High-bypass Learning: Automated Detection of Tumor Cells That Significantly Impact Drug Response

Justin M. Wozniak,* Hyunseung Yoo,* Jamaludin Mohd-Yusof,† Bogdan Nicolae,‡
Nicholson Collier,§ Jonathan Ozik,§ Thomas Brettin,¶ and Rick Stevens¶

* *Data Science and Learning, Argonne National Laboratory, Lemont, IL, USA*
† *Computer, Computational and Statistical Sciences, Los Alamos National Laboratory, Los Alamos, NM, USA*
‡ *Mathematics and Computer Science, Argonne National Laboratory, Lemont, IL, USA*
§ *Decision and Infrastructure Sciences, Argonne National Laboratory, Lemont, IL, USA*
¶ *Computing, Environment, and Life Sciences, Argonne National Laboratory, Lemont, IL, USA*

***Abstract*—Machine learning in biomedicine is reliant on the availability of large, high-quality data sets. These corpora are used for training statistical or deep learning -based models that can be validated against other data sets and ultimately used to guide decisions. The quality of these data sets is an essential component of the quality of the models and their decisions. Thus, identifying and inspecting outlier data is critical for evaluating, curating, and using biomedical data sets. Many techniques are available to look for outlier data, but it is not clear how to evaluate the impact on highly complex deep learning methods. In this paper, we use deep learning ensembles and workflows to construct a system for automatically identifying data subsets that have a large impact on the trained models. These effects can be quantified and presented to the user for further inspection, which could improve data quality overall. We then present results from running this method on the near-exascale Summit supercomputer.**

1. Introduction

Biomedical research is entering a new phase in which highly automated learning methods may be able to quickly and accurately produce highly specific predictions, inferences, and insights. These capabilities have the potential to greatly accelerate the state of the field as well as have direct impact in clinical settings in diagnosis and treatment. Advanced learning techniques, however, are limited by the quality of the input training data available to them. Continued advancement in this area will rely on the availability of large, high-quality data sets. Ideally, similar data sets could be ingested by both traditional/statistical and/or emerging deep learning -based models. These techniques can then be cross-validated against each other as well as against other data sets. Such careful cross-validation is required before these methods can be allowed to guide decisions. Obtaining, evaluating, and curating large data sets is thus a critical aspect of modern approaches in machine learning for biomedicine. Thus, there is a strong need for automated techniques to identify and inspect outliers or anomalous data.

Many techniques are available to look for outlier data, but it is not clear how to evaluate the impact on highly complex deep learning methods. In this paper, we use deep learning ensembles and workflows to construct a system for automatically identifying data subsets that have a large impact on the trained models. These effects can be quantified and presented to the user for further inspection, which could improve data quality overall. We then present results from running this method on the near-exascale Summit supercomputer.

A variety of methods are available to identify outliers and anomalies, ranging from standard statistical methods to those which utilize ML methodologies themselves. Among the latter Isolation Forest [1] and Extended Isolation Forest [2] methods seem to be the most robust for the multivariate data sets we are concerned with. Within the context of the machine learning in cancer, the nature of the anomalies varies across the problem sets. Within the studies based on drug response data, there are likely to be both true anomalies (drugs which are unusually effective or ineffective, cancers which are unusually resistant) as well as noise due to experimental conditions or other confounding factors. In studies where the data derives from numerical simulations, what appear as anomalies could be real rare events or results of numerical errors. In clinical text analysis, the problem consists of multi-task classification of reports which may suffer from various forms of noise due to missing or incorrect labels, missing data or poor transcription.

In this work, we consider the problem of predicting tumor dose response across multiple data sources. Uno (Figure 1) uses multiple cell RNA and drug response data sets, and it is desirable to automatically discover which data sets are most beneficial to automated drug response prediction and which are detrimental. Information about either case could be useful to researchers. The beneficial cases could point to experimental conditions of broad applicability to a wide range of cancers and patients, and could be used to generate smaller data sets for rapid, prototype studies. The

detrimental cases could indicate at least two possibilities: 1) that a learning approach lacks the capability to assimilate a certain class of data, or 2) that a subset of the data is anomalous or erroneous, necessitating data correction.

The approach taken here is a "high-bypass learning" approach, in which machine learning is not used primarily to make predictions, but rather to produce statistics about learning that reveal information about the underlying data. We apply this method to the analysis of experimental cancer data.

