GitGraph - Architecture Search Space Creation through Frequent Computational Subgraph Mining

Kamil Bennani-Smires
Artificial intelligence and machine learning group, Swisscom
kamil.bennani-smires@swisscom.com

Claudiu Musat
Artificial intelligence and machine learning group, Swisscom
claudiu.musat@swisscom.com

Andreea Hossmann
Artificial intelligence and machine learning group, Swisscom
andreea.hossmann@swisscom.com

Michael Baeriswyl
Artificial intelligence and machine learning group, Swisscom
michael.baeriswyl@swisscom.com

ABSTRACT

The dramatic success of deep neural networks across multiple application areas often relies on experts painstakingly designing a network architecture specific to each task. To simplify this process and make it more accessible, an emerging research effort seeks to automate the design of neural network architectures, using e.g. evolutionary algorithms or reinforcement learning or simple search in a constrained space of neural modules.

Considering the typical size of the search space (e.g. ~10^{10} candidates for a 10-layer network) and the cost of evaluating a single candidate, current architecture search methods are very restricted. They either rely on static pre-built modules to be recombined for the task at hand, or they define a static hand-crafted framework within which they can generate new architectures from the simplest possible operations.

In this paper, we relax these restrictions, by capitalizing on the collective wisdom contained in the plethora of neural networks published in online code repositories. Concretely, we (a) extract and publish GitGraph, a corpus of neural architectures and their descriptions; (b) we create problem-specific neural architecture search spaces, implemented as a textual search mechanism over GitGraph; (c) we propose a method of identifying ***unique*** common subgraphs within the architectures solving each problem (e.g., image processing, reinforcement learning), that can then serve as modules in the newly created problem specific neural search space.

KEYWORDS

Architecture search, neuroevolution, corpus creation

1 INTRODUCTION

The work of a deep learning practitioner is more repetitive than we care to admit. A lot of the automation drive has focused on reducing the amount of repetitive work office workers do. Typically, however, this has not included AI researchers or data scientists. We automate the repetitive work of the customer support agent [4] who gives the same advice to hundreds of customers in need of a password reset. In a similar fashion, we should help the deep learning practitioner to avoid designing the same type of neural architecture every time they need to turn one type of sequence into another. While this is now a laborious manual process, attempts to automatically build new architectures from basic components are increasingly gaining traction.

Current automated neural architecture creation strategies rely on extensive expert knowledge and heavy handed supervision. They either use predefined modules [7] and the novelty lies in the recombination or they create new modules but within a very tightly controled structure [14, 18]. The reason for this heavy-handed supervision is that each step taken towards a better architecture is costly. This constraint is independent of the search method used. Whether it's employing reinforcement learning [2, 18] or evolutionary algorithms [14], for each change the system must evaluate candidates and each evaluation means training a full network on a usually complex task. The smaller the changes, the more candidates need to be evaluated. The space of possible options is too large to allow searching or evolving a full architecture from basic building blocks like matrix additions or multiplications. Shortcuts are thus necessary.

Neural evolution can be seen as a combination of two problems - defining a neural module search space and creating a policy to create that space. The question of finding the right policy has received almost all the community's attention [7, 14, 18], with the search space receiving almost none. A notable exception is a recent work on [10] that explicitly states that different domains require different operators, that are subsequently combined to form neural architectures.

We propose constructing the search space by using the known architectures for similar tasks. Expert supervision can guide the search and lower the network creation cost. In our view, however, this supervision need not be a laborious task linked to the task at hand. Instead, it can come from repositories of computation graphs that have been published for the tasks similar to this one before. This removes the need for handcrafted constraints built into the evolution task itself. It also leverages a previously unused resource - the network structures published by previous researchers.

As shown in figure 1, we split the task of search space definition into three parts:

- Search for architectures that solve similar problems. This step yields a collection of graphs.
- Common Subgraph Mining. Extract the neural modules and combinations of modules that are common between the found architectures. As shown in the corresponding task

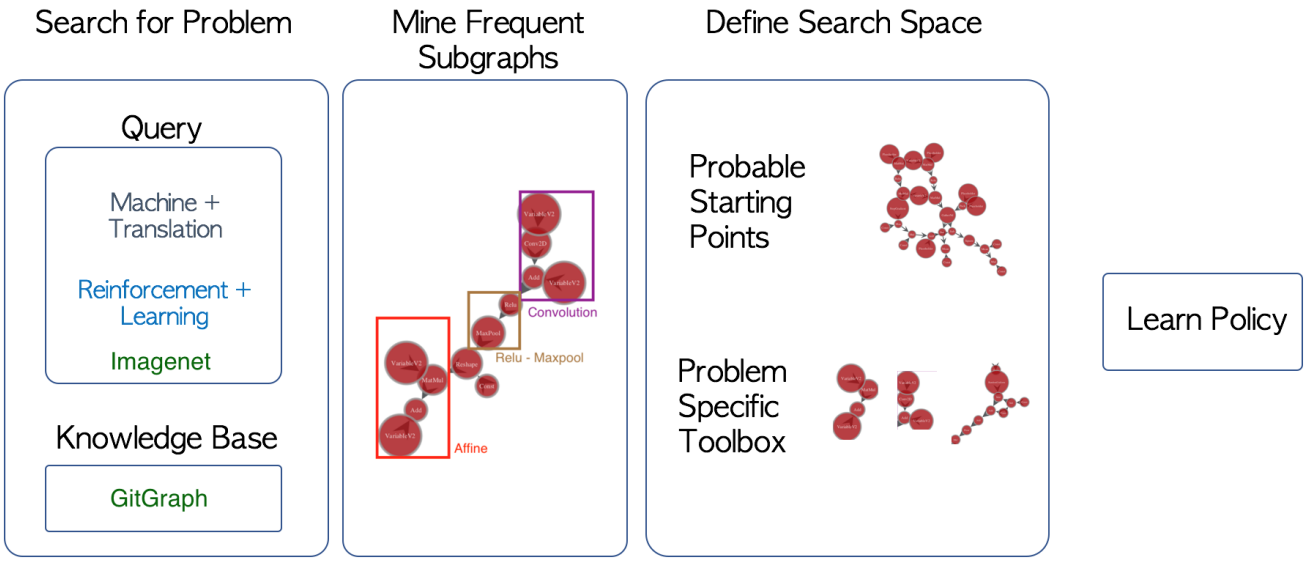


Figure 1: Automated Architecture Search Space Definition

in Figure 1, we can directly mine combinations like *convolution + Max pool + affine*.

