Dataset

With this last experiment, we aim at determining whether it is worthwile to train the SEE on the full dataset. Training on a larger dataset would be unnecessary if we could achieve the same performance with training on the tuning dataset. To justify the choice of the training dataset, we train the SEE with the final configuration 4.2.2 on the full dataset and compute our performance metrics on the test split of this dataset. For a comparison, we

also compute the performance metric on the full dataset's test split with a network that was trained on the tuning dataset[11]. Furthermore, we measure the performance of both models on the test split of the tuning dataset. Note that the formulas that are in the train split of the tuning dataset may be used in the test split of the full dataset and the other way around. Hence, the model trained on the full datset might have an advantage for the loss on the tuning dataset's test split and the model trained on the tuning dataset might have an advantage for the loss on the full dataset's test split. As can be seen from the plots in Figure 4.8 the model that was trained on the full dataset performs substantially better for all losses except for the constant margin loss that was computed on the test split of the tuning dataset. Therefore, we will also train the BEE on the full dataset with the final configuration 4.2.2 and evaluate the performance of both models in the following chapter.

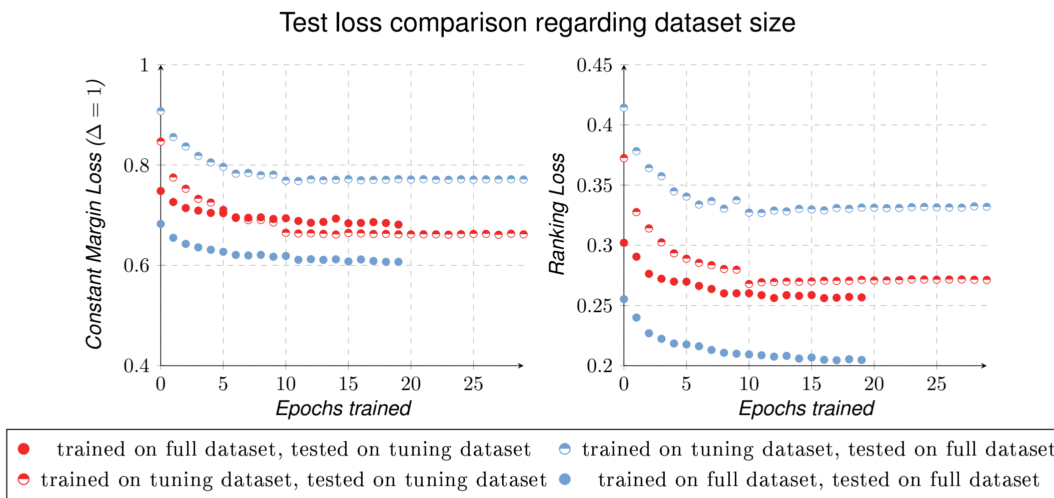


Figure 4.8: Results of the experiments regarding the dataset that is used for training. A red mark means that the loss was computed on the test split of the full dataset, whereas a blue mark means that the loss was computed on the test split of the tuning dataset. A full circle means that the model was trained on the full dataset, wheras a halfcircle means that it was trained on the tuning dataset. The training run with the full dataset had only 20 epochs because training on a bigger dataset is more time consuming. Nevertheless, it leads to better performance for three out of four performance measures.

[11] We use the weights from the network trained in Experiment 4.2.5 with a margin of one.

Chapter 5

Evaluation

In this chapter, we will look at the performance of our models. The focus lies on comparing the SEE and the BEE. First, we will compare the baseline and the proposed model via the ranking loss and the constant margin computed on the full test dataset. Second, there will be a comparison regarding the size and speed of both models. We will close the comparison with a small user study, for which we will use the qualitative evaluation dataset from Section 4.1.1 as the database of a fictive search engine and perform math-queries on with this search engine. The search results will be evaluated by a group of machine learning experts. This allows us again to compare the squeezed model against the original one. At the end of this chapter, we will use the results obtained from the user study to evaluate the SEE in terms of applicability as a backend of a real-word search engine.

5.1 Comparison

For the final comparison of the SEE and the baseline, both models were trained with the final configuration (see 4.2.2) derived from the preleminary experiments.

5.1.1 Loss on Test Data

Figure 5.1 shows the ranking loss and the constant margin loss that the BEE and the SEE obtain on the test split of the full dataset. In general, the plots show only a little difference between the performance of the two Equation-Encoders. But, we can still see that the baseline does perform a little bit better for both losses. In the end, the BEE achieves a ranking loss of 0.192 and a constant margin loss of 0.575 whereas the SEE achieves a loss of 0.205 and 0.607. The BEE manages to classify a few more triples correct i.e. it considers the positive fomula more similar to the anchor than the negative formula. Besides these differences, it is interesting to see that both models already achieve a pretty good loss after the first epoch and do not show big improvement after that. It seems that the dataset is so big that the networks have already learned the greater part of the desired function after

only one iteration over the dataset. For the tuning dataset, there is substantially more change in the test loss (see Figure 4.7). However, it is notable that - in case of the tuning dataset - the test loss stabilises after ten epochs. The full dataset is ten times bigger than the tuning dataset. This means that for both datasets the same amount of triples has been forwarded when the test loss starts to become more stable.

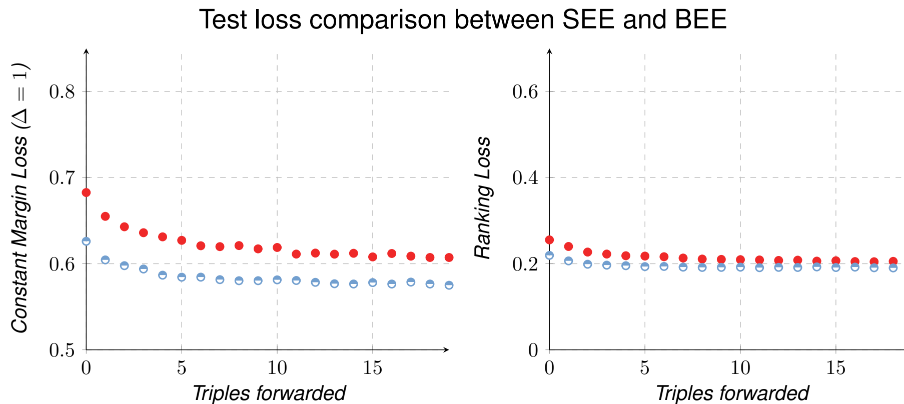


Figure 5.1: Performance of the BEE and SEE with their final configuration (see 4.2.2) and trained on the full dataset.

5.1.2 Size and Speed

For a comparison of the memory consumption of both models, we need to take a look at two quantities:

1. The pure number of trainable parameters that a network has.
2. The number of values that are passed through the network (i.e. the feature maps and the gradient).

These values and the bits needed to store them determine the memory consumption[1]. The values from Table 5.1 show that the SEE is more memory saving than the BEE. The

Model	Parameters Size	Forward/Backward Pass Size	Total Size
SEE	0.51 MB	6.00 MB	6.51 MB
BEE	0.97 MB	21.00 MB	21.97 MB

Table 5.1: Memory consumption (in MegaBytes) of the squeezed model and the baseline.

new architecture needs half of the space that the old one needed for storing the trainable parameters and uses nearly four times less memory to store the values during the forward and the backward pass.

[1]Technically, we also need to consider the number of values in the input for memory consumption, but since both models have the same input this quantity is negligible for a comparsion.

For a comparison of speed, we measure the time that the respective model needs for processing 150,000 formulas. Since processing a formula requires more operations if one wants to train the network instead of simply compute embeddings with a pretrained model, we measure time at training time and at inference time. The results are presented in Table 5.2. The measurements show that the SEE is clearly faster. For training the throughput is nearly four times higher and for inference it is even five times higher.

Model	Inference Total Time	Inference Throughput	Training Total Time	Training Throughput
SEE	8.3 sec	~18,000 $\frac{\text{formulas}}{\text{sec}}$	35.6 sec	~1,400 $\frac{\text{triples}}{\text{sec}}$
BEE	40.5 sec	~3,600 $\frac{\text{formulas}}{\text{sec}}$	137.7 sec	~360 $\frac{\text{triples}}{\text{sec}}$

Table 5.2: Speed and throughput of the SEE ande the BEE. For inference 149,084 singular formulas were forwarded and the training was done on a dataset with 50,000 triples. Thus, for both speed tests roughly the same number of formulas was processed.

In summary: Regarding the throughput and the memory consumption the Squeezed Equation-Encoder performs clearly better.

5.1.3 User Study

For this evaluation, we will set up a small scale search engine which will then be evaluated by members of the scientific staff of the Artificial Intelligence Group at the TU Dortmund University. The search engine allows a user to search for papers or formulas from the qualitative evaluation dataset (see Section 4.1.1) by providing a mathematical expression as a query. In principle, this search engine works as already described in Section 3.1. First, we use the model that we want to evaluate to embed all formulas from the dataset. The resulting vector representations are then inserted in an index structure [1] that allows us to perform an efficient approximate nearest neighbor search[2]. After the index structure has been built up, we are already prepared for performing queries. If we want to query some mathematical expressions, we can do that by providing this expression in LaTeX. The LaTeX-Code will be compiled to a png-file and the resulting png-file can be forwarded through our model. We end up with a distributed representation $\vec{q}$ (see Section 2.2.3) of our query. The final step is to search the index structure for a number of $\vec{q}$'s nearest neighbors $\mathbb{A} = \{\vec{a}_1, ..., \vec{a}_k\}$. The formulas that belong to the vectors in $\mathbb{A}$ are the results of our query. This study will only consider the ten nearest neighbors. This is the number of results that is usually shown to a user on the first page of a state-of-the-art search engine like Google Scholar.

[2]The implementation used in this work can be found here.

In order to test our search engine, we have seven formulas that are related to popular machine learning methods (see Table 5.3). For all of these methods, there are papers in our evaluation dataset that deal with the respective topics and thereby contain semantically similar formulas. Thus, it is ensured that finding meaningful results is possible.

