How realistic is a change coupling graph? Estimations with convolutional networks.

Philipp Czerner

April 15, 2018

Abstract. Applying advances in machine learning with neural networks to structured data, such as graphs, is an area under active research. We implement a practical application using convolutional neural networks for a real-world classification task and evaluate its performance. After generating change-coupling graphs from publicly available software projects, we train a classifier differentiating them from slightly perturbed instances and randomly generated ones, to estimate how 'realistic' a certain change-coupling graph is.

1. Introduction

In recent years, neural networks have had impressive successes across many different machine learning tasks, most notably in areas of image processing, natural language processing, and artificial intelligence. Consequently, there are efforts to apply them to other types of data, such as graphs.

Here, we implement a classifier based on convolutional neural networks for a real-world problem. In particular, we consider a certain type of graph, called change-coupling graph, that represents dependencies between files in a repository of a software project.

Our classifier tries to differentiate between graphs based on real repositories and ones that are corrupted or generated from scratch. In this manner we try to build a measure of 'realisticness', to be able to estimate how close to reality a graph is.

The motivation for this comes from the SimSE project [2]. SimSE tries, amongst other things, to predict the development of a software project with an agent-based simulation. In doing so, change-coupling graphs are generated, with the explicit goal of them being as realistic as possible. Our work is intended to function as a verifier, measuring how well this goal has been achieved. It should be noted, however, that our development was entirely independent of the SimSE project and at no point have we incorporated data generated by their simulation.

This work is structured as follows: Section 2 describes how we go about building our dataset, and gives a precise definition of our problem. Our neural network is presented in section 3. We then discuss the results we achieve on our dataset in section 4. Section

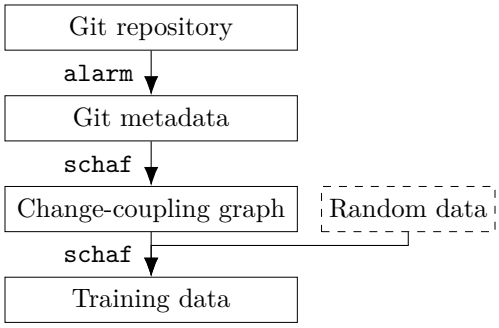


Figure 1: The different stages of transformations between the raw and training data. The labels indicate which part of our software is responsible for that step.

5 goes into the details of our implementation, giving both the general structure and the various algorithms and optimisation that were used. Finally, section 6 discusses our results and possible improvements and extensions.

2. Generating training data

A large portion of our work was focused on building a set of training data, the process of which we will describe in this section. Figure 1 gives a general overview of the different stages the data goes through until it is finally used as an input to the neural network.

First, the metadata of Git repositories is downloaded and stored in a compact format. We then parse this data to produce a number of change-coupling graphs, which are once again stored on disk. Finally, the training data is generated, adding randomly generated graphs to produce additional negative examples.

2.1. Downloading Git repository metadata

To properly train the neural network, a large dataset of graphs derived from real software projects is needed. To this end, the website GitHub [1] provides public access to an enormous number of repositories, including many important projects under active development and in widespread use (e.g. the Linux kernel).

We developed the program `alarm`, which can download metadata from GitHub and store it in an efficiently readable format. This data will then be processed further by the other part of our software, `schaf`.

Repositories on GitHub are managed with Git, a version control software. Git organises data as a number of objects with different types, with each object being identified by a SHA-1 hash of its contents. For us, the relevant objects are commits, trees and blobs. A commit object stores information about a single commit, such as its parents and an associated tree. A tree describes the contents of a directory at some point in time,

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

by listing all subdirectories (which are themselves trees) and files (which are blobs). References to other objects are always stored via their SHA-1 hash. In particular, blobs are identified by a hash of their contents, which therefore changes precisely when their contents change.

Hence, we only need to consider commits and trees when storing the metadata to generate change-coupling graphs. Changes in the contents of a file can be detected by its SHA-1 hash, without needing to consult the contents of a blob.

Git utilises a well-defined and documented protocol to communicate between a client and a server hosting a repository. There are different protocols for access over a local filesystem, HTTP, SSH or a dedicated Git daemon. GitHub provides both HTTP and SSH access.

To implement a more efficient transfer of data, we do not use Git to download the repositories. Instead, we implement Git's HTTP-based protocol ourselves. This enables a small optimisation with regards to network usage, as we can pretend to already have downloaded the contents of some files. The server may then opt to exclude these files while sending data.

More importantly, we parse the data on the fly, disregarding all objects except for trees and commits. Those we store on disk, in a compact format that includes multiple repositories in a single file. Additionally, we build an index to enable efficient access to any repository.

2.2. Change-coupling graphs

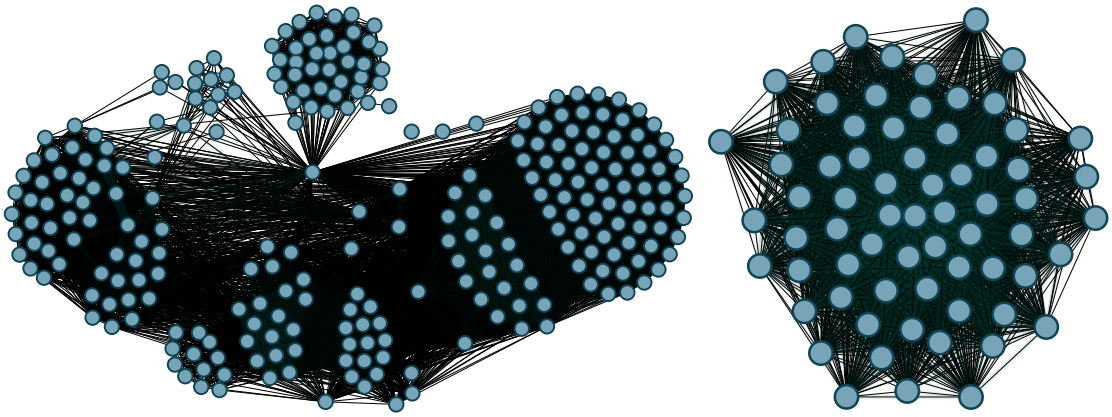
The objects of our analysis are change-coupling graphs, which are defined in terms of software projects. Broadly speaking, we take the set of all files as nodes and assign weight to an edge corresponding to the number of commits in which both adjacent nodes were changed simultaneously. We will now make that definition more precise.

Repositories consist of a set of *files*, containing blocks of binary data, called *blobs*. (We do not concern ourselves with the meaning of this data.) Files are organised via the use of *directories*, each of which is a set of files and directories it contains. Looking at files and directories via the 'is contained in' relation we get a directed rooted tree, with a root node that always is a directory.

In a software project, each *commit* is associated with such a tree, representing the state of each file and directory at the time the commit was made, and a number of parent commits, which the current commit is based on. Usually, a commit has exactly one parent.

Let us first consider commits for which this is the case. Here, we compile a list of changes based on the two trees associated with the commit and its parent. All files which are present in both trees, and have the same blob (i.e. contain the same data) we call unchanged. We then compile the set of changed files, so the set of all files which are not unchanged. (Typically there are only a few changes per commit.)