- Defining the Search Space by specifying which modules are large, frequent and unique enough to be useful. These subgraphs then become a toolbox of task-oriented modules.

The resulting task specific neural module toolbox becomes the starting point to evolve new architectures.

A great **example**, present in B, regards the common task of treating images inputs. We found that there is a subgraph (also shown in Figure 8b) that appears at least once in 30% or more of the distinct computational graphs for image-related tasks. This subgraph is actually a combination of known modules. It is a convolution operation followed by a Relu activation and Maxpooling, the result is reshaped and eventually an affine transformation is performed (weight multiplication and bias addition). This chain of operations is "typical" when dealing with images.

The advantage of finding the computational graphs and the common components for a task is twofold. Graphs that resemble many others for the same task have a high likelihood of being a good starting point in searching for a new architecture. In addition, common components can be transformed into modules that can be used as changes when searching for an architecture. We thus go into a supervised learning approach, where we use the knowledge of humans who manually devised the architectures previously to create new ones.

In this paper we introduce GitGraph [1] - a dataset of TensorFlow [1] computational graphs, alongside with the description of tasks they are useful for. In addition to publishing the dataset, we make three contributions:

- We show that GitGraph can be used to search for a problem and define the neural search space. We show that there are enough distinct computational graphs for common problems to make connections between them and extract common components.
- We propose a method to identify frequent neural subgraphs.
- We show that by using common subgraphs as modules, we can reduce the complexity of the resulting architectures by up to 70%.

[1] https://www.mycloud.ch/s/S00E8129370EFE75830040072AD8203611E4F9971E1

2 RELATED WORK

2.1 Architecture Search

A standard way of searching for a neural architecture is to build one from nothing.[18] use reinforcement learning build convolution stacks for image-related problems and recurrent networks for text-related ones. In [18] , they encoded the architecture of a neural network as a string and used an RNN (the controller) to sample architectures. This RNN is trained using REINFORCE [15] in order to maximize the expected reward (accuracy on validation dataset) of the architectures generated. Their method was applied in order to generate a CNN architecture and a reccurent cell by creating by hand one appropriate search space for each task. A known problem is the cost of the search, determined mostly by the number of candidates that need to be evaluated. [18] limit the number of possibilities by imposing a rigid structure (e.g. a stack of convolutions). In addition, they do not fully train the candidate networks. With these constraints, they showed that state of the art results could be obtained with learnt networks, but with a computational cost 10.000 times higher than training a single network.

MetaQNN [2] also generates CNN architectures based on reinforcement learning. The layer selection process is modeled as a Markov Decision Process MDP , where each is state is defined as tuple of relevant layer parameters such as the type of the layer (Convolution , Pooling , Fully connected, Global Average Pooling,

SoftMax) associated with some parameters dependant on the type (e.g. number of neurons in a fully connected layer, stride for convolution), the optimal architecture is found using Q-learning with an epsilon-greedy strategy. Finally they sampled from the optimal policy which is not deterministic (stopping at epsilon=0.1) 5 models and ensemble them to make the predictions. [2] rely on the search space when doing comparison with other networks : better performance on networks that used only the same components as they have in their search space.

[14], following in the footsteps of [9], replace reinforcement learning with evolution strategies. The latter are shown to be a viable alternative to gradient descent in neural architecture creation. Neuroevolution passes from learning weights of a predefined architecture to learning the links between modules and the weights attached to them jointly. However, it still relies on the existence of modules that can be joined. The reinforcement or genetic evolution of architectures have the advantage of coverage - if there is an architecture that is superior for a task but unbeknown to the research community at this point, there is hope that it will be found.

A distinct advantage of evolving using borrowed modules is their improbable nature. As shown by NEAT [13], there is a tendency for new architectures to go extinct from an evolving population before they could realize their potential. We believe this effect can be countered by adding whole blocks to the net, instead of their individual components. In addition, neuroevolution has moved from direct to indirect gene encodings, because of the growing size of the networks. As shown in [12], the number of genes can be much lower than the number of connections and neurons in the brain. A compositional pattern producing network (CPPN) compresses a pattern with regularities and symmetries into a relatively small set of genes. By constructing a toolbox of task-oriented neural modules, we also dissociate the number of mutations needed from the number of actual connections in the resulting ANN.

Instead of building networks from basic building blocks like additions and multiplications, the Neural Architect [7] searches a space large, prebuilt modules to achieve the same outcome. The downside of this approach is that the toolbox is universal, with the same Relu, conv or affine layers used irrespective of the task. The addition of a module that is useful to the task at hand can only be done manually. Moreover, a data scientist or researcher's knowledge is needed to define which modules are useful and how they can be combined. The Neural Architect is much faster than the methods above, because of trying a small number of candidates to get to a competitive final result.

A work that prioritizes the search space over the search policy is the recently introduced search [10] based on domain specific languages. Like the Neural Architect, Salesforce [10] introduced a novel way of searching for neural architectures relying on human supervision. The main improvement of their approach is the creation of a domain specific language (DSL). This contrasts strongly to the standard toolbox that the Neural Architect uses and allows the creation of architectures with modules that are especially created for that task. They study the case of recurrent nets and show that a very simple domain specific search space leads to good architectures. For instance, when defining a DSL for recurrent networks, it will contain 4 unary operators, 2 binary operators, and a single ternary operator. This very specific choice is given by the researchers' prior knowledge of the structure of GRUs [3] and LSTMs [5]. The intuition is that, once the search space is created, finding the right search policy is feasible - the right quantity and combination of the modules can be found with ease. We thus focus on the creation of the initial search space, with the goal of automating this initial step that is, for now, a human prerogative.

2.2 Frequent Subgraph Mining

To find the frequent neural modules for a chosen problem, we employ methods typically used for subgraph mining. Given a graph dataset $D = G_0, G_1, \text{fi}.G_n$, $support(g)$ denotes the number of graphs in D in which g is a subgraph. The problem of frequent subgraph mining is to find any subgraph g s.t. $support(g) >= minSup$.