The remainder of this paper is organized as follows. First, we describe more detail about the machine learning application of interest and its data. Second, we describe the data analysis we desire to perform and the computing infrastructure used to accomplish it. Third, we present results from performing the analysis under a range of conditions. Finally, we offer conclusions and opportunities for future work.

2. Methods

In this section we describe the neural network -based cancer problem of interest and our approach using a novel training and data analysis workflow.

2.1. Uno: A neural network for drug response prediction

A highly desirable goal in the application of deep learning is to enable cross-comparison of cancer studies and integrate results into a unified drug response model. The overall idea is to train a neural network (NN) on a corpus of tumor dose responses based on given combinations of cell RNA sequences, drug descriptors, and drug fingerprints. The model can then provide predictions for combinations of RNA sequences and drugs that it was not trained on. The goal of this paper is to study the effects of training ensembles of on different subsets of the data and use the trained NN to make predictions on the held-out data. By incrementally training on different subsets as part of a controlled workflow, various observations may be made about the components of the data.

The Uno benchmark [3] integrates experimental cancer data from 2.5 million samples across six research centers to examine study biases and to build a unified drug response model. The associated manually designed DNN has four input layers: a cell RNA sequence layer, a dose layer, a drug descriptor layer, and a drug fingerprints layer. It has three feature-encoding submodels for cell RNA sequence, drug descriptor, and drug fingerprints. Each submodel is composed of three hidden layers. The last layer for each of the submodels is connected to the concatenation layer along with the dose layer. This is connected to three hidden layers. The scalar output layer is used to predict tumor dose.

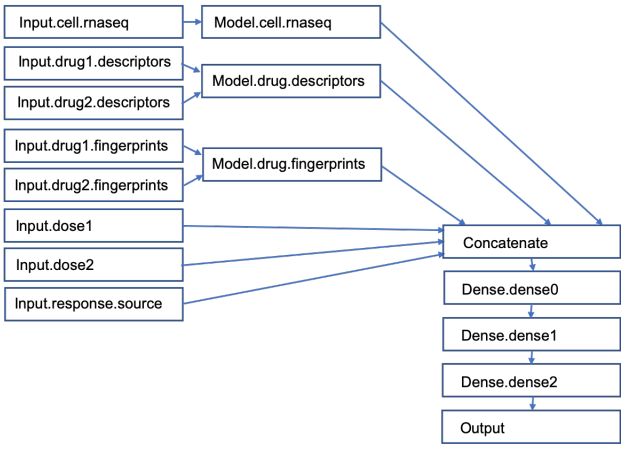

Figure 1: Uno neural network architecture.

2.2. The Uno data set

Uno training data consists of a composite data set that can be constructed in multiple ways. Multiple user-selected data sets can be dynamically assembled before training by selecting from drug response, gene expression, and molecular descriptors of interest; all from separate studies.

For this study, a master dataset was created. The drug response, gene expression and molecular descriptor datasets were joined to create a single data frame called "Top21" consisting of 529,940 rows and 6,213 columns, which will be used for training and validation. (This file is available for download.) Samples are defined by the composite (cell id, drug id), which map to a label, the tumor dose response.

2.3. The Sublearning Workflow

In this section, we describe the workflow used to train on partial subsets of the overall data and integrate statistics from the trained models to produce insight into the underlying data.

Conceptually, the goal of this workflow is to train on "most" of the data set, with some small portion left out. Then, in subsequent "stages", additional data is added to the total training set (without removing any data from the training set). The previous, partially-trained model is restarted and thus is expected to quickly incorporate the new data. This process is depicted in Figure 2. The x and y axes could represent cell lines and candidate drugs, where the data is the drug response. Thus, gray squares consist of cell line $\times$ drug response values that are used for training in that stage, where white squares are left out. Each stage consists of all possible subsets; thus Stage 1 trains 4 different models on 4 different training sets, Stage 2 trains 16 models (based on the models from Stage 1), and so on.

