Amazon SageMaker Debugger: A System for Real-Time Insights into Machine Learning Model Training

Nathalie Rauschmayr [1] Vikas Kumar [1] Rahul Huilgol [1] Andrea Olgiati [1] Satadal Bhattacharjee [1] Nihal Harish [1] Vandana Kannan [1] Amol Lele [1] Anirudh Acharya [1] Jared Nielsen [1] Lakshmi Ramakrishnan [1] Ishaaq Chandy [1] Ishan Bhatt [1] Zhihan Li [1] Kohen Chia [1] Neelesh Dodda [1] Jiacheng Gu [1] Miyoung Choi [1] Balajee Nagarajan [1] Jeffrey Geevarghese [1] Denis Davydenko [1] Sifei Li [1] Lu Huang [1] Edward Kim [1] Tyler Hill [1] Krishnaram Kenthapadi [1]

Abstract

Manual debugging is a common productivity drain in the machine learning (ML) lifecycle. Identifying underperforming training jobs requires constant developer attention and deep domain expertise. As state-of-the-art models grow in size and complexity, debugging becomes increasingly difficult. Just as unit tests boost traditional software development, an automated ML debugging library can save time and money. We present Amazon SageMaker Debugger, a machine learning feature that automatically identifies and stops underperforming training jobs. Debugger is a new feature of Amazon SageMaker that automatically captures relevant data during training and evaluation and presents it for online and offline inspection. Debugger helps users define a set of conditions, in the form of built-in or custom rules, that are applied to this data, thereby enabling users to catch training issues as well as monitor and debug ML model training in real-time. These rules save time and money by alerting the developer and terminating a problematic training job early.

1 Introduction

Prototyping and training machine learning (ML) models is an iterative process that typically consists of much trial and error (Cadamuro & Gilad-Bachrach, 2016; Xin et al., 2018). The ML lifecycle consists of several steps including data preprocessing, data augmentation, model creation and configuration, hyperparameter tuning, and model deployment. A mistake in any of these may lead to an underperforming model, or simply the training not converging at all without any immediate feedback to the model developer. Training deep learning models requires substantial computational resources (Strubell et al., 2020), so it is imperative to discover issues early to shorten the model production cycle and to avoid expensive training runs that lead to sub-par results.

We present *Amazon SageMaker Debugger*[1], a framework-agnostic system that allows ML developers to capture important data during ML model training. Debugger automatically identifies problems such as vanishing or exploding gradients, neuron saturation, and overfitting. It addresses the needs of inexperienced ML practitioners who may not have the expertise to detect and analyze such issues and also of experienced ML scientists who want to directly inspect and analyze tensors. It can be used either as a stand-alone open source library or as a fully managed service, integrated with Amazon SageMaker[2] (Liberty et al., 2020), the ML platform offered by Amazon Web Services (AWS).

Our key contributions are as follows:

- Design and implementation of Debugger, including (1) A concise framework-agnostic API to store, read, and analyze tensors during model training; (2) Automatic recording of all model parameters from XGBoost, PyTorch, TensorFlow, and MXNet models; (3) Integration with Amazon SageMaker as a scalable, secure, and fully managed service (§3).
- Description of Debugger built-in rules to identify common issues & terminate failing training jobs early (§4).
- Deployment results & insights (§5) and case study (§6).

Amazon SageMaker Debugger (hereafter referred to as Debugger) enables users to gain insights into model training without requiring them to write their own framework-specific probes. These insights help to guide model design

[1]Amazon AWS AI. Correspondence to: Nathalie Rauschmayr <rauscn@amazon.com>, Lu Huang <llhu@amazon.com>, Krishnaram Kenthapadi <kenthk@amazon.com>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

[1]https://aws.amazon.com/sagemaker/debugger

[2]https://aws.amazon.com/sagemaker

decisions: in one customer use case, Debugger helped reduce model size by 45% and the number of GPU operations by 33% while improving accuracy (§6). Early termination of training jobs via built-in rules saves time and costs. While the amount saved depends on the given model and training configuration, we have observed savings of up to 30%. Debugger has been used by AWS customers from many different industry segments and has often been helpful to improve model performance. We note that the core component of Debugger is the open source library, *smdebug*[3], which allows users to benefit from many of Debugger's features even if they do not use Amazon SageMaker.

The rest of the paper is organized as follows. We first present a high level overview of debugging in the ML lifecycle in §2. We describe the design and architecture of Debugger in §3 and the built-in rules in §4. We present deployment results and insights in §5, followed by a case study of how Debugger helps in the ML development journey in §6. Finally, we discuss related work in §7 and conclude in §8.

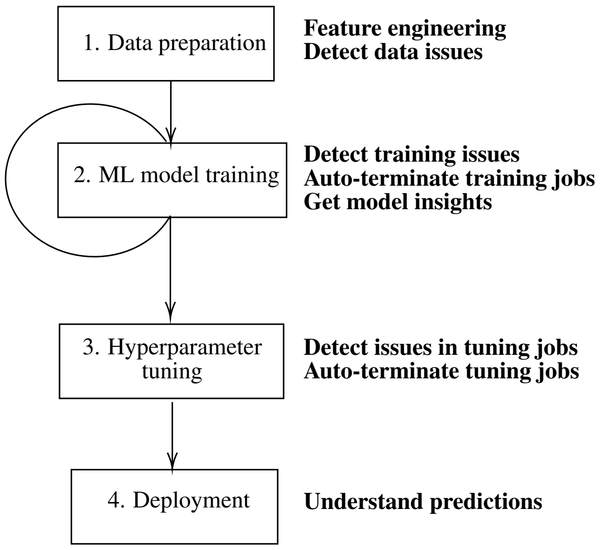


Figure 1. Debugging in the ML lifecycle

2 Debugging in the ML lifecycle

Debugging an ML training script is quite different from debugging traditional software. While a software bug sometimes causes a compilation error, a poor hyperparameter initialization will converge to poor accuracy rather than fail explicitly. So how do we approach ML debugging? The key is to isolate root causes for low model performance, such as bad hyperparameter settings, low model capacity, biased training data, and numerically unstable operations. Fig. 1 shows a high level overview of the ML lifecycle and how debugging can be done at each stage.

1. Data preparation encompasses data cleaning, data preprocessing, and feature engineering. It ensures that data contains representative samples and that training and test sets are randomly sampled and do not overlap. If the data contains too many correlated features the model may be more likely to overfit.