Generative Adversarial Network (GAN):	$min\, max\, V(D, G) = \mathbb{E}[log(1 - D(x)]+$ $\mathbb{E}[log(1 - D(G(z)))]$
Gradient Descent (GD):	$\theta_{i+1} = \theta_i - \eta \nabla L(\theta_i)$
k-means:	$argmin\, \sum_k \sum_S \|\|x - \mu_i\|\|^2$
Long Short-Term Memory (LSTM):	$f_t = \sigma(W_f * x_t + U_f * h_{t-1} + b_f)$
Policy Gradient (PG):	$\nabla \mathbb{J}(\theta) = \mathbb{E}_\pi[q_\pi(s, a) \nabla_\theta \pi(a\|s, \theta)]$
Q-Learning (QL):	$Q(s, a) = R(s, a) + \sum Q(s', \pi(s'))$
Variational Autoencoder (VAE):	$l_i(\theta, \phi) = -\mathbb{E}_{z \sim q_\theta}(z\|x_i)[log\, p_\phi(x_i\|z)]+$ $\mathbb{KL}(q_\theta(z\|x_i)\|p(z))$

Table 5.3: The queries that are used for our user study.

The results of the queries will be evaluated in a small study with 14 participants. All participants have either a master's or a bachelor's degree in computer science and have experience in the field of machine learning. They were asked to classify each search result (consisting of a paper and a formula from that paper) for a given query as either relevant or not relevant. The participants could choose which of the queries they wanted to evaluate based on their expertise regarding the respective topic and how much queries they wanted to review. If a participant decided to review a query, he reviewed the search results of both models. This gives us a direct comparison of the real-world performance of the BEE and the SEE.

To compare the SEE against the baseline, we calculate a score that results from the user study: For this score, we look at every single review of a query and count the number of search results that the reviewer classified as relevant. From this, we get a score for each reviewer for each query for both models. For each query, we summarise the resulting scores by computing the median. The median was chosen to make the scores more robust to outliers. This score will be referred to as the *per-reviewer-score*.

As can be seen in Table 5.4 there is only little difference in the overall performance of the BEE and the SEE. There are topics for that the baseline works better (LSTM, QL and VAE) and there are some topics for which the squeezed version achieves higher scores (GAN, GD and PG). The BEE achieves an overall score that is a little bit higher than the score of the SEE. Most notably the scores differ much for the variational autoencoder query. For this particular topic, the SEE performs poorly while the BEE performs quite well. However, it is difficult to declare a clear winner because the overall difference is small and for both models there are exactly three queries for that the respective model achieves

a higher per-reviewer-score. Additionaly, the study is small and thereby prone to outliers. There are for example two search results that the SEE found for the VAE query that were classified as not relevant by both reviewers, but are arguably relevant since they come from a paper that deploys VAEs [21]. To obtain more reliable results, one would have to conduct a study with more queries and far more participants. However, this is out of the scope of this work.

Query	BEE	SEE
Generative Adversial Networks	6.0	8.0
Gradient-descent	5.0	8.0
k-means	0.0	0.0
Long Short-Term Memory	9.0	8.0
Policy Gradient	5.0	6.0
Q-Learning	7.0	5.0
Variational Autoencoder	6.0	1.0
Total	38.0	36.0

Table 5.4: Results of the user study: The table shows the per-review-score for all seven queries.

5.1.4 Conclusion

The BEE has a small advantage regarding the loss. It seems to be a bit better at fitting the function that is given by the data and objective function. However, it is much slower and bigger. This makes hyperparameter tuning very time consuming and might be unsatsifactory in a real-world application because embedding a set of formulas has a higher computational cost. Also, the small advantage from the loss comparison does not result in a clearly better performance in the user study. There are queries for which the SEE performs better. Given all this, it is difficult to give clear advice on which model should be used. It depends on the circumstances. If for the given application time and resource awareness are not important, the BEE might be a reasonable choice. Otherwise, I would advise to use the squeezed version. It has comparable performance to the BEE, but is easier to train because experiments do not take that long and it is more ressource-saving in general.

5.2 Quality of the Search Engine

The user study does not only enable us to compare both models. It also gives us the opportunity to take a first look at the quality of a search engine that runs with an Equation-Encoder model as its backend. This should help us to identify strengths of our approach,

but more importantly it also allows us to identify the weaknesses of the Equation-Encoder architecture. This second part is important for future work in which one might overcome some of these weaknesses and thereby improve the quality of math-based search engines.

For this evaluation, we take the results of the search results of the SEE and calculate yet another score out of this. This score looks at every single search result and accumulates the number of reviewers that had the opinion that this particular search result is relevant. This means when two out of four reviewers found the top-1-ranked search result for a given query to be relevant, the score for this search result is two. This score will be referred to as the *per-result-score*. The coloring of the search results in the appendix (see A.4) matches the per-result-score. A green background means that the score is as high as possible (i.e. all reviewers voted "relevant"), a red background means the score is zero and for all other scores the background is yellow.

Rank	GAN	GD	k-means	LSTM	PG	QL	VAE
(1)	3/3	6/6	0/3	2/3	3/3	1/3	2/2
(2)	3/3	6/6	0/3	3/3	3/3	3/3	0/2
(3)	1/3	5/6	0/3	3/3	2/3	3/3	0/2
(4)	3/3	1/6	0/3	3/3	0/3	1/3	0/2
(5)	0/3	6/6	0/3	3/3	0/3	3/3	0/2
(6)	3/3	0/6	0/3	1/3	2/3	3/3	0/2
(7)	2/3	6/6	0/3	3/3	3/3	2/3	0/2
(8)	3/3	6/6	0/3	0/3	2/3	0/3	0/2
(9)	2/3	6/6	0/3	3/3	3/3	2/3	0/2
(10)	3/3	5/6	0/3	1/3	0/3	1/3	0/2

Table 5.5: Results of the user study. The table shows the per-result-score for all seven queries. The left value is the achieved score and the right value is the highest possible score (#reviewers).

For the queries GAN, GD and LSTM the SEE performs pretty well. The majority of the search results are colored green and there is only one result with a red background. For both queries that are related to reinforcement learning (QL and PG) our model does not perform as good as for the queries mentioned earlier, but still most of the results are meaningful. The remaining two queries (k-means and VAE) yield very unsatisfying results. For the VAE query, there is only one relevant result and for the k-means query there are no relevant results at all. Although the results for these two queries are unsatisfying, they are not completely meaningless. Regarding the VAE query, the Results (2) and (3) come from papers that deal with generative models, the results (5) and (6) come from a paper that actually deals with VAEs and some other results do at least share some symbols (Results (4), (8) and (9)). Especially Result (4) shows us again the difficulty of math-based literature search. The model probably assumes this result to be equal because it contains

the subexpression $q(...)$. But this is a misleading commonality. The q from the query refers to some distribution whereas the q from Result (4) stands for the expected reward of a particular action in a particular state. This emphasizes how important context is. For example, a human expert may be able to distinguish both q subexpression because some other symbols from Result (4) (e.g. π, s_t, a_t and r_t) indicate that this formula comes from the field of reinforcement learning and not from a generative model. This problem is even more remarkable for the k-means query. The given results do often have shared symbols (most notably $\sum$) or a similar overall structure, but the context is always different and thereby the results were not relevant for the participants in the user study. Note that the bad quality of the results for the k-means query is reflected by the similarity predicted by the SEE. Out of all results, the ones from the k-means query have the lowest similarity.

Chapter 6

Conclusion and Outlook

The objective of this thesis was to train a model similar to the one from Pfahler et al. [27] on a larger dataset and to derive a model that has higher throughput in order to be able to handle the increased data volume. With the full dataset (see Section 4.1.1) we had access to a dataset that is not only larger but also more diverse (i.e. it includes formulas from different subjects). For training on that dataset we derived a new model by combining the original Equation-Encoder [27] and the SqueezeNet architecture [11]. In order to justify some of the architectural choices and hyperparamter settings, we also conducted experiments with this new model. The most important findings from the experiments are:

1. The SEE is applicable to our math-similarity learning task and is thereby an adequate strategy to increase the throughput while roughly maintaining performance (see Section 5.1).

2. A bigger and more diverse dataset does indeed improve performance (see Section 4.2.7).

3. Deploying the SELU nonlinearity together with LeCun initialisation can be a good strategy to deal with input that has a low variance (see Section 4.2.2).

4. The citation sampling approach from this work does not result in triples that are more profitable for learning (see Section 4.2.4).

5. A loss with a bigger margin that allows our model to learn from almost every triple leads to faster convergence and pushes embeddings further apart (see Section 4.2.5).

All this are empirical findings for the task of this thesis and the applicability to different tasks is uncertain.

The user study from Chapter 5 showed that the Equation-Encoder approach has the potential for being used as the backend of a math-based literature seach engine, but it the study also showed that there is a lot of work do be done. There are still some unsatisfactory search results for certain queries. An obvious point of attack for future work is the

labeling/sampling of the triples. Because of our loose definition of similarity between the anchor and the positive formula, the trained models seem to have a very loose notion of what they consider similar or dissimilar. This property can be useful in some cases since not only the exact same formula is found but also formulas that appear in the same context. However, this loose notion of similarity is also a likely reason for the rather unrelated search results. Therefore, developing a more sophisticated method for sampling triples that defines a more strict concept is a reasonable step towards improving the Equation-Encoder approach. The following sampling strategies could be interesting for future work:

- Split mathematical expressions found in papers at $=$, $>$, $\geq$, $<$ or $\leq$. The obtained equations could be used as the anchor and the positive formula of a triple. Their semantic and syntactic is very strong since one formula is the result of a mathematical transformation of the other formula.

- Sometimes scientific publications have a small summary of their basic methods. Two papers that use the same method and have such a summarisation might contain formulas which describe the same concept with a different notation or from a different perspective. So, we could do something like this: Take all papers with a certain keyword (e.g. *Q-Learning*) in its abstract and put all formulas from those papers that appear in an section called *background*, *introduction*, *preliminary* etc. in one class. Then we can sample anchor and positive formula from the same class.