We define N as the number of possible subsets at each stage, and S as the number of stages. In the figure, N=4 and S=3. The workflow is not specific to our Uno-based application and could be used to train on any model data

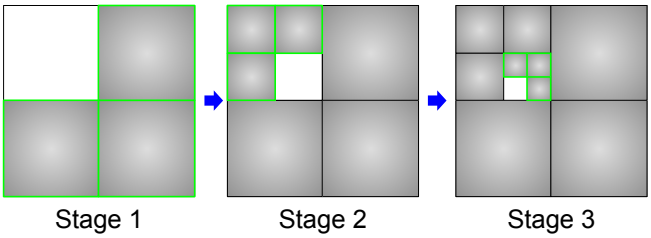

Figure 2: Data subsetting conceptualization. The subsets of training data shown in green are added to the total training data set (gray) at each stage (validation data is left out).

that can be subdivided. In practice, we used N=4 and S=5, thus training 1,364 models in each workflow. Each potential model is considered a "node" in our workflow.

There is thus an exponential number of models to be trained in the number of stages, however, the later models are only adding a small amount of data and thus are expected to train very quickly.

The contents of the Top21 data set may be divided as follows. At each node in the workflow, a training set is created by starting with the training set of its parent (or of the empty set for Stage 1). An additional fraction of the parental validation set is provided to each child node of the parent. This additional data is divided into two subsets, with one subset being added to the training set and the other subset becoming the validation set at that node. The data is divided based on cell id only, not by drug. It is important to note that each node training set is different, and members of the validation set are not in the training set for the node or in the training set for any ancestor node. A diagram of this process is shown in Figure 3.

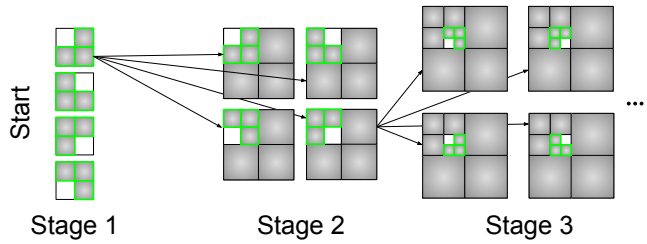

Figure 3: DAG diagram of learning dependencies in Sub-learning workflow. (See Figure 2 for context.)

At each stage transition, the parent NN weights are passed to the child before training is restarted on the new data subset. Thus, a large number of "trajectories" through the training data are made, each resulting in a leaf node in the final stage. The object is to obtain training statistics for each node in the workflow and compare with its parent. A variety of measurements are made using the computing infrastructure described in the following.

2.4. Computing infrastructure

In this section we describe the computing framework used in this work. The Cancer Deep Learning Environment (CANDLE) is an open framework for rapid development, prototyping, and scaling deep learning applications on high-performance computing (HPC) systems. CANDLE was initially developed to support a focused set of three pilot applications jointly developed by cancer researchers and deep learning / HPC experts, but is now generalizable to a wide range of use cases. It is designed to ease or automate several aspects of the deep learning applications development process. CANDLE runs on systems from individual laptops to OLCF Summit, and enables researchers to scale application workflows to the largest possible scale.

Machine learning (ML) has the capability to transform many scientific problems. In response to the growing power of ML techniques and the increasing available computing power at large scale computing facilities, the U.S. Department of Energy Exascale Computing Project (ECP) launched the Cancer Distributed Learning Environment (CANDLE). CANDLE is developing a suite of software to support scalable deep learning on DOE supercomputing resources. While the CANDLE project is explicitly aimed at supporting deep learning in the three cancer pilot projects in the near-term, its longer-term goal is to support a wide variety of deep learning applications across DOE science domains.

Our system, CANDLE/Supervisor, is a workflow application framework used to develop multiple deep learning workflows. It is a Supervisor in the sense that it manages the execution of many (thousands) of concurrent, subordinate deep learning training or inference runs. CANDLE/Supervisor was designed to run a wide range of model types, including Keras, TensorFlow, and PyTorch. It is also generic with respect to the application area. For example, CANDLE runs models in RNA expression data, molecular dynamics data, and clinical text data.

Key components of the CANDLE architecture are shown in Figure 4. A handful of top-level workflows have been developed by the CANDLE team, but these can easily be modified or extended. These include:

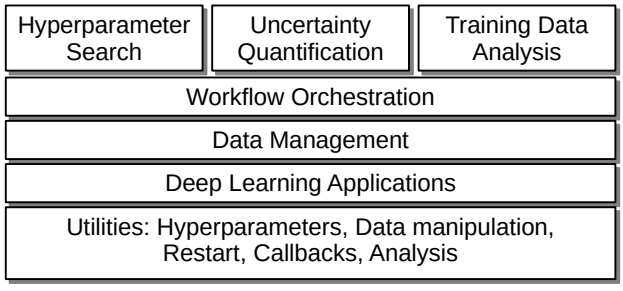

Figure 4: CANDLE component architecture.

