# UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior

Eldon Schoop
eschoop@berkeley.edu
UC Berkeley
Berkeley, California, USA

Forrest Huang
forrest_huang@berkeley.edu
UC Berkeley
Berkeley, California, USA

Björn Hartmann
bjoern@berkeley.edu
UC Berkeley
Berkeley, California, USA

## ABSTRACT

Training deep neural networks can generate non-descriptive error messages or produce unusual output without any explicit errors at all. While experts rely on tacit knowledge to apply debugging strategies, non-experts lack the experience required to interpret model output and correct Deep Learning (DL) programs. In this work, we identify DL debugging heuristics and strategies used by experts, and use them to guide the design of Umlaut. Umlaut checks DL program structure and model behavior against these heuristics; provides human-readable error messages to users; and annotates erroneous model output to facilitate error correction. Umlaut links code, model output, and tutorial-driven error messages in a single interface. We evaluated Umlaut in a study with 15 participants to determine its effectiveness in helping developers find and fix errors in their DL programs. Participants using Umlaut found and fixed significantly more bugs compared to a baseline condition.

## CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**; • **Computing methodologies** → Machine learning; • **Software and its engineering** → *Software testing and debugging*.

## KEYWORDS

ML Debugging, ML Development, End-User ML

**ACM Reference Format:**
Eldon Schoop, Forrest Huang, and Björn Hartmann. 2021. UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3411764.3445538

## 1 INTRODUCTION

The surge of interest in Machine Learning (ML) has resulted in groundbreaking advances in many domains, from healthcare [37, 54], to transportation [72], to entertainment [1]. An enabling factor of these applications are Deep Neural Network (DNN) models, which can extract and discriminate features from raw data by using massive amounts of learned parameters [9]. These "Deep Learning" (DL) approaches are incredibly powerful, even surpassing human-level accuracy on some tasks [23]. DNNs also enable many new interactions over "Classical ML", such as generating high-dimensional data [20], and supporting *transfer learning*, where selected parameters from a DNN may be retrained to generalize to new applications, creating high performing models without needing millions of data points or massive computational resources.

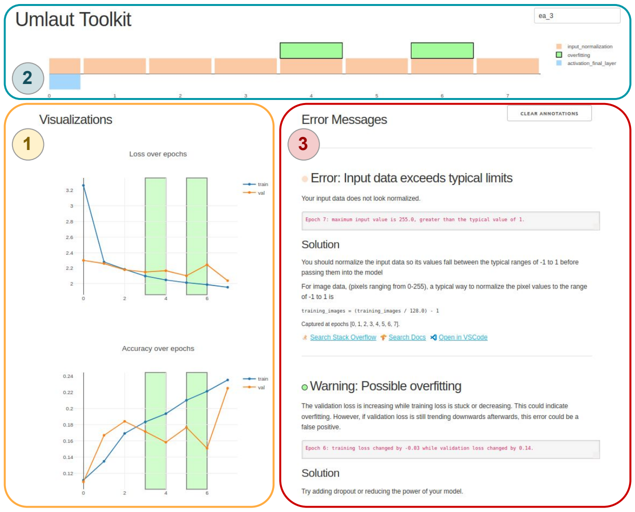
Non-expert ML programmers, such as software engineers, domain experts, and artists, can use transfer learning to create their own models by using recent frameworks which make this task more approachable with high-level APIs [12, 34]. However, when bugs are introduced, the default failure mode of DL programs is to produce unexpected output without explicit errors [70]. ML novices often expect models to behave as APIs, and have limited mental models to facilitate debugging, sometimes even abandoning ML approaches altogether when they fail [10, 29]. Further compounding the issue, DNNs are considered "black box" models, and cannot be debugged with traditional means such as breakpoints. Experts rely on their experience and tools such as Tensorboard [2] and tfdbg [11] to begin inspecting model behavior, but often fall back on trial-and-error approaches guided by intuition [10]. Adding structure to the DL development process through explanations and guidance could help users close this debugging loop and bridge theory with practice [3, 62].

We introduce Umlaut, the Usable Machine LeArning UTility, a system which uses a multifactor approach to assist non-experts in identifying, understanding, and fixing bugs in their DL programs[1] (Figure 1). Umlaut draws inspiration from tools and metaphors in software engineering which inspect code to provide warning messages and suggestions to developers. This includes linting [38], unit testing, dynamic analysis [56], and explanation-based debugging [45]. Umlaut attaches to the DL program runtime, running heuristic checks of model structure and behavior that encode the tacit knowledge of experts. Umlaut then displays results of checks as error messages that integrate program context, explain best practices, and suggest code recipes to address the root cause(s). Our aim is not to define new heuristics or outperform experts, but to show how existing heuristics used by experts can be automatically checked and made accessible to a broader set of users.

A key objective of Umlaut is to support users in overcoming three critical "gulfs" of the DL debugging process: mapping from symptoms to their root cause(s), choosing a strategy to address the underlying problem, and mapping from strategy to concrete code implementation. Umlaut uses an automated checking infrastructure to detect errant model behavior and raise error messages reflecting the surrounding context. Error messages are presented

*CHI '21, May 8–13, 2021, Yokohama, Japan*
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8096-6/21/05.
https://doi.org/10.1145/3411764.3445538

[1] Source code for our system is available at https://github.com/BerkeleyHCI/umlaut



Error: Input data exceeds typical limits

Your input data does not look normalized.

```
Epoch 7: maximum input value is 255.0, greater than the typical value of 1.
```

Solution

You should normalize the input data so its values fall between the typical ranges of -1 to 1 before passing them into the model

For image data, (pixels ranging from 0-255), a typical way to normalize the pixel values to the range of -1 to 1 is

```
training_images = (training_images / 128.0) - 1
```

Captured at epochs [0, 1, 2, 3, 4, 5, 6, 7].

Search Stack Overflow Search Docs Open in VSCode

Warning: Possible overfitting

The validation loss is increasing while training loss is stuck or decreasing. This could indicate overfitting. However, if validation loss is still trending downwards afterwards, this error could be a false positive.

```
Epoch 6: training loss changed by -0.03 while validation loss changed by 0.14.
```

Solution

Try adding dropout or reducing the power of your model.

**Figure 1: The Umlaut web interface combines visualizations of model metrics (1); a timeline showing errors over epochs (2); and explanations of underlying error conditions with the program context and suggestions for best practices with code examples (3). Plots and the timeline are automatically annotated with with relevant data when errors are clicked.**

in a web interface that tightly couples errors with visualizations of model output, linking root causes to their symptoms. Error messages include descriptions of their underlying theoretical concepts, and suggest potential debugging strategies to bridge theoretical and practical knowledge gaps. To translate these strategies into actionable changes, Umlaut errors include code recipes which implement suggested fixes, outbound links to curated Stack overflow and documentation searches, and links to the suspect lines of code in source.

Our work makes the following contributions:

- A discussion of opportunities for supporting the DL debugging process, in contrast to Classical ML, through novel user interfaces
- A novel approach of encoding expert heuristics into computational checks of DL program structure and DL model behavior
- The Umlaut system, a tool which implements several automatic checks to assist in finding, understanding, and fixing bugs in Keras programs
- An evaluation which shows Umlaut helps non-expert ML users find and fix significantly more bugs in DL applications.

## 2 BACKGROUND: CHALLENGES IN DEEP LEARNING (DL) DEVELOPMENT

The recent success of deep learning in a variety of domains has led to an increase in users of DL, and a corresponding growth of tools that have emerged to help developers with DL workflows. This section summarizes background information and design considerations for tools that aim to aid the DL development process.

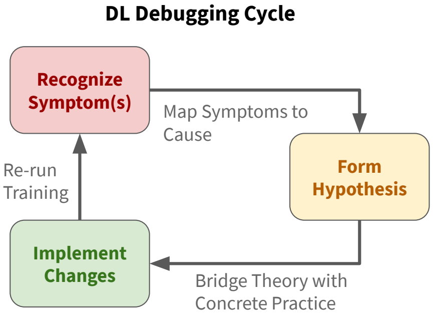


**Figure 2: To debug DL programs, users first recognize symptoms from errant model behavior or code structure. Experts use mental models built from experience to translate from these symptoms to hypotheses of underlying root causes. Finally, code changes are implemented to test the underlying hypotheses, and training is rerun to check them.**

### 2.1 Key Differences of Designing for DL over Classical ML

Both “classical” and “deep” ML development processes are often exploratory [62], where the data, model, and scaffold code are iteratively refined to reach target benchmarks [29]. However, there are critical differences between the implementation of classical ML and DL approaches which significantly alter the developer experience. While classical ML can be effectively applied to many problems, DL can handle high-dimensional, unstructured input and output spaces, such as object detection and audio-cue detection. We characterize the fundamental differences between classical ML and DL in this section and introduce a unique set of challenges that DL support tools should address.

*Data Requirements:* Both DL and classical ML models require ground truth labeled data to train. However, DNNs often require significantly more data: a rule of thumb suggests a minimum of 5,000 samples per class [19], while classical algorithms such as SVM or Random Forests require far fewer data points. Handling large-scale datasets drives up costs for data collection and processing, particularly in domains with noisy or incomplete data [54].

*Featurization:* Classical ML algorithms require hand-engineered features to maximize signal from input data. In contrast, DNNs learn features from patterns in the data directly, eliminating the developer-driven feature engineering step [3, 19]. While this provides DNNs tremendous flexibility in handling unstructured input data, this offers less control and means developers cannot verify whether the model has received “features” from extraneous patterns in the data that confound the effectiveness of the model.