It is possible for a commit to have no parents (this is necessarily true for the first commit), or to have multiple (when merging multiple commits into a single one); computing a set of changes does not make sense in either case. Practically speaking, commits



(a) (b)

Figure 2: Example of change-coupling graphs. (a) Graph based on a real repository[2] (b) Randomly generated graph

without a parent are very rare (usually just the initial commit). Merges (commits with multiple parents) do happen from time to time. They generally do not make substantial changes, but rather merge accumulated changes from two sources. Hence, we ignore all commits that do not have exactly one parent.

We now have a set of changed files C_i for each commit i with exactly one parent. Then we define $V := \bigcup C_i$ our set of nodes, $E := V^2 \setminus \{(v, v) : v \in V\}$ our set of edges, $G = (V, E)$ an undirected simple graph, and $c : E \to \mathbb{N}$, $c(v, w) := |\{i : v, w \in C_i\}|$ the edge weights, each weight being the number of commits in which both files were changed.

In our implementation, edges with weight 0 are omitted to save memory. Many graphs only have a fraction of the maximum number of edges, which makes the memory savings quite substantial.

Figure 2a shows an example of a change-coupling graph based on real data. It is immediately visible that the graph is well connected, but it also exhibits distinctive structural properties, like the clustering of nodes.

2.3. Training data for the neural network

Our network takes n receptive fields of size m as input. In terms of our change-coupling graph $G = (V, E)$, we need to find n subgraphs of size m, such that each subgraph has high locality. (An edge with higher weight is considered shorter.)

For simplicity, we define the locality of any subgraph $G' = (V', E') \subseteq G$ as the sum of its edge weights, so $C(G') := \sum_{e \in E'} c(e)$.

We choose the first node of each subgraph simply by chance, with each node being equally likely. Finding a neighbourhood with optimal locality is NP-complete (prove by reducing Clique), so initially we relied on a simple, greedy heuristic: We would always

[2]The repository can be found at `https://github.com/mikechambers/as3corelib`.

choose the node that maximises the locality w.r.t. the current subgraph. So given a subgraph $G' = (V', E') \subseteq G$, we have

$$v' := \arg\max_{v \in V \setminus V'} \sum_{w \in V'} c(v, w)$$

Then we add v' to our subgraph. This procedure is repeated until either $|G'| = m$ or no node v' with positive value can be found. In the latter case, all outgoing edges of our subgraph G' have weight 0.

However, this heuristic tends to yield very similar neighbourhoods, which do not achieve a good coverage of the whole graph. The main problem is that it would deterministically move towards a few well-connected nodes, which most resulting subgraphs then include. The starting node provides too little variance to offset this tendency.

So we modified our heuristic to work probabilistically. Each potential node v' is weighted by

$$w(v') := \Big(\sum_{w \in V'} c(v, w) - 1 \Big)^2$$

This function was chosen empirically, based on the fact that it is cheap to calculate and gives good results on our dataset.

Once we have found a set of neighbourhoods, we then normalise the edge weights based on empirical data.

We divide each weight in a graph by the maximum weight beforehand, mapping them to $[0, 1]$. After sampling the distribution of edge weights over a large number of real graphs, we approximated it in closed form. This yields the following cumulative distribution function:

$$F_W(w) := \Pr(W \leq w) = c_1 \log(c_2 w + c_3)$$

with W the edge weight and c_1, c_2, c_3 being constants. (Their values can be found in table 1.) Consequently, our normalised weights are

$$\hat{c} : E \to [-1, 1],\ \hat{c}(e) := 2F(c(e)) - 1\ \forall e \in E$$

Once the edge weights are normalised, we generate the full adjacency matrix for the neighbourhood. Let $V' =: \{v_1, ..., v_m\}$ denote the nodes in the neighbourhood. Then the adjacency matrix is given by

$$\begin{pmatrix} 0 & \hat{c}(v_1, v_2) & \cdots & \hat{c}(v_1, v_m) \\ \hat{c}(v_2, v_1) & 0 & \cdots & \hat{c}(v_2, v_m) \\ \vdots & \vdots & \ddots & \vdots \\ \hat{c}(v_m, v_1) & \hat{c}(v_m, v_2) & \cdots & 0 \end{pmatrix}$$

Note that this matrix is symmetric and the diagonal is always zero. Those redundancies could easily be eliminated by discarding everything but the upper half. We did not explore this option, as it would somewhat increase the complexity of our implementation, and this was neither a performance bottleneck nor does it influence the neural network (it just results in some superfluous parameters in the first layer).

Table 1: The different numerical approximations used for normalising and generating training data, with W, X, Y, Z denoting edge weight, node count, commit count and commit size, respectively. $\hat{\Phi}$ denotes the cumulative distribution function of the normal distribution truncated to $[0, 1]$.

Function	c_1	c_2	c_3
$F_W(w) = c_1 \log(c_2 w + c_3)$	0.2326	64.54	1.193
$F_X(x) = 1 - (c_1 x + c_2 x^2 + c_3 x^3)^{-\frac{1}{2}}$	0.01604	$6.048 \cdot 10^{-6}$	$9.667 \cdot 10^{-10}$
$F_X^{-1}(p) = c_1 \log(1 + c_2(1-p)^{-2})$	2566	0.01869	
$F_Y(y) = 1 - c_1 y^{c_2}$	0.7548	-1.149	
$F_{Z\|W}(z; w) = c_1\big(1 - \hat{\Phi}^{-1}(z; F_W(w), c_2)\big)^{c_2} - c_1$	582.1	0.2096	0.7807

Besides the instances based on real graphs, we generate two kinds of negative examples: Instances based on graphs which are generated from scratch (called random instances), and ones from graphs where the edge weights have been changed by a small, random amount.

For the latter ones, we add an amount to a $\frac{1}{16}$ fraction of the edges. This amount is equal to $8 + Y$, with $\Pr(Y = k) = 2^{-(k+1)}$, for $k \in \mathbb{N}_0$.

To generate graphs randomly, we created closed form approximations for the number of nodes, the number of commits, and the size of each commit. The distribution of the number of commits is based on the number of nodes. The precise distributions are shown in Table 1.

Generating a graph is then done by simply sampling from these distributions. We first sample the number of nodes and use that to generate the number of commits. For each of those commits we create a commit size and take that many nodes randomly.

This creates graphs which have similar statistical properties as real change-coupling graphs but do not exhibit their typical structure. Illustrating this, figure 2b shows a sample random graph. While the distribution of edge weights is similar, the graph does not show any relationships between groups of nodes.

3. Classifying graphs with Convolutional Neural Networks

Machine learning on structured data represented in the form of graphs faces the problem of choosing an appropriate representation. While graphs are highly structured and admit a measure of locality, this structure cannot easily be embedded spatially. Furthermore, the size of graphs may vary immensely. To feasibly process large graphs, efficient algorithms are required.

There are different approaches to these problems, revolving around various methods to 'squash' the graph into a more convenient representation, e.g. a real-valued vector. Examples include identifying important substructures [15], quantifying similarities with graph kernels [13], or using learned representations via recurrent neural networks [9].

