SCHEDULED CONTRASTIVE TRANSFER LEARNING FOR SOFTWARE ENGINEERING TASKS

Aaron Harris, Afia Farjana
Computer Science
College of William & Mary
Williamsburg, VA, USA
{amharris04, afarjana}@wm.edu

ABSTRACT

Our current research efforts into contrastive learning in the context of software engineering machine learning has shown orthogonal results sets versus the baseline. We have also seen utility in using a scheduled temperature rate for the contrastive loss measure. We combine these factors with a "fine-fine-tuning" (FFT) approach; continuing transfer learning to train the fine-tuned model from the baseline to capture the orthogonal results in one model. Through the use of a training period scheduled temperature with contrastive loss, we are able to exceed the baseline accuracy of three software engineering tasks by varying margins, ranging from 5% to 264% during single beam tests.

1 INTRODUCTION

Machine learning methodologies related to software engineering have been a focus of considerable research. In particular, work related to the repair of software and injection of code mutants via neural machine translation (NMT) have seen progress via the usage of transformers and transfer learning Mastropaolo et al. (2022). These developments are of great importance to the software engineering community; the time and resources dedicated to code repair, for example, are estimated to make up approximately 16% of software development costs Krasner (2018), and taking a considerable portion of development time usage Minelli et al. (2015).

Repairing a bug, for a software developer, is not simply replacing the faulty text with the appropriate variation; it is localizing and identifying the bug, determining the appropriate fix, and considering the impact of that fix on the code as a whole. Other tasks, such as the injection of code mutants, contribute to security-related software engineering tasks like test suite creation. This presents its own unique set of challenges, such as determining which mutants are plausible with a given input of code, though it keeps in common the time-related detriments of bug repair. The capability to automatically repair software bugs and generate code mutants, along with myriad other text-to-text software engineering tasks, lends itself well to transformer-based approaches; we consider the problem in the same way one might consider the translation between two human languages.

Of note in this area of research is the overlap between text-based and vision-based problems. Methodologies such as masked auto-encoders, which started in the text domain and made vast improvements in the vision domain, are one such example He et al. (2021). In that vein, we seek to employ contrastive learning to improve the accuracy of bug repair and mutant generation tasks. Contrastive loss is highly effective in vision-based tasks Khosla et al. (2020), and has shown usefulness crossing over into code-related NMT tasks Ding et al. (2023).

In this effort, we propose to extend current research by focusing on the use of contrastive learning as an alternative loss measure to the baseline presented in Mastropaolo et al. (2022). We have found that contrastive learning is capable of finding orthogonal results to the baseline, though at a lower rate [pending]. Therefore, we will use transfer learning to incorporate the benefits of the baseline method and our novel contrastive implementation to determine whether these methods can create a combined result of greater magnitude than either method independently, capturing a large proportion of the correct results of the baseline while adding new correct results from the novel method.

2 BACKGROUND

Consider a software bug and a given fix - solutions can be wide-ranging, but in general the fix is some variation on the bug. For our discussion of this, we refer to these two code segments as m_{bug} and m_{fix}, and seek to find an m_{fix} such that the new code segment no longer produces the erroneous behavior present in the original code segment. We refer to the tuple (m_{bug}, m_{fix}) as a Bug-Fix Pair (BFP), noting that the given m_{fix} is not necessarily the only solution, simply the solution present in the dataset. Conversely, a code mutant can be expressed as the inverse of such a tuple, (m_{fix}, m_{bug}), given that a correct code segment will be translated into a buggy code segment.

Given these parameters, the goal is to determine a method that effectively translates between the two code segments. In the domain of human language translation, transformers have demonstrated a capability on par with that of human interpreter output Popel et al. (2020); given the similarity of the problem, it is reasonable to consider the transformer architecture as an appropriate vehicle for our task. In their 2017 work Vaswani et al. (2017), Vaswani et al. introduced the transformer architecture, which has been used considerably in research related to natural language processing (NLP) and NMT. Specifically it excels at text translation and adjacent fields. Of particular importance in the architecture is the concept of an attention layer, which attends to each element of the input sequence to determine its importance to other input sequence elements, providing the model a variety of context when considering those elements. This development gave the architecture an advantage over previous methods, such as Long Short-Term Memory models Hochreiter & Schmidhuber (1997).