2. Model training is an iterative process, where different configurations and model architectures are applied. If the model has too few parameters it cannot learn meaningful patterns; if it has too many parameters it can likely overfit. For model training to converge it is important to choose the best possible training configuration that encompasses initialization schema, optimizer settings, layer configurations, and other hyperparameters. While most incorrect configurations lead to a non-decreasing loss, the root cause may not always be obvious. For deep learning models, monitoring the distribution of gradients and weight updates across layers in real-time can reveal if the model is over-parameterized or whether optimizer settings were incorrectly set. Similarly, activation outputs can show how many neurons suffer from saturation. Monitoring training and validation losses help to detect overfitting. Overall it is crucial to detect issues early on and stop training runs destined to produce a low-quality model. Another key aspect of debugging is to get a high-level understanding of what the model learns, e.g., which features in the input data are most relevant.

3. Hyperparameter tuning will further refine a good model configuration. At this stage, ML practitioners often create different versions of a model with different hyperparameter combinations, whereby sub-optimal models may be produced. Debugging sub-optimal training jobs ahead of time helps save time and costs.

4. Deployment is the final step after model training and hyperparameter tuning. At this stage, problems such as data drift can occur, where the distribution of the data during inference is significantly different from the distribution of data the model has been trained on. This can lead to unexpected model behavior and debugging could help to understand such behavior and find possible root causes.

3 SageMaker Debugger Design and Architecture

We next describe the design and architecture of Debugger. Before diving deep into its core functionalities we first give a brief overview of Amazon SageMaker to set the context.

[3]https://pypi.org/project/smdebug

3.1 Amazon SageMaker

Amazon SageMaker is a fully managed service provided as part of Amazon Web Services (AWS) that enables data scientists and developers to build, train, and deploy ML models in the cloud at any scale. It offers purpose-built tools for every step of ML development, including data labeling, data preparation, feature engineering, auto-ML, training, automatic model tuning, debugging, deployment, hosting, model monitoring, bias detection, explainability, and workflows. SageMaker automatically spins up the training instance requested by the user, pulls the training image from Amazon Elastic Container Registry (ECR)[4], and downloads data and training scripts into the container.

Once the training is initiated, model data is retrieved by Debugger at specific intervals. The SageMaker platform asynchronously uploads the Debugger data to the customer's S3 bucket.

3.2 Debugger Components

Debugger consists of two major components:

- **smdebug**: this is the core library used to record and load tensors. It provides basic functionalities to read, query, and filter data by step, training phase, and tensor names. smdebug's design strives to satisfy two goals:

  1. ease of use by providing a framework agnostic, concise API for data collection and analysis
  2. customization and extensibility

  With just two lines of code, users can enable data collection and use any of the default collections such as weights, biases, gradients, and losses. During analysis, data is loaded as NumPy arrays independently of the underlying framework that generated the tensors. The library has been open-sourced on PyPI[5] and GitHub[6].

- **Built-in rules**: As part of SageMaker, users can run a set of pre-defined rules that analyze a training job while in progress. Debugger uses the smdebug library to record tensors from the training job which are then uploaded to Amazon Simple Storage Service (S3). Rules are executed in a dedicated Docker container where the data is fetched and checked for certain conditions such as vanishing gradients or overfitting.

3.3 Data Collection

All major deep learning frameworks provide mechanisms to insert hooks into a model in order to run custom code. For instance, users can define forward hooks in PyTorch (Paszke et al., 2019) and MXNet (Chen et al., 2015) that allow one to overwrite the default behavior during a forward pass. Keras (Chollet et al., 2015) provides functions such as `on_batch_end` and `on_batch_begin`. Debugger leverages those mechanisms under the hood to store tensors during model training. Debugger provides users a concise API to enable and configure those hooks.

The first step is to register the smdebug hook on the model. For example, in the case of TensorFlow Keras this can be done in the following way:

```
import smdebug.tensorflow as smd
hook = smd.KerasHook("/opt/ml/tensors")
model.fit(x, y, epochs=10, callbacks=[hook])
```

SageMaker's default deep learning containers for XGBoost, TensorFlow, PyTorch, and MXNet are modified such that the hook registration and configuration is automatically done within the container, so users do not need to modify their training script (i.e., zero code change).

Users can specify collections of tensors to be emitted and they can choose any of the default ones such as weights, biases, gradients, and losses, or define their own custom collection. For example, the following custom collection captures ReLU activation outputs:

```
custom_collection=CollectionConfig(
    name="relu_ouput",
    parameters={
        "include_regex": ".*relu_output",
        "save_interval": "500"
    }
)
```

The regular expression `.*relu_output` specifies tensor names to be included in the collection and tensors are saved every 500 steps. A step consists of one forward pass and one backward pass. Users can also set a mode to distinguish between tensors saved during training and validation phases (if not specified, steps are recorded as global steps). Instead of a `save_interval` one can also specify a list of specific step numbers.

3.4 Data Analysis

Once the training has started, tensors will be saved under the path specified during hook construction. Tensors are written in the form of protobuf files and the SageMaker platform uploads these to Amazon S3. This is done asynchronously to reduce impact on the training job. Users can then fetch, analyze, and query the data using the smdebug library. This can be done while the training is still in progress, thereby supporting advanced real-time analysis of the model training. Debugger's API allows retrieval of tensors given a regular expression, for a particular step, or given training/validation

[4] https://aws.amazon.com/ecr
[5] https://pypi.org/project/smdebug
[6] https://github.com/awslabs/sagemaker-debugger

mode. The data is loaded as framework-agnostic NumPy arrays, so an analysis routine written for data retrieved from a PyTorch model will also work with data from a TensorFlow model.

To access tensors, users create a trial object that takes the location of tensors, which can either be an S3 path or a local path:

```
from smdebug.trials import create_trial
trial = create_trial("/opt/ml/tensors")
```

The trial object allows one to access and query the tensors. The following function returns the list of tensor names:

```
trial.tensor_names(regex=".*")
```

Given the tensor name, we can retrieve the tensor for a specific step:

```
trial.tensor("conv0").value(step)
```

With just four lines of code, we can setup a custom monitoring routine that uses Debugger to automatically fetch and visualize the latest data, for example, the distribution of weights from a specific layer for the latest available validation step. Once the training job has finished, the parameter `trial.loaded_all_steps` is set to True and the monitoring loop is terminated:

```
while not trial.loaded_all_steps:
  steps = trial.steps(mode=modes.EVAL)
  t = trial.tensor("conv1").value(steps[-1])
  plt.hist(t.flatten(), bins=100)
```

3.5 Integration with Amazon SageMaker

As shown in Fig. 2, a SageMaker training job runs in a training container and Debugger will emit tensors to Amazon S3. The built-in rules run on separate instances in a rule container and as such they do not interfere with the training job in terms of resource usage. The rules emit metrics to Amazon CloudWatch[7], a monitoring and observability service, to indicate whether an issue was found or not. Users can then set up a CloudWatch alarm and Lambda[8] function to auto-terminate training jobs after a rule has triggered (Fig. 2). Furthermore users can specify their own custom rules, that run within SageMaker's managed rule container.