*Interpretability:* A key feature of classical ML algorithms are that they are often more interpretable than DNNs. While there are many meanings and subsets of model interpretability, it is widely accepted that we do not yet fully understand the exact rules and features that DNNs rely on to produce specific outputs, and how well DNNs generalize to new problems [48]. This makes pinpointing the exact source of numerical errors in DNNs very difficult and gives rise to “silent errors” in the model that can only be spotted by experts with experience pattern-matching code smells to possible errors [78]. Visualization has been a critical tool in interpreting DNN behavior, but this still remains an open research question [31]. In contrast, some classical approaches intrinsically attribute the hand-engineered features most relevant to any prediction.

*Training:* Unlike classical ML algorithms, DNNs require nonconvex optimization of a large number of parameters. This requires proper initialization of neural network weights [24] and an involved search process for network hyperparameters [6]. DNN training time can take days or weeks, often even requiring online tuning in order to converge [69], lengthening the feedback loop. Experts rely on experience to determine an 'educated guess' of the typical range of hyperparameters which can drastically decrease the search space. Novices encounter difficulty in this process, especially when it generates unknown or ambiguous symptoms.

*Transfer Learning:* DNNs allow developers to reuse the "feature-picking" parts of the NN, and "fine-tune" the bottom layers to use those feature for new domains and applications. A common interaction is fine-tuning a model trained on many images to a new, smaller, dataset.

## 2.2 Detecting Errors during DL Training and Evaluation

To show how UMLAUT fits in the DL development process, we identify four high-level stages of DL development from prior work [3, 61]: (1) Data Processing, (2) Training and Tuning, (3) Evaluation; and (4) Deployment. We focus on the challenges that DL developers face in Phases (2) and (3).

Typical DL workflows require developers to iteratively train and evaluate their models to identify bugs and modeling issues [3, 70]. We characterize this debugging process using the *DL debugging cycle* shown in Figure 2. During this cycle, developers repeatedly train models with a specific experimental setup of network architectures, loss functions, and hyperparameters. The model performance is then evaluated by qualitatively inspecting the classification results of various data examples, and quantitatively by calculating accuracy on a validation set. Using the results generated by the training run, developers recognize symptoms, form hypotheses to the root causes of problems, and make decisions to modify the experimental setup using their theoretical understanding of the models. They will then re-run the experiment and this cycle continues until developers obtain a model with satisfactory performance.

Debugging DL models is challenging because even though errors occur in both the training and evaluation phase, the symptoms often only materialize in the evaluation phase in the form of poor model performance [3]. While experts often rely on a continuously refined set of best practices that pattern-match model outputs to effective modifications, novices often think of DL models as black boxes and can have difficulty in recognizing and understanding symptoms [3, 10, 29].

## 2.3 Mapping Symptoms to Root Causes

One critical step in the DL debugging cycle (Figure 2) is to map modeling issues from symptoms to their root causes. This step requires developers to analyze model outputs and training curves, classify specific issues from these statistics, and convert them into actionable items. Current error mitigation practices are often ad hoc, such that developers usually only have tools that document performance metrics and general theory resources, but are required to manually draw connections between them. For example, a developer might need to consult best practices collected from literature, expert blogs, and academic lectures [7, 39, 40, 66] to derive a set of actionable items that resolve their issues. Developing this skill requires extensive time and exposure to errors at different stages and levels of abstraction: the program, theory, data, etc. These skills are essential for successfully training a model with high performance, yet helping novices gain the tacit knowledge needed to successfully diagnose and debug model issues remains an open challenge [5].

# 3 RELATED WORK

We map prior work in three axes that correspond to Section 2 based on their contributions, and discover design opportunities for UMLAUT in complementary areas.

## 3.1 Interfaces for Supporting Classical Machine Learning Workflows

HCI research has produced novel interfaces which allow users to interactively train and tune ML models as early as 2003 [14, 15, 42, 50]. Gestalt is a toolkit which adds structure to the ML development process with an IDE [61]. Makers can use ESP to interactively train and deploy gesture recognition models on hardware [52]. Other works help compare DL model performance, but only once the models are trained [55]. While these tools support the feature engineering workflow required for classical ML, UMLAUT focuses on training and tuning DNNs. DNNs instead learn features from input data and enable powerful new applications.

## 3.2 Tools for comparing and improving DL Model Performance

Research and engineering teams have produced novel interfaces to compare model performance [55, 71, 76] and subsequently debug modeling issues. Because of the intrinsic relationship between training data and a model, these tools can highlight relevant training data contributing to outliers [30, 60, 74] and refine the model itself [4]. Taking steps towards debugging these issues, TensorFuzz adapts coverage-based fuzzing to identify model inputs which generate numerical errors [57].

In addition, UMLAUT is inspired by a field of academic research in Explainable Artificial Intelligence (XAI) which help practitioners *interpret* the output and behavior of their ML models. DNNs often have too many parameters to easily understand, and explaining their output is an active area of research [17]. Methods like Saliency Maps can highlight the specific parts of an input image used to make a prediction [41, 58], while Concept Activation Vectors (CAV) can explain the higher-level concepts used [43].

Evaluating the performance of ML models is a critical step, but all of the aforementioned prior work depends on having an already-trained model. UMLAUT assists users in the training step required *before* evaluation. We believe UMLAUT is an early step in both debugging and providing explanations of neural network output during the training process.

## 3.3 Prescribing Best Practices and Code Changes in Context

As mentioned in Section 2, current tools mostly help inform code changes in DL development workflows by tracking and instrumenting experiments for large-scale deployments [35, 47, 63]. While these tools are critical for developers to track the progress of their experiments, they typically do not directly report any potential errors. ML practitioners can also add instrumentation and visualizations to their DL models using toolkits such as TensorWatch [65] and Lucid [58], but the choice of visualization and its interpretation requires expertise.

Several studies conduct empirical analyses of bugs found in ML programs using data from Stack Overflow and GitHub [33, 36, 77, 78]. These works create a high-level classification of common bugs, but don't link between symptoms, root causes, and actionable items in context. On the other hand, some tools in research [7, 64] and deployment (such as EarlyStoppingHooks [12]) use algorithmic checks for training. However, while these actions are taken in context during training, they do not produce error messages, link to root causes, or tie back to other information (e.g. learning curves).

Inspired by work in supporting traditional software development [8, 16, 18, 22], Umlaut also suggests code examples from official documentation and best practices pulled from StackOverflow, which helps users to directly address errors and dive deeper into the code. Umlaut builds upon established paradigms in software engineering such as linting [38], unit testing, dynamic analysis [56], and explanation-based debugging [26, 45]. Umlaut works *in context* to help users interpret the behavior and inspect the points of failure of their ML applications [28], as similar paradigms have not been extensively explored for DL development. We draw additional inspiration from software visualization [68] and tutorial systems for complex user interfaces [21]. Umlaut also adapts an automated-checking infrastructure that enables running tests over model runtime behavior to flag problems for non-expert users. This approach has been used in other HCI research to assist debugging electrical circuits and embedded systems [13, 51].

# 4 DEBUGGING ML PROGRAMS WITH UMLAUT

To use Umlaut, users attach a Umlaut client to their program, which injects static and dynamic heuristic checks on the program, parameters, model structure, and model behavior. Violated heuristics raise error flags which are propagated to a web-based interface that uses interlinked visualizations, tutorial explanations, and code snippets to help users find and fix errors in their model. Umlaut also emits flagged error messages to the command line, inline with Keras training output, to reduce context switching. Heuristics, errors, and their implementation are described further in Section 6.

To illustrate how Umlaut works in practice, consider Jordan, a park ranger who wants to receive a notification when rare birds appear in a bird feeder camera. Jordan has domain expertise in ornithology and birding, and has taken an online data science course, but they are not an ML expert. Jordan was able to prepare a labeled dataset of birds at the feeder using previous recordings, and they found a template project from the data science course to use a pretrained Resnet image classification model [25] for transfer learning. Jordan's next step is to fine-tune the pretrained model on their new dataset. After fixing the input image shapes from a bug produced by Keras, Jordan is able to get the training loop to run. The loss is now decreasing, and accuracy rising, but only to 60%–not enough for their application. Jordan manages to contact their former data science instructor, who volunteers a quick look at the program, but can't seem to find anything wrong. Jordan hears that Umlaut can help detect and fix bugs in DL programs, and gives it a try.

## 4.1 Importing Umlaut and Creating a Session

*To use Umlaut, Jordan adds three simple lines of code to import and attach it to their program: they import the Umlaut package, pass the model and other inputs to the Umlaut object, and add the Umlaut callback to the training loop.*

A key design principle of Umlaut is to ensure it integrates smoothly into existing DL frameworks and development tools. We choose to integrate Umlaut into the Keras API of Tensorflow 2, because of its high-level API and its broad community support. At runtime, the Umlaut client adds a callback and injects shims into the Keras training routine. While the model is training, the client runs several heuristic checks, sending metrics and raised errors to a Umlaut server through a named session. Colliding session names are appended with an auto-incremented integer.