The transformer architecture is delineated into an encoder and decoder, which are responsible for learning the formation of latent space representations from the input sequence and generating the output sequence, respectively. The loss function for the encoder and decoder is typically cross-entropy loss, due to its ability to determine the (dis)similarity between the training target and the predicted probability distribution.

Of course, the approach does come with its limitations, in that such an architecture learns from the provided data, and any output produced at test time is restricted to a rearrangement of the tokens which it has learned Tufano et al. (2019b). Variables and other named elements in code present a risk of creating a large vocabulary problem. To subvert this issue, we employ code abstraction to reduce the number of idioms present in both training and test. We also note a key difference between the use case of human language translation and that of code repair and mutant generation: in a human language translation, the intent is to convey the same meaning, while in the software engineering tasks presented, it is to correct or break the meaning.

3 RELATED WORKS

3.1 TRANSFORMERS AND CONTRASTIVE LOSS

In 2020, Raffel et al. continued this line of effort with the Text-to-Text Transfer Transformer (T5) architecture Raffel et al. (2020). The default T5 transformer continues to employ cross entropy loss. The baseline we measure against follows this method.

Contrastive loss seeks to create a distribution space where similar embeddings have a lower distance between them while ensuring dissimilar embeddings have a greater distance. To enable this dual functionality, datasets used with contrastive learning require augmentation to provide additional inputs that are similar in nature. While image dataset augmentation has numerous methodologies for augmentation, it is a more difficult problem with text datasets, specifically code-centric ones Ding et al. (2023); we cannot simply change the color or horizontally flip our code inputs. Instead, we perform this augmentation using a rules-based Java-language code augmentation tool Yu et al. (2022), which generates semantically identical variations of the code input using a rules-based approach.

While contrastive learning has made impact in research related to image-based tasks Khosla et al. (2020), there is area for growth in the domain of NMT and software engineering related machine learning. Research related to the use of contrastive learning in software engineering tasks has been performed by Ding et al., with emphasis on clone and bug detection; the work primarily focuses on the problem of out-of-distribution elements and how to correct the issue Ding et al. (2023).

An important aspect of a contrastive loss function is the idea of temperature - a lower temperature parameter tends to produce more determinant results, while a higher temperature tends towards the exploratory. Additional research has been performed concerning the use of scheduling changes in temperature throughout training Kukleva et al. (2023), which has shown increased representational learning. We will employ scheduled temperature using several growth models (linear, Fibonacci, decreasing) in our efforts.

3.2 Transfer Learning

Transfer learning is not a new concept; the first related paper with regard to neural networks was published in 1976 Bozinovski & Fulgosi (1976). The essential element is that a network trained on one topic can utilize the learning already performed to enhance its capabilities in learning a separate, but related, topic. These processes are referred to as ”pretraining” and ”fine-tuning” the model. Multi-task learning (MTL), where a single pretrained model is simultaneously fine-tuned on several downstream tasks, has been shown to be an effective method Zhang & Yang (2017).

In our work we use a T5 model that has been pretrained on a large dataset of code and code-related text, then fine-tuned on task-specific datasets for bug fixing (small and medium variants) and mutant generation; this model has been trained using cross-entropy loss. We additionally use a second model trained using the same methods, with the key differences being (a) the datasets have been augmented to include modified, though similar, elements and (b) contrastive learning is used for the encoder portion of the transformer.

3.3 Learning Bug Fixes

Our research seeks to extend the work by Tufano et al. Tufano et al. (2019b). This study demonstrates a method for repairing code by learning the patterns associated with bug-fix pairs in a large corpus of software changes. The authors were able to attain between 9% and 50% accuracy in selecting the correct fix for a given bug. Additional work by many of the same authors Tufano et al. (2019a) further clarifies the design decisions of their research. Further research by Chen et al. Chen et al. (2019) similarly tackled the task of bug fix selection, though specifically on one-line changes, unlike Tufano et al. (2019b) and Tufano et al. (2019a) which performed at the function level.