While the smdebug library provides all functionalities to write and read tensors via a concise API, the key advantage of running Debugger in the SageMaker environment is that ML models can be trained at large scale, be monitored, and be auto-terminated when issues arise.

[7] https://aws.amazon.com/cloudwatch/
[8] https://aws.amazon.com/lambda/

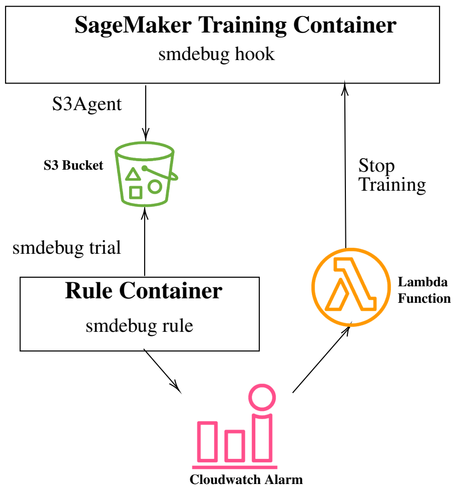


Figure 2. Debugger workflow

3.6 Technical Challenges

This section summarizes the key technical challenges that Debugger addresses:

1. Scale rule analysis by offloading into separate containers
2. Reduce overhead when recording and fetching tensors
3. Separate compute and storage and minimize impact on training

1) Offloading analysis into separate containers on a different instance allows users to run an arbitrary number of rules without impacting the training process itself. Users can write their own custom rules and define the appropriate instance, thereby enabling the execution of compute-intensive analysis at scale.

2) Debugger incorporates several optimizations to reduce storage costs and improve I/O performance, for instance, saving only a subset of tensors, computing aggregations, and supporting different save intervals. Further tensors are stored in an efficient binary file format and data retrieval is optimized with the help of index files that store metadata such as name, shape, and step along with the location of tensor objects in Amazon S3.

3) In the system design, data storage has been decoupled from compute, and a storage component uploads data asynchronously to S3. Being a managed service, Debugger optimizes storage of large amounts of data (in GBs) across local disk and S3 to be able to handle data emission at millisecond intervals.

4 SageMaker Debugger Rules

This section gives an overview of existing built-in rules (Tab. 1).

4.1 Training and test datasets

Real world data-sets are typically imbalanced and noisy. If the model training does not account for these factors, the resulting model is likely to have poor predictive power for the classes with very few samples (Buda et al., 2018) (Guo et al., 2008). There are multiple ways to address this problem: for example, during data-loading more samples can be drawn from the under-represented classes or the loss function can be adjusted to assign a higher penalty to incorrect predictions using class weights (Japkowicz & Stephen, 2002) (Buda et al., 2018). Debugger provides a built-in rule that analyzes the model inputs and predictions. The rule uses the imbalance ratio (Johnson & Khoshgoftaar, 2019) which is defined as: $IR = \frac{max_i \zeta_i}{min_j \zeta_j}$ where ζ is the number of class occurrences. Since Debugger has access to the inputs after data-loading, the rule can determine if preprocessed data has been re-sampled or not. The rule triggers if the imbalance ratio is above a predefined threshold.

Further Debugger provides a rule to verify if data has been correctly normalized checking for zero mean and unit variance. In case of NLP models Debugger can check the ratio of specific tokens given the rest of the input sequence which is useful for optimizing performance.

4.2 Activation functions

Deep neural networks consist of activation functions that determine the output of a node given a set of inputs. This allows the network to learn non-linear mappings and without such non-linear transformations the network would not be able to perform any complex tasks (Leshno et al., 1993). Depending on the type of activation, neurons may suffer from saturation which means that a neuron outputs values close to the asymptotic bounds of the function (Rakitianskaia & Engelbrecht, 2015)(Lau & Lim, 2017). This results in vanishing gradients leading to insignificant parameter updates and preventing the model from learning. This is a well known problem for tanh and sigmoid: activation functions that have been commonly used in the early days of deep learning and may prevent the training of complex models to converge.

Commonly used activation functions such as sigmoid and ReLU can saturate. For inputs outside of the range of [-5,5] sigmoid produces vanishing gradients. Many state of the art models use ReLU activation function, which is defined as the positive part of the argument. As the activation function is set to zero for negative inputs, such inputs produce zero gradients. A model can suffer from the dying ReLU problem, where the gradients become zero due to the activation output being zero (Lu et al., 2019). A dead ReLU can be defined as: $\sum_{i=0}^{n} ReLU(x_i) == 0$ where n is the number of batches. If too many neurons suffer from this problem, then the model cannot effectively learn. Debugger has built-in rules that retrieve activation outputs across steps and determine how many neurons in a model output zero values based on the above formula. If a predefined threshold is exceeded, an issue is raised.

The root causes of neuron saturation and dead ReLUs can be mitigated by scaling the input data to have zero mean and unit variance. This symmetric initialization avoids a dead ReLU problem. Additionally, it is desirable to prevent weights from growing too large which could push activation functions to their asymptotic bounds. ReLU is an unbounded function and adding a penalty term on large activation values helps to keep activations small and sparse (Glorot et al., 2011). Variants such as LeakyReLU, ELU, and SiLU circumvent the dying ReLU problem by allowing small negative outputs. As shown in (Glorot & Bengio, 2010), initialization also plays a crucial role in avoiding neuron saturation which we discuss next.