Both possible sampling methods would lead to much stronger relation between the x and x_+. Thus, they would require a more strict sampling for the negative formula too. Otherwise it would probably be very easy for our model to fit the triple data.

For the overall performance of an Equation-Encoder backed search engine, we also should consider other ranking criteria than pure similarity (e.g. number of citations or recency). We also could combine the similarity ranking of different models that learned on datasets with different sampling approaches.

Appendix A

Further Information

A.1 Proof: Why do we need nonlinear activation functions?

Let l_i and l_{i+1} be two fully-connected layers with weight matrices W_i, W_{i+1} and bias vectors $\vec{b}_i, \vec{b}_{i+1}$. The activation function of of both layers is the identity function. Input to l_i is denoted with $\vec{x}$ and the output of l_{i+1} with $\vec{y}$. The following equation defines the output:

$$\vec{y} = W_{i+1}(W_i\vec{x} + \vec{b}_i) + \vec{b}_{i+1} \tag{A.1}$$

Because of the distributivity of matrix multiplications, the term could be transformed as follows:

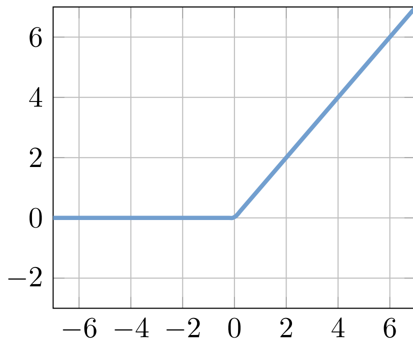
$$\vec{y} = W_{i+1}W_i\vec{x} + W_{i+1}\vec{b}_i + \vec{b}_{i+1} \tag{A.2}$$
$$= W'\vec{x} + \vec{b}' \tag{A.3}$$
$$\text{with} \tag{A.4}$$
$$W' = W_{i+1}W_i \text{ and} \tag{A.5}$$
$$\vec{b}' = W_{i+1}\vec{b}_i + \vec{b}_{i+1} \tag{A.6}$$

In this case, we did not gain anything by stacking these two layers. We could used just one layer with weights W' and biases $\vec{b}'$ instead. So, to benefit from stacking layers we need some nonlinear activation function after each layer.

A.2 Activation Functions

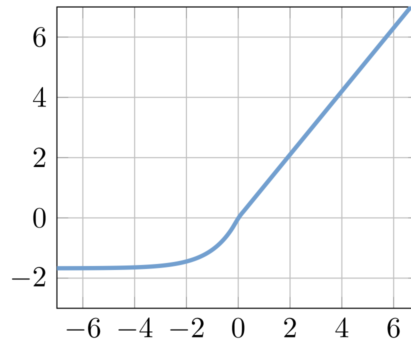
ReLU (Rectified Linear Units)

$$ReLU(x) = max(0, x)$$



SELU (Scaled Exponential Linear Units)

$$SELU(x) = \lambda * \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$



$\alpha = 1.6732632423543772848170429916717,$

$\lambda = 1.0507009873554804934193349852946$

A.3 Triples

1606.00642/Radii Stabilization_8.png - hep-th (hep-th)

1504.05511/Conclusions_4.png - hep-ex (hep-ex)

, S. Burke[131], I. Burmeister[43], E. Busato[34], D. Büscher
Buszello[167], J.M. Butler[22], A.I. Butt[3], C.M. Buttar[53], J.I

1.

1901.00115/Morse index and bifurcation_0.png - math-ph (Mathematics)

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} = \frac{\partial L}{\partial q_i}, \ i = 1, 2, \ldots, 6,$$

hep-ph0412378/The [56,2+] multiplet revisited_4.png - hep-ph (hep-ph)

$$\frac{\bar{B}_2}{\bar{B}_3} = \frac{O_2}{O_3},$$

2.

hep-ph0412378/The Mass Operator_0.png - hep-ph (hep-ph)

$$M = \sum_i c_i O_i + \sum_i b_i \bar{B}_i$$

1502.01874/Governing equations and numerical method_1.png - physics.flu-dyn (Physics)

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\partial_i p + \sqrt{\frac{Pr}{Ra}} \nabla^2 \mathbf{u} + \theta \mathbf{e}_z,$$

1212.0901/Advances in Training Recurrent Networks_4.png - cs.LG (Computer Science)

$$= \theta_{t-1} + \mu_{t-1} v_{t-1} - \epsilon_{t-1} \nabla f(\theta_{t-1})$$

3.

1206.5538/Probabilistic Models noteAC_6.png - cs.LG (Computer Science)

$$P(h_i \mid x) = \sum_{h_1=1} \cdots \sum_{h_{i-1}=1} \sum_{h_{i+1}=1} \cdots \sum_{h_{d_h}=1} P(h \mid x)$$

1411.0040/The Slepian zero set and path decomposition_32.png - math.PR (Mathematics)

$$\frac{2}{2-a} \int_R \frac{1}{\sqrt{4\pi}} \exp\left(-\frac{x^2}{4}\right) \cdot P^{\widetilde{\mathbf{W}}_x}(F \in da) \exp\left(\frac{x^2}{4} - \frac{x^2}{4(2-a)}\right)$$

1409.3826/Results: energy and lifetime_1.png - cond-mat.mes-hall (Condensed Matter)

$$\Delta\Gamma^{\delta}_{s,t} = \Gamma^{0} - \Gamma^{\delta}_{s,t},$$

4.

1409.3826/Results: energy and lifetime_0.png - cond-mat.mes-hall (Condensed Matter)

$$\Delta\varepsilon^{\delta}_{s,t} = \varepsilon^{0} - \varepsilon^{\delta}_{s,t},$$

1604.02405/Coarse Property C_15.png - math.GT (Mathematics)

$$||\frac{u}{||u||} - \frac{v}{||v||}|| \leq \frac{1}{||u||} \cdot 2 \cdot ||u - v||$$

cond-mat0607415/Model and results_2.png - cond-mat.supr-con (Condensed Matter)

$$\omega_q^2 = \omega_0^2 \left(1 + \beta_q \frac{\sin^2(q_x/2)\left(\omega_q^2 - \Omega_q^2\right)}{\left(\omega_q^2 - \Omega_q^2\right)^2 + \left(\Gamma_q \omega_q\right)^2}\right),$$

5.

cond-mat0607415/Model and results_0.png - cond-mat.supr-con (Condensed Matter)

$$\omega_0 D^{-1}(q,\omega) = \omega^2 - \omega_0^2 \left(1 + \alpha \sin^2(q_x/2) P(q,\omega)\right),$$

cond-mat9610121/Theoretical Model_6.png - cond-mat.str-el (Condensed Matter)

$$\kappa^{\alpha}_{ij} = \left| \sqrt{\left| \frac{\partial^2 q^{\alpha}_{ij}}{\partial k_x^2} \frac{\partial^2 q^{\alpha}_{ij}}{\partial k_y^2} - \left(\frac{\partial^2 q^{\alpha}_{ij}}{\partial k_x \partial k_y}\right)^2 \right|} \right|$$

1608.06075/Stronger sum uncertainty relation for n incompatible observables_1.png - quant-ph (quant-ph)

$$(\Delta A_1)^2 + (\Delta A_2)^2 \geq \frac{1}{2}[\Delta(A_1 + A_2)]^2.$$

6.

1608.06075/Stronger sum uncertainty relation for n incompatible observables_3.png - quant-ph (quant-ph)

$$\geq \frac{1}{n-2} \left\{ \sum_{1 \leq i < j \leq n} [\Delta(A_i + A_j)]^2 - \frac{1}{(n-1)^2} \left[\sum_{1 \leq i < j \leq n} \Delta(A_i \right.\right.$$

hep-th0109057/Conformal Komar mass_7.png - hep-th (hep-th)

$$\tilde{M} = -\frac{1}{8\pi} \oint_{\tilde{S}} \tilde{\epsilon}_{\alpha_1 \ldots \alpha_{n-2} \mu\nu} \tilde{\nabla}^{\mu} \xi^{\nu} d\tilde{S}^{\alpha_1 \ldots \alpha_{n-2}}$$

1703.02775/Problems and Questions_2.png - math.CA (Mathematics)

$$\mathbb{F}_{\lambda}(T) = \min_{S} M_m(T - \partial S) + \lambda M_{m+1}(S).$$

7.

1703.02775/Measures: A Brief Reminder_2.png - math.CA (Mathematics)

$$\mathcal{V}^k_{\mathcal{E}}(A) = \sum_i \alpha(k) \left(\frac{\operatorname{diam}(E_i)}{2}\right)^k,$$

1601.05881/Measure of qubit quality_3.png - cond-mat.mes-hall (Condensed Matter)

$$\Gamma_{\rm ph} \approx \gamma |\mathbf{v}|^2,$$

1501.03969/Application to Control of a Gasoline Homogeneous Charge Compression Ignition Engine_14.png - cs.SY (Computer Science

$$\max_{\lambda_L} \left\{ \frac{1}{2} \Delta U^T(k) W_1(k) \Delta U(k) + W_2^T(k) \Delta U(k) \right.$$

1501.03969/Application to Control of a Gasoline Homogeneous Charge Compression Ignition Engine_7.png - cs.SY (Computer Science)

8.

$$\lambda_{L_{k+1}} = max(\lambda_{L_k} + \lambda_{step}(\Lambda_1 \lambda_L + \Lambda_2), 0)$$

1606.08060/Consistency_12.png - math.AP (Mathematics)

$$B_{1,i} = (\phi_\alpha \phi^{(3)} + \frac{1}{4} \phi_{\alpha\alpha}^2)_i, \quad |B_{2,i}| \leq c,$$

A.4 Search Results from User Study - SEE

Query:

$$min\ max\ V(D,G) = \mathbb{E}[log(1 - D(x)] + \mathbb{E}[log(1 - D(G(z)))]$$

Generative Adversarial Network (Generative Model)