Work by Mousavi et al. Mousavi et al. (2020) logically delineates automatic software repair into two categories, runtime (preventing/rescuing from fault at execution) and source code (repairing prior to runtime); our research is focused on the latter.

3.4 Learning Mutant Generation

Mutant generation has been approached in baseline research Tufano et al. (2019b). Additional work by Sergen et al Aşik & Yayan (2023) produced a method of generating python-based mutants, using transformer architecture and abstracted code. The paper considers those results that are perfect predictions and similar to perfect predictions, and relays a 6% to 35% accuracy rate, with higher accuracy when counting ”other buggy codes” produced, with a high degree of lexical accuracy in produced code.

4 Methodology

Figure 1 provides a visual representation of our process.

Our method will continue the transfer learning approach on the baseline by performing a second round of fine tuning on the existing model that uses a scheduled contrastive loss objective function to attempt to increase the performance of the T5 model in the selected tasks. We focus on three tasks from the baseline Tufano et al. (2019b), namely small bug fixes (BFS), medium bug fixes (BFM), and mutant generation (MG). We do not consider the other two tasks present in the baseline, Assert Generation and Code Summarization, as these tasks are different in concept. Assert Generation, while still a code-to-code task, creates a small amount of code based on the given input. Code Summarization is not a code-to-code task, producing a plaintext description from the supplied code snippet.

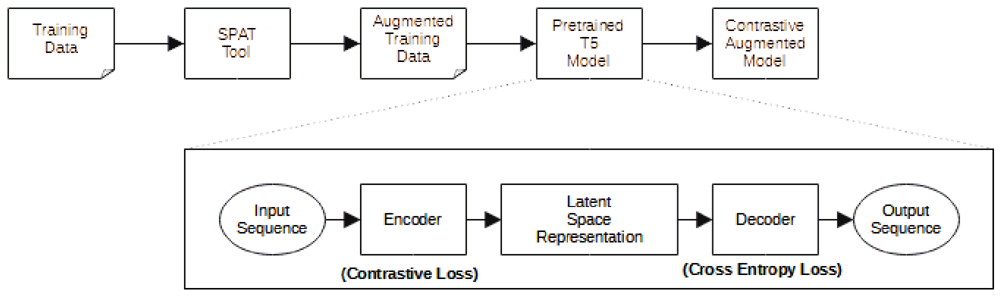

Figure 1: Process diagram for including contrastive loss as an additional learning pass.

We implement a custom training loop to perform FFT using temperature-based contrastive learning with the stated schedulers (linear, Fibonacci, and decreasing). The base contrastive learning function has already been implemented during current research; it will be expanded to use both static and dynamic scheduling rates for linear, Fibonacci, and decreasing schedules. The selection of an appropriate temperature for this work is regarded as an additional hyperparameter; we will vary the temperature ranges and schedules to tune it.

The schedulers, at each iteration, increment (or decrement) the temperature value of the contrastive loss function by a rate that is determined by dividing the range of temperature change steps over the course of the planned training period. For linear schedules, this is simply the difference in the high- and low-temperature values. For the Fibonacci-based schedulers, it is the cardinality of the indices of Fibonacci values to be used.

Our primary concern is increasing the accuracy at various beam sizes, aligned with the baseline. Further, we will analyze the differences in the accurately predicted results with those of the baseline to determine orthogonality. We will also observe the differences in training duration; we begin this effort with the expectation that contrastive scheduled learning will take longer to perform, due to the increased size of the dataset due to data augmentation.