## 4.2 User specification of Umlaut checks

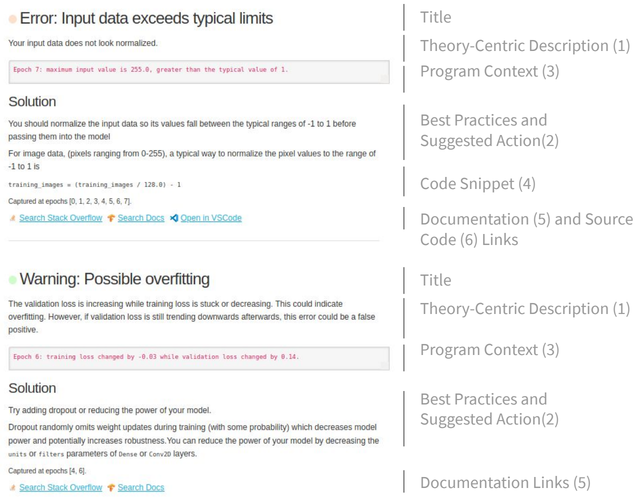
*Before running Umlaut with their training loop, Jordan tells Umlaut that their model expects images as input and a sparse vector output indicating the predicted class by passing the* `inputtype='image'` *and* `outputtype='classification'` *arguments to their Umlaut call. These flags tell Umlaut to run additional checks (e.g., ensuring the input dimensions are consistent with image formats and that a softmax layer is used on the output).*

Users can supply arguments to Umlaut which specify the expected input and output formats of the model, reflecting the high-level problem statement. Umlaut supports image or sparse text inputs, and classification or regression outputs. Depending on the user's guidance, Umlaut selects different checks to run based on the input, and alters the content of output error messages (e.g., ensuring an RGB color image has 3 dimensions and is normalized, or that a classification loss function such as cross entropy is not used for a regression output). This specification is optional, but leads to more detail in error messages and a wider selection of checks. This is a novel interaction for DL debugging, and can be used to ensure the model architecture and data preparation match with the intended problem type.

## 4.3 Actionable Error messages

*When Jordan runs Umlaut with a training session, they see some errors appear in the web interface. They first turn to an error marked as Critical ("Missing Activation Functions").*

A significant, novel component of the Umlaut system is that it generates error messages to *explain* silent error conditions. Umlaut lists suspected DL program issues, highlights their root cause(s)



Error: Input data exceeds typical limits

Your input data does not look normalized.

```
Epoch 7: maximum input value is 255.0, greater than the typical value of 1.
```

Solution

You should normalize the input data so its values fall between the typical ranges of -1 to 1 before passing them into the model

For image data, (pixels ranging from 0-255), a typical way to normalize the pixel values to the range of -1 to 1 is

```
training_images = (training_images / 128.0) - 1
```

Captured at epochs [0, 1, 2, 3, 4, 5, 6, 7].

Search Stack Overflow Search Docs Open in VSCode

Warning: Possible overfitting

The validation loss is increasing while training loss is stuck or decreasing. This could indicate overfitting. However, if validation loss is still trending downwards afterwards, this error could be a false positive.

```
Epoch 6: training loss changed by -0.03 while validation loss changed by 0.14.
```

Solution

Try adding dropout or reducing the power of your model.

Dropout randomly omits weight updates during training (with some probability) which decreases model power and potentially increases robustness.You can reduce the power of your model by decreasing the `units` or `filters` parameters of `Dense` or `Conv2D` layers.

Captured at epochs [4, 6].

Search Stack Overflow Search Docs

Title
Theory-Centric Description (1)
Program Context (3)
Best Practices and Suggested Action(2)
Code Snippet (4)
Documentation (5) and Source Code (6) Links
Title
Theory-Centric Description (1)
Program Context (3)
Best Practices and Suggested Action(2)
Documentation Links (5)

**Figure 3: Umlaut errors include several elements to help developers close the DL debugging loop. Errors include short and long descriptions (1) with suggested solutions (2), often incorporating program context (3). Solutions can include code snippets or hints (4), and outbound documentation and Stack Overflow links (5). To help users pinpoint the root cause(s) in code, some errors include links to open the source file in VSCode at the specific location of the suspected root cause (6).**

with integrated program context, offers potential solutions from collected best practices, and directly links error messages to visualizations of model output. Error messages produced by Umlaut contain the following elements:

*4.3.1 Title and Severity Qualifier.* Error messages produced by Umlaut have titles which reflect their respective root causes. Titles are given severity qualifiers (*Warning*, *Error*, and *Critical*) depending on the expected impact on model performance. Warnings have minimal impact on accuracy, but may lead to issues in the future (e.g., an issue with validation data). Critical errors can prevent the model from learning from data at all (e.g., a hyperparameter causing loss to reach NaN). Severity qualifiers are added manually to error message titles, but future iterations of Umlaut could automatically assign them based on predicted impact.

*4.3.2 Instructional Description with Program Context.* Studies of the experiences of non-expert ML developers show that building an understanding of ML theory and bridging that theory with practice are significant hurdles [10, 29]. In Umlaut error messages, descriptions explain the surrounding ML theory, describe the heuristic check used to raise the error, and suggest actionable bug resolution steps in order to bridge knowledge gaps for non-expert users. Error messages can also include *program context* to shed light on the particular conditions which raised an error during program execution. The context is dependent on the particular error, and includes runtime data, such as values of variables which exceeded the limits of a heuristic, prototypes of API calls with invalid arguments, or names of model layers with invalid hyperparameters assigned.

*Jordan remembers learning about different activation functions for DNNs in their class, helped by the quick refresher from Umlaut's error description. Jordan looks at the program context in the error and sees that Umlaut printed the names of layers in the model with linear activation functions—the bottom layers which were swapped in for transfer learning on the bird dataset. It was a simple mistake: Jordan simply forgot to add an activation argument, and Keras assigns linear activations when the argument is omitted.*

*4.3.3 Bridging to Best Practices with Code Examples.* While theory is critical for building mental models to aid in DL debugging, theory alone is not enough to guide users in decision making when debugging. Furthermore, understanding the proper API usage of DL frameworks themselves can remain challenging to novices [29]. Umlaut makes error messages actionable by including descriptions of potential solutions based on best practices and by instantiating them with concrete code examples.

Beyond code snippets, error messages in Umlaut can provide outbound links to curated Stack Overflow searches (e.g., `[keras] is:closed from_logits` to search for closed issues with a "Keras" tag for a search query) and links to Tensorflow documentation for relevant APIs. Altogether, program context, grounded in explanations of *why* it has raised errors, helps develop user mental models of DL debugging; while code snippets embodying best practices help users close the debugging loop by making the appropriate fixes to their application.

*Jordan remembers learning about many kinds of activations in their class—sigmoid, tanh, relu, . . . —but can't remember when to use which one. Reading further in the Umlaut error message, a code hint suggests adding activation='relu' when working with image data. Jordan copies this hint to paste into their program.*

*4.3.4 Referencing the Suspected Root Cause in Code.* To further assist users in closing the debugging loop in larger models or more complex programs, the Umlaut client ingests the source of the program being debugged and inspects stack frames in the Python runtime to guess the closest line of code to the source of a given error. The Umlaut web interface renders these links as URLs which open the Visual Studio Code editor to the specified line and character in the file where the bug occurred.

*Jordan notices the error message has an "Open in VSCode link". They click the button and are taken directly to first layer missing an activation function. They paste the code hint from Umlaut there and into the other layers missing nonlinear activations. Relieved it wasn't something more serious, Jordan restarts the training process.*

## 4.4 Bidirectional Link Between Errors and Interactive Visualizations

Inspired by development tools such as Tensorboard [2] and Weights and Biases[2], Umlaut uses simple visualizations to show how model training progresses over time. Line plots show loss and validation accuracy values at the end of every training epoch (a complete iteration over the training dataset), with multiple traces for training and validation. As a rule of thumb, decreasing loss and increasing accuracy plots that slow over time are a positive indicator. When

[2] https://www.wandb.com/

errors are present in a DL program, anomalies may appear in these plots, which are often subtle and require expertise to decipher [39].

*Jordan keeps an eye on the UMLAUT plots as new training metrics stream in. They notice the validation loss plot starts decreasing, then plateaus and starts increasing. A new warning message pops up: "Possible Overfitting". Clicking the error highlights the epochs in the loss plot where the validation curve started increasing while the training curve decreased, confirming Jordan's suspicion that this was an undesired result. Following the recommendations of the overfitting warning, Jordan adds Dropout to the model and reruns training.*

*4.4.1 Error Timeline.* UMLAUT also displays a *timeline* visualization, which encodes the type and frequency of errors encountered in the DL program over time. For every training epoch, unique errors are stacked on a vertical axis, distinguished by a 4-element categorical color scale. This visualization allows the user to inspect the behavior of the model and training process over time, e.g., spotting errors flagged before the beginning of the training process (plotted below the horizontal axis) or errors which only appear later during training (such as overfitting or spiking loss from an outlier in data). Users can click on the timeline or on error messages to highlight specific regions in the line plots. Plot annotations show which epoch(s) the errors occurred, and where the behavior of the curves caused a heuristic to raise an error. Inspecting the timeline may also help determine when a raised error was a false positive, e.g., when an error appears sporadically, or rarely.

*After the last training run, Jordan keeps an eye on the loss and validation plots. They seem to look fine this time, but the Overfitting warning pops up again. They're skeptical, since they just implemented a fix earlier, so they click the error to highlight parts of the error timeline, and the loss and accuracy plots. Jordan sees there were two epochs when the validation loss went up a small amount, but the overall trend looks fine. They make the judgment call that the error was likely a false alarm, and save the model checkpoint, at an accuracy of 84%.*

*With the model trained, Jordan writes a quick program that uses it to classify live images from the camera feed and notify them by email when a rare bird appears. The system not only helps Jordan enjoy the wildlife, but logging the rare birds' feeding activity from the classifier output also helps in their conservation efforts.*

## 5 UMLAUT HEURISTICS

In order to codify best practices from experts into UMLAUT's automated checking infrastructure, we identify and implement 10 preliminary heuristics based on commonalities in various sources including lecture notes [39, 66, 67], industry courses and articles [32, 49], textbooks [19], expert practitioner blogs [40], default values in APIs [2, 12], and early-stage research cataloguing tensorflow program tests [7]. We prioritized heuristics which covered bugs and conceptual misunderstandings shown to be common themes in Stack Overflow questions, open source DL projects, and expert interviews from existing literature [33, 77, 78].