Problem class	Rules
Datasets	Class imbalance Not normalized data Ratio of tokens in sequence
Loss and accuracy	Loss not decreasing Overfitting Underfitting Overtraining Classifier confusion
Weights	Poor initialization Updates too small
Gradients	Vanishing Exploding
Tensor	All values zero Variance of values too small Values not changing across steps
Activation function	Tanh saturation Sigmoid saturation Dying ReLU
Decision trees	Depth of tree too large Low feature importance

Table 1. Overview of existing Debugger rules

4.3 Parameter initialization

Initialization assigns random values to parameters. If all parameters have the same initial value, they receive the same gradient and as a consequence the model would not be able to learn. If parameters are initialized with too small or large values, it may lead to vanishing or exploding gradients and neuron saturation (Glorot & Bengio, 2010). The goal of initialization is to break the symmetry such that weights connected to the same neuron do not have the same values and can learn independently. (Glorot & Bengio, 2010) show that keeping weight gradients similar across layers helps to achieve quicker training convergence. They propose a normalized initialization requiring: $Var(a^{|l-1|}) = Var(a^l)$ where l is a layer in the model and a the layer output. Debugger has a built-in rule that runs at the start of model training and verifies that layer outputs follow the above properties. A large difference between the variance values may indicate an incorrect initialization. As this prevents convergence, Debugger can automatically stop the training, thereby saving time and costs.

4.4 Gradients

Vanishing gradients becomes a challenge with the increase in the number of layers in a deep neural network, and can prevent deeper layers from learning (Hanin, 2018). Residual connections (He et al., 2016) or BatchNorm layers (Ioffe & Szegedy, 2015) can mitigate this problem. Vanishing gradients can also be caused by activation functions such as sigmoid and tanh that saturate more quickly. Choosing an alternate activation function such as LeakyReLu can stabilize the training. Vanishing gradients is especially a problem in recurrent neural networks, where they can prevent the network from developing a long-term memory (Ribeiro et al., 2020). Contrarily gradients may also explode and gradient clipping can help to stabilize the training (Tan & Lim, 2019). In both cases, it is wasteful to continue training when the model has too small or large gradients. Debugger provides a built-in rule to raise an alarm when statistical properties of the gradients such as min, max, and mean deviate from predefined thresholds. Further, if the majority of weights remain small throughout the training, it may indicate that the model can be pruned, which we discuss in more detail in §5.

4.5 Optimizer settings

Given the gradients, the optimizer decides by how much the parameters are updated. In addition to the learning rate which is given as input to the optimizer, parameters such as momentum, learning rate decay, and weight decay could be provided to stabilize the model training. These values need to be carefully chosen to avoid issues such as vanishing or exploding gradients and the training simply not converging. While Debugger does not currently store optimizer settings, one can capture gradients and weights of consecutive steps. The weight updates can be defined as: $W_i - W_{i+1}$ where i is the step number. Comparing the weight updates with the gradients δ_i for step i can reveal issues with optimizer settings. For instance, in the beginning of the training gradients are typically large, but if parameter updates are small then it may indicate that either the learning rate is too small or weight decay is too large. As a result, the training may not converge or may get stuck in a non-optimal solution. Contrarily if gradients are small but parameter updates are large, then the learning rate is too large which can cause the training to oscillate.

4.6 Loss and accuracy

Incorrect configuration of the model training often prevents the training from converging. One possible scenario is for the training loss to decrease in contrast to the validation loss ($L_{val} \gg L_{train}$). This is a clear indication of overfitting, since the model works well on the data seen during training but is unable to generalize to unseen data. If neither training loss nor validation loss decreases then the model underfits the data and misses relevant relations between features and outputs. On the other hand, if both training loss and validation loss decrease but at a certain point in time validation loss rises, then early stopping needs to be applied to prevent overfitting. This can easily be automated with Debugger. If training loss does not decrease throughout the training, then there can be multiple reasons such as too small learning rate, vanishing/exploding gradients, and incorrect hyperparameters. In such cases, auto-termination is recommended since continuing to train a model that is overfitting or not generalizing is wasteful. Debugger provides built-in rules to detect all the aforementioned issues. We observed that these are the most frequently used built-in rules by customers.

4.7 Decision trees

Debugger can also capture data during XGBoost model training. In gradient boosting a tree with very large depth may be prone to overfitting while a too shallow tree might not capture enough details and hence underfit the data. Debugger provides a built-in rule that checks the depth of the tree in every step and raises an alarm if the tree grows beyond a certain limit. Debugger also provides a built-in rule to record feature importance and shows how often a feature was used in a split. Potentially a feature may not be used at all if it has low variance or if it is highly correlated with another feature. In several customer use-cases, Debugger was used to auto-terminate XGBoost training jobs if a small subset of features held the majority of weights in the model.

5 Deployment Results and Insights

Debugger has helped numerous customers to find training issues, from which we highlight a few use cases. As discussed in §1, Debugger can be used to analyze and visualize tensors in real-time from within a notebook and the analysis can easily be modified while the training is still in progress.

Optimizing visual search model - a customer story: In one customer project the real-time analysis gave significant insights (Carlson et al., 2020). The customer had been training an auto-encoder model consisting of several convolutional and deconvolutional layers, and a bottleneck layer. An auto-encoder learns a compressed representation of the input data and then attempts to reconstruct the original input as closely as possible. The customer observed that the loss itself was not a good indicator of whether the model had learnt meaningful representations. They often noticed that the model produced blurry output images despite the loss having converged. By monitoring the latent space and t-SNE embeddings, they were able spot this issue much earlier. By using Debugger they were able to monitor gradients and activation outputs in real-time and create their own custom real-time visualization for the latent space. This analysis revealed that the model suffered from vanishing gradients and also that most neurons in the bottleneck layer were inactive. Based on these observations they decided to change the model architecture. By using the Debugger built-in rules and auto-terminating feature, they were able to perform a large parameter sweep to create different model configurations and get a more optimal model in a shorter amount of time and with reduced costs. Their final model was 45% smaller in disk size and performed 33% less GPU operations.

Using Debugger for iterative model pruning: Based on these results numerous customers expressed interest in achieving similar model size reductions. As state of the art models are growing in size and the number of parameters, model pruning is becoming increasingly more important. Pruning aims to remove the non important weights without reducing accuracy. Many types of pruning techniques are known, for example, structured versus unstructured pruning, randomly removing weights versus removing by size or rank, and iterative pruning versus one-shot pruning (Blalock et al., 2018). In case of CNNs, iterative filter pruning is known to achieve state of the art results (You et al., 2019) (Li et al., 2017) (Molchanov et al., 2017).