2.5. Model state transfer for sublearning

As mentioned in Section 2.3, at each stage transition, the parent DNN weights are passed to the child before training is restarted on the new data subset. However, it is important to note that each child is an independent learning instance that may run on a different device (GPU) on the same compute node or even a different compute node altogether. In this case, the problem of efficient transfer of the model state becomes an important, because it affects the performance and scalability of the entire sublearning workflow.

A naive solution to this problem would be to checkpoint the state of the model on the parent and then restart multiple children from the same checkpoint. However, such a solution has several disadvantages. First, it is important to note that the parent does not necessarily have to be killed: instead, it could continue as one of the children, which effectively reduces the overhead of instantiating multiple children from the same checkpoint. Second, the parent does not necessarily have to be stopped for the duration of checkpointing: instead, asynchronous checkpointing techniques that take advantage of fine-grain parallelism as implemented in modern deep learning runtimes (e.g., Tensorflow) can be used to mask the checkpointing overhead.

Our previous work DeepFreeze [4] illustrates such techniques based on the idea of augmenting the execution graph with fine-grain tensor copy operations, which can run in parallel with gradient computations and weight updates involving different layers during the back-propagation. Using this approach, a full checkpoint of the DNN model can be produced in-memory or on local storage with minimal impact on the learning performance, which can then be again transferred asynchronously to the memory of a different GPU and/or remotely to a different compute node. Third, even if the parent can continue as a child with minimal overhead, asynchronous checkpointing may cause significant overhead in instantiating the other children, because they need to wait until the checkpoint becomes available.

To mitigate this issue, the notion of *cloning* the DNN model can be used instead: unlike checkpointing, it involves the notion of replicating the DNN model state such that three objectives are simultaneously addressed: (1) introduce as little runtime overhead as possible on the initial training instance; (2) minimize the amount of time necessary to construct the replicated training instance (to avoid wasting core hours; (3) continue training on the replicated instance as early as possible (to finish the work as early as possible). By comparison, checkpointing addresses only objective (1), therefore missing the opportunity to co-optimize the capture of the DNN model state and its reconstruction as one or many independent replicas.

As illustrated in our previous work DeepClone [5], the augmentation techniques for the execution graph during the back-propagation introduced by DeepFreeze can be extended with additional techniques such as zero-copy transfers of tensors and optimized reconstruction that takes into account the ordering needed for the forward pass. Using this approach, the sublearning workflow can be optimized to effectively use cloning as a "fork" primitive that branches off in alternative directions with minimal overhead.

Such considerations have inspired new data models that address the data management requirements of deep learning. In this context, we are exploring the notion of *data states* [6], which are intermediate snapshots of datasets (e.g., DNN models) that can be either captured or cloned asynchronously at scale, while adopting the FAIR principles [7] (findable, accessible, interoperable, reusable) in form of a lineage, which makes it easy to navigate through their evolution and/or search for interesting snapshots that can be reused.

2.6. Hyperparameter search

This is the process of refining the architecture of the underlying neural network in the deep learning application. Given the basic design of a neural network, there are many flexible parameters that can be tuned to produce a particular result, such as accuracy or performance. These are typically optimized using generic optimization routines. Such searches produce a great many (thousands) of parallel trial training cycles, which run for minutes to hours.

The simplest, though most costly, methods for hyperparameter optimization include exhaustive space search (the brute force method), simple gradient descent, multiple gradient descent, and random search. Though these search algorithms can be tuned to execute quickly (random search) or to find the optimal solution (exhaustive search), the marginal optimization with respect to the utilized resources is not efficient for problems with 109 or greater reasonable discrete parameter combinations. There are two primary drawbacks to utilizing an a priori user-specified set of discrete hyperparameters for reducing loss: 1) it requires the user to make assumptions concerning topological efficiencies and efficacies and 2) it is limited to a small, finite set of models (i.e., it is forcing a complex algorithm into constrained bounds). By including effective reductions possible using gradient descent, we may gain one or two orders of magnitude of search space, however, this is still well below the 1021 complexity that is possible in the current CANDLE workflows.