Our heuristics map to common issues in data preparation, model architecture, and parameter tuning. We implement a check for each which is *static* (using a snapshot of the program prior to training) or *dynamic* (analyzing the program during model training runtime). Each heuristic check has an associated severity qualifier *(Critical, Error, Warning)* and error message written by the authors. These error messages include context and suggestions summarized from the heuristic's sources, and are described in Appendix A. Our list is not exhaustive, and we discuss how UMLAUT may be extended with additional heuristics checks in Section 6.

Although the heuristics we select are widely-accepted and often apply to common use cases, they may not always apply to a user's specific context (resulting in a false positive or negative). In particular, some heuristics used in the ML community suggest concrete values, e.g., for learning rate or image dimensions. UMLAUT adopts commonly used values, e.g., input normalization between -1 and 1. There values might change with new developments in underlying algorithms or community conventions. Future versions of UMLAUT could have such values exposed as configuration parameters that users can update over time.

### 5.1 Data Preparation

*5.1.1 Input Data Exceeds Typical Limits (dynamic).* Normalizing input data to a common scale can help models converge more quickly, weigh features more evenly, and prevent numerical errors [49, 66]. Normalization is often regarded as an important "default" setting [70]. UMLAUT checks if the input data exceeds the typical normalization interval of $[-1, 1]$.

*5.1.2 NaN Encountered in Loss or Input (dynamic).* The loss value of a training batch can overflow and become NaN during training as a result of non-normalized inputs or an unusually high learning rate [39, 49]. UMLAUT checks whether NaN values appear in the loss output, and, if so, whether they appear in the input. UMLAUT separately checks if the current learning rate is unreasonably high (Section 5.3.1) which could also be causing NaN loss values.

*5.1.3 Image input data may have incorrect shape (dynamic).* DL frameworks expect image inputs to convolutional layers to follow a particular format (typically "NHWC" or "NCHW")[3]. If these dimensions are not ordered as expected, the program may still run without an error, but the network will have incorrect calculations of convolutions in those layers (i.e. convolving over the wrong channels). This can reduce accuracy and speed due to a resulting incorrect number of parameters. UMLAUT checks the input sizes of these dimensions (assuming input image height matches width, common for many vision tasks) against the configured ordering. UMLAUT raises an additional error message if the configured channel ordering is not optimal for the hardware the program is running on (CPU or GPU).

*5.1.4 Unexpected Validation Accuracy (dynamic).* When a model's prediction accuracy on a validation set is unusually high or when it exceeds the value of its training set accuracy, this may indicate leakage between the training and validation data splits [49]. UMLAUT checks if the validation accuracy exceeds the training accuracy or exceeds 95% after the third epoch (to reduce noise).

[3] https://www.tensorflow.org/api_docs/python/tf/nn/conv2d

## 5.2 Model Architecture

*5.2.1 Missing Activation Functions (static).* When multiple linear, or "Dense" layers are stacked together without a non-linear activation in-between, they mathematically collapse into a single linear layer rendering additional parameters useless. Therefore, a non-linear activation function must be used between them to produce nonlinear decision boundaries [19, 66]. Umlaut inspects the model architecture and raises an error if two linear layers are stacked together without a nonlinear activation in between.

*5.2.2 Missing Softmax Layer before Cross Entropy Loss (static).* Some loss functions expect normalized inputs from a softmax layer (i.e., classification outputs that model a probability distribution, such that each class's probability is between 0-1 and sums to 1) [70]. The Keras defaults for cross-entropy loss expect softmax inputs, so omitting a softmax layer (or omitting the `from_logits=True` argument to the loss function) can result in a model that learns inefficiently due to improper gradients. Umlaut checks that softmax is being calculated before the cross-entropy loss calculation.

*5.2.3 Final Layer has Multiple Activations (static).* A complementary problem to a missing softmax layer prior to the loss calculation is the addition of an extra activation function. Umlaut checks for stacked activation functions, which is redundant or may even impact model performance negatively.

## 5.3 Parameter Tuning

*5.3.1 Learning Rate out of common range (dynamic).* Setting the learning rate too high or too low can cause drastic changes to model behavior and cause several symptoms in output. A learning rate which is too high can cause NaN outputs or a non-decreasing loss value during training, while a low learning rate can cause loss to converge to non-optimal values early [39, 70]. Best practices for initializing learning rates vary: Keras initializes the Adam optimizer with a learning rate of 0.001, while some experts suggest 0.0003 [40]. Because selecting a learning rate is highly problem-specific, Umlaut checks that the optimizer's learning rate falls between 0.01 and $10^{-7}$ (near the limit of precision for 32-bit floating point numbers) and raises an error if it falls outside this range.

*5.3.2 Possible Overfitting (dynamic).* Overfitting occurs when a model fits training data too closely, reducing its ability to generalize to new data. This is a core challenge to DL development since features created by a DNN may capture subtle elements disproportionately common in training data [48]. To check for overfitting, Umlaut determines if the generalization error of its model has started to increase while the training error continues to drop, a widely-accepted indicator of overfitting [19, 39, 49, 66]. Our implementation of this check is reproduced in pseudocode below:

```
function DetectOverfitting(epoch, model, logs)
    d_loss = logs.loss - model.history.prev_loss
    d_val_loss = logs.val_loss - model.history.prev_val_loss
    if d_val_loss > 0 and d_loss <= 0 then
        raise OverfittingError(epoch, context=(d_loss, d_val_loss))
    end if
end function
```

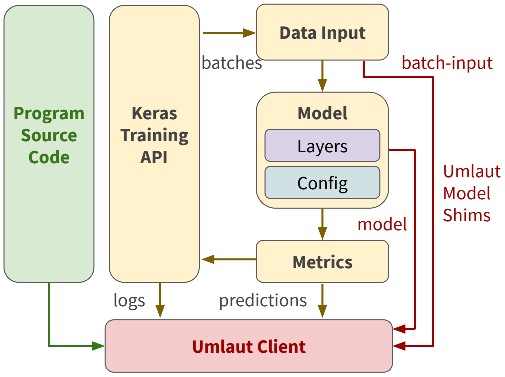


**Figure 4: Umlaut uses the Keras callback system to collect metrics about the training process during runtime. Umlaut also injects variables into the underlying Tensorflow model graph to capture input and output values, and collects a reference to the model object.**

*5.3.3 High Dropout Rate (static).* In order to prevent overfitting and aid in generalization, dropout can be used, which probabilistically prevents a percentage of neurons from receiving gradient updates. Umlaut checks the model configuration before training and raises a warning if the dropout probability exceeds 50%, which could lead to redundancy in the model and a reduction in accuracy due to lower-than-desired number of parameters. This error is often be caused by the users' confusion between the 'drop' and 'keep' probability, which are opposites. Our implementation of this check is reproduced in pseudocode below:

```
function DetectHighDropout(model)
    flagLayers = list()
    for layer in model do
        if layer is Dropout and layer.dropoutRate >= 0.5 then
            flagLayers.append(layer.index, layer.name, layer.dropoutRate)
        end if
    end for
    if flagLayers then raise HighDropoutError(context=flagLayers)
    end if
end function
```

# 6 IMPLEMENTATION

Umlaut is comprised of 2 major components. The first is a client program which interfaces with a Keras training session, injects checks into the runtime, then uses those checks to raise errors. Metrics and errors are streamed from the Umlaut client to the second component, the Umlaut server. The server logs errors and metrics in a database, and renders data and error messages with a web application.

## 6.1 Umlaut Client Shims and Structure

The Umlaut client is packaged as a Python library which can be imported and configured for use with a Keras program in 3 lines. Users import the library, configure and initialize the imported `UmlautCallback` object which returns a `tf.Keras.callbacks.Callback`

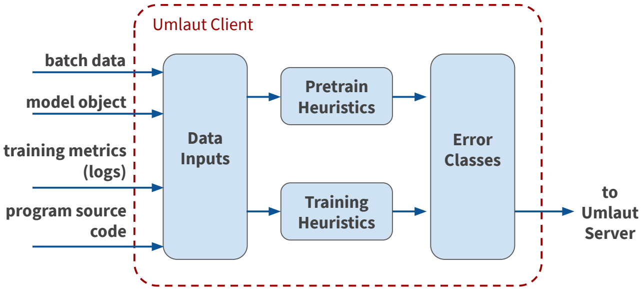


**Figure 5: The Umlaut client uses data collected from shims to run static checks of the model before training, and dynamic checks during training. Heuristic checks and errors (reflecting root causes) are distinct concepts in Umlaut's architecture, allowing similar, yet subtly different symptoms to raise different root causes from within the same check.**

instance, and pass that callback instance as an argument to the `model.fit` training function.

In order to access and diagnose a broad range of error symptoms, Umlaut requires several data sources from the DL program runtime. Because these sources must be transparently instrumented, we refer to the instrumentation as "Model Shims". Umlaut uses various APIs and shims to ingest the following runtime information (Figure 4).