A well-known method is graph based Substructure pattern mining, gSpan [16]. It finds all the frequent subgraphs without candidate generation and false positive pruning. Relying on Depth-First search , it introduces two novel techniques DFS lexical order and minimum DFS Code which makes the frequent subgraph mining task solved efficiently.

Compared to gSpan [16], CloseGraph [17] aims to mine **closed** frequent subgraphs. A graph g is closed in a database if there exists no proper super graph of g that has the same support as g. Mining closed subgraph helps to get rid off redudant subgraphs. The search space is pruned by introducing two novel concepts: equivalent occurence and early termination on top of gSpanś concepts. This method is also highly effective it has been shown by the authors that it outperforms gSpan [16] by a factor of 4 to 10 when the frequent subgraphs are large.

2.3 Code Corpora

We apply the subgraph mining methods on a corpus of computational graphs published on GitHub [2] - GitGraph. While this is the first corpus of its kind, GitGraph is conceptually similar to existing code corpora from two points of view. Firstly - just like code corpora, we are scraping a corpus from a publicly available resource and thus we have to verify its properties like the amount of duplicate contents. Secondly, the graphs themselves are derived from Tensorflow code and thus, if a generic compiler becomes available, a computational graph corpus could be obtained from the original Tensorflow code. This would reduce the graph corpus building to a code corpus building problem.

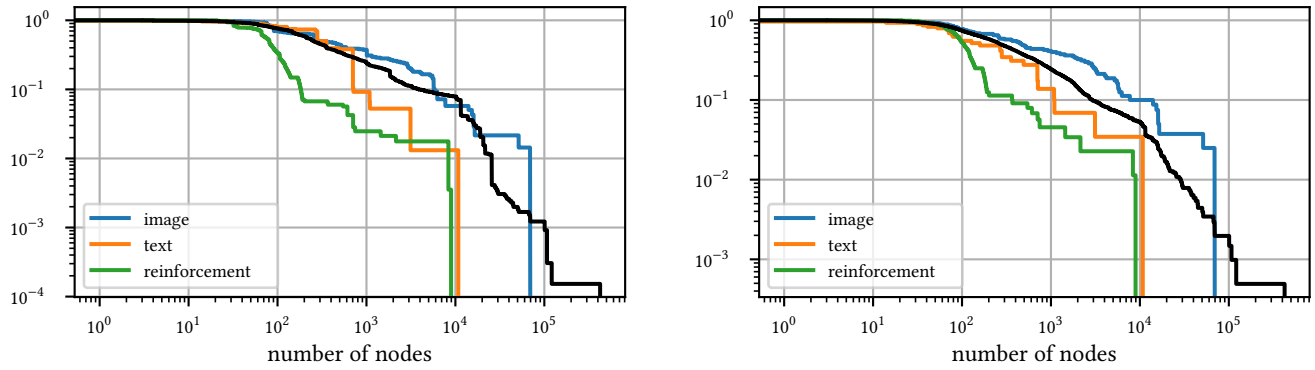
We have a keen interest in spotting duplicated graphs, as this may affect the statistics extracted about the frequency of subgraphs. In [11] they show that a large amount of code is duplicated on several projects coming from the same open-source software eco-system. The importance of this phenoma yield to different techniques that had been developed to detect code duplication, These methods can be token-based such as [8] or rely on abstract syntax trees [6] or hash-based [11].

3 THE GITGRAPH CORPUS

3.1 Corpus Creation

The first contribution of this paper is the creation of a searchable database of computational graphs along with a description of the

[2] www.github.com



(a) With duplicated graphs

(b) Without duplicated graphs

Figure 2: Distribution of nodes in the graphs per task

problem they solve. We checked out Github repositories of neural networks written in Tensorflow [3]. An observation led to a quick progress in the corpus creation - creating automated compiling routines is not necessary in a first instance. We can instead obtain the computational graphs from the checkpoints *.ckpt files* that the authors did not include in their .gitignore . We then stored the graphs, alongside with the descriptions in the *readme* files and the github descriptions in a non relational database, in this case MongoDB. We use the search functionality of the database to retrieve the graphs that are linked to specific problems or techniques, like *reinforcement learning*. In its current version, GitGraph contains 6863 graphs in total, coming from 1449 repositories, for an average of 4.73 graphs per repository.

A node in Tensorflow graph contains the operation performed, from a set of standard operation such as addition, matrix multiplication or convolution. It may also contain additional information depending on the type of operation such as certain hyperparameters values. The contents of the Tensorflow checkpoints cannot are not compatible with the graph libraries used afterwards [16, 17]. A preprocessing phase is needed, where the graph definitions are extracted from these files and cleaned. We convert the checkpoint contents to Graph-tool [4] graphs. Graph-tool is a python library for graph manpilation where the core algorithms and data structures are written in C++ for fast computations.

The distribution of nodes is shown by the black "all" curve in Figure 2 a. We can observe that most architectures contain between 10^2 and 10^3 nodes, with the smallest 20% having less than 10^2 and the largest 20% more than 10^3 nodes.

3.1.1 Deduplication and Limitations. Many of the graphs are duplicates, due to multiple checkpoints for the same model and forked repositories. To counter the effect that these duplicated graphs will have on the subsequent analysis, we perform a deduplication step. We only remove exact graph duplicates. To assess equality between two nodes, we only factor in the type of operation, not any possible additional information like hyperparameter values.

A fully duplicated architecture distorts the results when mining frequent subgraphs. From a different point of view, it can be interesting, since we believe people will tend to duplicate good architectures. We do not explore this in the current work. Even after the deduplication phase, it is still possible for two distinct graphs to the same behavior. For example, stacking two times an Affine Layer $matrix multiplication + addition of bias$, without an activation will result in the same behavior as only one affine layer. We do not include this type of duplication in the analysis.

The deduplication leads to a subset of 2033 unique graphs from the original 6863. We see the result of the deduplication in figure 2 b. Since the number of nodes varies wildly, we opt for a complementary cumulative distribution function (CCDF) curve. The values on the curves show the proportions of the graphs for which the number of nodes exceeds a given threshold, represented by the value on the X axis. We can observe from the difference between Figures 2 a and b that, for the black curve, corresponding to all the graphs, the shape of the CCDF curves stays almost unchanged when the graphs are deduplicated.

