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 Doddla 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 Debugger1, 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 over-fitting. 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 SageMaker2 (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

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

2https://docs.aws.amazon.com/sagemaker/latest/dg/train-debugger.html
decisions: in one customer use case, Debugger helped reduce model size by 45% and 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, smdebug3, 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 §3.5 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.

2 DEBUGGING IN THE ML LIFECYCLE

Debugging an ML training script is quite different from traditional software. While software bugs sometimes cause 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, or 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 don’t overlap. If the data contains too many correlated features the model may be more likely to over-fit.

2. Model training is an iterative process, where different configurations and model architectures are applied. If the model has too few parameters it can not learn meaningful patterns, if it has too many parameters it will more likely overfit. For model training to converge it’s 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. 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 over-fitting. Overall it is important to detect issues early on and to stop training runs destined to produce a low-quality model. Another important aspect of debugging is to get a high-level understanding of what the model learns e.g. which features in the input data are more 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 saving 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 to the distribution of data the model has been trained on. This can lead to ambiguous model behavior and debugging should help to understand those predictions and find possible root causes.

3 SAGEMAKER DEBUGGER DESIGN AND ARCHITECTURE

Here we describe the design and architecture of Debugger. Before diving deep into its core functionalities we will 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 an ML platform provided as part of Amazon Web Services (AWS) enabling customers to do scalable and cost-effective machine learning in the cloud. It removes the heavy lifting from each step in the ML development cycle such as data labeling, training at scale, automatic hyperparameter tuning, debugging, model monitoring, and deployment. 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 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 the 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 that are then uploaded to Amazon Simple Storage Service (S3). Rules are executed in a dedicated Docker container where they fetch the data and check 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., 2017) and MXNet (Chen et al.) 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, in the case of TensorFlow Keras this can be done in the following way:

```python
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 (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:

```python
custom_collection=CollectionConfig(
    name="relu_output",
    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 is defined as one forward and one backward pass. Users can also set a mode to distinguish between tensors saved from 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 and allows for 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 mode. The data is loaded as framework-agnostic NumPy ar-
Amazon SageMaker Debugger

rays, 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:

```python
1. from smdebug.trials import create_trial
2. 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:

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

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

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

With just 4 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 terminates the monitoring loop:

```python
1. while not trial.loaded_all_steps:
2.     steps = trial.steps(mode=modes.EVAL)
3.     t = trial.tensor("conv1").value(steps[-1])
4.     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/](https://aws.amazon.com/cloudwatch/)

\(^8\) [https://aws.amazon.com/lambda/](https://aws.amazon.com/lambda/)

### 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) Several optimizations were applied 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, 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 those factors, it will lead to a model that has low or no predictive power for the classes with few samples (Buda et al., 2018) (Guo et al., 2008). There are different ways this problem can be addressed: for instance during data-loading more samples can be drawn from the under-represented classes or one can adjust the loss function 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 which is defined as (Johnson & Khoshgoftaar, 2019): $IR = \frac{\max_{i} \; \sum_{j} \; \sigma_{i,j}}{\min_{i} \; \sum_{j} \; \sigma_{i,j}}$, where $\sigma$ 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 pre-defined 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 (Rakitskai & 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 function such as sigmoid and ReLU can saturate. For inputs outside of the range of [-5,5] sigmoid produces vanishing gradients. In case of ReLU, negative inputs produce zero gradients. Many state of the art models use ReLU activation which is a non-linear rectified unit: a function that is linearly increasing for increasing positive values and outputs zero otherwise. 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 can’t effectively learn. Debugger has built-in rules that retrieve activation outputs across steps and based on the above formula identify how many neurons in a model output zero values. If a pre-defined 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, avoiding weights 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). Activation functions such as LeakyReLU, Elu, Selu etc. are non linear functions that allow small negative outputs and circumvent the dying ReLU problem. As shown in (Glorot & Bengio, 2010), initialization also plays a crucial role in avoiding neuron saturation and this will be discussed in the next section.

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

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 con-
nected to the same neuron do not have the same values and can learn independently. (Glorot & Bengio, 2010) shows that keeping weight gradients similar across layers helps to achieve quicker training convergence. They propose a normalized initialization that holds: \( \text{Var}(a^{l-1}) = \text{Var}(a^l) \) 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 variances may indicate an incorrect initialization. Since this prevents convergence, stopping the training is advised and can be automated with Debugger saving users time and costs.

4.4 Gradients

Deep neural networks consists of many layers and quite commonly gradients may vanish the more layers a network has (Hanin, 2018). This prevents deeper layers learning because they only receive small gradients. 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 instead an activation function like 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, continuing training when the model has too small or large gradients, would not be useful. Debugger provides a built-in rule to determine the statistical properties of the gradients such as min, max, mean and raises an alarm if \( \sum_{i=1}^{n} \delta_i < t \) where \( t \) is a predefined threshold. If the majority of weights remain small throughout the training, it may indicate that the model can be pruned. This will be discussed in more detail in section 5.

4.5 Optimizer settings

Given the gradients, the optimizer decides by how much the parameters are updated. Stochastic gradient descent algorithms take at least a learning rate. Additional parameters such as momentum, learning rate decay and weight decay allow one to stabilize the model training. Those values need to be carefully chosen, otherwise it may lead to vanishing or exploding gradients or 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 \( \delta_i \) for the 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 the learning rate is too small or weight decay is too large. This means the training will 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 and the training starts oscillating.