**Keras Callbacks Provide Epoch Number and Training `logs`:** The Keras framework implements a callback mechanism which provides hooks at various steps during the model training process. The Umlaut client is primarily implemented as a Keras callback which runs static program checks before training starts and dynamic checks after the completion of every training epoch. Pre-training checks are not provided data from Keras, and rely on access to the model object and source module (described below). Callbacks fired during training are passed epoch numbers for indexing, and a `logs` object which contains the loss and accuracy values from the current epoch on the training and validation data.

**Users Provide the `tf.keras.Model` Object:** When initializing the `UmlautCallback`, Users must pass the model being trained as an argument. The model object exposes many critical elements for diagnosing errors. The `logs` object provided by Keras callbacks only provides a snapshot of the model's loss and accuracy metrics. Having access to the model instance exposes a `tf.Keras.callbacks.History` object which stores loss and accuracy values from every epoch in the current training run. The history object allows Umlaut to check the *behavior* of the model over time, enabling more complex heuristics (e.g., detecting overfitting). The model object also exposes its underlying structure, e.g., the individual layers and optimizer.

**Model `call` Overrided to Access Input and Output:** In order to access copies of data passed into and predictions from the model during training, we override its *call* function. To do this, we add two `tf.Variable` objects to the model execution graph (before and after). The variables store copies of the model's latest input and output data, and can be evaluated in the Tensorflow session used in the Keras backend.

**Module Source Code Captured by Searching Stack Frames:** Some error messages rely on the location and contents of the program source code. Umlaut uses the Python `traceback` library to guess which source module contains the training loop, and then stores the contents of the file for searching.

## 6.2 Umlaut Client Logic: Running Checks and Raising Errors

During the training process, Umlaut aggregates inputs from model shims and dispatches them to test runners. Test runners run *static* checks before training starts, and *dynamic* checks during program execution, after every training epoch. Checks during either of these stages can raise errors, which include client program context, and are stored on the Umlaut server. A key design choice in the implementation of Umlaut was to decouple checks and errors. This allows more flexibility and brevity in cases where one heuristic could detect similar symptoms that map to different errors.

Static checks inspect the structure of the model and its parameters without any context from runtime. Dynamic checks use context from program runtime in concert with model structure and parameters. Dynamic checks can capture *snapshots* of the program execution environment (e.g., to find input data with NaN values), or can track the *behavior* of the model over time (e.g., capturing overfitting when training loss decreases and validation loss increases). The performance impact of static checks is minimal, and model size impacts performance on the order of ms. For dynamic checks, Umlaut mostly operates on aggregate metrics already collected by Keras, and the added operations from shims have no noticeable effect on performance. We confirmed this by running Umlaut on more complex models (see Section 7).

When a check raises an error, it initializes that error with the program context necessary to render the error in Umlaut's web application (e.g., including the names of layers with missing activation functions or the value of a high learning rate). At the end of every epoch, the client sends metrics (loss and accuracy for test and validation sets) and errors to the server. For errors, only a unique error key and the related context is sent to the server, and the server renders the error's static description and contents. Since these requests use aggregate data, they impact performance on the order of tens to hundreds of ms (including network latency) per epoch.

Umlaut's design allows new checks and errors to be added in a standardized way. To do this, a developer must add a new error message by subclassing a base error template in the Umlaut server, and a check function that raises the new error to a check runner in the Umlaut client.

## 6.3 Umlaut Server

The Umlaut web interface is implemented using Plotly Dash with the Flask web framework, and MongoDB for the database. The web application exposes a REST API to accept updates from the Umlaut client which stores errors and metrics associated with their session in the database. When a user navigates to the Umlaut session view, the page polls the database and rerenders the page when new data is present. Interactive graphing features in the web application are implemented using Plotly Dash's Pattern Matching

Callbacks feature. This functionality allows click events on an error, the timeline, or a plot to update the other corresponding elements.

# 7 USER EVALUATION

We evaluate the usability of Umlaut's interface as well as its ability to help developers find and fix bugs in ML programs in a within-subjects user study with 15 participants. We introduce bugs into two image classification programs, and measure the number of bugs participants find and fix, with and without Umlaut.

## 7.1 Participants

We recruited 15 participants (12 male, 3 female; ages 18-30, $\mu = 23.8$, $\sigma = 3.1$) from university mailing lists to participate in our study. Through a recruiting survey, we accepted participants who were at least familiar with ML concepts and development, but who did not identify as an expert or professional (i.e., excluding ML reseracher who primarily develop ML models). Of our participants, 12 had integrated existing machine learning models into projects, and 9 had retrained the last layers of an existing machine learning model to adapt it to a use case. 4 participants had developed new machine learning models, and 2 had contributed to open source machine learning projects. Questions determining expertise were adapted from Cai et al. [10]. 14 participants were graduate students, and 1 undergraduate. 11 had academic backgrounds in computer science, 3 in electrical or computer engineering, and 1 in mechanical engineering. Participants were compensated $20 USD. Evaluations lasted under 60 minutes.

## 7.2 Setup

Due to the COVID-19 pandemic, the study was conducted remotely using Zoom video-conferencing software on the experimenter's laptop, a 2016 MacBook Pro. Participants used the Visual Studio Code IDE with the Pylance Python language server [59] and VS IntelliCode [53], which together provide relevance-ranked auto-completion and syntax checking. Python files for the debugging tasks were loaded and executed by the IDE on a Google Cloud Platform instance with an Nvidia Tesla T4 GPU to reduce model training time. For the CIFAR-10 task, training the provided model for 10 epochs took under 1 minute.

## 7.3 Study Design and Tasks

We modeled the design of our user evaluation after that of Gestalt [61]. Our study was a within-subjects design, comparing Umlaut to a baseline condition across two debugging tasks. To account for interaction effects from the ordering of these conditions, tasks were counterbalanced by condition (baseline vs Umlaut) and by order (Program A vs Program B). We measured the number of bugs *found* (i.e., the explicit root cause verbally indicated by the participant) and *fixed* in each task. Bugs which only had a partial fix (e.g., adding missing nonlinearities in convolutional but not linear layers) were not counted as fixed.

For the debugging tasks, we created a simple Keras program which loads the CIFAR-10 dataset [46], constructs a 7-layer convolutional neural network, configures cross entropy loss and Adam optimization [44], trains the model for 10 epochs, and evaluates model accuracy on the CIFAR-10 test set. We designed this program to be as simple as possible—under 35 lines of code (under 40 when adding Umlaut)—for two key reasons. First, simplicity strengthens the baseline condition by being easier for the participant to fully understand. Second, the model is able to train quickly (under one minute on a GPU) before the test accuracy plateaus around 77%, making more iteration feasible in the study timeframe compared to a larger (but potentially more accurate) model.

We created two modifications of this program, *Program A* and *Program B*, and inoculated both with three unique bugs. These programs both execute without any explicit Python errors or warnings, but the bugs impact the accuracy of the model at different levels of severity: *low* (approx. 0-5% reduction in accuracy), *medium* (approx. 6-20% reduction in accuracy), and *high* (accuracy will not increase beyond random chance). The bugs in both programs were also chosen to span common stages of the ML development process: *Model Architecture*, *Parameter Tuning*, and *Data Preparation*. Finally, the bugs may generalize well to different learning tasks, e.g., image classification, sentiment classification, pose estimation,…).

The bugs introduced into Program A were:

- **A1:** No softmax function was added after the final `Dense` layer, causing the optimizer to receive unnormalized logits and not improve loss (High severity, model architecture)
- **A2:** Dropout rate set to 0.8, resulting in only 20% of model capacity being used (Medium severity, parameter tuning)
- **A3:** Input images were not normalized, with values ranging from 0-255 (Low severity, data preparation)

The bugs introduced into Program B were:

- **B1:** Learning rate was set to -1e3 instead of 1e-3, resulting model being unable to learn from data (High severity, parameter tuning)
- **B2:** No ReLU activation functions were added to the model, resulting in stacked convolution or dense layers collapsing into a single layer (Medium severity, model architecture)
- **B3:** Validation data overlapped with the training set, picking the first 100 training images (Low severity, data preparation)

As a test, we connected Umlaut with VGG16 and ResNet101 from the `Keras.applications` API and ran it in the same scenarios as our user evaluation (adding bugs A1, A3, B1, B3). A2 and B2 were not considered as they would require source code changes to Keras. We verified the same errors as our small test model were raised.

## 7.4 Procedure

After completing an entry survey, participants were shown a minimum-example Keras program which fit a linear model (2 `Dense` layers) on a different, simpler dataset (Fashion-MNIST [75]) in the Visual Studio IDE. Participants were shown the dataset Readme, the structure of the program was explained (imports, data loading, model architecture, training configuration, training, and evaluation), and the program was executed in the editor. For participants starting in the Umlaut condition, the application had lines of code added for invoking Umlaut. The Umlaut web interface was loaded on a web browser on the researcher's laptop, and the example program was run with an Umlaut session attached. The Umlaut command line and web interfaces were explained, and participants were told error messages were based on heuristics, so there may be false positives. For participants starting in the baseline condition, the example program with Umlaut code added was demonstrated between completing the baseline and Umlaut tasks.

Before starting the first debugging task, participants were shown the Readme for the dataset used by their debugging task programs, CIFAR-10 [46]. Participants were told they would be shown a program with multiple bugs, and their task would be to find, explain, and fix all the bugs they found in that program. Participants were told they should not need to make major architectural changes to the models (e.g., by adding or removing layers, or changing the sizes of `Conv2D` or `Dense` layers), but were able to if desired. Participants were told they could use any online resources needed, e.g., documentation, Stack Overflow, or web search; and the researcher could troubleshoot the apparatus or explain the Umlaut interface, but not assist with debugging. Finally, participants were told a bug free version of this program could have a target test accuracy of 77%, but were reminded their goal was not to maximize accuracy, and that a high accuracy does not guarantee a bug-free model. The training period took approximately 15 minutes (plus 5 for Umlaut).