3.2 Subgraph Mining Scope Definition

Defining the scope of the problem. If, for instance, someone is interested in *machine translation for Swiss German*, we may not find any prior researchers who tackled that specific problem. A reasonable assumption is that neural architectures made for *machine translation* or, in the best case for *machine translation German* are similar to the given task.

To attract the interest of a varied audience, we focus on three tasks, from three different domains of machine learning: *image processing*, *text processing* and *reinforcement learning*. GitGraph contains:

- For image : 139 graphs with duplicates , 80 without.
- For text : 77 graphs with duplicates, 29 without.
- For reinforcement : 283 with duplicates, 88 without.

[3] www.tensorflow.org
[4] https://graph-tool.skewed.de

We visualize the distribution of nodes in graphs linked to the three chosen tasks before and after the deduplication step in figure 2 a. and b. The most important observation that can be drawn from figure 2 a and b is that the shapes are remarkably similar. Removing the duplicates had no major impact on the node distribution of the graphs.

Unsurprisingly, the largest graphs are the ones that deal with image processing. Text processing comes second, with the smallest number of nodes being found in reinforcement learning tasks. An important element that is visible 2 b. in the *text* task is that there are graphs with 0 nodes. This is more visible after the deduplication because they represent a higher proportion of the total number of graphs.

We observe an interesting behaviour - for image oriented architectures, after deduplication we have a higher proportion of large (more than 10^3 nodes). This shows that for every day tasks the large architectures like Resnet may prove too much and researchers favor smaller ones.

3.3 Graph Cleaning

After defining the datasets of unique graphs for each task, we perform a sequence of preprocessing steps in order to focus only on the core neural architecture. We remove the auxiliary discernible components in order to make the graph lighter in terms of nodes. We remove nodes that are not useful for the solving the central problem, but are instead used for connected tasks. We thus:

- remove all nodes created by the optimizer including all the subsequent gradient computation nodes.
- remove all nodes concerning saving/restoring variables as well as summarization (visualisation on the tensorboard).
- remove the nodes used to initialize a variable
- reduce multiple nodes pertaining to a variables definition to a single one. We remove assign and identity nodes, and forward the edges directly to the variable node.

By doing so, we ensure that the subgraphs that are common to the graphs resulting from the reduction phase are useful in computing the core elements of the task. In the absence of reduction, a common problem is that the common subgraphs are actually outside the core. Instead of finding elements that can be then recombined into a better atchitecture for the same task, we would mix core with non-core modules, making the automated learning process slower and more cumbersome instead of more streamlined.

4 FREQUENT SUBGRAPHS

The second contribution of this work is a method of mining reusable neural modules. We focus on subgraphs that are repeated in the published graphs in GitGraph. We define a common subgraph, for a certain task, as one that appears in the graphs made for that task more than a given threshold. As in the deduplication phase, we only employ the type of operation within the node when computing the node equalities.

We define a subgraph that is common at τ% for task T as one that appears in a *minimum* of τ% of the graphs for task T. The higher the maximum threshold of a subgraph, the more likely it is that subgraph is actually relevant for the given task.

In Figure 3a we portray the number of distinct frequent subgraphs, for each support τ and each example task. For low τ values the number of frequent subgraphs is very high and uninformative. We thus plot the subgraph count for τ values above 30%. While for image and text processing we notice the expected steady decrease in the number of common subgraphs with the increase of τ, for reinforcement we notice that there are exactly 25 common subgraphs for any τ below 70%. This high similarity between various distinct reinforcement architectures is explained by multiple forking of the same repository.

4.1 Subgraph Mining and Matching

The biggest common subgraphs are more informative than smaller ones contained within them. The main goal of GitGraph is to find neural components that reduce the complexity of the space when creating neural evolution or search policies. We thus eliminate the small common subgraphs that are contained within bigger ones that are still common given the same value of τ.

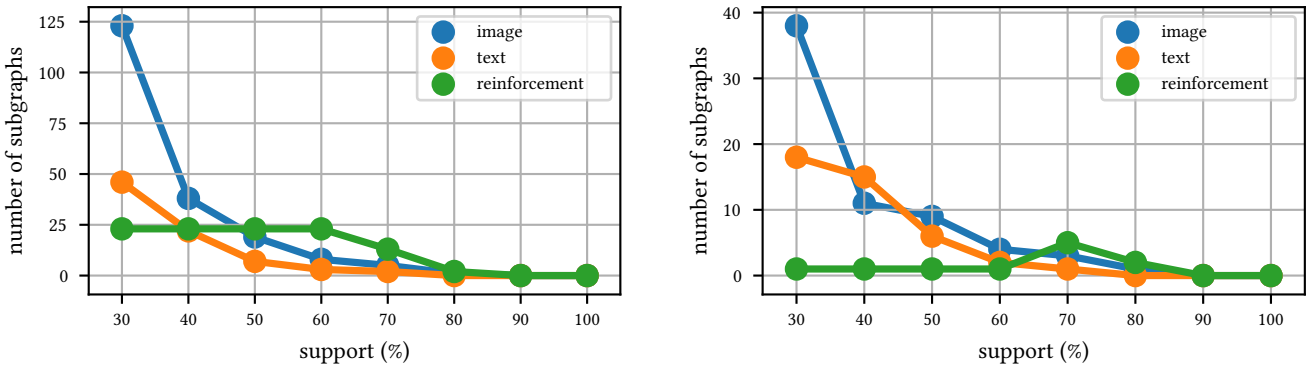
CloseGraph is a GSpan [16] option that ensures that the subgraphs we mine are distinct and not mere variations of the same one. The functioning of CloseGraph is the following. Let A-B-C be the notation corresponding to a subgraph where node A is connected to node B and node B is connected to node C. If, within a set of graphs, the subgraph A-B-C is present k_{ABC} times and graph A-B-C-D is present k_{ABCD} times, with $k_ABC = k_{ABCD}$, then A-B-C is always present only in A-B-C-D. If a subgraph is only present inside a bigger subgraph, then only the biggest subgraph is returned. If $k_{ABC} > k_{ABCD}$, then A-B-C is also present outside of A-B-C-D $k_{ABC-ABCD}$ times. If $k_{ABC-ABCD} > \tau$ then we also consider A-B-C a subgraph on its own. In order to this, once we have all the frequent subgraph fetched by gSpan, we counted in how many graphs they appear (unique count) without considering them if they appear in a bigger subgraph returned by gSpan. The unique count $k_{ABC} - ksupersetABC$ is then compared to the threshold to determine if the subgraph should be included in the returned set.

