SCRAM: Simple Checks for Realtime Analysis of Model Training for Non-Expert ML Programmers

Eldon Schoop
UC Berkeley EECS
Berkeley, CA 94720 USA
eschoop@berkeley.edu

Forrest Huang
UC Berkeley EECS
Berkeley, CA 94720 USA
forrest_huang@berkeley.edu

Björn Hartmann
UC Berkeley EECS
Berkeley, CA 94720 USA
bjoern@eecs.berkeley.edu

CHI'20 Extended Abstracts, April 25–30, 2020, Honolulu, HI, USA.

ACM ISBN 978-1-4503-6819-3/20/04.
https://doi.org/10.1145/3334480.3382879

Abstract
Many non-expert Machine Learning users wish to apply powerful deep learning models to their own domains but encounter hurdles in the opaque model tuning process. We introduce SCRAM, a tool which uses heuristics to detect potential error conditions in model output and suggests actionable steps and best practices to help such users tune their models. Inspired by metaphors from software engineering, SCRAM extends high-level deep learning development tools to interpret model metrics during training and produce human-readable error messages. We validate SCRAM through three author-created example scenarios with image and text datasets, and by collecting informal feedback from ML researchers with teaching experience. We finally reflect upon our feedback for the design of future ML debugging tools.

Author Keywords
Machine Learning; Debugging; Tutorial Systems; Interactive Visualization.

CCS Concepts
•Human-centered computing → Interactive systems and tools;

Introduction

Many domain experts, hobbyists, and makers wish to adopt Machine Learning (ML) models such as neural networks into their applications, but lack formal training in ML. These users often have some programming expertise and their own novel datasets for a particular domain problem. For example, a farmer may want to classify the types of cucumbers from their farm, or an independent app developer may want to recommend workouts in their fitness app [35, 38].

Several Deep Learning (DL) toolkits, including Keras [7] and Apple Create ML [15], make these tasks more approachable by providing high-level APIs which reduce the effort needed to preprocess data, train, and evaluate DL models (neural networks). However, when non-expert ML developers use these APIs to train these models on novel datasets, the models can produce unexpected output without explicitly throwing errors.

While experts can rely on experience and tools such as TensorBoard [1] and tfdbg [6] to inspect and correct model behavior, non-experts often lack the theoretical and practical knowledge to interpret results from these tools [14, 5] and could benefit from explanations and guidance through this unstructured process [3, 33].

We introduce SCRAM, a prototype system which can interpret potential error conditions in the DL training phase and provide descriptive, actionable warning messages to help users debug and produce well-trained models (see Figure 1). SCRAM draws inspiration from tools in software engineering which inspect code to provide warning messages and suggestions to developers. Our goal is to develop a system that can encode this tacit knowledge of experts into heuristics which check model output over time during training. This system will guide non-expert users to correct errors with human-readable error messages that explain

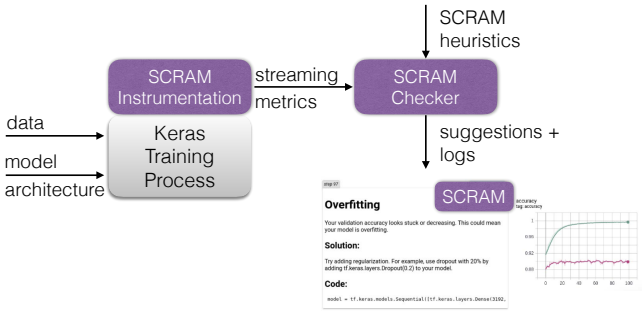

Figure 1: The Keras framework outputs data batches and model metrics to SCRAM (left), and SCRAM outputs error messages and visualizations to Tensorboard (right).

best practices and code recipes to bridge theoretical and practical knowledge gaps. During the tuning phase, users interpret these error messages to make changes to hyperparameters, correcting model behavior.

In this paper, we describe the SCRAM prototype; share three heuristics for detecting common problems during neural network training; and validate error messages produced by SCRAM in three author-created scenarios with experienced ML instructors for future iterations of our system.