Participants were then shown program A or B, in the baseline or Umlaut condition. The only differences between conditions were that Umlaut code was added to the program and opened in a web browser. Participants were not allowed to use Umlaut software in the baseline condition. After completing the first task, the other program was shown, in the other condition. Participants were limited to 15 minutes of debugging time per program.

# 8 RESULTS AND DISCUSSION

## 8.1 Umlaut Helped Participants Find and Fix Significantly More Bugs

Across both programs, participants using Umlaut found more bugs ($\mu = 2.8, \sigma = 0.4$) compared to the baseline condition ($\mu = 1.8, \sigma = 1.1$). This difference is statistically significant (Wilcoxon Signed-Rank test, $Z = 2.67, p = 0.004$). Furthermore, participants using Umlaut were able to implement fixes for more bugs ($\mu = 2.5, \sigma = 0.5$) compared to the baseline condition ($\mu = 1.5, \sigma = 0.9$). Again, this difference is statistically significant (Wilcoxon Signed-Rank test, $Z = 2.65, p = 0.004$).[4]

Furthermore, survey responses collected from participants confirm and strengthen these findings. On 5-point Likert scale questions (1= strongly disagree, 5=strongly agree), participants indicated that Umlaut helped them find ($\mu = 4.3, \sigma = 0.70$) and fix ($\mu = 4.0, \sigma = 1.1$) bugs they would have not noticed without it. Participants also indicated a high likelihood of integrating Umlaut as a regular part of [their] ML development processes ($\mu = 4.3, \sigma = 0.60$). The distribution of ratings for these questions is shown in Figure 6.

## 8.2 Open-Ended Feedback

We asked participants to share open-ended comments on the advantages and disadvantages of Umlaut, what they liked and disliked about its interface, and what additional features would make it truly useful. We conducted an open coding phase over the qualitative responses, and further grouped codes into related topics [73].

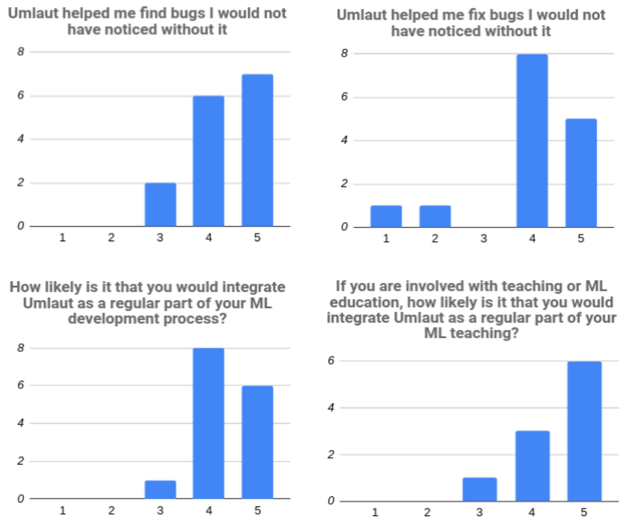


**Figure 6: Distribution of participants' ratings on likert-scale questions (Top row: 1=Strongly Disagree to 5=Strongly Agree; Bottom Row: 1=Very Unlikely to 5=Very Likely)**

### 8.2.1 Model Checks Illuminate Silent Errors and Save Time.

Many participants commented on the general difficulty of debugging ML code, and remarked that Umlaut was a significant step in making the process quicker and easier. P7 related ML debugging to trial-and-error, and suggested Umlaut added missing structure: *"[Umlaut] Makes the whole guess-and-check debugging flow a lot faster and smoother. Instead of having to comb over the code and form my own hypotheses about what could be wrong, Umlaut will provide you with a list of possible issues."*

Others validated the prevalence of silent errors in ML debugging, and how Umlaut shed light on these difficult-to-find errors, saving time: *"[The primary advantage of using Umlaut was] automatic checking for common "errors" like missing activations or strange learning rates that don't cause runtime errors but prevent successful training"* (P2); *"[Umlaut helped] me quickly find bugs in my machine learning model that are difficult to detect through code inspection. I have always found debugging machine learning models to be a time-consuming and error-prone process"* (P9); *"[Umlaut can] Identify basic bugs (e.g. out-of-distribution that aren't trivially caught through type/shape checks)"* (P8).

One participant noted Umlaut's time savings could reach beyond debugging itself, as validating bug fixes can also be time consuming: *"[Umlaut's primary advantage was] Finding 'bugs' that otherwise would not have produced an actual error (bad parameters, values outside recommended ranges, overfitting). These bugs are by far more time consuming to debug because they usually require me to train a model for at least some time (5 epochs?) to verify that they've been fixed"* (P5).

### 8.2.2 Best Practices and Code Examples Help Close the Debugging Loop.

Participants appreciated the explanations of underlying error conditions and suggestions based on best practices: *"The potential*

[4] We measure significance using a non-parametric test to account for the possibility that our participants' actual skill levels may not be normally distributed due to recruiting graduate students in engineering departments.

*diagnosis along with reasoning was quite helpful"* (P3); *"Error messages were descriptive and gave me specific actions to do. Also good values for parameters e.g. dropout rate is much better than saying the value is too high"* (P13); *"Moreover, it does not only suggest to me what's potentially wrong, but also how to fix it. Very useful"* (P12).

Code snippets were helpful in translating theory into practice and navigating complex APIs: *"Web interface had very helpful blurbs- e.g. for overfitting it immediately suggests to adjust filter count or add dropout, and it gives the one line fix for the sparse cross entropy loss issue"* (P5); *"Because my goal was to make fixes to the code, it was helpful to have concrete code snippets that I could copy into the code and modify lightly. It can be tricky to find up-to-date code snippets for machine learning libraries on the web as the libraries can change quite frequently, and often there are many different API members that can accomplish the same goals. Umlaut saved me a lot of time"* (P9).

Some participants wished Umlaut provided code samples more frequently: *"I'd prefer to see more code suggestions (e.g. suggestions of what class to use for the logits case in the second example)"* (P15). Integrating direct comparisons between snippets and the underlying program could help bridge gulfs in debugging: *"Suggested code changes in context of actual source code (similar to GitHub PR suggested changes feature), would make debugging even easier"* (P13); and counterexamples could potentially help users search for faulty code *"There were a few instances where I felt like Umlaut could skip some of the prose (even though it's only a few sentences long) and lead in with a code snippet showing an anti-pattern, and another code snippet that fixes it"* (P9).

Some error messages in Umlaut include explanations of API components, but not explicit snippets which can be copied and pasted into the source program, e.g., in cases where an unknown root cause could be addressed by one of many candidate solutions. This is discussed further below.

*8.2.3 Effectively Communicating the Heuristic Nature of Umlaut.* Effectively communicating the uncertainty of ML models and intelligent systems is an open research question. Umlaut uses heuristic checks which have the potential to miss errors or raise false positives. Some participants took this into consideration while using Umlaut: *"I would use Umlaut with the understanding that it might not be perfect, so in my particular case, I don't think I would be misled into thinking I had debugged all of the issues in my model if Umlaut didn't report any issues"* (P9). However, P5 cautioned against potential over-reliance on Umlaut: *"[The primary disadvantage is] "Autograder-driven development" effect [. . . ] I feel like relying on Umlaut to point out errors means I'm less likely to scrutinize parts of the program that Umlaut did not pick up on. [...] The second time around, without Umlaut providing feedback I felt more compelled to look at the entire program top to bottom."* Suggestions provided by Umlaut use qualifying language and offer multiple solutions in cases where there is not a single guaranteed fix (e.g., overfitting). Identifying effective ways to communicate the underlying uncertainty of Umlaut is an important direction of future work.

*8.2.4 Umlaut as a Pedagogical Tool.* 10 participants who indicated involvement with teaching or ML education also responded to a 5-point Likert scale question indicating a high likelihood of integrating Umlaut as a regular part of ML teaching ($\mu = 4.5, \sigma = 0.67$). Open-ended comments also suggested the potential for Umlaut as an instructional aid: *"I think this would be a fantastic tool especially for new students of deep learning"* (P6); *"It can point out areas where there are potential problems that someone especially someone new to ML might not notice"* (P4); *"It definitely helped out in the debugging process, especially as someone returning to machine learning after a long time"* (P15).

*8.2.5 UI Tweaks.* Several participants (P1, P2, P3, P4, P7) suggested changing the sessions dropdown menu to automatically refresh (currently, the entire webpage must be refreshed). P15 suggested more deeply linking visualizations with errors: *"It would also be nice to see an icon saying what warning/critical errors are associated with each epoch when I hover over it, instead of just the accuracies."*

Some users appreciated the detailed descriptions and suggestions from error messages: *"The error messages were designed and structured well. (having both short and long versions of the error message, and identifying the particular layer/epoch)"* (P14). However, others thought the detail cluttered the Umlaut user interface, and should be hidden unless expanded by the user: *"the textbox displaying the error messages cannot be resized, so it is difficult to see all the errors at once"* (P6); *"the longer blurbs tend to clog the screen so you have to scroll to see all of the errors & recommended solutions, if there's a way to expand/collapse and just show a one-line blurb"* (P5). These visual design issues could be addressed in a future iteration of Umlaut.

## 9 LIMITATIONS AND FUTURE WORK

Because of the stochastic nature of the DL training process itself, Umlaut has important limitations. As a prototype, it also has limitations from engineering constraints.