Results:

$$L(G, D; w) = \frac{1}{m}\sum_{i=1}^{m}\log(D(x_i)) + \sum_{i=1}^{m} w_i \log(1 - D(G(z_i))).$$

From: *Training Generative Adversarial Networks with Weights*, Similarity: 0.981

$$\min_G \max_D V(D,G) = \min_G \max_D \mathbb{E}_{x \sim p_d(x)}[logD(x)] + \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))].$$

From: *Generate the corresponding Image from Text Description using Modified GAN-CLS Algorithm*, Similarity: 0.979

$$\sum_{i=1}^{N} \mathbb{E}_{q(W)} \log p(y^i|x^i, W) - KL(q(W)||p(W)) \to \max_{m,\sigma}$$

From: *Bayesian Sparsification of Gated Recurrent Neural Networks*, Similarity: 0.968

$$\max_D \mathbb{E}_{x \sim p_s}[logD(F(x))] + \mathbb{E}_{x \sim p_t}[log(1 - D(F(x)))]$$

From: *Domain Confusion with Self Ensembling for Unsupervised Adaptation*, Similarity: 0.962

$$Q^*\left(m, f, s, a\right) = max_{\pi_m}\mathbb{E}\left[R_t | M_t = m, F_t = f, S_t = s, A_t = a, \pi_m\right]$$

From: *Catastrophic Importance of Catastrophic Forgetting*, Similarity: 0.962

$$V(D,G) = \mathop{\mathbb{E}}_{x \sim p_{data}(x)}[\log D(x)] + \mathop{\mathbb{E}}_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

From: *A Survey on Data Collection for Machine Learning: a Big Data – AI Integration Perspective*, Similarity: 0.962

$$L_{class} = \frac{1}{|\mathcal{D}_{real}|}{}_{(x,c) \in \mathcal{D}_{real}} \log P(C = c \mid x) + \mathbb{E}_{W,C \sim p_c}[\log P(C \mid G(W,C))],$$

From: *Teacher-Student Compression with Generative Adversarial Networks*, Similarity: 0.962

$$\min_{G_{\mathcal{J}}} \max_{D_{\mathcal{J}}} V(D_{\mathcal{J}}, G_{\mathcal{J}}) = \mathbb{E}_{x \sim P^{\mathcal{J}}_{data}}[\log(D_{\mathcal{J}}(x)] +$$

From: *StackNet: Stacking Parameters for Continual learning*, Similarity: 0.96

$$\text{REGRET}_T(L; \{w_t, y_t\}_{t=1}^T) = \sum_{t=1}^{T} \mathbb{E}\left[|f_t(w_t) - y_t|\right] - \min_{h \in H} \sum |h(w_t) - y_t|,$$

From: *Passing Tests without Memorizing: Two Models for Fooling Discriminators*, Similarity: 0.96

$$V_{\text{MM}}(D_{\theta_D}, G_{\theta_G}) = \mathbb{E}_{x \sim p_r}[\log(D_{\theta_D}(x))] + \mathbb{E}_{z \sim p_z}[\log(1 - D_{\theta_D}(G_{\theta_D}(z)))].$$

From: *Convergence Problems with Generative Adversarial Networks (GANs)*, Similarity: 0.959

Model: Squeezed Equation-Encoder

Query:

$$\theta_{i+1} = \theta_i - \eta \nabla L(\theta_i)$$

Gradient-Descent (Optimisation)

Results:

$$\theta_{i+1} = \theta_i + \alpha \nabla_\theta \mathcal{L}(\theta_i) \quad (1)$$

From: *Toward Interpretable Deep Reinforcement Learning with Linear Model U-Trees*, Similarity: 0.982

$$\theta_{k+1} = \theta_k - \eta \nabla_{\theta_k} J(\theta_k) \quad (2)$$

From: *A block-random algorithm for learning on distributed, heterogeneous data*, Similarity: 0.975

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i}) \quad (3)$$

From: *Deep Learning based Estimation of Weaving Target Maneuvers*, Similarity: 0.965

$$\frac{\tilde{\partial}\Lambda_{kk}}{\partial u_{k'}} = \begin{cases} \frac{\partial \Lambda_{kk}}{\partial u_{k'}} & \text{if } k = k' \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

From: *Spectral Inference Networks: Unifying Deep and Spectral Learning*, Similarity: 0.96

$$\theta_{t+1} = \theta_t - \alpha_{t+1} \nabla \mathcal{L}(\theta_t) \quad (5)$$

From: *Few-shot Learning with Meta Metric Learners*, Similarity: 0.958

$$\lambda_b = \frac{\sum_{i=1}^{k} n_i \|\mu_i - \bar{\mu}\|^2}{k-1} \quad (6)$$

From: *Understanding V2V Driving Scenarios through Traffic Primitives*, Similarity: 0.955

$$\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \nabla \hat{L}_t(\theta_t) \quad (7)$$

From: *Quasi-hyperbolic momentum and Adam for deep learning*, Similarity: 0.952

$$\theta_{k+1} = \theta_k - \nabla \tilde{U_k}(\theta_k) h_{k+1} + \sqrt{2h_{k+1}} \xi_{k+1} \quad (8)$$

From: *Cyclical Stochastic Gradient MCMC for Bayesian Deep Learning*, Similarity: 0.946

$$\theta_{t+1} = \theta_t - \eta w_{I_t} \nabla_{\theta_t} \mathcal{L}(\Psi(x_{I_t}; \theta_t), y_{I_t}) \quad (9)$$

From: *Not All Samples Are Created Equal: Deep Learning with Importance Sampling*, Similarity: 0.945

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t \quad (10)$$

From: *Deep Learning based Estimation of Weaving Target Maneuvers*, Similarity: 0.944

Model: Squeezed Equation-Encoder

Query:

$$argmin \sum_k \sum_S ||x - \mu_i||^2$$

k-means (Clustering)

Results:

$$\approx \sum_a p_a \sum_b q_{ab} d(a,b)^2 \quad (1)$$

From: *Introducing user-prescribed constraints in Markov chains for nonlinear dimensionality reduction*, Similarity: 0.911

$$x^{adv_{t+1}} = x^{adv_t} + \alpha \cdot (g_{t+1}) \quad (2)$$

From: *Improving the Generalization of Adversarial Training with Domain Adaptation*, Similarity: 0.905

$$P(h|x) = \prod_i P(h_i|x) \quad (3)$$

From: *Deep Generative Networks For Sequence Prediction*, Similarity: 0.904

$$R_{i \leftarrow j}^{(l,l+1)} = \frac{z_{ij}}{z_j + \epsilon \cdot sign(z_j)} \cdot R_j^{(l+1)} \quad (4)$$

From: *Analyzing Neuroimaging Data Through Recurrent Deep Learning Models*, Similarity: 0.903

$$p_i = \sum_{l \neq i} p_{il} y_{il} \quad (5)$$

From: *Feature Dimensionality Reduction for Video Affect Classification: A Comparative Study*, Similarity: 0.902

$$x \mapsto \sum_{k=1}^{N} a_k \sigma(\langle w_k, x \rangle + b_k) \quad (6)$$

From: *Some Approximation Bounds for Deep Networks*, Similarity: 0.896

$$c_j^i \leftarrow \lambda c_j^i + (1-\lambda) \sum_l 1\left[z_q(y_l) = e_j^i\right] \quad (7)$$

From: *Fast Decoding in Sequence Models using Discrete Latent Variables*, Similarity: 0.896

$$\jmath_{\text{entropy}} = -\sum_{g \in G} \mathrm{I}_g \ln p_g \quad (8)$$

From: *Domain Adaptation for sEMG-based Gesture Recognition with Recurrent Neural Networks*, Similarity: 0.895

$$= \sum_{i=1}^{*} *i*i_i^* = \sum_{i=1}^{*} *i_i^* \quad (9)$$

From: *How do infinite width bounded norm networks look in function space?*, Similarity: 0.894

$$h_j = \sigma_f \left(\sum_i W_{ji} v_i + b_j \right) \quad (10)$$

From: *A Deep Learning Approach with an Attention Mechanism for Automatic Sleep Stage Classification*, Similarity: 0.894

Model: Squeezed Equation-Encoder

Query:

$$f_t = \sigma(W_f * x_t + U_f * h_{t-1} + b_f)$$

Long Short-Term Memory (Deep Learning)

Results:

$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (1)$$

From: *Time is of the Essence: Machine Learning-based Intrusion Detection in Industrial Time Series Data*, Similarity: 0.991

$$r_t = sigmoid\left(W_r x_t + U_r h_{t-1} + b_r\right) \quad (2)$$

From: *Recurrent Neural Networks for Time Series Forecasting*, Similarity: 0.99

$$o_t = sigmoid\left(W_o x_t + U_o h_{t-1} + b_o\right) \quad (3)$$

From: *Recurrent Neural Networks for Time Series Forecasting*, Similarity: 0.99

$$f_t = sigmoid\left(W_f x_t + U_f h_{t-1} + b_f\right) \quad (4)$$

From: *Recurrent Neural Networks for Time Series Forecasting*, Similarity: 0.989

$$v_t = tanh(W_v h_t + b_v) \quad (5)$$

From: *Deep Neural Net with Attention for Multi-channel Multi-touch Attribution*, Similarity: 0.988

$$h_t = (1 - z_t) * n_t + z_t * h_{t-1} \quad (6)$$

From: *Foresee: Attentive Future Projections of Chaotic Road Environments with Online Training*, Similarity: 0.987

$$r_t = \sigma(W_{ir} * x_t + b_{ir} + W_{hr} * h_{t-1} + b_{hr}) \quad (7)$$

From: *Foresee: Attentive Future Projections of Chaotic Road Environments with Online Training*, Similarity: 0.987

$$exponent = \frac{1}{N} * \sum_{i=0}^{N} log2(|X_i|) \quad (8)$$

From: *Low-Precision Floating-Point Schemes for Neural Network Training*, Similarity: 0.986

$$i_t = sigmoid\left(W_i x_t + U_i h_{t-1} + b_i\right) \quad (9)$$

From: *Recurrent Neural Networks for Time Series Forecasting*, Similarity: 0.986

$$\begin{gathered} h_t = ([tpr_t, fpr_t] \times W_1 + b_1) \\ cd_L(t, D, C) = \sigma\left(h_t \times W_2 + b_2\right) \end{gathered} \quad (10)$$