Related Work

Interactive ML Development

HCI research has produced novel interfaces which allow users to interactively train and tune ML models as early as 2003 [9]. Gestalt is toolkit which adds structure to the ML development process, allowing developers to iteratively modify and analyze their models in an IDE [32]. Makers can alternatively use ESP to interactively train and deploy gesture recognition models on Arduino hardware [26]. While these tools support the feature engineering workflow re-

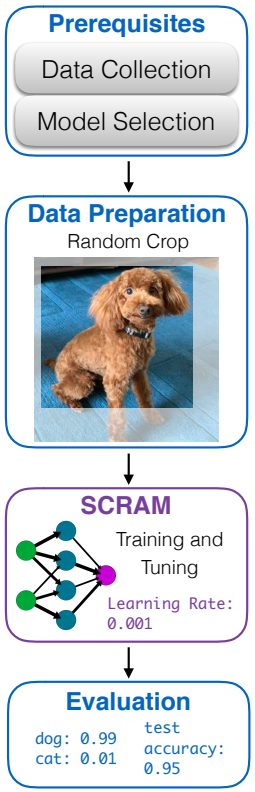

Figure 2: Once a dataset is collected and an ML algorithm is selected, users must (1) preprocess data, (2) train and tune their model, and (3) evaluate their model on test data.

quired for classical ML, SCRAM focuses on training and tuning deep neural networks, which instead learn features from the input data itself. These DL models enable powerful contemporary applications in domains where manual feature selection is infeasible (e.g., image/speech recognition), but come with the trade-off of an extended and hard-to-interpret training process. ML practitioners can add instrumentation and visualizations to their DL models using toolkits such as TensorWatch [36] and Lucid [30], but the choice of visualization and its interpretation requires expertise. SCRAM uses heuristics to produce text error messages which can guide novices in the DL debugging process. One recent commercial tool may help non-experts fine-tune pre-trained models, but does not interpret model output [2]. SCRAM integrates with a open source Python framework, Keras [7], which has a large support community and can provide more advanced functionality as needed.

Software Engineering Support Tools

The software engineering community has created a variety of tools which produce useful warnings and suggestions to guide software development. SCRAM draws upon established paradigms in software engineering such as linting [17], unit testing, dynamic analysis [28], and explanation-based debugging [21] to help users interpret the behavior and inspect the points of failure of their ML applications. We draw additional inspiration from software visualization [39] and tutorial systems for complex user interfaces [12] which guide novices through complex tasks.

Model Visualization and Inspection Tools

Research and engineering teams have produced novel interfaces to compare model performance [27]. Because of the intrinsic relationship between training data and a model, these tools can highlight relevant training data contributing to outliers [31] and refine the model itself [4]. TensorFuzz can further assist debugging by adapting coverage-based fuzzing to identify model inputs which generate numerical errors [29]. Other tools track and visualize test results to help select models for large-scale deployments [23, 16, 34]. Evaluating the performance of ML models is a critical step, but depends on having an already trained model. SCRAM assists users in the training step required *before* evaluation.

Explanations and Interpretability

SCRAM is inspired by systems which help practitioners *interpret* the output and behavior of their ML models. Deep neural networks often have too many parameters to easily understand, and explaining their output is an active area of research [10]. Activation Maps highlight the parts of an input image used to make a prediction [30]. A more recent algorithm, Concept Activation Vectors (CAV), can explain the higher-level concepts used in an output classification [20]. Training and tuning neural networks similarly produces output which is difficult to interpret [18, 19], relying on tacit knowledge and expertise to understand [14]. We believe SCRAM is an early step in providing explanations of neural network output during the training process.

A key component of SCRAM is an automated checking infrastructure that enables running tests over model runtime behavior to flag problems for non-expert users. Other systems in HCI research use this approach to assist debugging electrical circuits and embedded systems [8, 25]. SCRAM adapts this approach to ML debugging.