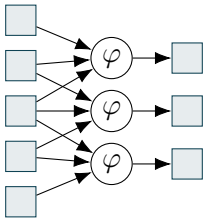


Figure 3: A convolutional layer applied to a one-dimensional input. Note that the function applied to each triple is the same.

We make use of the technique described in [8]. Initially, a sequence of nodes is chosen, the length of which is fixed. We then generate a neighbourhood for each of those nodes, which is a group of nodes preserving locality as much as possible. These neighbourhoods are also of constant size. For details on how they are generated, please refer to section 2.3.

3.1. Structure of the neural network

Our neural network is based on convolutional neural networks (CNNs), which are based on concepts proposed in [4]. These have been used to great effect in many different areas, for example in image classification [6], drug discovery [14], and board games like Go and Chess [10, 11].

CNNs revolve around applying a filter sequentially to a grid of input data (see figure 3). The moving window of input is called receptive field. This cannot be applied directly to graphs, as they cannot be embedded into a grid without losing a significant amount of structure.

Instead we generate receptive fields by considering local neighbourhoods and move the filter to another random node in each iteration. Consequently, the locality of the *receptive fields* w.r.t. each other is not preserved. Stacking another convolutional layer on top of the first one would thus not be meaningful.

As there are only a constant number of receptive fields, we cannot consider the whole graph in a single input step. However, it is possible to run the network multiple times over different sections of a large graph, and aggregate the results.

Figure 4 shows the structure of the neural network. We have one convolution consisting of two layers, which is followed by two fully-connected layers. There is a single, scalar output.

We use tanh as nonlinearity in all layers, except the last one. There, the logistic function is used instead, normalising the value to be a probability. The functions applied in each hidden layer can thus be written down as follows:

$$f : \mathbb{R}^i \times \mathbb{R}^{j,i} \times \mathbb{R}^j \to \mathbb{R}^j,\ f(x, W, b) = \tanh(Wx + b)$$
$$f' : \mathbb{R}^i \times \mathbb{R}^{1,i} \times \mathbb{R} \to \mathbb{R},\ f'(x, W, b) = \frac{1}{1 + e^{-(Wx+b)}}$$

Here, x is a vector of inputs to the layer, W is a matrix of weights and b is a vector of biases.

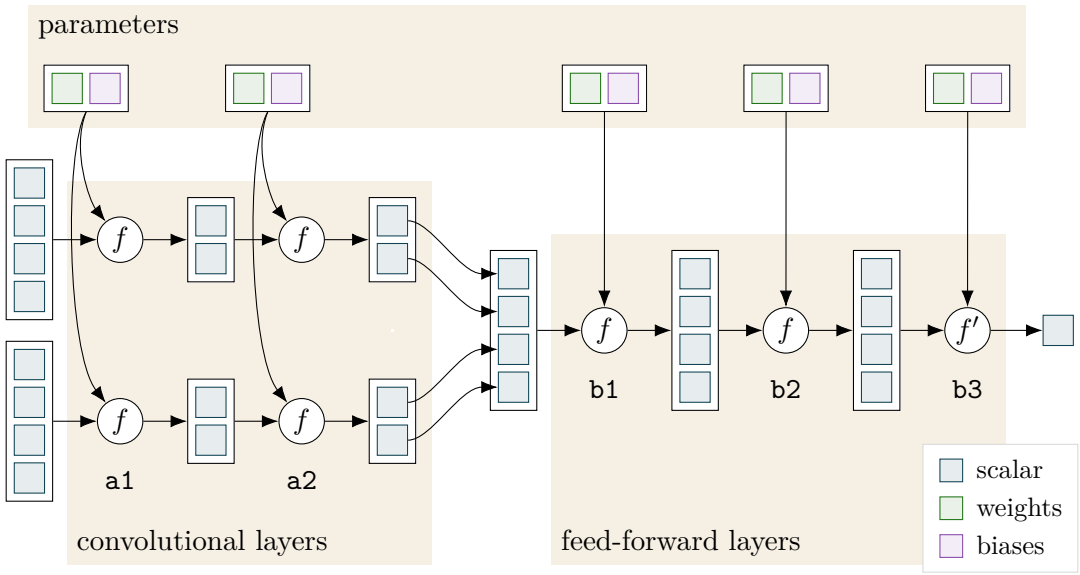


Figure 4: The layout of the neural network. Sizes of the different layers are not representative of the sizes used during training.

We train our network by minimising the mean cross-entropy loss of a batch of instances. So our cost function is defined as

$$E(p) := \frac{1}{N} H\left(\frac{y+1}{2}, \hat{y}\right)$$

Here N is the batch size, $y \in \{-1, 1\}^N$ the correct labels and $\hat{y} \in (0, 1)^N$ the output of the NN. p is the vector of parameters, so all weights and biases. H is the cross-entropy.

$$H(y, \hat{y}) := -(y \log \hat{y} + (1 - y)(1 - \log \hat{y}))$$

3.2. Training the NN

We split our `top100` dataset into both training and testing data, such that about 10% of instances are used for the latter. For cross-validation of our hyperparameter choices we use the `small` dataset. The datasets are described in section 5.2.

To initialise the weights we use random values drawn from the normal distribution $\mathcal{N}(0, \sigma^2)$, with $\sigma = 0.05$. The biases are set to 0.

We train the neural network using stochastic gradient descent [3]. We shuffle the instances in the training data and group them into batches. For each batch, we compute the average loss and update the parameters based on their gradient multiplied by the learning rate. After each epoch (a traversal of the whole training data) we re-shuffle the instances and start anew.

3.3. Additional techniques

In addition to the basic layout presented in the previous two sections, we apply additional techniques to improve training speed and reduce overfitting.

First, we employ a simple exponential learning rate decay. In the earlier stages of training, it is more beneficial to use a higher learning rate, to enable the network to improve quickly. Later on, however, it is more about fine-tuning the parameters, and a high learning rate prevents the NN from improving. While there are many different ways to choose an appropriate learning rate, we elected to focus on simplicity: After a set number of epochs, we halve the training rate.

Then, we have dropout [12]. Essentially, dropout means that in each training step a number of neurons are 'dropped' from the NN, i.e. their output is set to 0. The probability of a single neuron not being dropped is a fixed constant $d \in (0, 1]$. (A value of $d = 1$ would disable dropout.) Common values of d are in the range $[0.5, 1]$, with 0.5 being a robust choice across different contexts.

More precisely, we modify our function f in the following way ($\odot$ denotes component-wise vector multiplication):

$$f'(x, W, b) := \frac{1}{d}(f(x, W, b) \odot v)$$

where the components of $v \in \mathbb{R}^j$ are sampled independently from $\mathcal{B}(d)$ (the Bernoulli distribution with success probability d). This is done for all layers except the output layer.

Scaling inversely with d is done to preserve the expected output of each layer, which is relevant when evaluating the neural network. For testing and validation dropout is disabled by setting d to 1 and having all neurons participate.