From: *Learning to Weight for Text Classification*, Similarity: 0.985

Model: Squeezed Equation-Encoder

Query:

$$\nabla \mathbb{J}(\theta) = \mathbb{E}_{\pi}[q_{\pi}(s,a)\nabla_{\theta}\pi(a|s,\theta)]$$

Policy Gradient (Reinforcement Learning)

Results:

$$\nabla_{\theta} V(\theta) = \mathbb{E}_{s\sim\rho^{\pi}, a\sim\pi_w}[\nabla_{\theta}\log\pi_{\theta}(a|s)\hat{Q}(s,a)] \tag{1}$$

From: *An Introduction to Deep Reinforcement Learning*, Similarity: 0.97

$$\nabla_{\theta} J(\pi) = \mathbb{E}_{\pi}\left[\nabla_{\theta}\log\pi(a|s)\left(Q^{\pi}(s,a) - b(s)\right)\right] \tag{2}$$

From: *CM3: Cooperative Multi-goal Multi-stage Multi-agent Reinforcement Learning*, Similarity: 0.96

$$= -\mathbb{E}_{\epsilon\sim\rho_0(\cdot)}\left[\nabla_a c_{\psi^*}(f(\theta,\epsilon),s)\nabla_{\theta} f_{\theta}(s,\epsilon)\right] \tag{3}$$

From: *Implicit Policy for Reinforcement Learning*, Similarity: 0.956

$$\sum_{i\in High_d} \nabla J_i(\theta) = \nabla J(\theta) - \sum_{i\in Low_d} \nabla J_i(\theta) = \nabla J(\theta) \tag{4}$$

From: *Accelerating Minibatch Stochastic Gradient Descent using Typicality Sampling*, Similarity: 0.952

$$\nabla^2_{\theta'} KL(\theta,\theta')|_{\theta'=\theta} := -\nabla^2_{\theta'} \int_{\tau\in\Upsilon} f(\tau;\theta)\left[\log\left(f(\tau;\theta')\right) - \log\left(f(\tau;\theta)\right)\right] d\tau|_{\theta'=\theta} \tag{5}$$

From: *Trust Region Policy Optimization for POMDPs*, Similarity: 0.951

$$\hat{f}(\psi) = \alpha(\nabla_{\theta}\log\pi_{\theta}(a|s)(R - V_{\theta}(s)) \tag{6}$$

From: *Fast Efficient Hyperparameter Tuning for Policy Gradients*, Similarity: 0.948

$$U(\pi) = \int_s \int_a \int_{s'} u(s,a,s')p(s'|s,a)p(s,a|\pi)\, ds'\, da\, ds \tag{7}$$

From: *Model-Based Active Exploration*, Similarity: 0.946

$$\mathcal{T}Q(s,a) = r(s,a) + \gamma \int_{s'\in\mathbb{S}} P(s,s',a) \max_{a'\in\mathbb{A}} Q(s',a')\mathrm{d}s \tag{8}$$

From: *Off-Policy Actor-Critic in an Ensemble: Achieving Maximum General Entropy and Effective Environment Exploration in Deep Reinforcement Learning*, Similarity: 0.946

$$\nabla_a Q(s_t,a)|_{a=\mu(s_t)}\nabla_{\theta^{\mu}}\mu(s_t) \tag{9}$$

From: *QUOTA: The Quantile Option Architecture for Reinforcement Learning*, Similarity: 0.94

$$= -\int_{\tau\in\Upsilon} f(\tau;\theta)\nabla^2_{\theta'}\log\left(f(\tau;\theta')\right) d\tau|_{\theta'=\theta} \tag{10}$$

From: *Trust Region Policy Optimization for POMDPs*, Similarity: 0.939

Model: Squeezed Equation-Encoder

Query:

$$Q(s,a) = R(s,a) + \sum Q(s', \pi(s'))$$

Q-Learning (Reinforcement Learning)

Results:

$$Q(s,a) \leftarrow \frac{N(s,a) \cdot Q(s,a) + V(s)}{N(s,a)+1} \tag{1}$$

From: *Improved Reinforcement Learning with Curriculum*, Similarity: 0.925

$$\bar{R}(s,a,s') = \tilde{r}(s,a) + \gamma \tilde{V}(s') \tag{2}$$

From: *Reward-estimation variance elimination in sequential decision processes*, Similarity: 0.925

$$Q(s_2,a_1) = r^+ + \gamma Q(s_3,a_2) \tag{3}$$

From: *Interpretable Reinforcement Learning via Differentiable Decision Trees*, Similarity: 0.924

$$\frac{\partial \ln 1 - \beta_o(s',\vartheta) + \pi_{\mathcal{O}}(s',o)\beta_o(s',\vartheta)}{\partial \vartheta} \tag{4}$$

From: *Natural Option Critic*, Similarity: 0.923

$$Q(s_3,a_2) = r^+ + \gamma Q(s_2,a_1) \tag{5}$$

From: *Interpretable Reinforcement Learning via Differentiable Decision Trees*, Similarity: 0.923

$$Q^{\pi}_{\delta t}(s,a) = r(s,a)\delta t + \tag{6}$$

From: *Making Deep Q-learning methods robust to time discretization*, Similarity: 0.922

$$Q(s_1,a_1) = Q(s_1,a_2) = Q(s_4,a_1) = Q(s_4,a_2) = r^- \tag{7}$$

From: *Interpretable Reinforcement Learning via Differentiable Decision Trees*, Similarity: 0.92

$$D_{\xi,\varphi}(s,a,s') = \frac{\exp[f_{\xi,\varphi}(s,a,s')]}{\exp[f_{\xi,\varphi}(s,a,s')] + \pi(a|s)} \tag{8}$$

From: *Adversarial Imitation via Variational Inverse Reinforcement Learning*, Similarity: 0.92

$$A_i(o,a) = Q_i^{\psi}(o,a) - b(o,a_{\backslash i})), \text{ where} \tag{9}$$

From: *Actor-Attention-Critic for Multi-Agent Reinforcement Learning*, Similarity: 0.917

$$= \sum_{s',o} \mu_{\mathcal{O}}(s',o)\Big(\beta_o(s',\vartheta)(1-\pi_{\mathcal{O}}(s',o))\frac{\partial_i \partial_j \beta_o(s',\vartheta)}{\beta_o(s',\vartheta)} \tag{10}$$

From: *Natural Option Critic*, Similarity: 0.916

Model: Squeezed Equation-Encoder

Query:

$$l_i(\theta, \phi) = -\mathbb{E}_{z\sim q_\theta}(z|x_i)[log\, p_\phi(x_i|z)] + \mathbb{KL}(q_\theta(z|x_i)|p(z))$$

Variational Autoencoder (Generative Model)

Results:

$$\mathcal{L}_x(\theta_x, \phi_x; x_u, z_u, \beta_x) = \underbrace{\mathbb{E}_{q_{\phi_x}(z_u|x_u)}[\log p_{\theta_x}(x_u|z_u)]}_{\text{Reconstruction loss}} - \beta_x \underbrace{KL(q_{\phi_x}(z_u|x_u)||p(z_u))}_{\text{Capacity limitation regularization}} \quad (1)$$

From: *Variational Collaborative Learning for User Probabilistic Representation*, Similarity: 0.986

$$\mathrm{MMD}^2(p||q) = \mathbb{E}_{x,y\sim p}[k(x,y)] + \mathbb{E}_{x,y\sim q}[k(x,y)] \quad (2)$$

From: *GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models*, Similarity: 0.976

$$H(\epsilon|x) = -\mathbb{E}_{p_\theta(x,\epsilon)} \log p_\theta(\epsilon|x) \quad (3)$$

From: *Adversarial Learning of a Sampler Based on an Unnormalized Distribution*, Similarity: 0.975

$$q^\pi(s_t, a_t) = r_t + \gamma \max_{a_{t+1}} q^\pi(s_{t+1}, a_{t+1}) \quad (4)$$

From: *Improved robustness of reinforcement learning policies upon conversion to spiking neuronal network platforms applied to ATARI games*, Similarity: 0.974

$$u \sim q(u|\vec{x}, \vec{a}, \vec{r})\mathbb{E}\left[\log p(x_t|z_t, u) + \log p(a_t|z_t, u) + \log p(r_{t+1}|z_t, a_t, u)\right] \quad (5)$$

From: *Deconfounding Reinforcement Learning in Observational Settings*, Similarity: 0.973

$$= \sum_{t=1}^{T} \mathop{\mathbb{E}}_{\substack{z_t\sim q(z_t|z_{t-1},\vec{x},\vec{a},\vec{r}) \\ u\sim q(u|\vec{x},\vec{a},\vec{r})}} [\log p(x_t|z_t, u) + \log p(a_t|z_t, u) + \log p(r_{t+1}|z_t, a_t, u)] - \mathrm{KL}\left(q(u|\vec{x}, \vec{a}, \vec{r})||p(u\right. \quad (6)$$

From: *Deconfounding Reinforcement Learning in Observational Settings*, Similarity: 0.972

$$= \mathbb{E}_{(,)\sim p(G_D(,))} \sum_{y\in\mathcal{Y}} post(, y,)\nabla \log(p(y|;)) \quad (7)$$

From: *Skeptical Deep Learning with Distribution Correction*, Similarity: 0.971

$$l'_{ty} = l_{ty} + [[y = \mathrm{eos}]] \log\left(\sigma(w_1(c_t - w_2))\right) \quad (8)$$

From: *Calibration of Encoder Decoder Models for Neural Machine Translation*, Similarity: 0.97

$$\geq_{q(z|x_{1:n},y_{1:n})} \left[\sum_{i=m+1}^{n} \log p(y_i|z, x_i) + \log \frac{p(z|x_{1:m}, y_{1:m})}{q(z|x_{1:n}, y_{1:n})}\right] \quad (9)$$