We intend to answer the following research questions:

- RQ1: What effect does contrastive loss in continued transfer learning have on code repair and mutant generation prediction accuracy?
  Our figure of merit will be the accuracy at selected beam sizes (1, 5, 10, 25, 50) for the Bug Fix (Small) and Bug Fix (Medium) tasks, and beam size 1 for the Injection of Code Mutants task, in keeping with the baseline paper.
- RQ2: What are the limitations of temperature-based contrastive loss in these tasks?
  In our current research, we note that using too similar of temperature values can be detrimental.
- RQ3: What overlap do the results have with the baseline method?
  I.e., are some previously accurate predictions lost in exchange for new results?

4.1 Data Augmentation & Processing

Our datasets are derived from those presented in the baseline Tufano et al. (2019b). We employ separate datasets for BFS, BFM, and MG. For each, we generate variants using Yu et al. (2022) (specifically, a configurable implementation available at Devy99 (2023)) based on the following rules:

- Boolean Exchange
- Conditional Expression to Single If
- Infix Expression Dividing
- Loop Exchange

Dataset	Initial Count	Augmented Count	% Increase
BFS	46680	53154	13.87%
BFM	52364	119234	127.70%
MG	92477	110316	19.29%

Table 1: Summary of dataset size increases using the semantic-preserving augmentation method.

Original	public void METHOD_1 (android.view.View view) { if (!((context) instanceof TYPE_1)) { } else { ((TYPE_1) (context)).METHOD_2(string); } }
Variation	public void METHOD_1 (android.view.View view) { if ((context) instanceof TYPE_1) { ((TYPE_1) (context)).METHOD_2(string); } else { } }

Table 2: A sample from the BFS dataset, processed through the augmentation and abstraction processes.

- Loop If Continue to Else
- Reverse If Else
- Single If to Conditional Expression

While there are additional rules available in the augmentation tool, we limited the list based on two criterion. First, after an initial test run, which rules reliably provided multiple results for our input set. Second, we removed rules that provided only boilerplate insertions, such as inserting a single dummy log statement; the reasoning behind this decision is that we seek changes with functional equivalence and substance in the code difference. As every input sequence would, keeping with the example, be capable of receiving an extra meaningless line, it would not generate a semantically similar and substantially, textually different variation.

Only the training data is augmented; the test sets are left in their original state to make comparison with the baseline clear. Once the data has been augmented, it is passed through the src2abs tool Tufano (2019) to create the necessary abstractions. Finally, we create a ”stacked” dataset to ensure that the variations are distributed well throughout. For each task, ten variations of the input dataset are generated and then concatenated.

Table 1 provides a summary of the increases in dataset size produced by this augmentation. We note that BFM received the largest increase in size, which follows from the nature of the datasets. BFS and MG both tend to have smaller inputs and targets, while BFM has more room for elements that the tool can consider for augmentation.

Table 2 provides a sample of an augmentation, post-abstraction - in this case, the swapping of branches in an if...else statement via the Reverse If Else rule. The condition at the beginning of the statement is reversed, and the branches are exchanged. This provides the semantically similar, textually different desired variation.

5 RESULTS

Of the three contrastive temperature schedulers, the Fibonacci-based scheduler outperformed the linear and linear-decreasing schedulers in most cases. As such, we continued with this scheduler for our further attempts, increasing the temperature from the first through tenth Fibonacci numbers over the course of training.

As each dataset is composed of ten variations, including some duplicates in cases where ten variations were not generated by Devy99 (2023), is approximately the length of ten of the baseline datasets, this can be considered equivalent to 200 epochs. This is done to increase the instances in which differentiation between similar and dissimilar samples are encountered, giving the contrastive loss function opportunities to increase the distance in the representation.

Task		Beam 1	5	10	25	50
Bug Fix (Small)	Baseline	15.08	**32.08**	**37.01**	**42.51**	**45.94**
	Experiment	**21.22**	31.68	35.40	40.23	43.80
Bug Fix (Medium)	Baseline	11.85	19.41	**23.28**	**28.60**	**32.43**
	Experiment	**12.91**	**19.64**	22.34	25.57	28.42
Mutant Generation	Baseline	28.72				
	Experiment	**28.98**				

Table 3: Experimental results for single-task models.