In figure 3b we portray the number of distinct frequent subgraph after the subgraph reduction step for each support τ ranging from 30 to 100%. For image and text processing, the figure shows an expected decrease in the number of subgraphs - from 123 to 38 for images and from 46 to 18 for text.

In the special case of reinforcement - with $\tau = 30\%$ we obtain 23 frequent subgraphs but only 1 remains after the subgraph reduction - that is reused in almost all of the others. A counterintuitive aspect that occurs at $\tau = 70\%$ is that the number of subgraphs increases from 1 to 5. This means that the only common subgraph for $\tau <= 60\%$ actually occurs in less than $\tau = 70\%$ of all the reinforcement graphs. However, 5 parts of it are common for $\tau = 70\%$ of the graphs, which explains the unexpected jump in the number of common subgraphs, with the increase of τ. These observations are thus indicative of a low level of architectural changes in reinforcement-oriented graphs.

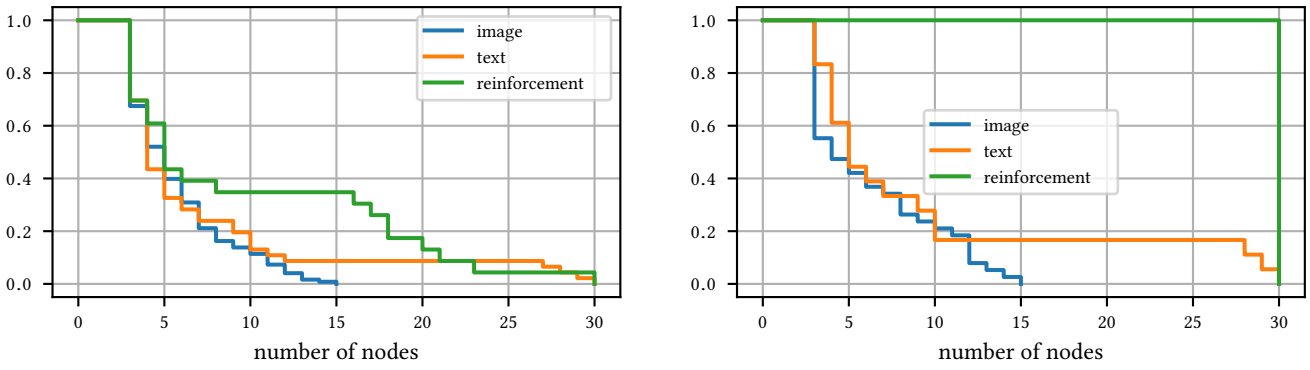
4.2 Subgraph Size

The frequent subgraphs are meaningful, large chains, containing tens of nodes. If they are replaced with single nodes, they can lead



(a) Before reduction **(b) After reduction**

Figure 3: Number of distinct frequent subgraphs given support



(a) Before reduction **(b) After reduction**

Figure 4: Size of frequent subgraphs (in no of nodes) (min support = 30%)

to a significant reduction of the complexity of the network. We thus do not show common subgraphs smaller than 3 nodes.

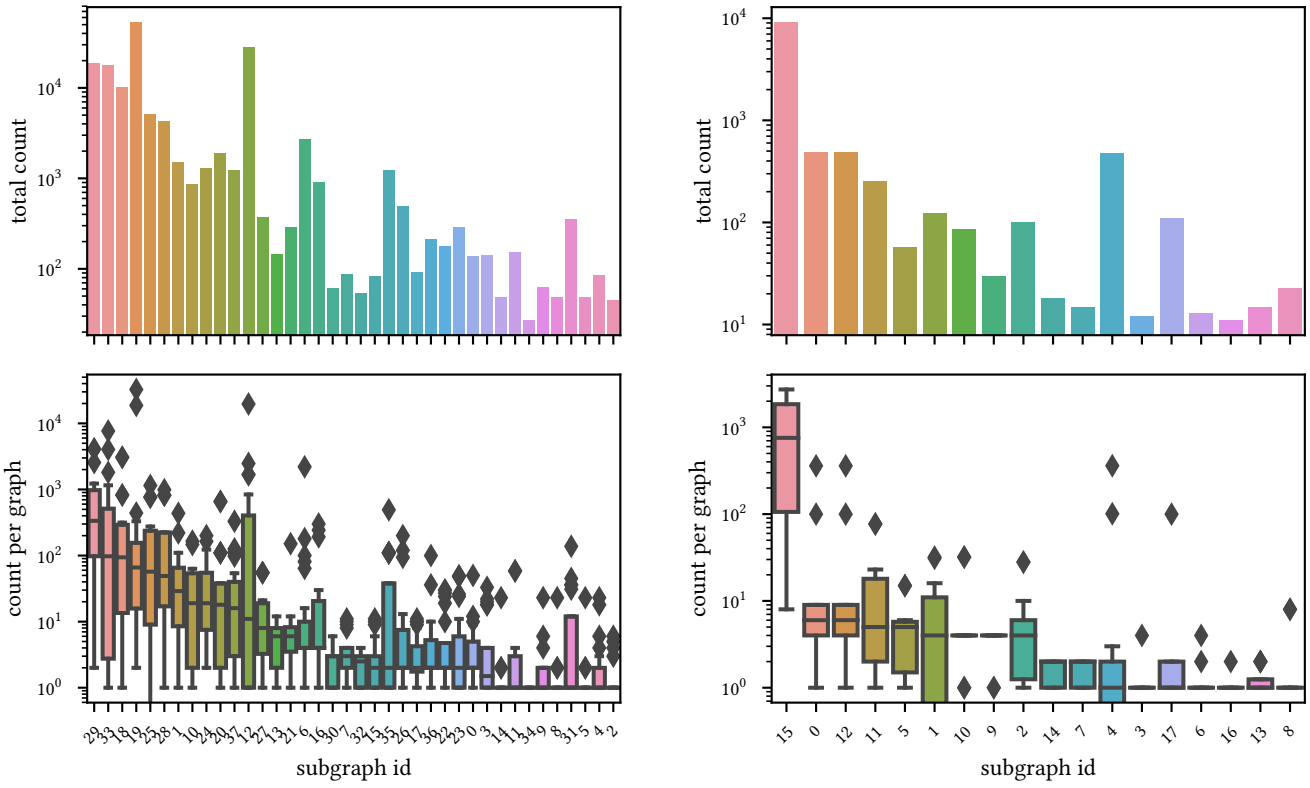
In figure 4a and 4b we show the number of nodes in the frequent subgraphs, before and after the subgraph reduction respectively. Each CCDF curve shows the percentage of distinct frequent subgraphs mined for this task with the size exceeding the threshold on the abscissa.