The last technique is l_2 regularisation [7], which modifies the cost function by adding the squared l_2 norm of the vector of parameters, scaled to the number of parameters, to the cost function:

$$E'(p) := E(p) + \gamma |p|_2^2$$

γ is a constant that determines the strength of regularisation.

l_2 regularisation encourages the NN to have small weights, which allows it to better generalise from training data and prevents overfitting.

3.4. Hyperparameter optimisation

As usual, the behaviour of our NN is influenced by a number of hyperparameters, which are listed in table 2. To determine the optimal values of those, we employed random search. In each iteration, we sample a set of hyperparameters uniformly from manually chosen intervals, and then train a neural network for a set amount of wall-clock time.

The parameters concerning the size and count of the receptive fields determine how the inputs to the neural network are structured, and so are used when generating the training data. This means that changing them would require a full rebuild of the training data, and so it is not feasible for us to search them in this manner. As we observed

Table 2: Hyperparameters of the NN, as well as their values used for the final network and the range considered for the random search.

Name	Value	Range	Description
receptive field nodes	12	-	Number of nodes per receptive field
receptive field count	32	-	Number of receptive fields
batch size	170	[64, 256]	Number of instances per batch
learning rate	0.25	[0.03, 0.27]	Learning rate initially used for training
learning rate decay	70	[10, 120]	Number of epochs after which the learning rate is halved
layer `a1` size	55	[2, 128]	Size of the first convolutional layer
layer `a2` size	65	[2, 128]	Size of the second convolutional layer
layer `b1` size	120	[2, 128]	Size of the first fully-connected layer
layer `b2` size	120	[2, 128]	Size of the second fully-connected layer
dropout	0.7	[0.5, 1.0]	Probability of a neuron *not* being dropped during a training iteration
l_2 regularisation	0	[0, 50]	Factor controlling the strength of l_2 regularisation

a significant improvement by using a larger number of inputs when exploring different choices, we fixed them at their current values, which still allow us to fit the training data into main memory.

Some parameters have simple scaling dependencies amongst each. For example, dropout reduces the expected number of neurons that are active in any given layer during training. To compensate, we scale the sizes of those layers by $\frac{1}{d}$.

4. Results

We ran out calculations locally on a single GPU. The specifications of our system are reproduced in table 3.

Table 3: Specifications of our training environment.

CPU	Intel i5-6600K
GPU	Nvidia GeForce GTX 1080
OS	Ubuntu 17.10

Our set of training data consisted of 131072 instances, amounting to 2.3 GiB of data. The instances were split evenly between positive and negative instances, the latter of which are also split evenly between randomly generated ('random') and slightly modified real instances ('perturbed').

First, we determined a suitable range of values for each of the hyperparameters. They are shown in table 2. Then we searched for a good set of values by employing the random search described in the previous section. We ran each iteration for 1 minute which corresponds to about 60-90 epochs, depending on the size of the NN.

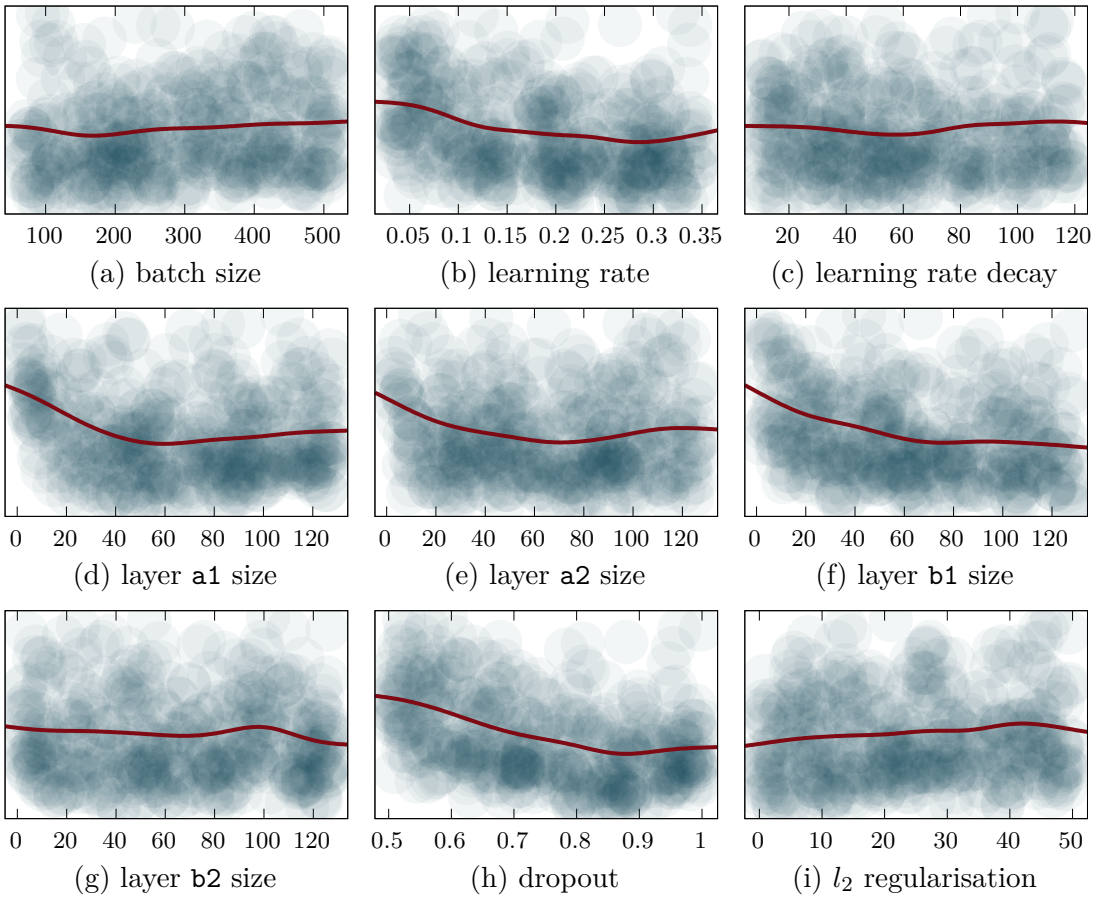


Figure 5: Results of the hyperparameter search, projected onto the different hyperparameters. Y-axis is mean squared error on testing data, lower values indicate better results. Sizes of the layers are before scaling them to the dropout value. The worst 3% of results are omitted from the scatter plot. (They are still included for the smoothed curve.)

The results of the random search are shown in figure 5. We note that most parameters did not influence the performance of our NN strongly and there is a large amount of noise.