2.7. Uncertainty quantification (UQ)

This is the analysis of the sensitivity of the deep learning method to small changes in the training data or other quantities in the neural network. UQ provides estimates on how robust the overall method is in the presence of data errors and other variability. UQ studies also produce a great many (thousands) trial training and inferencing runs , which are then analysed using statistical techniques.

2.8. Training data analysis

This is the analysis of the given training data sets for their applicability to target application problems, generalizability across problems (transfer learning), or unexpected

negative impacts. Training data analysis is performed by training neural networks on various subsets of the whole training data corpus, then performing cross-comparisons of the neural networks produced, or applying transfer learning to so far unused parts of the training data corpus or other problems entirely. Since training data can be broken up and reordered in a great many ways, this methodology also produces a great many (thousands) of training cycles.

2.9. Supporting infrastructure

Under the top-level workflows, several supporting components are provided. The Workflow Orchestration component is based on the Swift/T workflow system [8], [9], a previously developed MPI-based dataflow programming language and runtime. The Data Management component includes capabilities for data input and output, data caching [10], and metadata storage in databases. The Deep Learning Applications are somewhat independent of CANDLE as they are developed by focused applications teams, but they are built around common abstractions and APIs designed for integration into CANDLE workflows. These include options for handling hyperparameters formatted in a common way. As most of our applications are based on Keras/Tensorflow [11], CANDLE provides wrappers and simplifications for managing neural network weights data, input and output tools, features to manage restartability and callbacks (a Keras feature to execute Python code dynamically during neural network training), and so on. CANDLE also includes an analysis library for evaluating UQ, as well as support for codes written in PyTorch.

To execute a training run, the workflow produces a list of parameter tuples that are encoded as arguments to a Python-based wrapper script. These wrapper scripts are the interfaces to the various CANDLE Benchmark applications or external user applications. The parameters are encoded in JavaScript Object Notation (JSON) format which can be easily converted by the Python wrapper script into a Python dictionary, from which a CANDLE Benchmark application can retrieve the parameter values. These scripts are run concurrently across the available nodes of the Swift/T allocation, typically one per node. Thus, the Benchmark has access to all the resources on the node. The Benchmarks are Python programs that implement the application-level logic of the cancer problem in question. The are coded (using the CANDLE common library protocol) to enable the hyperparameters to be inferred from a suitable default model file, or to be overwritten from the command line. It is this construction that allows the parameter tuples to be easily ingested by the respective Benchmarks, and use a standardized interface developed as part of the project.

The result of a wrapper execution is a performance measure on the parameter tuple p, typically the validation loss. Other metrics could be used, including training time or some combination thereof. These are fed back to the workflow to produce additional parameters to sample. The results are also written to a Metadata Store, which contains information about the wrapper execution. The Metadata Store records which parameter tuples have already been run, enabling efficient workflow restarts. Thus, a progress history is available for each learning trial run, as well as for the overall optimization workflow.

At execution time, workflow configuration, control, and model runs are clearly separated as shown in Figure 5. In this figure, user-defined components are shown in blue, and static Supervisor components are shown in white. The workflow starts on the login node with a user provided test script that sets up the run. This script simply defines the data to be used and the other configurations. The system configuration (cfg sys) configures the system settings, such as the queue, and project to be used, number of compute nodes, and requested wall time. These settings are abstracted over the many schedulers supported by our system, which includes all large DOE systems and many others (LSF, SLURM, PBS, Cobalt, etc.). The parameters configuration (cfg params) configures the parameter space used to start the run, for example, these could define the hyperparameter space to be searched by a hyperparameter optimization workflow. Site settings are selected from an extensible library containing systems Summit, Theta, Cori, and others. At run time, the workflow launches inside a Workflow Shell that loads appropriate libraries for the system and configures the Swift/T system, which manages workflow progress. Swift/T launches the workflow on the compute nodes using the system scheduler tools (e.g., qsub).