From: *Neural Processes*, Similarity: 0.97

$$\log p(y_t, y_c) \geq \mathbb{E}_{q(z|y_c,y_t)}\Bigg[\log p(y_t|z) \quad (10)$$

From: *Neural Processes*, Similarity: 0.97

Model: Squeezed Equation-Encoder

A.5 Search Results from User Study - BEE

Query:

$$min\ max\ V(D,G) = \mathbb{E}[log(1 - D(x)] + \mathbb{E}[log(1 - D(G(z)))]$$

Generative Adversial Network (Generative model)

Results:

$$MMD(X_S, X_T) = \left\| \frac{1}{n_1} \sum_{i=1}^{n_1} f(x_{S_i}) - \frac{1}{n_2} \sum_{i=1}^{n_2} f(x_{T_i}) \right\|_{\mathcal{H}}^2 \quad (1)$$

From: *TLR: Transfer Latent Representation for Unsupervised Domain Adaptation*, Similarity: 0.982

$$I_{Loss} = \left| I(X,Y) - \mathbb{E}\left[log_2 \frac{\hat{p}(x,y)}{\hat{p}(x)\hat{p}(y)} \right] \right| \quad (2)$$

From: *Interpreting Active Learning Methods Through Information Losses*, Similarity: 0.974

$$E[y_c z^\intercal] = E[E[y_c z^\intercal | y_c = 1]] = E[z^\intercal | y_c = 1]\mu_c \quad (3)$$

From: *Deep Learning under Privileged Information Using Heteroscedastic Dropout*, Similarity: 0.972

$$\min_G \max_D V(G,D) = \mathbb{E}_{x \sim p_{data}}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]. \quad (4)$$

From: *Training GANs with Centripetal Acceleration*, Similarity: 0.972

$$AM_{score}(\{x\}_1^N) = \mathbb{E}[KL(\bar{c}^{train} || c(x)] - KL(\bar{c}^{train} || \mathbb{E}[c(x)])] \quad (5)$$

From: *DeSIGN: Design Inspiration from Generative Networks*, Similarity: 0.968

$$\max_D L_D = \mathbb{E}_{x \sim p_{data}}[-\log D(x)] \quad (6)$$

From: *Task Transfer by Preference-Based Cost Learning*, Similarity: 0.968

$$\log(P(I|x^*)) = I * \log(f(x^*|x)) + (1 - I) * \log(1 - f(x^*|x)) \quad (7)$$

From: *Modelling Latent Travel Behaviour Characteristics with Generative Machine Learning*, Similarity: 0.968

$$\hat{y} = W_2((1-p)I + pr)W_1 x = W_2 r_p(W_1 x) \quad (8)$$

From: *Drop-Activation: Implicit Parameter Reduction and Harmonic Regularization*, Similarity: 0.968

$$\left(\frac{1}{n} \left(1 - \left(1 - Pr(-u,v)^K\right)^L \right) \frac{|C_v \cap S_{-u}|}{|S_{-u}|} \right) \quad (9)$$

From: *Scaling-up Split-Merge MCMC with Locality Sensitive Sampling (LSS)*, Similarity: 0.967

$$MMD^2(F,X,Y) = \frac{1}{m(m-1)} \sum_i^m \sum_{i \neq j}^m k(x_i, x_j) \quad (10)$$

From: *A Theoretical Investigation of Graph Degree as an Unsupervised Normality Measure*, Similarity: 0.966

Model: Baseline Equation-Encoder

Query:

$$\theta_{i+1} = \theta_i - \eta \nabla L(\theta_i)$$

Gradient Descent (Optimisation)

Results:

$$\theta_{i+1} = \theta_i + \alpha \nabla_\theta \mathcal{L}(\theta_i) \quad (1)$$

From: *Toward Interpretable Deep Reinforcement Learning with Linear Model U-Trees*, Similarity: 0.968

$$\frac{\partial J(\theta)}{\partial \theta} = -\sum_i I(y = C_i) \frac{1}{\hat{y}_i(\theta)} \frac{\partial \hat{y}_i(\theta)}{\partial \theta} \quad (2)$$

From: *When Work Matters: Transforming Classical Network Structures to Graph CNN*, Similarity: 0.956

$$\theta = \arg\min_\beta \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} w_{ij} \ell(x_i, j, \beta) + \lambda \Omega(\beta) \quad (3)$$

From: *Training Set Debugging Using Trusted Items*, Similarity: 0.949

$$i \neq k (w_i - w_i^*)^2 + 2(w_k - w_k^*)^2 + \sigma^2 \quad (4)$$

From: *Byzantine-Robust Distributed Learning: Towards Optimal Statistical Rates*, Similarity: 0.944

$$\theta_{k+1} = \theta_k - \eta \nabla_{\theta_k} J(\theta_k) \quad (5)$$

From: *A block-random algorithm for learning on distributed, heterogeneous data*, Similarity: 0.943

$$w_i^{t+1} = w_i^t - \eta \frac{\partial E}{\partial w_i^t} - \lambda w_i^t \quad (6)$$

From: *Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes*, Similarity: 0.942

$$= -\frac{1}{m} \sum_{j=1}^{m} \sum_{l=1}^{L} \frac{\partial f_j^v}{\partial \theta_l} |_{\theta_l = \theta_{l,t}}^\top \frac{\partial f_i}{\partial \theta_l} |_{\theta_l = \theta_{l,t}} \quad (7)$$

From: *Learning to Reweight Examples for Robust Deep Learning*, Similarity: 0.941

$$u_i^{t+1} v_i^{t+1} = u_i^t \exp(-\eta g_i^t) v_i^t \exp(\eta g_i^t) = u_i^t v_i^t \ . \quad (8)$$

From: *Exponentiated Gradient Meets Gradient Descent*, Similarity: 0.941

$$\frac{\partial L}{\partial w_{ij}^{l+1}} = \frac{\partial L}{\partial h_j^{l+1}} a_i^l \quad (9)$$

From: *Detecting Dead Weights and Units in Neural Networks*, Similarity: 0.940

$$L(\theta) = L^s(\theta) + \lambda L^t(\theta) \quad (10)$$

From: *Multimodal Deep Domain Adaptation*, Similarity: 0.940

Model: Baseline Equation-Encoder

Query:

$$argmin \sum_k \sum_S ||x - \mu_i||^2$$

k-means (Clustering)

Results:

$$HammingLoss = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{q} \left| C(x^i) \triangle y^i \right| \quad (1)$$

From: *Domain-Adversarial Multi-Task Framework for Novel Therapeutic Property Prediction of Compounds*, Similarity: 0.987

$$F1(\tilde{y}, \tilde{\tilde{y}}) = 2 \cdot \frac{precision \cdot recall}{precision + recall} \quad (2)$$

From: *Recurrent Neural Networks for Time Series Forecasting*, Similarity: 0.980

$$R_{block} = -\|x_{obj} - x_{goal}\|_2 \quad (3)$$

From: *Meta-Reinforcement Learning of Structured Exploration Strategies*, Similarity: 0.979

$$L_{Lipschitz} = \beta * \max(0, k(x) - L_n) \quad (4)$$

From: *Towards Robust Neural Networks with Lipschitz Continuity*, Similarity: 0.978

$$\text{minimize } \mathcal{D}(x, x_{adv}) \quad (5)$$

From: *ECGadv: Generating Adversarial Electrocardiogram to Misguide Arrhythmia Classification System*, Similarity: 0.977

$$s_{i,l} = \frac{\exp(-||p_{cell_i} - p_{proj_l}||^2)}{||[x, y] - [x_{source_l}, y_{source_l}]||} \quad (6)$$

From: *Learning agent's spatial configuration from sensorimotor invariants*, Similarity: 0.977

$$Precision@k = \frac{Number\ of\ postive\ items\ in\ Top\ k}{k}, \quad (7)$$

From: *Distributed Collaborative Hashing and Its Applications in Ant Financial*, Similarity: 0.976

$$\ell_{weight} = \sum_{i=1}^{c} o_i (y_i - f(x))^2 \quad (8)$$

From: *Wrapped Loss Function for Regularizing Nonconforming Residual Distributions*, Similarity: 0.975

$$x_{out} = SimQuant(x) \quad (9)$$

From: *Quantizing deep convolutional networks for efficient inference: A whitepaper*, Similarity: 0.975

$$\gamma = max_{||\Delta x||_2} \mid sign(y_i f(x_i)) = sign(y_i f(x_i + \Delta x)). \quad (10)$$

From: *Theory of Deep Learning IIb: Optimization Properties of SGD*, Similarity: 0.975

Model: Baseline Equation-Encoder

Query:

$$f_t = \sigma(W_f * x_t + U_f * h_{t-1} + b_f)$$

Long Short Term Memory (Recurrent Neural Networks)

Results:

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{1}$$

From: *Towards Binary-Valued Gates for Robust LSTM Training*, Similarity: 0.998

$$f_t = \sigma(W_f h_{t-1} + V_f x_t + b_f) \tag{2}$$

From: *Neural Tensor Factorization*, Similarity: 0.996

$$r_t = \sigma(U^r X_t + W^r s_{t-1} + b^r) \tag{3}$$

From: *Decentralized Flood Forecasting Using Deep Neural Networks*, Similarity: 0.995

$$J(w_{t+1}) = J(w_t) - \frac{(d_t^T K(1 - w_t))^2}{d_t^T K d_t} \tag{4}$$

From: *Greedy Frank-Wolfe Algorithm for Exemplar Selection*, Similarity: 0.994

$$o_t = \sigma\left((W_o x_t + U_o h_{t-1} + b_o\right) \tag{5}$$

From: *A deep learning approach to real-time parking occupancy prediction in spatio-temporal networks incorporating multiple spatio-temporal data sources*, Similarity: 0.994

$$o_t = g(W_o.x_t + U_o.h_{t-1} + b_o) \tag{6}$$

From: *A Multi-variable Stacked Long-Short Term Memory Network for Wind Speed Forecasting*, Similarity: 0.993

$$f_t = g(W_f.x_t + U_f.h_{t-1} + b_f) \tag{7}$$

From: *A Multi-variable Stacked Long-Short Term Memory Network for Wind Speed Forecasting*, Similarity: 0.993

$$\widetilde{c_t} = \phi(BN(W_c h_{t-1} + V_c x_t + b_c)) \tag{8}$$

From: *Neural Tensor Factorization*, Similarity: 0.9993

$$o_t = \sigma(W_o h_{t-1} + V_o x_t + b_o) \tag{9}$$

From: *Neural Tensor Factorization*, Similarity: 0.993

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} \tag{10}$$

From: *A Neural Network Approach to Missing Marker Reconstruction in Human Motion Capture*, Similarity: 0.993