4.6 Loss and accuracy

If the model training is incorrectly configured then this almost always leads to the problem that the loss during training won’t converge. Multiple possible scenarios can occur such as training loss decreases in contrast to the validation loss \( L_{val} \gg L_{train} \). This is a clear indication of over-fitting, a model that works well on the data it has seen during training, but that can not generalize to unseen data. If neither training nor validation loss decrease then the model is under-fitting the data and misses relevant relations between features and outputs. If training 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 learning rate too small, vanishing/exploding gradients, or incorrect hyperparameters etc. In such cases, auto-termination is recommended because continuing to train a model that is over-fitting or not generalizing, would be wasted compute and experimentation time. Debugger provides built-in rules to detect all the aforementioned issues. 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 large depth may be prone to over-fitting because it captures too many details. A tree too shallow might not capture enough details. 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 subset of features are holding the majority of weights in the model.

5 Deployment Results and Insights

Debugger has helped numerous customers to find training issues and this section highlights a few use cases. As described in the introduction 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.). They had been training an auto-encoder model consisting of several convolutional, deconvolutional layers and a bottleneck layer. An auto-encoder learns a compressed representation of the input data and its goal is to reconstruct the original input as closely as possible. The customer observed that the loss itself was not a good indicator whether the model had learnt meaningful representations. They often found the model would produce blurry output images despite the loss having converged. By monitoring the latent space and t-SNE embeddings, they were able spot such an 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 most neurons in the bottleneck layer were inactive. Based on these observations they changed 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 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 asked how they can achieve similar model size reductions. As state of the art models are growing in size and number of parameters, model pruning is becoming increasingly more important. Pruning aims to remove the non important weights without reducing accuracy. Many different techniques exist such as structured versus unstructured pruning, randomly removing weights versus removing by size or rank, iterative pruning versus one-shot pruning. Numerous publications have shown that in case of CNN’s iterative filter pruning achieves state of the art results (You et al., 2019) (Li et al.) (Molchanov et al.). For further details on pruning we refer to (Blalock et al., 2018).

Pruning helps to avoid overfitting and smaller models tend to be more robust. When the development of Debugger was started, it was not foreseen that it could also be used as core component within a model pruning workflow. 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., c):

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.

**Model understanding with Debugger:** As discussed in Section 2 debugging a model is more than 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 ambiguous model behavior. Based on customer requests several end-to-end analysis workflows were created on top of Debugger that can be used 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 that highlight class discriminative features of the input (Fig. 8) (Rauschmayr et al., b). 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 ModelMonitor⁷. ModelMonitor captures inference requests and predictions of the models hosted on SageMaker and

---

⁷https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-model-monitor.html
Amazon SageMaker Debugger 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. For more details on saliency maps we refer to the subsequent section.

10

200

300

400

500

1.8

1.6

1.4

1.2

1.0

0.8

0.6

0.4

0.2

0.0

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

6 CASE STUDY

The previous section has demonstrated the value of Debugger via different example customer use cases, however detailed reporting of such is limited due to restrictions surrounding customer data. Therefore in this section we examine how Debugger can simplify and accelerate model development and find critical model bugs throughout the ML lifecycle using a publicly available dataset and model (Rauschmayr et al., a). Specifically we have taken 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 section 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. In the subsequent subsections we mainly analyse the model inputs and outputs. The performance impact for collecting those 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 Run the 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 those rules to trigger. With just few lines of codes one can retrieve and visualize the loss values as 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 shown in section 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.

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 in-
puts saved by Debugger and simply select those where the label and prediction do not match:

```python
defutter = "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 decent test accuracy of 90.6%. The root cause of this is the data augmentation pipeline that performs a random rotation on the training data.

```python
    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 2 example inputs that were incorrectly predicted: red pixels indicate where the model is paying attention to and it reveals 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 as 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.

### 7 RELATED WORK

This section gives an overview of existing debugging and visualization tools for ML. As stated in (Sculley et al., 2015) maintaining real-world ML systems is difficult and expensive because such system 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). The authors of (Cadamuro & Gilad-Bachrach, 2016) developed 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 authors of (Ma et al., 2018) propose a technique...
for conducting state differential analysis to identify neurons that are responsible for mis-classification and then perform training input selection to mitigate the issues. (Koh & Liang, 2017) tackles 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 subset of problem: detecting data related issues such as mis-labeled training data. Authors of (Kang et al., 2018) adapt program assertions to machine learning models that ensure that model outputs are consistent. These assertions are implemented as user-defined callbacks and they are similar to Debugger rules. In contrast to Debugger those assertions only run on model inputs and outputs.

As part of the debugging process, one may require to get visual insights into the model training. Several visualization and monitoring tools exist. TensorBoard is among the most 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 TensorFlow 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 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 is uploading 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, XGBoost and others. Weights & Biases provides 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 meaning that 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 which just a few lines of code. In contrast to TensorBoard and Weights & Biases smdebug reads tensors as NumPy arrays allowing users to quickly analyse 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.

The two most novel aspects of Debugger are: 1) a more flexible API to record and retrieve tensors. For example, if user wants to get data from second layer instead of first layer, then this usually requires to interrupt the training and to manually update the configuration. In Debugger, with just one line of code, users can enable data collection of weights from all layers without modifying the training script itself. User can visualize and analyze the data in any form and change it while the training is still in progress. Users can also easily analyse and compare Debugger data across different training jobs. 2) Many use cases require automatic error detection which is for instance the case 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, thus it does not impact the training and enables the execution of compute-intensive analysis at scale. Further smdebug and SageMaker Debugger is designed to be extensible, so that existing and future advances in tensor analysis and visualizations can be supported. This allows users to customize Debugger for their specific needs.

8 Conclusion

We presented 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


Amazon SageMaker Debugger


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.

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.

### 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:

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