- Higher learning rates seem to improve the performance, although this may be an artefact of the short time of each iteration.
- As higher values of learning rate decay essentially turn that functionality off (a single set of parameters runs for about 60-90 epochs), using learning rate decay has a small positive impact.
- We only see a strong effect of the size of the first three layers, `a1`, `a2` and `b1`.
- Dropout does not improve our performance much, at least in the time frame

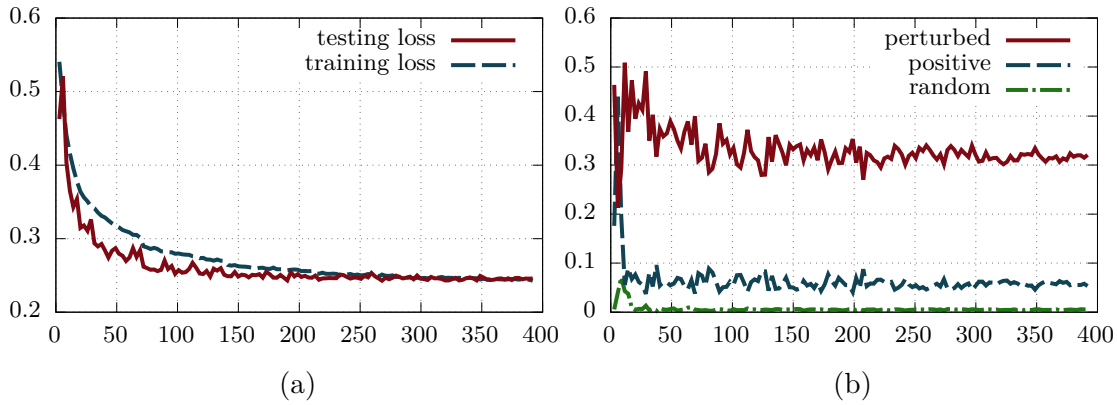


Figure 6: Performance of the NN during training. Time is given in epochs. (a) Cross-entropy loss on the training and test datasets. (b) Percentage of misclassified instances, split by type of instance.

measured by our random search. However, it is needed when training for longer periods of time.

- l_2 regularisation does not seem to provide a benefit.

Our final choice of hyperparameters is shown in table 2. To reduce overfitting, which occurs when training for longer periods of time, we set the dropout to 0.7, whereas the values for the other layers are more or less unchanged based on the search results.

During training we observe consistently better loss on the testing data than the training data (figure 6), likely due to using dropout. (Note that dropout is turned off during testing.) In particular, we do not see overfitting.

Table 4 shows the results of our final NN, both on the testing data and the separate `small` dataset. We can see that our neural network does quite well on positive instances, and has almost no errors on random instances. The perturbed instances are more difficult, with about a third being misclassified.

Curiously, we achieve better results on the `small` dataset than the testing data. This is a strong indicator that our network did not overfit, and the performance was not due to choosing the right hyperparameters for the dataset. To explain the improvement, we speculate that the repositories in the `small` dataset have less variance: The `top100` dataset incorporates the same number of repositories for each of the 49 most popular programming languages, in contrast to the `small` dataset that is biased towards more popular languages. Hence `top100` might include a wider range of repositories.

The fact that our NN correctly identifies our randomly generated instances without difficulty indicates that it does not rely on general statistical properties of the graphs, but rather considers their structure. Alternatively, it could mean that our random generation is not very good.

[3]Note that the random instances of both datasets were generated in exactly the same manner. Any differences are by chance alone.

Table 4: Results of the final NN. '`top100` test' refers to the part of the `top100` dataset that was used as test data. Precision is the fraction of classifications that were correct, recall is the fraction of positive instances that were correctly labelled as positive and F_1 score is their harmonic average.

Measure	`top100` test	`small`
number of instances	13090	131070
mean cross-entropy loss	0.2465	0.1888
false negatives	0.0590 (337/5714)	0.0236 (1324/56125)
false positives	0.1483 (1094/7376)	0.1330 (9968/74945)
misclassified positive	0.0509 (337/6619)	0.0200 (1324/66301)
misclassified random[3]	0.0059 (19/3200)	0.0036 (114/31904)
misclassified perturbed	0.3286 (1075/3271)	0.2998 (9854/32865)
precision	0.8517	0.8670
recall	0.9491	0.9800
F_1 score	0.8977	0.9201

5. Implementation details

Our implementation is split into two parts: `alarm` and `schaf`. The former is responsible only for downloading the repository metadata and organising it into files, while the latter generates the change-coupling graphs, the training data, and trains and runs the NN.

5.1. `alarm` – Application for Lazily Acquiring Repository Metadata

`alarm` is a script written in Python 3. It is organised as a single file, with 1091 LOC (excluding comments and blanks). We chose Python as it is easy to include libraries for various functionalities (e.g. sending HTTP requests, gzip compression) and downloading repositories is constrained mostly on network bandwidth. The source code is publicly available on GitHub[4].

It uses GitHub's API to search through their list of repositories. We implemented Git's smart HTTP protocol, which negotiates the files that have to be sent and organises the file transfer. The data consists of Git objects, so `alarm` contains functionality to parse this format, including the delta encoding used to compress similar files. This was all implemented in Python, with no use of existing Git libraries.

Git's metadata (as was described in section 2.1) is stored. The format is basically the same as the one used for the transfer of data (Git packfiles), with some small modifications to be able to start writing before having received all the objects. On disk, the repositories are grouped into large files, each containing about a hundred repositories. This facilitates compression, as multiple small repositories are compressed together.

We maintain an index of all repositories and their locations in the storage files. When

[4] `https://github.com/suyjuris/alarm`

searching for a repository we are able to skip ahead in the file. In particular, when downloading repositories from different sources we can safely ignore those that were already downloaded, and collect them when exporting into graphs.

The grouping of repositories on disk is thus mostly transparent. To be able to selectively export certain types of repositories each repository can be assigned a number of tags. Those are also stored in the index and can be modified without touching the actual stored data.

Originally, we tried to also generate the graphs in `alarm`, as the code to read Git's objects is already there. However, we quickly ran into performance problems on some of our larger graphs, both due to running out of memory and low throughput. So we moved that part over to `schaf` and just generate 'jobfiles' from `alarm`. Those describe precisely which repositories to include and in which files they are located.

5.2. Repository search

In this section we describe our criteria that determine which repositories are downloaded.

For the `top100` dataset we include the 49 most popular programming languages on GitHub, based on the number of active repositories[5] (see table 5). Then we query the GitHub API for the Top100 repositories for each language based on the number of stars. More specifically, we request a page of 100 results, sorted by stars.

The `small` dataset is a bit more involved. Here, we query for repositories in a range of sizes, so ones between 1000 and 10000 KiB. Those we sort by the number of stars. We query for pages each containing 100 results. When then page limit is reached, we filter to only include repositories below a certain number of stars, which is the value of the last result.

[5] `http://githut.info/`

Table 5: The programming languages used in our `top100` dataset.

1. JavaScript	14. Go	27. Puppet	40. Scheme
2. Java	15. Perl	28. Rust	41. Dart
3. Python	16. CoffeeScript	29. PowerShell	42. Common Lisp
4. CSS	17. TeX	30. Erlang	43. Julia
5. PHP	18. Swift	31. Visual Basic	44. F#
6. Ruby	19. Scala	32. Processing	45. Elixir
7. C++	20. Emacs Lisp	33. Assembly	46. FORTRAN
8. C	21. Haskell	34. TypeScript	47. Haxe
9. Shell	22. Lua	35. XSLT	48. Racket
10. C#	23. Clojure	36. ActionScript	49. Logos
11. Objective-C	24. Matlab	37. ASP	
12. R	25. Arduino	38. OCaml	
13. VimL	26. Groovy	39. D	

Some repositories are empty, or not available for other reasons, so we ignore them. Also, repositories which yield graphs that are too large to store in memory are skipped, although that only happens later during graph generation.