Model: Baseline Equation-Encoder

Query:

$$\nabla \mathbb{J}(\theta) = \mathbb{E}_{\pi}[q_{\pi}(s,a)\nabla_{\theta}\pi(a|s,\theta)]$$

Policy Gradient (Reinforcement Learning)

Results:

$$\theta' = \theta + \alpha \mathop{\mathbb{E}}_{s,a\sim\rho}\left[\left({}^{*}Q_{\theta}(s,a) - Q_{\theta}(s,a)\right)\nabla_{\theta}Q_{\theta}(s,a)\right], \tag{1}$$

From: *Towards Characterizing Divergence in Deep Q-Learning*, Similarity: 0.979

$$= m\int_{c}\left[\nabla_{\theta}p_{\theta}(s,c)Q^{\pi}(s,c) + p_{\theta}(s,c)\nabla_{\theta}Q^{\pi}(s,c)\right]g^{s}(c)\mathrm{d}c \tag{2}$$

From: *Policy Gradients for Contextual Recommendations*, Similarity: 0.978

$$\nabla_{\theta}\mathcal{L}(\theta) = \mathbb{E}_{s,a}[(Q^{*} - Q(s,a;\theta))\nabla_{\theta}Q(s,a;\theta)]. \tag{3}$$

From: *Joint Modeling of Dense and Incomplete Trajectories for Citywide Traffic Volume Inference*, Similarity: 0.976

$$= -\mathbb{E}_{\epsilon\sim\rho_0(\cdot)}\left[\nabla_{a}c_{\psi^{*}}(f(\theta,\epsilon),s)\nabla_{\theta}f_{\theta}(s,\epsilon)\right] \tag{4}$$

From: *Implicit Policy for Reinforcement Learning*, Similarity: 0.976

$$\nabla_{\theta}V(\theta) = \mathbb{E}_{\rho}[\nabla_{\theta}\pi_{\theta}(s)\nabla_{a}Q_{w}(s,a)|_{a=\pi_{\theta}(s)}] \tag{5}$$

From: *An Introduction to Deep Reinforcement Learning*, Similarity: 0.973

$$\mathbb{E}[\nabla_{\theta}\log\pi(a|s)b(s,a) - \nabla_{\theta}f(\theta,s,\xi)\nabla_{a}b(s,a)] = 0. \tag{6}$$

From: *Policy Optimization with Second-Order Advantage Information*, Similarity: 0.973

$$\nabla_{\theta}\log(f(\tau;\theta)) = \nabla_{\theta}\log\left(\Pi_{h=1}^{|\tau|}\pi_{\theta}(a_h|y_h)\right) \tag{7}$$

From: *Trust Region Policy Optimization for POMDPs*, Similarity: 0.972

$$\nabla_{\theta}\mathbb{H}\left[\pi_{\theta}(\cdot|s)\right] = -\nabla_{\theta}\mathbb{E}_{\epsilon\sim\rho_0(\cdot)}\left[\log\pi_{\theta}(f_{\theta}(s,\epsilon)|s)\right] \tag{8}$$

From: *Implicit Policy for Reinforcement Learning*, Similarity: 0.969

$$\nabla_{\theta}J(\theta) = \sum_{t=0}^{H-1}\mathbb{E}_{s_t\sim d^{t}_{\pi_{\theta}}}\left[\nabla_{\theta}\pi(\theta,s_t)\nabla_{a}Q^{t}_{\pi_{\theta}}(s_t,\pi(\theta,s_t))\right] \tag{9}$$

From: *Contrasting Exploration in Parameter and Action Space: A Zeroth-Order Optimization Perspective*, Similarity: 0.969

$$\nabla_{\omega}L = \mathrm{E}_{\tau}[\nabla_{\omega}log(D_{\omega}(s,a))] + \mathrm{E}_{\tau_h}[\nabla_{\omega}log(1 - D_{\omega}(s,a))] \tag{10}$$

From: *Hindsight Generative Adversarial Imitation Learning*, Similarity: 0.969

Model: Baseline Equation-Encoder

Query:

$$Q(s,a) = R(s,a) + \sum Q(s', \pi(s'))$$

Q-learning (Reinforcement Learning)

Results:

$$Q(s,a) \leftarrow \frac{N(s,a) \cdot Q(s,a) + V(s)}{N(s,a)+1} \quad (1)$$

From: *Improved Reinforcement Learning with Curriculum*, Similarity: 0.986

$$Q(s,a) = R(s,a) + \gamma max_{a'}(Q(s',a')) \quad (2)$$

From: *Mitigation of Policy Manipulation Attacks on Deep Q-Networks with Parameter-Space Noise*, Similarity: 0.979

$$Q(s,a) = R(s,a) + \gamma max_{a'}(Q(s',a')) \quad (3)$$

From: *The Faults in Our Pi Stars: Security Issues and Open Challenges in Deep Reinforcement Learning*, Similarity: 0.979

$$P(s_0|s,a) = 0 \forall s,a \quad (4)$$

From: *Off-Policy Deep Reinforcement Learning by Bootstrapping the Covariate Shift*, Similarity: 0.976

$$Q(s,a) := R(s,a) + \gamma \max_{a'} Q(s',a') \quad (5)$$

From: *Interpretable Reinforcement Learning via Differentiable Decision Trees*, Similarity: 0.970

$$\mu(s)P(s,s') = \mu(s')P(s',s) \quad \forall s,s' \quad (6)$$

From: *Approximate Temporal Difference Learning is a Gradient Descent for Reversible Policies*, Similarity: 0.969

$$\hat{r}(s,a,s') = f(s,a,s') - \log \pi(a|s) \quad (7)$$

From: *Adversarial Imitation via Variational Inverse Reinforcement Learning*, Similarity: 0.968

$$P(s'|s,a) = \begin{cases} 0, & \text{if } s' = 1,4 \\ p, & \text{if } s' = s+1, a = a_1 \\ p, & \text{if } s' = s-1, a = a_2 \\ \frac{1-p}{|S|-1}, & \text{otherwise} \end{cases} \quad (8)$$

From: *Interpretable Reinforcement Learning via Differentiable Decision Trees*, Similarity: 0.965

$$\forall_{s,a} : Q(s,a) = Q(s,a) + \alpha \delta_t e(s,a) \quad (9)$$

From: *Using Deep Reinforcement Learning for the Continuous Control of Robotic Arms*, Similarity: 0.963

$$\pi(s) = \text{argmax}_a Q(s,a) + M \cdot l_n(s,a) \quad (10)$$

From: *Safety-Guided Deep Reinforcement Learning via Online Gaussian Process Estimation*, Similarity: 0.960

Model: Baseline Equation-Encoder

Query:

$$l_i(\theta, \phi) = -\mathbb{E}_{z \sim q_\theta}(z|x_i)[log\, p_\phi(x_i|z)] + \mathbb{KL}(q_\theta(z|x_i)|p(z))$$

Variational Autoencoder (Generative model)

Results:

$$\log p(y_t|C, x_t) = \log \int p(y_t|z, x_t)p(z|C)dz \tag{1}$$

From: *Neural Processes*, Similarity: 0.987

$$logp_\theta(x) = E_{q_\phi(h|x)} log \left(\frac{p_\theta(x,h)}{q_\phi(h|x)}\right) + KL\left[q_\phi(h|x)||p_\theta(h|x)\right] \tag{2}$$

From: *A Review of Learning with Deep Generative Models from Perspective of Graphical Modeling*, Similarity: 0.986

$$\min_G \max_D \mathbb{E}_{x \sim p_r}[log(D(x))] + \mathbb{E}_{x' \sim p_g}[1 - log(D(x'))]. \tag{3}$$

From: *Collaborative Sampling in Generative Adversarial Networks*, Similarity: 0.985

$$\mathbb{E}_{x \sim p_\theta(x|z)}(\exp(1 + \log(\frac{\hat{p}_{data}(x)}{p_\theta(x|z)}) - 1))) \tag{4}$$

From: *Biadversarial Variational Autoencoder*, Similarity: 0.984

$$= \mathbb{E}_{x \sim \hat{p}_{data}} KL(q_\phi(z|x)||p(z)) \tag{5}$$

From: *Biadversarial Variational Autoencoder*, Similarity: 0.984

$$=_{z \sim q_\phi(z|x)} [\log p_\theta(x|z)] - D_{KL}[q_\phi(z|x)||p(z)]. \tag{6}$$

From: *Bounded Information Rate Variational Autoencoders*, Similarity: 0.984

$$= KL[q_\psi(x|c)||p(x)] + KL[p_\phi(c_g|z)||p_\theta(c|x)] \tag{7}$$

From: *Adversarially Approximated Autoencoder for Image Generation and Manipulation*, Similarity: 0.984

$$= \Delta_w \int q(w|\theta, \alpha)] \log p(y|x, w)dw \tag{8}$$

From: *A Survey on Methods and Theories of Quantized Neural Networks*, Similarity: 0.983

$$\mathcal{L}_{MAP}(w) = \sum_{n=1}^{N} \log p(y_n|x_n, w) - \log p(w) \tag{9}$$

From: *Distributed Weight Consolidation: A Brain Segmentation Case Study*, Similarity: 0.983

$$\begin{aligned} p(z|x) &= \underset{p(z|x)}{\arg\min}\ \mathbb{E}_{x \sim \tilde{p}(x)}\left[\int p(z|x) \log \frac{p(z|x)}{q(z|x)q(x)} dz\right] \\ &= \underset{p(z|x)}{\arg\min}\ \mathbb{E}_{x \sim \tilde{p}(x)}\left[KL\left(p(z|x)\|q(z|x)\right) - \log q(x)\right] \\ &= \underset{p(z|x)}{\arg\min}\ \mathbb{E}_{x \sim \tilde{p}(x)}\left[KL\left(p(z|x)\|q(z|x)\right)\right] \end{aligned} \tag{10}$$

From: *Variational Inference: A Unified Framework of Generative Models and Some Revelations*, Similarity: 0.983

Model: Baseline Equation-Encoder

Notation

Scalars and Arrays

a	A scalar.
$\vec{a}$	A vector.
A	A matrix or tensor (two or more dimensions).