5.1 RQ1: What effect does contrastive loss in continued transfer learning have on code repair and mutant generation prediction accuracy?

The results of these experiments can be seen in Tables 3 & 4. For the single-task models, our methods provide a marked increase in the accuracy of BFS and BFM tasks (40.7% and 8.9%, respectively), while providing a very minor increase in MG accuracy (0.9%) on Beam 1. In later beam sizes, only the BFM task improved on the baseline at Beam 5, showing a 1.1% increase. All other trials for single task models fail to pass the baseline.

For the multi-task model, markedly higher improvements in accuracy are seen on Beam 1. BFS and BFM increase by 99.3% and 264.1%, respectively, with MG continuing to have a lesser increase at 5.6%. For Beam 5, BFS and BFM show 2.2% and 5.5% increases. As with the single-task models, later beam sizes do not see an improvement over the baseline.

5.2 RQ2: What are the limitations of temperature-based contrastive loss in these tasks?

Due to the nature of the temperature scheduler, which changes over the course of the planned training length and not on a per-epoch basis, we conduct initial model runs to determine the appropriate number of epochs to perform. This is done empirically, with 20 epochs of the stacked augment datasets providing the highest accuracy in each case. This adds some time considerations to training, as training at e.g. 10, 20 and 30 epochs requires three separate training runs; early stopping is not an option with this implementation.

5.3 RQ3: What overlap do the results have with the baseline method?

For this analysis, we consider whether results in one result set are present in the other; for example, in the Bug Fix (Small) task, what portion of the baseline results does the experimental contrastive model capture? This is done by comparing the inclusion of indices from the test set in each result set.

As shown in Table 5, the highest proportion of results are in the joint category; that is, those that belong to both the baseline and experimental models. This is to be expected, as we are starting with the baseline model when fine-tuning. What is of interest to our research is that the proportion of accurate predictions that are exclusive to the baseline model is lower than those unique to the experimental model, demonstrating that the trade-off for the additional fine-tuning is gaining more perfect predictions than are lost.

6 Future Work

There is still room for future research in this area. Firstly, there is the issue of hyperparameter tuning. At present, we do not have an *a priori* method for determining the appropriate temperature ranges or type of scheduler to be used; all results are currently obtained empirically.

Further, our implementation's temperature scheduler changes the temperature over the entirety of the training period. This makes it difficult to incorporate early stopping while still traversing the entirety of the given temperature range. It would be beneficial to explore methods that perform the scheduled temperature changes on a per-epoch basis.

Task		Beam 1	5	10	25	50
Bug Fix (Small)	Baseline	11.61	35.64	**43.87**	**52.88**	**57.70**
	Experiment	**23.14**	**36.45**	40.79	46.42	49.83
Bug Fix (Medium)	Baseline	3.65	19.17	**24.66**	**30.52**	**35.56**
	Experiment	**13.29**	**20.23**	23.52	26.80	29.81
Mutant Generation	Baseline	28.92				
	Experiment	**30.53**				

Table 4: Experimental results for the multi-task model.

Task	Joint	Baseline	Experiment
BF(S)	685 (11.74%)	192 (3.29%)	553 (9.48%)
MG	2141 (18.52%)	661 (5.72%)	1209 (10.45%)

Table 5: Overlap anaylsis of experimental vs baseline methods. Figures are presented as M (N%), where M is the count of predictions and N is the percentage of the entire test set represented.

7 Threats to Validity

We strive to provide a methodology for software engineering machine learning tasks that in generalized and rigorous; however, there are areas of this research that could introduce issue to the results we provide.