Pruning helps to avoid overfitting and results in smaller models which tend to be more robust. While designing and developing Debugger, we had not anticipated that it could also be used as a core component within a model pruning workflow. Considering the subsequent customer interest, we have developed an end-to-end workflow for iterative model pruning of ResNet models (He et al., 2016) that leverages Debugger in two ways (Rauschmayr et al., 2020c):

1. capturing feature maps, to identify the weights to prune
2. auto-terminating training jobs via a custom rule if the desired minimum model size is reached or if the validation accuracy drops below a predetermined threshold.

Fig. 3 shows the pruning experiment of a ResNet18 model trained on Caltech101 dataset: in each of the 10 pruning iterations the 100 smallest filters are removed and the model is fine-tuned so that it can recover from pruning and re-gain accuracy. Fig. 3 shows that model parameters were reduced from 11 to 2.5 million. Accuracy drops after 8 iterations: the custom Debugger rule terminates the experiment and prevents the execution of jobs that would otherwise produce low quality models.

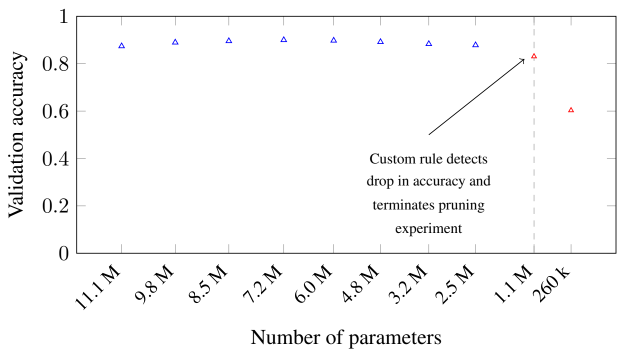


Figure 3. Iterative model pruning of a ResNet18 model. Debugger is used to compute filter ranks and to terminate the pruning experiment via a custom rule that triggers when accuracy drops.

Model understanding with Debugger: As discussed in §2 debugging a model extends beyond just making the model training converge. Quite commonly one wants to understand what the model has learned, for example, to determine if there is bias in the training data or to detect undesirable model behavior. Based on customer requests, we created several end-to-end analysis workflows on top of Debugger specifically for debugging NLP or CV models. For instance, monitoring attention scores in transformer models across heads and layers helps to get a better understanding of the relationships between sentences (Vig, 2019) (Clark et al., 2019). We have created a similar interface as (Clark et al., 2019) on top of Debugger which provides the flexibility to visualize transformer models without having to modify the model. Users can just plug in their own custom transformer model written in their framework of choice.

In the case of CV models, we created an end-to-end analysis workflow that generates saliency maps highlighting class discriminative features of the input (Fig. 8) (Rauschmayr et al., 2020b). Saliency maps typically require the gradients of the outputs with respect to the inputs or feature maps. In this workflow Debugger is used in combination with SageMaker Model Monitor[9]. Model Monitor captures inference requests and predictions of the models hosted on SageMaker and raises an alarm if data drift is detected. Once an alarm is raised, our workflow automatically enables Debugger to capture relevant tensors and compute saliency maps for incoming requests. We discuss saliency maps further in the next section.

[9] `https://aws.amazon.com/sagemaker/model-monitor`

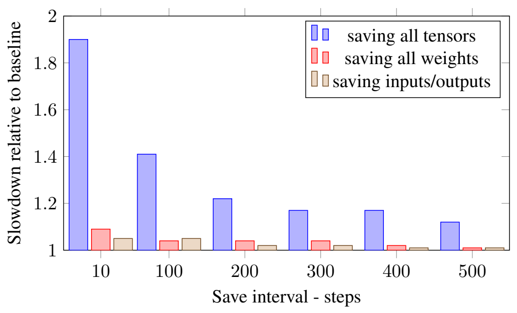


Figure 4. Performance impact in a ResNet18 model training: saving all tensors versus saving only weights and model inputs/outputs

6 Case Study

While we demonstrated the benefit of Debugger via different customer use case examples in §5, we next provide a case study using a publicly available dataset and model (Rauschmayr et al., 2020a). Our goal is to examine how Debugger can simplify and accelerate model development and find critical model bugs throughout the ML lifecycle. For this purpose, we use code from a GitHub repository that fine-tunes a ResNet18 model on the German Traffic Sign dataset (Stallkamp et al., 2012) using PyTorch.

6.1 Performance Impact of Data Collection

First we study the performance impact of Debugger. As described in §3.3, users can specify custom collections of tensors to be emitted. For demonstration purposes, we define a collection where all tensors are saved. The performance penalty is mainly a trade-off between the size of the tensors to store and the `save_interval`. In the worst case it can record each tensor of the model in every step. We fine-tune the ResNet18 model on a p3.2xlarge instance. Fig. 4 shows that the slowdown remains under 1.2 with a `save_interval` above 200 steps. Saving tensors every 10 steps slowed down the benchmark training job by a factor 1.9 when all tensors were saved and by a factor 1.1 when only weights were saved. For the remaining experiments, we mainly analyze the model inputs and outputs. The performance impact for collecting these data is minimal (Fig. 4).

Debugger provides different options to tune the data collection, for instance users can create custom collections and specify different save intervals for each of them or compute aggregates before data is saved. Further benchmark numbers for different frameworks and different state of the art CV and NLP models can be found in the appendix.

6.2 Training with Built-in Rules

After the collections are defined, we run the training with several Debugger built-in rules enabled. Out of those *loss-not-decreasing* and *class-imbalance* rules triggered. The model trained for 10 epochs and reached a final test accuracy of 90.6%. We use Debugger to investigate what caused these two rules to trigger. With just a few lines of codes one can retrieve and visualize the loss values while training is still in progress.

```
loss = "CrossEntropyLoss_output"
losses = trial.tensor(loss).values()
```

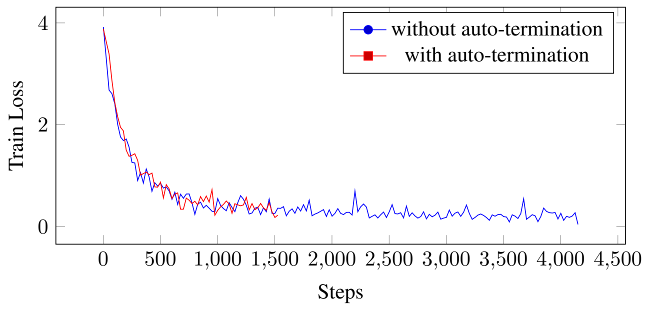


Figure 5. Training with and without Debugger auto-termination

Fig. 5 (blue curve) demonstrates that the default training configuration ran the training for too long. Instead of training for 4000 steps, early-stopping should have been applied after 1000 iterations. As discussed in §3.5, we can enable the auto-termination feature of Debugger and stop the training after a rule triggers. In this example, doing so reduces compute time by more than half (red line).