5.3. `schaf` – Statistical Commit History Analysis Framework

`schaf` is our main application, responsible for the generation of graphs and training data, as well as training and executing the neural network. It is written in C++, mainly for reasons of performance and the benefits of static type checking. The source code is publicly available on GitHub[6].

As shown in table 6, the main parts of our program are general purpose utilities and data structures. However, much of those has been taken from previous projects and was only adapted for use in `schaf`.

Development was done on Ubuntu Linux 17.10[7], using `g++` 7.2.0 as compiler[8]. Additional tools include `gdb`[9] for debugging, the sampling profiler of `gperftools`[10] and GNU Emacs [11] for actually writing the code.

In terms of libraries, we use LZ4[12] and zlib[13] for compression, xxHash[14] for hashing, and Steve Reid's SHA-1 implementation[15] when parsing Git objects. We use the TensorFlow framework for our neural network [1]. TensorFlow and zlib are the only external dependencies.

We will now give an overview of `schaf`, beginning with the general structure of the application. Then, we describe in detail the three main modes of operation: Generating change-coupling graphs from repository metadata, creating training data for the NN, and training and running the NN. Finally, we close with a discussion of techniques used for debugging and visualisation during development.

5.4. General structure of `schaf`

Our style of programming is focused on simplicity and performance. Consequently, we do not use many of the features of C++, such as exceptions or inheritance hierarchies. Instead, we employ a data-oriented view, abstracting functionality based on the data processing that is done, whereas the representation of data is not hidden. This is reflected in our architecture.

Our first important abstraction is the reader for the Git objects representing the metadata of repositories which `alarm` outputs. It takes a file as input and produces a stream of metadata, which is a list of Git objects.

[6] `https://github.com/suyjuris/schaf`
[7] `https://www.ubuntu.com/`
[8] `https://gcc.gnu.org/`
[9] `https://www.gnu.org/software/gdb/`
[10] `https://github.com/gperftools/gperftools`
[11] `https://www.gnu.org/software/emacs/`
[12] `http://www.lz4.org/`
[13] `https://zlib.net/`
[14] `http://www.xxhash.com/`
[15] `https://github.com/clibs/sha1`

Table 6: Lines of code for different parts of `schaf`. All counts exclude comments and blanks.

Description	LOC
General utilities and debugging	1638
Data structures and memory management	1369
Neural network and training data creation	1290
Generation of change-coupling graphs	925
Git object parsing	526
Command-line interface	481
Operating system interaction	278
Total	6507

The component responsible for graphs then takes that stream of data and writes the graphs into a file. Reading the graphs from a file is another abstraction, once again taking a file as input, and giving the graphs as output. Additionally, it supports the generation of random graphs.

This is used by the component responsible for the NN to create the training data and, more importantly, to provide an abstraction for the neural network. This includes low-level operations on the state of a single network (e.g. processing a single batch in training, or saving the parameters to a file), and high-level functions mapping directly to the user interface (which is a simple command-line interface).

We do not need to interact much with the operating system, apart from querying wall-clock time and retrieving stack traces when crashing. The specifics are handled by our platform layer, but it is only a few hundred lines.

5.5. Generating change-coupling graphs

Algorithm 1 shows the general process of generating a change-coupling graph. To be able to process large graphs in a reasonable time frame, we applied several optimisations.

First, all the file paths in the graph are represented using 64-bit hashes[16]. For paths we only need the operations of appending elements and of comparing for equality, both of which can easily be implemented on an integer value with only a vanishing probability of falsely comparing two different paths as equal. (A repository with 10^6 files, which is ten times as large as the largest one in our dataset, would have a probability of 10^{-8} of two files colliding for a random hash function.)

We keep the contents of all directories sorted, so we can determine the set of changes by a recursive algorithm, iterating over the two directories simultaneously.

To build the graph we use a hash table, more specifically the sparsehash library originally developed at Google[17]. We use their `dense_hash_map`, which is implemented

[16] We use specialisations of `xxHash`.

[17] `https://github.com/sparsehash/sparsehash`

Algorithm 1 Generating a change-coupling graph

Input set of commits
Output change-coupling graph
Initialise empty graph $G = (V, E)$, with weights $c : E \to \mathbb{N}$, $c = 0$
for all $c \in$ *commits* **do**
  changed := files changed in c
  $V := V \cup$ *changed*
  for all $v, w \in$ *changed*, $v < w$ **do**
    $c(v, w) := c(v, w) + 1$
  end for
end for
return (G, c)

using an open addressing scheme and quadratic probing.

We map the 64-bit hashes of files to indices so that our nodes are always a set $\{0, ..., |V| - 1\}$. As 32-bit are sufficient to represent those (again, our largest graph has only 10^5 nodes), edges are just 64-bit integers made up of two nodes. In particular, this makes our innermost loop quite simple:

```
std::sort(changed_nodes.begin(), changed_nodes.end());
for (int i = 0; i < changed_nodes.size(); ++i) {
    uint64_t node_i = changed_nodes[i] << 32;
    for (int j = 0; j < i; ++j) {
        uint64_t node_j = changed_nodes[j];
        edge_weights[node_i | node_j] += 1;
    }
}
```

Apart from reading the data (which is limited by the speed of `gzip` decompression), this loop dominates the execution time, especially on repositories with large commits. We also gained significant speed-ups by choosing a custom hash-function for the hash table, which exploits the special structure of the edges and can be evaluated cheaply[18].

The important part goes as follows:

```
val = (val ^ (val >> 30)) * 0xbf58476d1ce4e5b9ull;
val = val ^ (val >> 31);
```

As some graphs are too large to keep in memory (change-coupling graphs use memory quadratic w.r.t. the commit size), we skip graphs having more than $5 \cdot 10^7$ edges.

After having assembled the edges in a hash table, we pack the graph into a more compact format which is suitable for serialisation. The format is a simple binary one, which stores the adjacency lists of all nodes contiguously. For each node there is an offset into this array. An example is shown in figure 7.

[18] It is based on `splitmix64`, see `http://xorshift.di.unimi.it/splitmix64.c`.

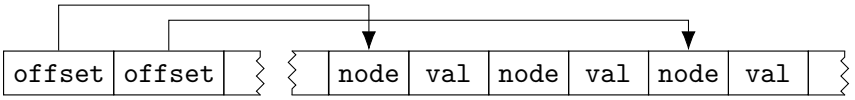


Figure 7: Example illustrating the layout used for storing our graphs. Here, the first node has two adjacent edges, which are stored beginning at its offset, and until the offset of the following node.