Design Considerations

SCRAM targets scenarios when users have an existing problem formulation for applying ML to their applications. In these cases, a novel dataset has already been collected and a neural network architecture chosen. The remain-

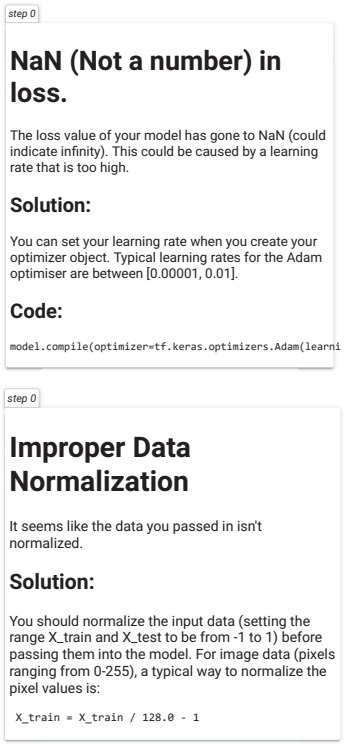

Figure 3: Error messages produced by SCRAM explain high-level concepts as well as suggest code snippets.

ing steps, shown in Figure 2, are: (1) *data preparation*, in which the data are split into training and tests sets, normalized, and formatted for input; (2) *training and tuning*, where model hyperparameters are tuned during training to help the model fit the data; and (3) *evaluation*, in which model performance is tested and compared. SCRAM focuses on guiding users through the second phase, *training and tuning*. During this phase, hyperparameters, such as the optimizer learning rate or model regularization, tune how the model fits batches of data, so it can generalize accurate predictions to new input data. However, this phase can produce confusing output, leading many users to even abandon applying ML altogether [5]. During tuning, experts rely on tacit knowledge to interpret model output, e.g., by visually inspecting the model loss and accuracy curves, or running small scale tests [18, 19].

We chose to integrate our system with existing, popular DL frameworks, Keras [7], and Tensorboard [1]. Keras is a popular choice for ML novices because it requires little code to construct and train neural networks, but its capabilities can also expand to meet advanced needs such as those of ML researchers. SCRAM outputs plots, suggestions, and error messages to TensorBoard, a visualization framework built for DL instrumentation that also integrates with Keras. Tensorboard supports real-time data loading during training as well as keeping track of older runs. To add SCRAM support to a Keras program with a Tensorboard, users only need to make a one-line code change.

Using SCRAM

Sam, a molecular biologist, wants to count the number of Gram positive bacteria in samples taken from an experiment on microscope slides. They already have access to thousands of annotated photos from previous experiments, and wish to repurpose a pretrained object detection neural network to count the bacteria in new samples. Formatting the dataset is easy, but once training begins, the model loss seems to increase, then reach NaN. A quick internet search turns up a Twitter thread [1] suggesting a lower learning rate. With the learning rate corrected, the model begins training, but the validation accuracy is much lower than expected and doesn't seem to be improving. Sam tweaks multiple parameters of the network and optimizer, but nothing seems to work. An ML engineer friend takes a quick pass over the code, but doesn't see anything obviously wrong and suggests using SCRAM. SCRAM detects that some input data points are reaching values as high as 255, and produces an error message stating the training data isn't being normalized properly. The message also suggests a code snippet to show how to normalize the training data to the model's expected input distribution, between -1 and 1. After Sam implements the suggested snippet, the model's accuracy increases rapidly. Sam verifies the model's correctness on test data. The model is integrated into Sam's lab workflow, saving hours of cell-counting time.

Implementation

SCRAM hooks into the built-in callback mechanism of Keras, which can invoke actions during model training. During training runtime, data batches and model metrics (loss and accuracies) are fed into SCRAM, where they are logged and checked against a list of heuristics to produce error messages. Checks are loaded individually, and can be swapped and customized as needed within SCRAM. Error messages and metrics are emitted directly to Tensorboard via the Tensorflow Summary API. (See Figure 1)

Model Checking Heuristics and Error Messages

SCRAM currently includes three heuristics which can identify common problems novices face when training their neu-

[1] https://twitter.com/karpathy/status/1013244313327681536

Heuristic	Description	Detection Method
Overfitting	When a model too closely fits its training data, it loses its ability to generalize to new data.	Check if validation accuracy decreases over two epochs while training accuracy increases, which likely indicates overfitting [18].
Improper Data Normalization	In transfer learning, new data should be normalized to a similar range as the original data the model was trained on.	Check if the values of input features of current batch lie within the conventional range of $[-1, 1]$ [37].
Unconventional Hyperparameters	Some hyperparameters for training deep models significantly affect model performance [11]. For instance, using too high of a learning rate will cause the model to produce NaN loss.	Check if the loss value reaches NaN, which indicates a possible incorrect range of hyperparameters [19].