To investigate the class imbalance issue, we retrieve the inputs into the loss function. The following line returns the labels for the last validation step.

```
steps = trial.steps(mode=modes.EVAL)

labels = "CrossEntropyLoss_input_0"
trial.tensor(labels).value(steps[-1])
```

We can then inspect the number of class instances the model was trained on. As shown in Fig. 6, there is a high imbalance and several classes have less than a hundred instances. To fix this, we change the default configuration of the dataloaders to take the class weights into account and to draw more samples from classes with smaller weights.

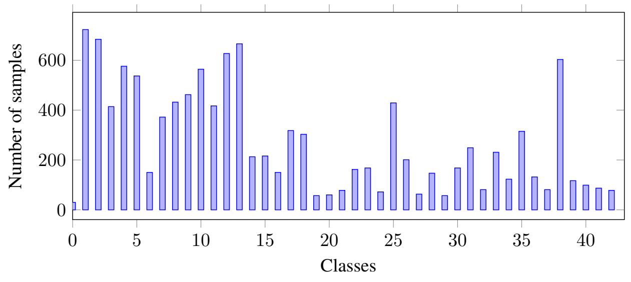


Figure 6. Number of class samples during training

To find out how to further increase test accuracy, we investigate the examples upon which the model made false predictions. We iterate over the predictions and model inputs saved by Debugger and simply select those where the label and prediction do not match:

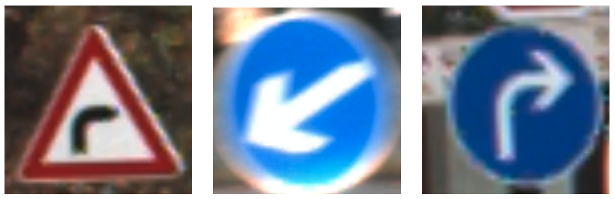
```
labels = "CrossEntropyLoss_input_0"
predictions = "CrossEntropyLoss_input_1"
inputs =  "ResNet_input_0"

for step in trial.steps():

  l = trial.tensor(labels).value(step)
  p = trial.tensor(predictions).value(step)
  i = trial.tensor(inputs).value(step)

  for prediction, label, img in zip(p,l,i):
      if prediction != label:
         plt.imshow(img)
```

Fig. 7 shows the result of above code segment. The analysis reveals that the model is often confused about traffic signs that involve a direction. Clearly this is a severe model bug despite the model achieving a reasonable test accuracy of 90.6%. We then identified the root cause to be the data augmentation pipeline that performs a random rotation on the training data.



Predicted: "Dangerous curve to the left" Groundtruth: "Dangerous curve to the right"

Predicted: "Keep right" Groundtruth: "Keep left"

Predicted: "Turn left ahead" Groundtruth: "Turn right ahead"

Figure 7. Inspecting incorrect predictions with Debugger

6.3 Debugger for Explainability

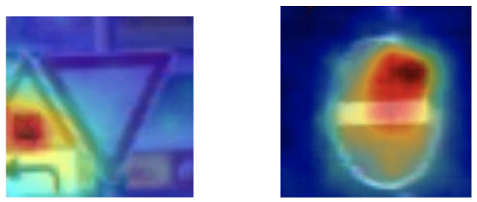
After fixing the training script, we create the final model, that is going to be deployed for production. As discussed in §2, debugging at this stage should help to understand unexpected model behavior, if, for instance, data drift occurs. As described in §5, we developed an end-to-end analysis analysis workflow that computes saliency maps based on the FullGrad method (Srinivas & Fleuret, 2019). This method requires bias and gradients of the outputs with respect to the intermediate activations. The product of both, the bias·gradients, is accumulated for each layer. With Debugger these tensors can automatically be retrieved. For our example application we retrieve all bias and gradients as follows:

```
g = trial.tensor_names(regex=".*gradient")
b = trial.tensor_names(regex=".*bias")

for gradient, bias in zip(g, b):
    trial.tensor(gradient).value(0)
    trial.tensor(bias).value(0)
```

Fig. 8 shows the saliency maps for our model for two example inputs that were incorrectly predicted: red pixels indicate where the model is paying attention to and we infer that the model does not take all features of the inputs into account.

While this section just showcases one example application, it demonstrates the power of Debugger. With the existing built-in rules users can automatically identify issues while the training is in progress and with just a few lines of code users can inspect the model and find critical bugs. Further, users running on Amazon SageMaker run built-in rules free of charge so there is no additional cost for using Debugger.



Predicted: "Stop"   Predicted: "Stop"

Figure 8. Using Debugger to compute saliency maps

7 Related Work

We give an overview of existing debugging and visualization tools for ML. As discussed in (Sculley et al., 2015) maintaining real-world ML systems is difficult and expensive because such systems may have traditional code but also ML-specific issues. In the past years, researchers proposed various methods to address the debugging problem in the ML lifecycle (Cadamuro & Gilad-Bachrach, 2016) (Ma et al., 2018) (Chakarov et al., 2016) (Zhang et al., 2018) (Koh & Liang, 2017) (Kang et al., 2018) (Li et al., 2019) (Wexler et al., 2020). (Cadamuro & Gilad-Bachrach, 2016) develop a framework that given an incorrect model prediction scans the training data and finds a set of items that needs to be altered to mitigate the test errors. Similarly (Ma et al., 2018) propose a technique for conducting state differential analysis to identify neurons that are responsible for misclassification and then perform training input selection to mitigate the issues. (Koh & Liang, 2017) tackle the problem of debugging and understanding incorrect predictions with influence functions that indicate how the model parameters change as a certain training point is up-weighted by an infinitesimal amount. (Zhang et al., 2018) propose a novel algorithm to detect both outliers and incorrect items in the training data. While the aforementioned techniques are very relevant to detect problems and to improve model accuracy, they are limited to a part of the problem: detecting data related issues such as mislabeled training data. (Kang et al., 2018) adapt program assertions for machine learning models to ensure that model outputs are consistent. These assertions are implemented as user-defined callbacks, and are similar to Debugger rules. However, in contrast to Debugger, these assertions only run on model inputs and outputs.