Algorithm 2 Converting from a hash table of edges into our graph storage format

Input Hashtable $H : E \to \mathbb{N}$
Output Graph in storage format
A, *edges* := arrays of size $2|E|$
for all $((v, w), c) \in H$, with $i = 1, ..., |E|$ **do**
  $A[2i] := (v, w, c)$
  $A[2i + 1] := (w, v, c)$
end for
Sort A lexicographically
for all $(v, w, c) \in A$, with $i = 1, ..., |A|$ **do**
  $edges[i] = (w, c)$
end for
offsets := array of size $|V|$
for all $i = 1, ..., |V|$ **do**
  $offsets[i] = \min\{j = 1, ..., |A| : A[j] = (i, *, *)\}$
end for
return (*offsets*, *edges*)

Algorithm 2 gives a high-level overview of how we convert our hash table of edges into this storage format. Crucially, it requires only iterating over the hash table once, reading its contents in arbitrary order. This can be done very efficiently, due to the hash table being implemented using an open-addressing scheme.

Sorting the array is the bottleneck here, so we implemented a specialised LSB radix sort. This fits well to our problem, as the array's elements consist of only two 32-bit integers and one 32-bit weight that does not influence the sorting. (So we are basically sorting 64-bit integers.) Consequently, our radix sort outperforms `std::sort` by a large margin.

Our radix sort considers 64-bit integers, using the bytes as digits. As we know that each edge is included twice, we only need to count the first four bytes, since the counts for the next four will be the same. After counting, there are 8 passes over the data, one for each byte, which copy the data between two arrays.

When the data is sorted, algorithm 2 copies it once again, omitting the first edge. It also sets the offsets. In our actual implementation, both of these steps are integrated into the last pass of our radix sort at no cost.

As we know the number of nodes beforehand, we implement an additional optimisation

for the (extremely) common case of $|V| < 2^{16}$. Here, each node only needs two of its four bytes, so we can safely skip half of our LSB sorting passes.

Finally, the graph has to be stored. We apply LZ4 compression, as it is very fast both in encoding and decoding, at the cost of compression ratio. We compress the graph in memory and then write the bytes to disk. (Our in-memory representation is chosen carefully so that it can be serialised simply by copying the bytes.)

To generate random instances for the NN we use the same process and just generate the set of commits used as input to algorithm 1. Those graphs are not stored on disk, but rather generated on-the-fly when creating training data.

5.6. Creating training data for the neural network

Creating training data involves three types of instances: positive (unchanged real data), perturbed (real data with slightly modified edge weights) and random (randomly generated graphs with similar statistical properties). They are all generated in a similar manner.

As was described in section 2.3, we need n neighbourhoods of size m. Algorithm 3 gives the general structure of our implementation. Note that we always have $f(x) = (x-1)^2$.

When implementing algorithm 3 we use a hash table to efficiently represent the non-zero entries of γ. To speed up checking whether a node is not in X, we use a bloom filter with parameter $k = 1$: We have an array of $l = 8192$ bits, and a hash function $h : V \to \{1, ..., l\}$. Whenever we add a node v to X, we set the bit $h(v)$ to 1. So if for a node $w \in V$ the bit $h(w)$ is *not* set, we know that it is not in X and can skip the more expensive check of iterating through X.

Algorithm 3 Creating a local neighbourhood from a graph

Input Graph $G = (V, E)$ with weights $c : E \to \mathbb{R}_{\geq 0}$
Input Weighting function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$
Output Set of nodes of size m
Initialise node weights $\gamma : V \to \mathbb{R}_{\geq 0}$, $\gamma = 0$
Choose node $v \in V$ at random
$X := \{v\}$
while $|X| < n, \delta^+(X) \neq \emptyset$ **do**
    for all $w \in \delta^+(v) \setminus X$ **do**
        $\gamma(w) := \gamma(w) + c(v, w)$
    end for
    Choose v' from $V \setminus X$ at random, weighted by $f \circ \gamma$
    $X := X \cup \{v'\}$
    $v := v'$
end while
return X

For the random instances we generate a graph as described in the previous section, so no modifications are required. When creating perturbed instances, we map our edge weights c using the following function:

```
uint32_t perturb(uint64_t v, uint64_t w, uint32_t weight) {
    uint64_t edge = std::min(v, w) << 32 | std::max(v, w);
    edge = (edge ^ (edge >> 30)) * 0xbf58476d1ce4e5b9ull;
    edge = (edge ^ (edge >> 27)) * 0x94d049bb133111ebull;
    edge = edge ^ (edge >> 31);
    if ((edge & 0xff) < 16) {
        return weight + __builtin_clz((edge >> 8) | 1);
    } else {
        return weight;
    }
}
```

This uses a hash function[19] to generate a random value based on the edge (v, w). Then, we check whether the last byte is in the range $[0, 15]$, which is true with probability $\frac{1}{16}$. `__builtin_clz` counts the number of leading zeros in the binary representation of a non-zero number, it is provided by `g++`. We know the first 8 bits of its argument are zero, and after that there each bit is 0 with probability $\frac{1}{2}$.

5.7. Training the neural network

To perform the calculations necessary for the NN we use TensorFlow [1]. Note that we are not using TensorFlow's Python API, but the less commonly used C++ API. Some functionality (such as gradients of some specific functions) is not yet available from C++, and some convenience features of Python do not exist, but it is certainly usable.

TensorFlow works as follows: The application specifies a graph representing computations on tensors. There are some conveniences provided, such as special functions like the `SoftmaxCrossEntropyWithLogits` function and automatic generation of gradients. Others we have to build by hand, for example dropout. The gradient generation of some functions was not yet implemented in the C++ API, so we had to add them manually in our code.

As linking C++ libraries built in other compilers is immensely difficult unless one is very careful about the types of features that are used, we have to link against a locally compiled TensorFlow.

Our set of training data fits comfortably into main memory (it is about 2 GiB large). As our model is quite simple, copying the data around has to be done efficiently to not become a bottleneck.

There are three main operations on our dataset that we have to support: Grouping the instances into batches of a certain size, shuffling all instances, and reading the data of one particular batch.

[19] Once again `splitmix64`.

We combine the first and second of these into a shuffle operation that reads the data from one location and shuffles it into another. Note that grouping the instances into batches is not free, as the features of all instances in a batch have to be contiguous in memory. (The same for their labels.)

Our shuffle is not in place, which enables us to run it concurrently when training the network. After both the epoch and the shuffle are finished, the original training data and the temporary copy containing the result of the shuffle are swapped.

We have to provide TensorFlow with the data for each batch. Originally we were using `std::memcpy`, but that turned out to be a bottleneck during training. Instead, we now initialise the tensors directly from our memory, by providing a custom allocator that simply returns a pointer to the appropriate memory region. This requires us to keep our data aligned, which is also taken care of by our shuffle.

5.8. Debugging and visualisation

The parts of our application that execute under our control (in particular all of generating change-coupling graphs and training data) were developed using traditional `printf`-style debugging and `gdb` in rare cases of memory corruption. Our program is mostly deterministic, so we did not have any problems in reproducing errors. Also, our debug builds make use of bounds-checking, which further reduces the chances of invalid memory accesses.

When optimising procedures, is it often helpful to look at the distribution of certain values (e.g. the length of a list to be sorted) when deciding which algorithms to use. We implemented the P^2 algorithm for generating quantiles of arbitrary data without storing the values [5]. This enables us to view (approximate) distributions of arbitrary values efficiently.