Table 1: SCRAM heuristics and associated detection methods.

ral networks (see Table 1). We collect these heuristics from research literature, course notes, and tutorials from ML experts [19, 18, 37, 11]. While these heuristics cover several common scenarios novices may encounter during training, there are many others which could be implemented in the future as well. Each heuristic has an associated checking function that tests if collected metrics violate the heuristic and an associated error message which is authored to give general theoretical advice as well as practical code snippets that can be used.

Inital User Experiences

To explore the utility of SCRAM, we constructed 3 example scenarios of errors with 3 different datasets: CIFAR-10 [22], Fashion-MNIST [40], and Large Movie Reviews [24]. Each scenario is inoculated with a potentially faulty model setup to generate errors from SCRAM: we use a large fully-connected network without any regularization to overfit on Fashion-MNIST; we set the model learning rate to 1e10 for Large Movie Reviews; and we use unnormalized pixels of the images in CIFAR-10 directly for model training.

To understand how SCRAM may help novices, we solicited feedback from 2 ML researchers with experience teaching introductory ML courses. We showed them our example scenario notebooks and allowed them to interact with the training code and error messages produced by SCRAM. Sessions lasted under half an hour each.

Both participants stated the notifications would be useful to novices, and that the heuristics capture common problems encountered by non-expert ML developers. One participant expressed its potential use to experienced ML developers—since training with large datasets may take days or weeks, notifications produced by SCRAM could direct attention to model training when needed. One participant remarked that

SCRAM can catch errors that might not even be detected by a novice at all, such as normalization. Both participants expressed interest in adding other heuristics, such as predicting when a batch size may be too large to fit in memory (usually resulting in an error and program interruption). Other suggestions were for tightening integration between SCRAM and the code itself, by differentiating warnings and errors for conditions that can break execution during runtime, or by identifying the particular lines of code that generated the error (e.g., which part of the model generated a NaN output). Finally, one participant remarked that debugging strategies aren't often taught in ML courses, and SCRAM could serve as an instructional aid.

Future Work and Conclusion

SCRAM represents a first step in making the neural network training and tuning process more manageable, thus making applying ML more approachable to non-experts. Beyond adding additional heuristics, we are excited to continue work on SCRAM in the following areas:

Dynamic Error Messages: Error messages produced by SCRAM are written to apply to general cases, and provide explanations to help users narrow down the root cause and implement fixes. Dynamically generated error messages such as those produced by software tutorial systems [13] could steer users closer to identifying the root causes of error conditions. Future iterations of SCRAM could check for finer numerical conditions or perhaps learn from examples to dynamically generate help messages.

Code-Aware Tutorial Content: Making the error messages from SCRAM interactive could significantly improve its use as a tutorial system. For example, SCRAM could highlight specific lines of user code or Tensorboard visualizations. Another potential approach could be gleaned from the Java Whyline, which allows users to ask questions about program output during runtime to identify bugs [21].

Integrating Active Tests with SCRAM: Further engineering work could enable SCRAM to run *static* checks of the ML program, enabling many more heuristics (e.g., checking initialization). SCRAM could also be extended to execute operations with the model, such as overfitting on small batches of data or running user-defined unit tests.

Controlled User Evaluation of SCRAM: Our exploratory validation of SCRAM had a limited number of participants and was conducted with experienced instructors, not target users directly. A controlled user evaluation of the next iteration of SCRAM would determine the effectiveness of its heuristics and error messages. One possible experiment design could be that of Gestalt [32], in which novices were asked to debug ML models inoculated with errors in randomized conditions.