As part of the debugging process, one may make use of visualization and monitoring tools to get visual insights into the model training. TensorBoard is a very popular tool used to track and visualize metrics and model graphs during training (Abadi et al., 2016). Furthermore it can display histograms of weights, biases, and model inputs such as images and text. Prior to training one must specify which data should be saved and the training script needs to be adjusted accordingly. Since TensorBoard 2.3 users can also access logs via DataFrames, which is however limited to scalar values. TensorBoard itself is tied to TensorFlow, however other frameworks such as MXNet and PyTorch have adopted it and provide custom extensions on top of TensorBoard.

Weights & Biases is a flexible and lightweight toolset (Biewald, 2020) that requires only a few lines of code to enable data collection. A background process uploads the data to the cloud where users can create dashboards and other types of visualizations. It supports frameworks such as TensorFlow, Keras, PyTorch, scikit-learn, and XGBoost. Weights & Biases supports experiment tracking, logging hyperparameters, and other metrics which can be exported via CSV files.

TensorWatch is an opensource tool for real-time interactive analysis of deep learning training (Shah et al., 2019). It leverages Jupyter notebooks to visualize data in different forms and supports data streaming and map-reduce style queries. TensorWatch allows users to create arbitrary streams dynamically wherein the training does not have to be stopped and restarted in order to change the data collection.

Debugger has a lot in common with the aforementioned tools, such as supporting multiple frameworks and enabling data collections with just a few lines of code. In contrast to TensorBoard and Weights & Biases, smdebug reads tensors as NumPy arrays allowing users to quickly analyze and visualize the data. Since the data is stored in Amazon S3 in real-time, the risk of running out of disk space during training is minimal.

Compared to existing tools, two key novel aspects of Debugger are as follows: 1) a more flexible API to record and retrieve tensors. For example, if the user wants to get data from the second layer instead of the first layer, then this usually requires to the training to be interrupted and the configuration to be manually updated. In Debugger, with just one line of code, users can enable data collection of tensors from all layers without modifying the training script itself. Users can visualize and analyze the data in any form and change it while the training is still in progress. Users can also easily analyze and compare Debugger data across different training jobs. 2) Automatic error detection is needed in many use cases, for instance, when performing a parameter sweep with hundreds of training jobs. Debugger introduces the concept of rules that allows users to automatically capture issues and to auto-terminate training jobs, therefore saving compute cost and time. The rule analysis is offloaded to separate instances, thereby not impacting the training and enabling the execution of compute-intensive analysis at scale. Further smdebug and SageMaker Debugger are designed to be extensible, so that existing and future advances in tensor analysis and visualizations can be supported. This flexible design allows users to customize Debugger for their specific needs.

8 Conclusion

We presented Amazon SageMaker Debugger, a framework-agnostic system to collect, query, and analyze data from ML model training and to automatically capture issues in real-time using a rich set of built-in rules. We described the design and architecture of Debugger, including its use either as a stand-alone open source library or as a scalable and fully managed service, integrated with Amazon SageMaker. By presenting deployment results, customer use cases, and a case study on a publicly available dataset, we demonstrated that the real-time insights provided by Debugger can help to significantly reduce both development time and cost by finding issues early and auto-terminating problematic training jobs. Debugging and monitoring of ML model training is a ripe area for further research. Instead of limiting ourselves to identification of issues during training, we could proactively anticipate issues during training and apply corrective measures to recover from them. We could also provide semantic insights such as global explanation, consistency, robustness, failure modes, and other characteristics of the model being trained, and their evolution over time.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 265–283, 2016.

Biewald, L. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

Blalock, D. W., Ortiz, J. J. G., Frankle, J., and Guttag, J. What is the state of neural network pruning? In *Proceedings of Machine Learning and Systems (MLSys)*, 2018.

Buda, M., Maki, A., and Mazurowski, M. A. A systematic study of the class imbalance problem in convolutional neural networks. *Neural Networks*, 106:249 – 259, 2018. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2018.07.011.

Cadamuro, G. and Gilad-Bachrach, R. Debugging machine learning models. 2016. CHI Workshop on Human Centred Machine Learning.

Carlson, A., Rauschmayr, N., Jetly, N., and Bhattacharjee, S. Autodesk optimizes visual similarity search model in Fusion 360 with Amazon SageMaker Debugger. https://aws.amazon.com/blogs/machine-learning/autodesk-optimizes-visual-similarity-search-model-in-fusion-360-with-amazon-sagemaker-debugger, 2020. AWS Machine Learning Blog.

Chakarov, A., Nori, A., Rajamani, S., Sen, S., and Vijaykeerthy, D. Debugging machine learning tasks. *arXiv preprint arXiv:1603.07292*, 2016.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*, 2015.

Chollet, F. et al. Keras, 2015. URL https://github.com/fchollet/keras.

Clark, K., Khandelwal, U., Levy, O., and Manning, C. D. What does BERT look at? An analysis of BERT's attention. In *BlackBoxNLP@ACL*, 2019.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.

Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 315–323, 2011.

Guo, X., Yin, Y., Dong, C., Yang, G., and Zhou, G. On the class imbalance problem. In *2008 Fourth International Conference on Natural Computation*, volume 4, pp. 192–201, 2008.

Hanin, B. Which neural net architectures give rise to exploding and vanishing gradients? In *Advances in Neural Information Processing Systems*, pp. 582—591. 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pp. 448–456, 2015.

Japkowicz, N. and Stephen, S. The class imbalance problem: A systematic study. *Intelligent Data Analysis*, pp. 429—449, 2002.

Johnson, M. J. and Khoshgoftaar, M. T. Survey on deep learning with class imbalance. In *Journal of Big Data*, pp. 2196–1115, 2019.

Kang, D., Raghavan, D., Bailis, P., and Zaharia, M. Model assertions for monitoring and improving ML models. In *Proceedings of Machine Learning and Systems 2 (MLSys 2018)*, 2018.

Koh, P. W. and Liang, P. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning*, pp. 1885–1894, 2017.

Lau, M. M. and Lim, K. H. Investigation of activation functions in deep belief network. In *2017 2nd International Conference on Control and Robotics Engineering (ICCRE)*, pp. 201–206, 2017.

Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861 – 867, 1993. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(05)80131-5.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient ConvNets. In *5th International Conference on Learning Representations (ICLR)*, 2017.

Li, L., Bai, Y., and Wang, Y. Manifold: A model-agnostic visual debugging tool for machine learning at Uber. In *USENIX Conference on Operational Machine Learning (OpML)*, 2019.