The differences between the pairs of CCDF curves portray the effect of the subgraph reduction. From the reinforcement curve in Figure 4b we notice that the size of the common reinforcement subgraph discussed in the previous section is 30 nodes. This subgraph is shown in Figure 10a. Without the reduction phase, this subgraph would not have stood out so evidently.

For image and text analyses, roughly half of the frequent subgraphs have a size of between 3 and 5 nodes. The other half is between 5 to 15 for image and 5 to 30 for text. We are primarily interested in the larger graphs, since a simple operation in Tensorflow can spawn multiple nodes. For example, adding two constants create 3 nodes: one node for each constant + one node for the addition operator. For *text*, we notice that roughly 20% of the reduced common subgraphs are actually very long - having more than 25 nodes. A manual analysis of these nodes shows that they correspond to commonly used recurrent units, like the LSTM in Figure 9a.

4.3 Graph Compositionality and Complexity Reduction

We show that the subgraphs mined with the methods presented in section 4.1 appear a large number of times in the original graphs. This insight is proven by the large complexity reduction possible through subgraph mining. This is an important element in building a small yet effective architecture search space.



(a) Image **(b) Text**

Figure 5: Count presence (multiple time) for each frequent subgraphs after reduction

In Figure 5 we show the fine-grained picture of individual subgraphs. The low number of subgraphs in the considered problems, after reduction, allows us to track each one individually. We opt for this visualization technique to emphasize the large number of occurrences some of these subgraphs have. The top figure indicates for each subgraph how many time they appears in total, sum of all the occurences of this frequent subgraphs. The bottom figure shows for each subgraph the number of time it appears per graph. As we can see some frequent subgraphs appears multiple times in the same graphs with a high variability. Others, like subgraphs id 13 for *image* or 14 for *text*, appear the same number of time in each graph.

Some examples are telling of the reusability potential of these graphs as individualized modules. For the image-oriented task, subgraph 12 containing 6 nodes has 28149 occurrences in total and in the 26 distinct graphs in which this module appear the median number of occurence per graph is 11. For text, the subraph 15 with 7 nodes has a total number of occurences of 9183 and in the 9 distinct graphs in which this graph occurs the median number of occurence per graph is 756 . For reinforcement learning, there is a single subgraph. It appears exactly once in each of the 59 dinstict graphs it occurs in. However its size is so great (30 nodes) that it actually eclipses the changes that each graph adds to this common component. This naturally leads to the question of what gains are achievable if the subgraphs are treated as individual nodes. We address this in the following section.

Many subgraphs have occurrence outliers in Figure 5. These portray the importance of using the found subgraphs as modules in the architecture evolution or search. If the cost of reusing the subgraph is low, it can be done a high number of times, thus mimicking the human creators of existing architectures.

Alternatively, from the lack of variability in other subgraph occurrences, we discern the idea of layer and we get architectural insights. For instance, if a subgraph is a convolution, we can determine, from its occurrence count, the usual number of convolutions needed to solve the problem. We only show plots for image and text, as for reinforcement there is a single subgraph, discussed previously and shown in 10a, that appears exactly once in each of the 59 dinstict graphs it occurs in.

4.3.1 Complexity Reduction. We inquire whether turning the found subgraphs into individual nodes helps in reducing the complexity of the neural search space. Currently, the GitGraph repository does not yet allow the creation of architectures using the frequent submodules as building blocks, which we leave for future

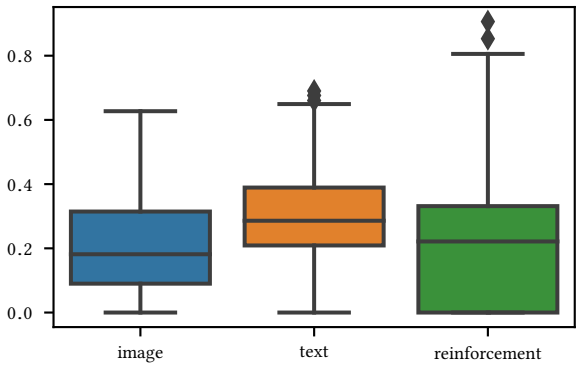


Figure 6: Distribution of node reduction on the whole data

work. We thus investigate the possible gains through a proxy - the ratio of the number of nodes that are in frequent subgraphs, with respect to the number of nodes found outside them. We use the term *complexity reduction* to capture the ratio.

To reduce a graph, we replace each of its frequent subgraphs by one node. We then compute the difference between the original graph node count and that of the reduced graph. Finally, we normalize by the original count to obtain a ratio of nodes that belong to the frequent components. Figure 6 plots the complexity reduction for the three studied tasks. The median reduction for the three tasks ranges from 20 to 30%, with the highest value being recorded for text.

This approach also allows us to identify the graphs where the reduction is 0% or close to it. In some cases, these can contain new concepts. In others, this can be a filter to identify mistakes - for instance empty graphs or ones that have been included in the analyses because of a failure of the TF-IDF based search method.

4.4 Generalization Capacity

A relevant critique of the analysis in the section above is that it is descriptive and does not show the capacity of the subgraphs found to generalize. The fact that a subgraph is found in 20% or more of the data in one dataset does not, in itself, guarantee that for a different set of graphs created for the same purpose will share these subgraphs.

We thus create a new batch of tests to test the hypothesis that the subgraphs found are general and lined to the task itself. For each task, we create five different experiments in which we randomly split the data into a ẗrainingṡet and a ẗestṡet, with the goal of seeing whether the subgraphs mined from the graphs within the training set will occur in the graphs within the test set. In each experiment, we held 80% of the graphs as training and the remaining 20% for testing. We performed the frequent subgraph mining, followed by the addtional reduction only on the training dataset and we computed the node reduction using these subgraphs on the test dataset. We report the results for each experiment individually in Figure 7. The median and variance change very little from the previous experiment. This result upholds the one in the previous section and shows the complexity reduction is achievable on unseen graphs, for the same task.