Communicating Uncertainty of Heuristics: The heuristics used by SCRAM are designed to detect and explain common errors, but these explanations are assumptions of model behavior and may not always be applicable. To mitigate this, the language of the messages are adapted to convey this intrinsic uncertainty, guiding the user to consider multiple possible underlying root causes and offering different solutions to mitigate them.

Recent advances in ML research have impacted numerous aspects of daily living, from transportation to healthcare to entertainment. We believe that artists, makers, domain experts, software engineers, and scientists can benefit from these advances by introducing these powerful tools to understanding and exploring their domain-specific data.

Acknowledgements

We wish to thank Philippe Laban, Mike Laielli, James Smith, and Andrew Head for their valuable insight in creating SCRAM. Thanks also to Ramon for modeling for Figure 2.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 265–283.

[2] Runway AI. 2019. Runway ML. (2019). https://runwayml.com/

[3] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19)*. IEEE Press, 291–300. DOI: http://dx.doi.org/10.1109/ICSE-SEIP.2019.00042

[4] Saleema Amershi, Max Chickering, Steven M. Drucker, Bongshin Lee, Patrice Simard, and Jina Suh. 2015. ModelTracker: Redesigning Performance Analysis Tools for Machine Learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. Association for Computing Machinery, New York, NY, USA, 337–346. DOI: http://dx.doi.org/10.1145/2702123.2702509

[5] C. J. Cai and P. J. Guo. 2019. Software Developers Learning Machine Learning: Motivations, Hurdles, and Desires. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 25–34. DOI: http://dx.doi.org/10.1109/VLHCC.2019.8818751

[6] Shanqing Cai. 2017. Debug TensorFlow Models with tfdbg. (Feb 2017). https://developers.googleblog.com/2017/02/debug-tensorflow-models-with-tfdbg.html

[7] François Chollet. 2015. keras. https://github.com/fchollet/keras. (2015).

[8] Daniel Drew, Julie L. Newcomb, William McGrath, Filip Maksimovic, David Mellis, and Björn Hartmann. 2016. The Toastboard: Ubiquitous Instrumentation and Automated Checking of Breadboarded Circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. Association for Computing Machinery, New York, NY, USA, 677–686. DOI: http://dx.doi.org/10.1145/2984511.2984566

[9] Jerry Alan Fails and Dan R. Olsen. 2003. Interactive Machine Learning. In *Proceedings of the 8th International Conference on Intelligent User Interfaces (IUI '03)*. Association for Computing Machinery, New York, NY, USA, 39–45. DOI: http://dx.doi.org/10.1145/604045.604056

[10] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. 2018. Explaining Explanations: An Overview of Interpretability of Machine Learning. (2018).

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

[12] Tovi Grossman, George Fitzmaurice, and Ramtin Attar. 2009. A survey of software learnability: metrics, methodologies and guidelines. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 649–658.

[13] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann. 2015. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 3–12. DOI: http://dx.doi.org/10.1109/VLHCC.2015.7356972

[14] C. Hill, R. Bellamy, T. Erickson, and M. Burnett. 2016. Trials and tribulations of developers of intelligent systems: A field study. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 162–170. DOI: http://dx.doi.org/10.1109/VLHCC.2016.7739680

[15] Apple Inc. 2019a. Apple Create ML. (2019). https://developer.apple.com/machine-learning/create-ml/

[16] Databricks Inc. 2019b. MLFlow. (2019). https://mlflow.org/

[17] S. C. Johnson. 1978. Lint, a C Program Checker. In *Technical Report*. Bell Telephone Laboratories, 78–1273.

[18] Andrej Kaparthy. 2016. Training Neural Networks, Part 1. *Convolutional Neural Networks for Visual Recognition. Lecture Slides* (20 January 2016). http://cs231n.stanford.edu/2016/syllabus.html

[19] Andrej Kaparthy. 2019. A Recipe for Training Neural Networks. (Apr 2019). https://karpathy.github.io/2019/04/25/recipe/

[20] Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas, and Rory Sayres. 2017. Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV). (2017).