Liberty, E., Karnin, Z., Xiang, B., Rouesnel, L., Coskun, B., Nallapati, R., Delgado, J., Sadoughi, A., Astashonok, Y., Das, P., Balioglu, C., Chakravarty, S., Jha, M., Gautier, P., Arpin, D., Januschowski, T., Flunkert, V., Wang, Y., Gasthaus, J., Stella, L., Rangapuram, S., Salinas, D., Schelter, S., and Smola, A. Elastic machine learning algorithms in Amazon SageMaker. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 731–737, 2020.

Lu, L., Shin, Y., Su, Y., and Karniadakis, G. E. Dying ReLU and initialization: Theory and numerical examples. *CoRR*, abs/1903.06733, 2019.

Ma, S., Liu, Y., Lee, W., Zhang, X., and Grama, A. MODE: Automated neural network model debugging via state differential analysis and input selection. *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. Pruning convolutional neural networks for resource efficient inference. In *5th International Conference on Learning Representations (ICLR)*, 2017.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

Rakitianskaia, A. and Engelbrecht, A. Measuring saturation in neural networks. In *2015 IEEE Symposium Series on Computational Intelligence*, pp. 1423–1430, 2015.

Rauschmayr, N., Huang, L., and Bhattacharjee, S. Detecting hidden but non-trivial problems in transfer learning models using Amazon SageMaker Debugger. https://aws.amazon.com/blogs/machine-learning/detecting-hidden-but-non-trivial-problems-in-transfer-learning-models-using-amazon-sagemaker-debugger, 2020a. AWS Machine Learning Blog.

Rauschmayr, N., Kumar, V., and Bhattacharjee, S. Detecting and analyzing incorrect model predictions with Amazon SageMaker Model Monitor and Debugger. https://aws.amazon.com/blogs/machine-learning/detecting-and-analyzing-incorrect-model-predictions-with-amazon-sagemaker-model-monitor-and-debugger, 2020b. AWS Machine Learning Blog.

Rauschmayr, N., Simon, J., and Bhattacharjee, S. Pruning machine learning models with Amazon SageMaker Debugger and Amazon SageMaker Experiments. https://aws.amazon.com/blogs/machine-learning/pruning-machine-learning-models-with-amazon-sagemaker-debugger-and-amazon-sagemaker-experiments, 2020c. AWS Machine Learning Blog.

Ribeiro, A. H., Tiels, K., Aguirre, L. A., and Schön, T. Beyond exploding and vanishing gradients: Analysing RNN training using attractors and smoothness. In *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 2370—2380, 2020.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*, pp. 2503–2511, 2015.

Shah, S., Fernandez, R., and Drucker, S. M. 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.

Srinivas, S. and Fleuret, F. Full-gradient representation for neural network visualization. In *Advances in Neural Information Processing Systems*, pp. 4124–4133. 2019.

Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural networks*, 32: 323–332, 2012.

Strubell, E., Ganesh, A., and McCallum, A. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 13693–13696, 2020.

Tan, H. H. and Lim, K. H. Vanishing gradient mitigation with deep learning neural network optimization. In *7th International Conference on Smart Computing Communications (ICSCC)*, pp. 1–4, 2019.

Vig, J. Visualizing attention in transformer-based language representation models. *arXiv preprint arXiv:1904.02679*, 2019.

Wexler, J., Pushkarna, M., Bolukbasi, T., Wattenberg, M., Viégas, F., and Wilson, J. The What-If tool: Interactive probing of machine learning models. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):56–65, 2020.

Xin, D., Ma, L., Song, S., and Parameswaran, A. G. How developers iterate on machine learning workflows - A survey of the applied machine learning literature. In *KDD Interactive Data Exploration and Analytics (IDEA) Workshop*, 2018.

You, Z., Yan, K., Ye, J., Ma, M., and Wang, P. Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 2133–2144. 2019.

Zhang, X., Zhu, X., and Wright, S. J. Training set debugging using trusted items. In McIlraith, S. A. and Weinberger, K. Q. (eds.), *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, pp. 4482–4489, 2018.

A Additional Benchmark for CV and NLP Models

This section studies the performance impact of smdebug on the model training using example state of the art model for CV and NLP. The first benchmark is training a Tensorflow ResNet50 model on 4 GPUs (p3.16xlarge) using Tensorflow's MirroredStrategy. As discussed in section 3.3 a collection specifies a list of tensors that are saved at a specific interval or step. For demonstration purposes, we define the following custom collection:

```
CollectionConfig(
    name="all",
    parameters={
        "include_regex": ".*",
        "save_interval": 100
    })
```

This configuration stores tensors including gradients, weights, biases, outputs and inputs. Fig. 9 shows that the slowdown remains under 1.2 with a `save_interval` above 200 steps. Saving tensors every 10 steps slowed down the benchmark training job by a factor 2.5 when all tensors were saved and by a factor 1.6 when only weights were saved.

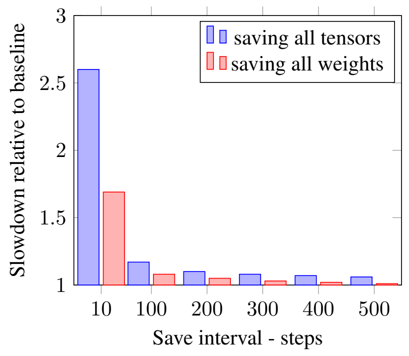


Figure 9. Performance impact in a ResNet50 model training: saving all tensors versus saving only weights

Fig. 10 shows the slowdown in training an MXNet BERT large model on the Stanford Question Answering dataset on 1 GPU (p3.2xlarge instance). The benchmark stores only the weights at different intervals. With a `save_interval` of 10 steps slowdown is below 1.5 and much lower compared to the TensorFlow benchmark. The difference can be explained with single versus multi-GPU training. In case of TensorFlow benchmark more data was saved from multiple GPUs.

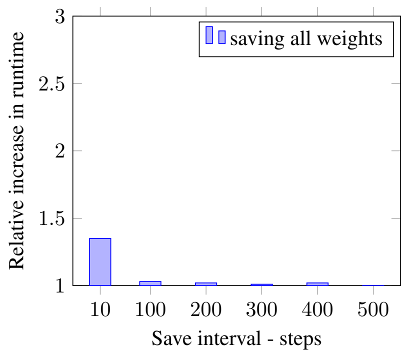


Figure 10. Performance impact in a BERT model training: saving weights