**Model Checks are Based on Heuristics:** Model checks are implemented as heuristics, so they may be raised as false positives or missed. For example, "Check Validation Accuracy" can be raised if random noise in data causes a spike in validation accuracy to exceed training accuracy during one epoch. While Umlaut errors include qualifying language and the error timeline can help determine if errors form a pattern, these mitigation strategies are not perfect and require some training to interpret. False positives could potentially be mitigated further with customizable filtering.

Umlaut may also miss errors (false negatives) for several reasons. Model checks were developed to apply to general cases, but these cases may not generalize to some specific conditions, e.g., omitting a nonlinear activation may sometimes increase performance, and the range of reasonable learning rates is highly dependent on the model structure and data. Future iterations of Umlaut could use deeper inspection of the data and model to adjust heuristic boundary conditions.

**Mappings from Heuristics to Root Causes May Not Always Hold:** In software debugging, there are often multiple possible root causes that lead to a common error symptom (e.g., null pointers). DL debugging is no exception, and Umlaut checks may miss the correct solution or possible suggest an incorrect one. Error messages include text to remind users of their inherent uncertainty, but this mitigation strategy is not perfect.

**Generalization to New Model Architectures:** New types of model architectures produced by research may require new debugging strategies, including different heuristics and parameter ranges. While significant work has been done to understand the taxononmy of DL errors [33], DL programming paradigms are still evolving, and the landscape of errors may change over time. Umlaut supports custom layers implementing the standard `Keras.layers.get_config` API. Umlaut also works with different types of input and output (e.g., NLP, tabular data, regression, etc.) and could be extended to work with novel data types.

**Crowd-based Error Message Creation:** In the future, error message content and heuristic check thresholds could include crowd-sourced best practices and tips from the broader DL community and others who have faced similar issues such as in HelpMeOut [22].

**Outbound Links are Hardcoded:** Error messages with outbound links to Stack Overflow and documentation currently only support hardcoded links, with the intent for documentation to provide more context on suggested code recipes, and Stack Overflow searches to search for a wider net of related issues. Hardcoded links will not capture all cases, and future versions of Umlaut could integrate program context into the links (e.g., searching Stack Overflow for normalization with the value 255 extracted from the program), but translating from a symptom to a well-formed search query is an open research problem.

**Umlaut Code Awareness is Incompatible with Python Notebooks:** Umlaut uses stack frame inspection to find a source module with a training loop. This routine currently fails on Python Notebooks, a common tool for developing DL programs [27]. This limitation could be overcome with additional engineering effort, or by implementing Umlaut as a Python Notebook extension.

**Version Control and Comparing Sessions:** Umlaut currently has no ability to *compare* training sessions side by side. This would allow faster verification that underlying program bugs have been solved, and better enable users to track their experiments over time.

## 10 CONCLUSION

Umlaut addresses critical gaps in the DL development process by discovering bugs in programs automatically, and using theory-grounded explanations to translate from their symptoms to their root causes. Umlaut assists in selecting a debugging strategy building from best practices, and guides the implementation of best practices with concrete code recipes. Umlaut unifies these principles into a single interface which blends together contextual error messages, visualizations, and code. An evaluation of Umlaut with 15 participants demonstrated its ability to help non-expert ML users find and fix more bugs in a DL program compared to when not using Umlaut in an identical development environment. We believe Umlaut is a stepping stone in the direction of designing user-centric ML development tools which enable users to learn from the process of DL development while making the overall process more efficient for users of all skill levels.

## ACKNOWLEDGMENTS

We thank James Smith and Michael Laielli for their insight, experience, and many conversations which helped make Umlaut possible. Thanks to Michael Terry for the support and suggestions. Finally, thank you to our reviewers and to our participants. This work was supported in part by NSF grant IIS-1955394.

## REFERENCES

[1] [n.d.]. NVIDIA DLSS 2.0: A Big Leap In AI Rendering. https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/

[2] 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* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[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* (Montreal, Quebec, Canada) *(ICSE-SEIP '19)*. IEEE Press, 291–300. https://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* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY, USA, 337–346. https://doi.org/10.1145/2702123.2702509

[5] Kanav Anand, Ziqi Wang, Marco Loog, and Jan van Gemert. 2020. Black Magic in Deep Learning: How Human Skill Impacts Network Training. *The British Machine Vision Conference* (2020).

[6] J. Bergstra, R. Bardenet, Yoshua Bengio, and B. Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *NIPS*.

[7] Houssem Ben Braiek and Foutse Khomh. 2019. TFCheck : A TensorFlow Library for Detecting Training Issues in Neural Network Programs. arXiv:1909.02562 [cs.LG]

[8] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-Centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 513–522. https://doi.org/10.1145/1753326.1753402

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[10] 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. https://doi.org/10.1109/VLHCC.2019.8818751

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

[12] François Chollet. 2015. keras. https://github.com/fchollet/keras.

[13] 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* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 677–686. https://doi.org/10.1145/2984511.2984566

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

[15] Rebecca Fiebrink and Perry R Cook. 2010. The Wekinator: a system for real-time, interactive machine learning in music. In *Proceedings of The Eleventh International Society for Music Information Retrieval Conference (ISMIR 2010)(Utrecht)*.

[16] Adam Fourney and Meredith Ringel Morris. 2013. Enhancing Technical QA Forums with CiteHistory. In *Proceedings of ICWSM 2013* (proceedings of icwsm 2013 ed.). AAAI. https://www.microsoft.com/en-us/research/publication/enhancing-technical-qa-forums-with-citehistory/ You can download the CiteHistory plugin at http://research.microsoft.com/en-us/um/redmond/projects/citehistory/.

[17] 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. arXiv:1806.00069 [cs.AI]

[18] M. Goldman and R. C. Miller. 2008. Codetrail: Connecting source code and web resources. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. 65–72. https://doi.org/10.1109/VLHCC.2008.4639060

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

[20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in neural information processing systems*. 2672–2680.

[21] 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.

[22] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1019–1028. https://doi.org/10.1145/1753326.1753478

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. arXiv:1502.01852 [cs.CV]

[24] K. He, X. Zhang, S. Ren, and J. Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*. 1026–1034.

[25] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[26] 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. https://doi.org/10.1109/VLHCC.2015.7356972

[27] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300500

[28] Andrew Head, Jason Jiang, James Smith, Marti A. Hearst, and Björn Hartmann. 2020. Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Outputs. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376798

[29] 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. https://doi.org/10.1109/VLHCC.2016.7739680

[30] Fred Hohman, Andrew Head, Rich Caruana, Robert DeLine, and Steven M. Drucker. 2019. Gamut: A Design Probe to Understand How Data Scientists Understand Machine Learning Models. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, Article 579, 13 pages. https://doi.org/10.1145/3290605.3300809

[31] F. Hohman, M. Kahng, R. Pienta, and D. H. Chau. 2019. Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers. *IEEE Transactions on Visualization and Computer Graphics* 25, 8 (2019), 2674–2693. https://doi.org/10.1109/TVCG.2018.2843369

[32] Jeremy Howard and Sylvain Gugger. 2020. Fastai: A Layered API for Deep Learning. *Information* 11, 2 (Feb 2020), 108. https://doi.org/10.3390/info11020108

[33] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2019. Taxonomy of Real Faults in Deep Learning Systems. arXiv:1910.11015 [cs.SE]

[34] Apple Inc. 2019. Apple Create ML. https://developer.apple.com/machine-learning/create-ml/

[35] Databricks Inc. 2019. MLFlow. https://mlflow.org/

[36] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[37] Andrew Janowczyk and Anant Madabhushi. 2016. Deep learning for digital pathology image analysis: A comprehensive tutorial with selected use cases. *Journal of pathology informatics* 7 (2016).

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

[39] 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

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

[41] Andrei Kapishnikov, Tolga Bolukbasi, Fernanda Viégas, and Michael Terry. 2019. XRAI: Better Attributions Through Regions. arXiv:1906.02825 [cs.CV]

[42] Jun Kato, Sean McDirmid, and Xiang Cao. 2012. DejaVu: integrated support for developing interactive camera-based programs. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*. 189–196.

[43] 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). arXiv:1711.11279 [stat.ML]

[44] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR (Poster)*. http://arxiv.org/abs/1412.6980

[45] Amy 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* (Boston, MA, USA) *(CHI '09)*. Association for Computing Machinery, New York, NY, USA, 1569–1578. https://doi.org/10.1145/1518701.1518942

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

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

[48] Zachary Chase Lipton. 2016. The Myth of Model Interpretability. *CoRR* abs/1606.03490 (2016). arXiv:1606.03490 http://arxiv.org/abs/1606.03490

[49] Google LLC. 2020. *Machine Learning Crash Course with TensorFlow APIs*. Retrieved February 2, 2020 from https://developers.google.com/machine-learning/crash-course

[50] Dan Maynes-Aminzade, Terry Winograd, and Takeo Igarashi. 2007. Eyepatch: prototyping camera-based interaction through examples. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. 33–42.