[21] Andrew J. Ko and Brad A. Myers. 2009. Finding Causes of Program Output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. Association for Computing Machinery, New York, NY, USA, 1569–1578. DOI: http://dx.doi.org/10.1145/1518701.1518942

[22] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[23] Lezhi Li and Yang Wang. 2019. Manifold: A Model-Agnostic Visual Debugging Tool for Machine Learning at Uber. (Aug 2019). https://eng.uber.com/manifold/

[24] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Portland, Oregon, USA, 142–150. http://www.aclweb.org/anthology/P11-1015

[25] Will McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. 2017. Bifröst: Visualizing and Checking Behavior of Embedded Systems across Hardware and Software. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. Association for Computing Machinery, New York, NY, USA, 299–310. DOI: http://dx.doi.org/10.1145/3126594.3126658

[26] David A. Mellis, Ben Zhang, Audrey Leung, and Björn Hartmann. 2017. Machine Learning for Makers: Interactive Sensor Data Classification Based on Augmented Code Examples. In *Proceedings of the 2017 Conference on Designing Interactive Systems (DIS '17)*. Association for Computing Machinery, New York, NY, USA, 1213–1225. DOI: http://dx.doi.org/10.1145/3064663.3064735

[27] Sugeerth Murugesan, Sana Malik, Fan Du, Eunyee Koh, and Tuan Lai. 2019. DeepCompare: Visual and Interactive Comparison of Deep Learning Model Performance. *IEEE Computer Graphics and Applications* PP (05 2019), 1–1. DOI: http://dx.doi.org/10.1109/MCG.2019.2919033

[28] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). Wiley Publishing.

[29] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, Long Beach, California, USA, 4901–4911. http://proceedings.mlr.press/v97/odena19a.html

[30] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. 2017. Feature Visualization. *Distill* (2017). DOI:http://dx.doi.org/10.23915/distill.00007 https://distill.pub/2017/feature-visualization.

[31] Google PAIR. 2019. What-if Tool. (2019). https://pair-code.github.io/what-if-tool/

[32] Kayur Patel, Naomi Bancroft, Steven M. Drucker, James Fogarty, Andrew J. Ko, and James Landay. 2010. Gestalt: Integrated Support for Implementation and Analysis in Machine Learning. In *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology (UIST '10)*. Association for Computing Machinery, New York, NY, USA, 37–46. DOI:http://dx.doi.org/10.1145/1866029.1866038

[33] Kayur Patel, James Fogarty, James A. Landay, and Beverly Harrison. 2008. Investigating Statistical Machine Learning as a Tool for Software Development. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. Association for Computing Machinery, New York, NY, USA, 667–676. DOI: http://dx.doi.org/10.1145/1357054.1357160

[34] Daniel Crankshaw Neeraja Yadwadkar Joseph Gonzalez Rolando Garcia, Vikram Sreekanti. 2019. flor. (2019). https://github.com/ucbrise/flor

[35] Kaz Sato. 2016. How a Japanese cucumber farmer is using deep learning and TensorFlow. (Aug 2016). https://cloud.google.com/blog/products/gcp/how-a-japanese-cucumber-farmer-is-using-deep-learning-and-tensorflow

[36] Shital Shah, Roland Fernandez, and Steven M. Drucker. 2019. A system for real-time interactive analysis of deep learning training. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2019, Valencia, Spain, June 18-21, 2019*. 16:1–16:6. DOI: http://dx.doi.org/10.1145/3319499.3328231

[37] Jonathan R. Shewchuk. 2019. Concise Machine Learning. (May 2019). https://people.eecs.berkeley.edu/~jrs/papers/machlearn.pdf

[38] Tom Simonite. 2018. The DIY Tinkerers Harnessing the Power of Artificial Intelligence. (Nov 2018).

https://www.wired.com/story/diy-tinkerers-artificial-intelligence-smart-tech/

[39] John T Stasko, Marc H Brown, and Blaine A Price. 1997. *Software Visualization*. MIT press.

[40] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. (2017).