Debugging large calculations is difficult because there is a lot of data involved. This is especially true when using a framework such as TensorFlow, which makes it difficult to access intermediate results of calculations. Luckily, we did not encounter any bugs in TensorFlow itself.

The TensorBoard tool, which is part of TensorFlow, was very helpful both in debugging problems with our NN and in understanding its performance. It shows graphs of scalar values and histograms of tensors that we output during training. For example, we can see how the outputs of our NN are distributed over time. Additionally, it shows the graph of computations that we provide, which helps to make sure that our program is working correctly.

We also use `gnuplot`[20] to plot distributions, sometimes combined with Python scripts to preprocess the data.

[20] `http://www.gnuplot.info`

6. Discussion and future work

We feel that there are many interesting possibilities that we could not consider based on the limited scope of this work.

First, the definition of change-coupling graphs could be modified to better reflect real-world development practices. For example, modifying the edge weight increase to be dependent on the commit size (i.e. having the total increase be constant) could help to differentiate between important and unimportant relations. For example, a commit that changes the line endings for all files in the repository would currently contribute significantly to the total edge weights, but it does not indicate an interesting relationship. Also, commits above a certain size could be ignored, to improve performance.

When creating the instances for the training data, we choose roots for the local neighbourhoods randomly. Maybe it would improve the performance of the NN to incorporate more of the structure of the graph into this choice.

Our NN has no problems telling apart random from real instances. While this does indicate that it does not rely on simple statistical heuristics, it could also mean that our random instances are missing important surface-level features, and that an improved generation would force the NN to learn more about the underlying structure.

Additionally, our perturbation is very basic. More varied changes (e.g. removing edges, adjusting the weight multiplicatively, or adding perturbation based on generated commits) could also improve the structural understanding of the NN.

Related to this, it could be interesting to gain insight into the way the NN operates. While this is notoriously difficult in general, simple visualisations such as marking influential areas of a graph are quite possible.

The layout of our network is quite fixed; while we are able to vary the sizes of layers we cannot programmatically add new ones, which would make it possible to find better layouts more easily.

In a similar vein, we did not consider different nonlinearities to use, and there is, of course, a multitude of variations on how exactly to train neural networks that may improve the performance.

An interesting possibility regarding different NN structures would be to train the network to recover corrupted input data. The output of the network would then have the same structure as the input, and the difference in values could quantify how corrupted—or 'unrealistic'—the inputs are. Additionally, one would immediately have a measure of 'unrealisticness' for different areas of the graph and suggestions on how to improve them. However, we think that this would require more effort in choosing the input representation, such that the network is able to make structural changes when constructing its output.

7. Conclusion

We implemented an application for classifying change-coupling graphs based on real repositories to estimate how 'realistic' such a graph is. This includes both the gathering of metadata from real repositories, generating the respective change-coupling graphs,

creating instances of training data for the NN, and training and running the NN.

We managed to achieve good results on our dataset, and are able to correctly classify real and random instances with high confidence.

Our chosen environment of C++ as a programming language, together with TensorFlow for computing the NN, was very suitable for our task. We were able to optimise our program in the necessary areas to get to a good level of performance.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, Jens Grabowski, Verena Herbold, Daniel Honsel, Stephan Waack, and Marlon Welter. Agent-based simulation for software development processes. In *Multi-Agent Systems and Agreement Technologies*, pages 333–340. Springer, 2016.

[3] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[4] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[5] Raj Jain and Imrich Chlamtac. The p 2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Communications of the ACM*, 28(10):1076–1085, 1985.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[7] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

[8] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023, 2016.

[9] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[10] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[11] David Silver, T. Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. 2017.

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[13] S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242, 2010.

[14] Izhar Wallach, Michael Dzamba, and Abraham Heifets. Atomnet: A deep convolutional neural network for bioactivity prediction in structure-based drug discovery. 10 2015.

[15] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *SIGKDD Explorations*, 5:59–68, 2003.

A. German Summary / Deutsche Zusammenfassung

Machine Learning mit neuronalen Netzen hat in letzter Zeit viele beeindruckende Erfolge erzielen können, u.a. in visueller Mustererkennung, Verarbeitung natürlicher Sprache und künstlicher Intelligenz. Infolgedessen gibt es Bemühungen, neuronale Netze auf andere Daten, z.B. Graphen, anzuwenden.

In dieser Arbeit beschäftige ich mich mit einem Klassifizierungsproblem auf Graphen und implementiere eine Lösung mithilfe eines Convolutional Neural Networks. Es geht speziell um eine bestimmte Klasse von Graphen, Change-Coupling Graphen, die die Abhängigkeiten zwischen Dateien in dem Repository eines Softwareprojektes darstellen.

Das Problem besteht daraus einzuschätzen, wie 'realistisch' ein bestimmter Change-Coupling Graph ist. Hierfür generiere ich Graphen basierend auf einem Datensatz an realen Repositories und füge sowohl veränderte Versionen von echten Graphen als auch zufällig generierte Graphen hinzu. Mein Klassifizierer wird dann darauf trainiert, diese von den echten Graphen zu unterscheiden.

Diese Arbeit ist motiviert durch das SimSE Projekt, das u. a. versucht die Entwicklung eines Softwareprojektes über eine agentenbasierte Simulation vorherzusagen. Hierbei werden Change-Coupling Graphen generiert, die möglichst realistisch sein sollen, damit die Simulation aussagekräftig ist.

Meine Arbeit soll messen, inwieweit dieses Ziel erreicht wurde. Dabei ist aber anzumerken, dass die Entwicklung meines Programms komplett unabhängig zum SimSE Projekt verlief, und dass ich zu keinem Zeitpunkt Daten, die von der Simulation generiert wurden, verwendete.

Um den Klassifizierer zu trainieren, habe ich einen Datensatz erstellt, basierend auf den Metadaten echter Repositories. Dafür wurde eine Anwendung in Python entwickelt, die das Git-Protokoll implementiert und darüber Metadaten in ein kompaktes Format speichert.

Aus diesen werden anschließend die Change-Coupling Graphen generiert, in einem anderen Programm, das ich in C++ implementiert habe. Dieses ist außerdem zuständig für das Erstellen der Trainingsdaten, also insbesondere das Abwandeln von bestehenden Graphen und das Generieren von neuen Graphen, welche als Negativbeispiele verwendet werden.

Dasselbe Programm ist für das Trainieren und Ausführen des neuronalen Netzes zuständig. Hierfür werden übliche Techniken verwendet (wie etwa Stochastic Gradient Descent, Dropout, l_2 Regularisation und Learning Rate Decay).

Die Güte des erstellten neuronalen Netzes wurde evaluiert, sowohl anhand der Validierungsdaten als auch anhand eines zweiten, unabhängigen Datensatzes. Mein Klassifizierer hat eine Fehlerrate von 5,09% (resp. 2,00%) auf realen Instanzen, 32,86% (resp. 29,98%) auf abgewandelten Instanzen und 0,38% auf zufällig generierten Instanzen.