- We note that an internal threat to validity exists from the decision to use only Java-based inputs. While this was done to keep with the baseline selected, it limits the nature of our findings to similar inputs.
- With regard to the construction of our method, the nature of using unshuffled, static datasets to ensure the non-overlapping variations from data augmentation introduces the question of whether the ordering of these datasets could produce differences in our outcomes.
- As with other research using similar architectures and methods, we maintain the limitations derived from the problem of outputs limited to a rearrangement of tokens learned during training.
- Due to the hyperparameter tuning issues noted above, we introduce a conclusion validity issue until a method is developed to determine exact temperature ranges and optimal schedules, and the implementation of an effective epoch-based approach.
- Our analysis of the accuracy of this method is based on the production of a perfect prediction; that is, we only count those result where the output is exactly matched to the ground truth. This is used as validating whether *any* appropriate m_{fix} for a given m_{bug} and vice versa for mutant generation would require considerable manual validation.

8 Conclusion

Software repair is time and resource consuming, and mutant generation encounters the same variety of problems. Due to the patterns in bugs and mutants, it is possible to generate fixes and breaks for a given input sequence; the question resolves to finding the best method of doing so given the current state of technology.

In this work, we implement our contrastive loss scheduler for text-to-text software engineering tasks in a transformer architecture. Our results indicate that fine-tuning the existing baseline models with additional iterations using these schedulers produces considerable accuracy improvements on one-shot inferences, with lower returns at later beam sizes. The improvements are primarily realized in the bug fixing tasks, with mutant generation improvements being more minor in nature.

REFERENCES

Sergen Aşik and Uğur Yayan. Generating python mutants from bug fixes using neural machine translation. *IEEE Access*, 11:85678–85693, 2023. doi: 10.1109/ACCESS.2023.3302695.

Stevo Bozinovski and Ante Fulgosi. The influence of pattern similarity and transfer learning upon the training of a base perceptron b2, 1976.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, September 2019. doi: 10.1109/TSE.2019.2940179. URL `https://ieeexplore.ieee.org/document/8827954`.

Devy99. Data-augmenter. `https://github.com/Devy99/data-augmenter`, 2023.

Yangruibo Ding, Saikat Chakraborty, Luca Buratti, Saurabh Pujar, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. Concord: Clone-aware contrastive learning for source code, 2023.

Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners, 2021.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *CoRR*, abs/2004.11362, 2020. URL `https://arxiv.org/abs/2004.11362`.

Herb Krasner. The cost of poor quality software in the us: A 2018 report. *Consortium for IT Software Quality*, 9 2018. URL `https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf`.

Anna Kukleva, Moritz Böhle, Bernt Schiele, Hilde Kuehne, and Christian Rupprecht. Temperature schedules for self-supervised contrastive methods on long-tail data, 2023.

Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Using transfer learning for code-related tasks, 2022.

Roberto Minelli, Andrea Mocci and, and Michele Lanza. I know what you did last summer: An investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pp. 25–35. IEEE Press, 2015.

S. Amirhossein Mousavi, Donya Azizi Babani, and Francesco Flammini. Obstacles in fully automatic program repair: A survey, 2020.

Martin Popel, Markéta Tomková, Jakub Tomek, Łukasz Kaiser, Jakob Uszkoreit, Ondřej Bojar, and Zdenek Zabokrtsky. Transforming machine translation: a deep learning system reaches news translation quality comparable to human professionals. *Nature Communications*, 11:4381, 09 2020. doi: 10.1038/s41467-020-18073-9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.

Michele Tufano. src2abs. `https://github.com/micheletufano/src2abs`, 2019.

Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. *41st ACM/IEEE International Conference on Software Engineering*, May 2019a. doi: 10.1109/ICSE.2019.00021. URL `https://doi.org/10.1109/ICSE.2019.00021`.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Pena, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology*, September 2019b. doi: 10.1145/3340544. URL `https://doi.org/10.1145/3340544`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Shiwen Yu, Ting Wang, and Ji Wang. Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304, 2022. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2022.111304. URL `https://www.sciencedirect.com/science/article/pii/S0164121222000541`.

Yu Zhang and Qiang Yang. An overview of multi-task learning. *National Science Review*, 5(1): 30–43, 09 2017. ISSN 2095-5138. doi: 10.1093/nsr/nwx105. URL `https://doi.org/10.1093/nsr/nwx105`.