5 CONCLUSION

We introduced GitGraph, the first corpus of neural computation graphs. The first goal of GitGraph is to serve as a knowledge repository that allows for an automated search of neural architectures that solve a specific problem. Using the search functionality, we can obtain a set of distinct architectures for problems related to the searched one. From the found architectures, in the form of computation graphs, we created a method of generating unique relevant common subgraphs.

The main aim of finding problem-specific GitGraph common subgraphs is to create a neural search space. Instead of searching or using reinforcement or evolution strategies using the basic building blocks, we reduce the complexity and cost of the subsequent neural architecture creation policy by optimizing the search space itself.

We show that the GitGraph common subgraphs cover between 20 and 40% of the nodes in their source graphs. Given the obtained complexity reduction, we believe they will be a basis for large problem-specific modules in future automated neural creation strategies.

REFERENCES

[1] MartÃŋn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan ManÃĺ, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda ViÃĺgas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. (2015). http://download.tensorflow.org/paper/whitepaper2015.pdf

[2] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2016. Designing Neural Network Architectures using Reinforcement Learning. *CoRR* abs/1611.02167 (2016). arXiv:1611.02167 http://arxiv.org/abs/1611.02167

[3] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv:1406.1078 http://arxiv.org/abs/1406.1078

[4] R.G. Goldberg and R.R. Rosinski. 1999. Automated natural language understanding customer service system. (April 20 1999). https://www.google.ch/patents/US5895466 US Patent 5,895,466.

[5] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[6] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone Detection Using Abstract Syntax Suffix Trees. (10 2006), 253–262.

[7] R. Negrinho and G. Gordon. 2017. DeepArchitect: Automatically Designing and Training Deep Architectures. *ArXiv e-prints* (April 2017). arXiv:stat.ML/1704.08792

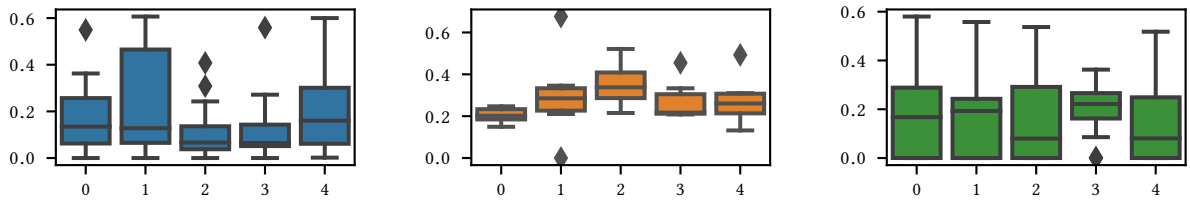
[8] Vaibhav Saini, Hitesh Sajnani, Jaewoo Kim, and Cristina V. Lopes. 2016. SourcererCC and SourcererCC-I: Tools to Detect Clones in Batch mode and During Software Development. *CoRR* abs/1603.01661 (2016). arXiv:1603.01661 http://arxiv.org/abs/1603.01661

[9] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. 2017. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. (2017), 1–13. https://doi.org/10.1.1.51.6328 arXiv:1703.03864

[10] Martin Schrimpf, Stephen Merity, James Bradbury, and Richard Socher. 2017. A Flexible Approach to Automated RNN Architecture Generation. (2017), 1–16. arXiv:1712.07316 http://arxiv.org/abs/1712.07316

[11] Niko Schwarz, Mircea Lungu, and Romain Robbes. 2012. On How Often Code is Cloned Across Repositories. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 1289–1292. http://dl.acm.org/citation.cfm?id=2337223.2337398

[12] Kenneth O. Stanley. 2007. Compositional Pattern Producing Networks: A Novel Abstraction of Development. *Genetic Programming and Evolvable Machines* 8, 2



(a) Image (b) Text (c) Reinforcement

Figure 7: Distribution of node reduction on five sets (train/test split)

(June 2007), 131–162. https://doi.org/10.1007/s10710-007-9028-8

[13] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks Through Augmenting Topologies. *Evol. Comput.* 10, 2 (June 2002), 99–127. https://doi.org/10.1162/106365602320169811

[14] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. 2017. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. (2017). arXiv:1712.06567 http://arxiv.org/abs/1712.06567

[15] Ronald J. Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning* 8 (1992), 229–256. http://www.cs.ualberta.ca/~sutton/williams-92.pdf

[16] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM '02)*. IEEE Computer Society, Washington, DC, USA, 721–. http://dl.acm.org/citation.cfm?id=844380.844811

[17] Xifeng Yan and Jiawei Han. 2003. CloseGraph: Mining Closed Frequent Graph Patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '03)*. ACM, New York, NY, USA, 286–295. https://doi.org/10.1145/956750.956784