Indexing

$a_{i,j}$	Element i, j of matrix A, i goes from left to right.
$\vec{a}_{i,\bullet}$	ith column of matrix A.
$\vec{a}_{\bullet,j}$	jth row of matrix A.

Sets, Graphs and Distributions

$\mathbb{A}$	A set.
$\mathcal{A}$	A graph
$\mathcal{N}(\mu, var)$	The normal distribution with mean = μ and variance = var.

Operations

$*$	The convolutional operation.
$\langle \vec{a}, \vec{b} \rangle$	The dot product of the two vectors.

Number Seperators

Decimal Sign	"."
Thousand Seperator	","

List of Acronyms

CNN Convolutional Neural Network

DFN Deep Feedforward Network

NN Neural Network

ILSVRC ImageNet Large Scale Visual Recognition Challenge

SEE Squeezed Equation-Encoder

GAN Generative Adversarial Network

GD Gradient Descent

LSTM Long Short-Term Memory

VAE Variational Autoencoder

PG Policy Gradient

QL Q-Learning

BEE Baseline Equation-Encoder

List of Figures

1.1 Yearly submission rates of computer science papers on the pre-print service arXiv.org. The letter combinations are acronyms for the different subject areas (e.g. LG stands for Machine Learning and CV stands for Computer Vision). A full list of the acronyms can be found here: here. Source: arXiv.org. 2

2.1 Illustration of how the kernel (red cube) is placed over the input tensor and of the movement of the kernel. Source: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. 8

2.2 Illustration of how a 3×3 pooling window is placed over the input (blue/bottom grid) to produce the output (cyan/top grid). Source [4]. 10

2.3 As can be seen from the left graph a sufficiently deep model showed to be an important factor for achieving good accuracy in the work of Goodfellow et al. [7]. The right graph shows that this effect is not reducible on the number of trainable parameters in deeper models. It showed that shallower models with about the same number of weights and biases performed inferior. Data from: [7]. 13

2.4 Subfigure (a) and (b) illustrate the abstraction performed by single layer as a folding of the input space and show how a successive folding can make regions linearly seperable. Subfigure (c) visualises how the abstraction from lower layers (bottom to middle) can be reused in the higher layers (middle to top) to summarise different input regions. In this case the first layer maps its five red input regions to two red output regions. Finally, this two red regions are then mapped to a common output region by the second layer. Source: [24]. 14

3.1 Architecture of the large Equation-Encoder used by Pfahler et al. [27]. Input size is 1x32x333. 19

3.2 A visualisation of a fire module. Source: [11]. 20

3.3 Architecture of SqueezeNet. Input size is 3x224x224. 22

4.1 Architecture of the SEE. Input size is 1x32x333. 27

4.2 A training run of the SEE with the initial setup. The optimisation process seems to suffer from the vanishing gradient problem. 29
4.3 The cosine similarity (blue squares) and the variance (red triangles) of the feature maps at a given layer within the SEE. The data points for layer zero show the variance and the cosine similarity for the input. Top Left: ReLU/Kaiming, Top Right: SELU/Kaiming, Bottom Left: ReLU/LeCun, Bottom Right: SELU/LeCun. 30
4.4 Experiments for determining the activation function and the initialisation scheme. 31
4.5 Experiments for determining an adequate learning rate. 33
4.6 Results of the experiments regarding citation sampling. 33
4.7 The four performance measures for the margin experiment. Only for the constant margin loss with a margin of 0.25 the version of the model trained with a margin of one does not perfom best. 34
4.8 Results of the experiments regarding the dataset that is used for training. A red mark means that the loss was computed on the test split of the full dataset, whereas a blue mark means that the loss was computed on the test split of the tuning dataset. A full circle means that the model was trained on the full dataset, wheras a halfcircle means that it was trained on the tuning dataset. The training run with the full dataset had only 20 epochs because training on a bigger dataset is more time consuming. Nevertheless, it leads to better performance for three out of four performance measures. 36

5.1 Performance of the BEE and SEE with their final configuration (see 4.2.2) and trained on the full dataset. 38

List of Tables

4.1 A Description of the Datasets in numbers. The value left from the slash refers to the train split of a dataset the other refers to the test split. 24

5.1 Memory consumption (in MegaBytes) of the squeezed model and the baseline. 38
5.2 Speed and throughput of the SEE ande the BEE. For inference 149,084 singular formulas were forwarded and the training was done on a dataset with 50,000 triples. Thus, for both speed tests roughly the same number of formulas was processed. 39
5.3 The queries that are used for our user study. 40
5.4 Results of the user study: The table shows the per-review-score for all seven queries. 41
5.5 Results of the user study. The table shows the per-result-score for all seven queries. The left value is the achieved score and the right value is the highest possible score (#reviewers). 42

Bibliography

[1] Yoram Bachrach et al. "Speeding up the XBox Recommender System using a Euclidean Transformation for Inner-Product Spaces". In: *Proceedings of the ACM Conference on Recommender systems*. 2014, pp. 257–264.

[2] Vassileios Balntas et al. "Learning Local Feature Descriptors with Triplets and Shallow Convolutional Neural Networks." In: *Proceedings of the British Machine Vision Conference*. 2016.

[3] Augustin-Louis Cauchy. "Méthode générale pour la résolution de systèmes d'équations simultanées". In: *Compte rendu des séances de l'académie des sciences*. 1847, pp. 536–538.

[4] Vincent Dumoulin and Francesco Visin. "A Guide to Convolution Arithmetic for Deep Learning". In: *arXiv preprint arXiv:1603.07285* (2016).

[5] Xavier Glorot and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks". In: *Proceedings of the International Conference on Artificial Intelligence and Statistics*. 2010, pp. 249–256.

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

[7] Ian J. Goodfellow et al. "Multi-Digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks". In: *Proceedings of the International Conference on Learning Representations*. 2014.

[8] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 1026–1034.

[9] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". In: *Diploma, Technische Universität München* 91.1 (1991).

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer Feedforward Networks are Universal Approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366.

[11] Forrest N. Iandola et al. "SqueezeNet: AlexNet-Level Accuracy with 50x fewer Parameters and $<$ 0.5 MB Model Size". In: *arXiv preprint arXiv:1602.07360* (2016).

[12] Piotr Indyk and Rajeev Motwani. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality". In: *Proceedings of the ACM Symposium on Theory of Computing*. 1998, pp. 604–613.

[13] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[14] Katarzyna Janocha and Wojciech Marian Czarnecki. "On Loss Functions for Deep Neural Networks in Classification". In: *Schedae Informaticae*. 2016, pp. 49–59.

[15] Shahab Kamali and Frank Wm. Tompa. "Retrieving Documents with Mathematical Content". In: *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2013, pp. 353–362.

[16] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *Proceedings of the International Conference on Learning Representations*. 2015.

[17] Günter Klambauer et al. "Self-Normalizing Neural Networks". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 971–980.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. 2012, pp. 1097–1105.

[19] Yann A. LeCun et al. "Efficient Backprop". In: *Neural networks: Tricks of the trade*. 2012, pp. 9–48.

[20] Min Lin, Qiang Chen, and Shuicheng Yan. "Network in Network". In: *arXiv preprint arXiv:1312.4400* (2013).

[21] Chaochao Lu, Bernhard Schölkopf, and José Miguel Hernández-Lobato. "Deconfounding Reinforcement Learning in Observational Settings". In: *arXiv preprint arXiv:1812.10576* (2018).

[22] Behrooz Mansouri et al. "Tangent-cft: An Embedding Model for Mathematical Formulas". In: *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval*. 2019, pp. 11–18.

[23] Thomas M. Mitchell et al. *Machine learning*. 1997.

[24] Guido F. Montufar et al. "On the Number of Linear Regions of Deep Neural Networks". In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 2924–2932.

[25] Yuval Netzer et al. "Reading Digits in Natural Images with Unsupervised Feature Learning". In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning.* 2011.

[26] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035.

[27] Lukas Pfahler, Jonathan Schill, and Katharina Morik. "The Search for Equations - Learning to Identify Similarities between Mathematical Expressions". In: *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*. 2019.

[28] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by back-propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536.

[29] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252.

[30] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529 (2016), pp. 484–503.

[31] Chen Sun et al. "Revisiting Unreasonable Effectiveness of Data in Deep Learning Era". In: *Proceedings of the IEEE International Conference on Computer Vision.* 2017.

[32] Jiang Wang et al. "Learning Fine-Grained Image Similarity with Deep Ranking". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2014, pp. 1386–1393.

[33] Ge Yang and Samuel Schoenholz. "Mean Field Residual Networks: On the Edge of Chaos". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 7103–7114.

[34] Richard Zanibbi et al. "NTCIR-12 MathIR Task Overview." In: *Proceedings of the NTCIR Conference on Evaluation of Information Access Technologies*. 2016.

[35] Wei Zhong and Richard Zanibbi. "Structural Similarity Search for Formulas Using Leaf-Root Paths in Operator Subtrees". In: *Proceedings of the European Conference on Information Retrieval.* 2019, pp. 116–129.