[51] 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* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/3126594.3126658

[52] 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* (Edinburgh, United Kingdom) *(DIS '17)*. Association for Computing Machinery, New York, NY, USA, 1213–1225. https://doi.org/10.1145/3064663.3064735

[53] Microsoft. 2020. *Automate code completions tailored to your codebase with IntelliCode Team completions.* Retrieved September 7, 2020 from https://github.com/microsoft/vs-intellicode

[54] Riccardo Miotto, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. 2017. Deep learning for healthcare: review, opportunities and challenges. *Briefings in Bioinformatics* 19, 6 (05 2017), 1236–1246. https://doi.org/10.1093/bib/bbx044 arXiv:https://academic.oup.com/bib/article-pdf/19/6/1236/27119191/bbx044.pdf

[55] 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. https://doi.org/10.1109/MCG.2019.2919033

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

[57] 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, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, California, USA, 4901–4911. http://proceedings.mlr.press/v97/odena19a.html

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

[59] Savannah Ostrowski. 2020. *Announcing Pylance: Fast, feature-rich language support for Python in Visual Studio Code.* Retrieved September 7, 2020 from https://devblogs.microsoft.com/python/announcing-pylance-fast-feature-rich-language-support-for-python-in-visual-studio-code/

[60] Google PAIR. 2017. FACETS. https://pair-code.github.io/facets/

[61] Kayur Patel, Naomi Bancroft, Steven M. Drucker, James Fogarty, Amy 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* (New York, New York, USA) *(UIST '10)*. Association for Computing Machinery, New York, NY, USA, 37–46. https://doi.org/10.1145/1866029.1866038

[62] 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* (Florence, Italy) *(CHI '08)*. Association for Computing Machinery, New York, NY, USA, 667–676. https://doi.org/10.1145/1357054.1357160

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

[64] Eldon Schoop, Forrest Huang, and Björn Hartmann. 2020. SCRAM: Simple Checks for Realtime Analysis of Model Training for Non-Expert ML Programmers. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '20)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3334480.3382879

[65] 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. https://doi.org/10.1145/3319499.3328231

[66] Jonathan R. Shewchuk. 2020. Concise Machine Learning. https://people.eecs.berkeley.edu/~jrs/papers/machlearn.pdf

[67] William Wang Alex Yang Jennifer Listgarten Anant Sahai Soroush Nasiriany, Garrett Thomas. 2019. A Comprehensive Guide to Machine Learning.

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

[69] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 3645–3650. https://doi.org/10.18653/v1/P19-1355

[70] Josh Tobin. 2019. *Troubleshooting Deep Neural Networks: A Field Guide to Fixing Your Model.* Retrieved September 17, 2020 from http://josh-tobin.com/troubleshooting-deep-neural-networks.html

[71] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. M<span class="smallcaps SmallerCapital">odel</span>DB: A System for Machine Learning Model Management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics* (San Francisco, California) *(HILDA '16)*. Association for Computing Machinery, New York, NY, USA, Article 14, 3 pages. https://doi.org/10.1145/2939502.2939516

[72] Matthew Veres and Medhat Moussa. 2019. Deep learning for intelligent transportation systems: A survey of emerging trends. *IEEE Transactions on Intelligent transportation systems* (2019).

[73] Robert Stuart Weiss. 1995. *Learning from strangers: the art and method of qualitative interview studies.* Free Press.

[74] J. Wexler, M. Pushkarna, T. Bolukbasi, M. Wattenberg, F. Viégas, and J. Wilson. 2020. The What-If Tool: Interactive Probing of Machine Learning Models. *IEEE Transactions on Visualization and Computer Graphics* 26, 1 (2020), 56–65. https://doi.org/10.1109/TVCG.2019.2934619

[75] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms.* arXiv:cs.LG/1708.07747 [cs.LG]

[76] Geoffrey X. Yu, Tovi Grossman, and Gennady Pekhimenko. 2020. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 126–139. https://doi.org/10.1145/3379337.3415890

[77] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 104–115.

[78] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

# A ERROR MESSAGE CONTENT

This section describes author-created error messages which may be raised by heuristic checks in the Umlaut client. At a minimum, these messages include a severity qualifier (Warning/Error/Critical), title, short description with related theory and background, and proposed solutions with examples instantiating best practices. Messages may additionally have a field which provides program context with explanations, an outbound link to a curated Stack Overflow search query, an outbound link to documentation, and a link to open the VSCode editor to the suspected problem line of code.

## A.1 Data Preparation

*A.1.1 Warning: Input Data Exceeds Typical Limits.* Your input data does not look normalized. You should normalize the input data so its values fall between the typical ranges of -1 to 1 before passing them into the model. For image data, (pixels ranging from 0-255), a typical way to normalize the pixel values to the range of -1 to 1 is: `training_images = (training_images / 128.0) - 1`

Program context: *Epoch <epoch>: <minimum/maximum> input value is <x_min>, <less/greater> than the typical value of <-1/1>.*

Stack overflow query: `[keras] closed:yes normalization`

Documentation link: https://www.tensorflow.org/tutorials/keras/classification#preprocess_the_data

This error includes a link to the source module where the model training loop is defined.

*A.1.2 Critical: NaN (Not a number) in input.* Some values in your model input are NaN (could indicate infinity). Please double check your input and make sure no NaN exists in it.

Stack overflow query: `[keras] nan input`

*A.1.3 Error: Image input data may have incorrect shape.* Your input images may have their dimensions in the wrong order. Your input is 4-dimensional with 2 equal dimensions, which is typically an image type. Most keras layers by default expect image data to be formatted as "NHWC" (Batch_size, Height, Width, Channel) unless otherwise specified. If running on CPU, setting the Keras image backend to "channels_first" and using "NCHW" (Batch_size, Channel, Height, Width) may sometimes improve performance. For example, you can tranpose your input data from "NCHW" to "NHWC", using `tf.transpose(X_train_images, [0, 2, 3, 1])`.

Program context: `Epoch <epoch>: Input shape is not <N,C,H,W/N,H,W,C>. Instead got <x_train.shape>`

*A.1.4 Warning: Check validation accuracy.* The validation accuracy is either higher than typical results (near 100%) or higher than training accuracy (which can suggest problems with data labeling or splitting). However, during early epochs, this could be a false positive. A high validation accuracy (around 100%) can indicate a problem with data labels, overlap between the training and validation data, or differences in preparing data for training and evaluation. Check to see if there is overlap between the training and validation sets, and inspect the validation set predictions by hand to ensure they make sense.

Program context: `Epoch <epoch>: validation accuracy is very high (<val_acc if val_acc > 95%>)`

`Epoch <epoch>: validation accuracy <val_acc> is higher than train accuracy (<train_acc>)`

Stack overflow query: `[keras] validation accuracy high`

## A.2 Model Architecture

*A.2.1 Critical: Missing activation functions.* The model has layers without nonlinear activation functions. This may limit the model's ability to learn since stacked `Dense` layers without activations will mathematically collapse to a single `Dense` layer. Make sure the `activation` argument is passed into your `Dense` and Convolutional (e.g., `Conv2D`) layers. A common practice is to use `activation='relu'`.

Program context: (for each problem layer) `Layer <index> (layer.name) has a missing or linear activation`

Documentation link: https://www.tensorflow.org/api_docs/python/tf/keras/activations

This error includes a link to the source module where the model was defined.

*A.2.2 Critical: Missing Softmax layer before loss.* The loss function of your model expects a probability distribution as input (i.e., the likelihood for all the classes sums to 1), but your model is producing un-normalized outputs, called "logits". Logits can be normalized to a probability distribution with a softmax layer.

Many Keras loss function classes can automatically compute softmax for you by passing in a `from_logits` flag:

```
tf.keras.losses.<your loss function class here>(from_logits=True)
```

where specifying `from_logits=True` will tell keras to apply softmax to your model output before calculating the loss function. Alternatively, you can manually add a softmax layer to the end of your model using `tf.keras.layers.Softmax()`.

Stack overflow query: `[keras] is:closed from_logits`

Documentation link: https://www.tensorflow.org/api_docs/python/tf/keras/losses

This error includes a link to the source module where the model was defined.

*A.2.3 Warning: Last model layer has redundant activation.* The last layer of the model has an extra (redundant) nonlinear activation function before Softmax (which is non-linear by itself). This can prevent the model from learning effectively. Remove the `activation` argument from the last layer of your model.

Program context: `Last layer in model <last_layer.name> has activation "<layer_config.activation>"`

This error includes a link to the source module where the model was defined.

## A.3 Parameter Tuning

*A.3.1 Warning: Learning Rate is <high/low>.* [5] The learning rate you set is <higher/lower> than the typical range. This could lead to the model's inability to learn. This can also lead to NaN loss values. You can set your learning rate when you create your optimizer object. Typical learning rates for the Adam optimizer are between 0.00001 and 0.01. For example:

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001))
```

Program context: `Epoch <epoch>: Learning Rate is <lr>`

Documentation link: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

*A.3.2 Warning: Possible overfitting.* The validation loss is increasing while training loss is stuck or decreasing. This could indicate overfitting. However, if validation loss is still trending downwards afterwards, this error could be a false positive. Try adding dropout or reducing the number of parameters in your model. Dropout randomly omits weight updates during training (with some probability) which potentially increases robustness. You can reduce the number of parameters of your model by decreasing the `units` or `filters` parameters of `Dense` or `Conv2D` layers.

Program context: `Epoch <epoch>: training loss changed by <d_loss> while validation loss changed by <d_val_loss>`

Documentation link: https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/Regularizer

*A.3.3 Warning: High dropout rate.* The dropout parameter of the indicated layer(s) is above 0.5, meaning less than half of the gradient updates will propagate through. This can prevent your model from learning. Lower the dropout rate. Typical values fall within [0.1, 0.5].

Program context (for each problem layer): `Layer <index> (layer.name) has dropout rate of <layer.dropout_rate>`

This error includes a link to the source module where the model was defined.

[5] The title of this error dynamically changes if the detected learning rate is above or below a common range.