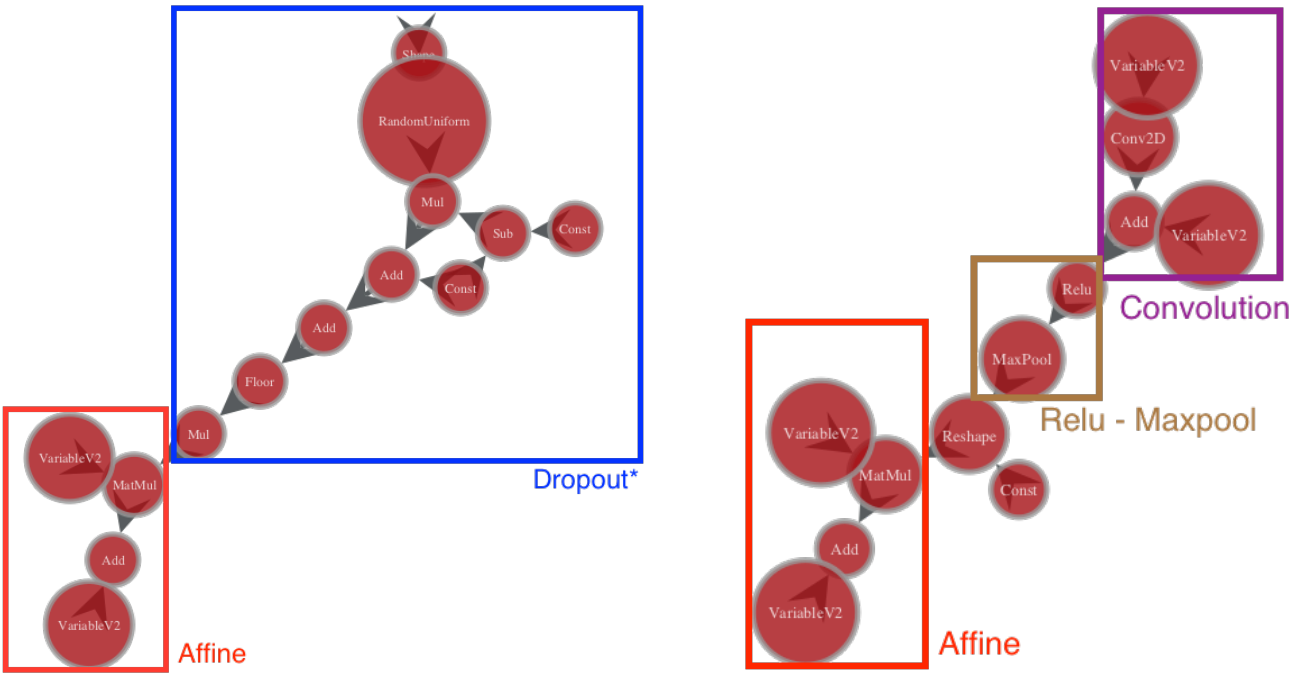
[18] Barret Zoph and Quoc V. Le. 2016. Neural Architecture Search with Reinforcement Learning. *CoRR* abs/1611.01578 (2016). arXiv:1611.01578 http://arxiv.org/abs/1611.01578

A DATABASE OF TENSORFLOW GRAPHS

All the meta checkpoints (from which we can extract the Tensorflow graphs) are contained in the following multi-part zip.

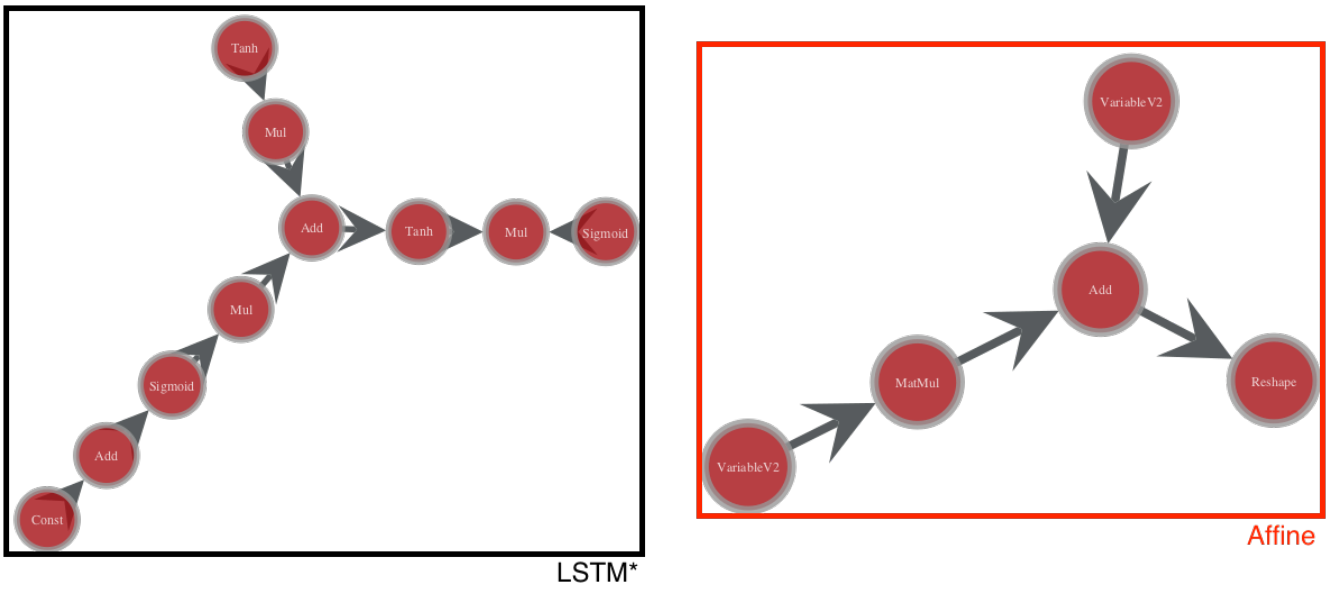
Download all the files below in the same directory and uncompress the main zip (Approx: 3go compressed , 15go uncompressed)
Main zip
Part 1
Part 2
Part 3
Part 4
Part 5
Part 6

B EXAMPLES OF FREQUENT SUBGRAPHS



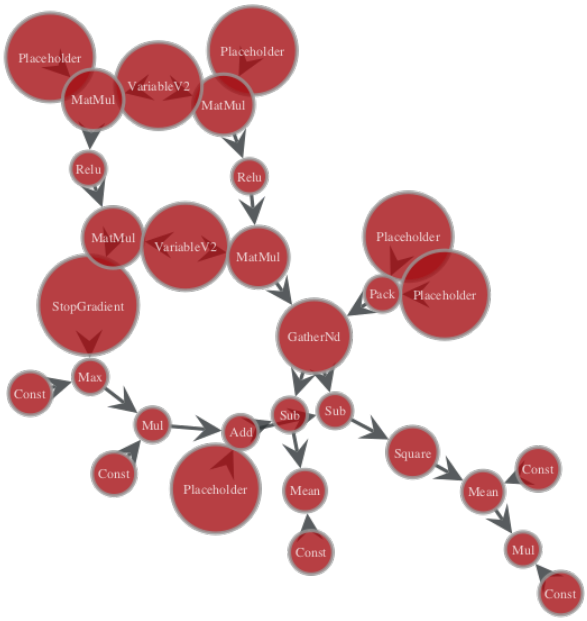
(a) (b)

Figure 8: Example of frequent subgraphs for the image dataset



(a) (b)

Figure 9: Example of frequent subgraphs for the text dataset



(a)

Figure 10: Frequent subgraph for the reinforcement dataset