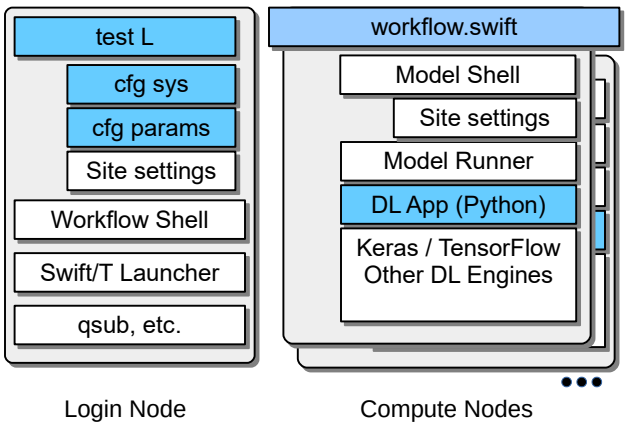

Figure 5: Distributed execution of workflow control and deep learning models in CANDLE/Supervisor.

The workflow.swift script prescribes the logic of the workflow. The Supervisor system contains the workflows described in this paper, and they can be extended for other problems, as they are only 100-200 lines each. Compute node execution starts with the Swift/T workers launched by the scheduler. For each task in the workflow, these launch a Model Shell which simply loads library configurations to run Python and the underlying deep learning engine. These receive the JSON parameter string from the workflow and pass them to a child Python process using the Model Runner, a reusable Python application that handles the input

and output parameters of the deep learning run. The Model Runner invokes the user deep learning application (DL App), which could be a CANDLE Benchmark or some other deep learning application. This then typically invokes a deep learning engine (DL Engine) that uses the compute resources on the node to perform the deep learning operation of interest (training or inferencing).

2.10. The Sublearning workflow implementation

The Sublearning workflow is implemented as a Swift/T workflow. This allowed for a high-level workflow description that constructs the data dependencies from one node to its children, as each child must wait for its parent to complete training so that training may resume on a larger data set. The Swift/T workflow is responsible for synchronizing on these dependencies and concurrently distributing work to the compute nodes of a large parallel computer. These data dependencies are expressed using the Swift/T code fragment shown in Figure 6.

```
1  /**   RUN STAGE: A recursive function that manages
2        the stage dependencies
3  */
4  (void v) run_stage(int N, int S, string node,
5                     int stage, void block)
6  {
7    //  Run the model for this workflow node
8    void parent = run_single(node, stage, block);
9
10   if (stage < S)
11   {
12     //  Recurse to the child stages
13     foreach id_child in [1:N] //  parallel loop
14     {
15       run_stage(N, S, node+"."+id_child,
16                 stage+1, parent);
17     }
18   }
19 }
20
21 /**  RUN SINGLE: Set up and run a single model
22      via candle_obj()
23 */
24 (void v) run_single(string node, int stage, void parent)
25 {
26   if (stage == 0)
27   {
28     v = propagate();
29   }
30   else
31   {
32     json_fragment = make_json_fragment(node, stage);
33     json = "{\"node\": \"%s\", %s}" %
34            (node, json_fragment);
35     wait (parent) //  waits for parent data
36     {
37       //  Run the model
38       obj_result = candle_obj(json, node);
39     }
40   }
41 }
```

Figure 6: Distributed execution of workflow control and deep learning models in CANDLE/Supervisor.

As shown, function `run_stage()` represents one node of the workflow graph. It performs a single training run for that node and recursively calls `run_stage()` for each child node. Following Swift/T semantics, these steps are concurrent; thus, a synchronization variable called parent is passed to the child nodes. The string node represents a string identifier for that node, for example, node "1.2" would have children "1.2.1", "1.2.2", "1.2.3", and "1.2.4".

Function `run_single()` runs a single NN training run, heavily relying on Supervisor utilities and libraries to run the model. It makes a JSON fragment (some string manipulation code not shown) based on the current node and stage, and invokes the `candle_obj()` function (CANDLE objective function) to run the model. Typical runs for this paper used 128 nodes of Summit, each running `candle_obj()` for the bulk of the time.

Results from each `candle_obj()` run are stored in a directory with the node name containing the inputs, output logs, stored NN weights, timing data, and statistics produced during training. These are used to produce the experimental results described in the following.

3. Experimental results

We applied the CANDLE system to the problem of identifying anomalous data in a cell line x drug pair data set for the Uno model.

All results in this paper were obtained on Summit, the IBM PowerPC and NVIDIA Volta -based supercomputer at Oak Ridge National Laboratory. Summit has 200 petaflops at double precision, across 4,608 nodes, each with two POWER CPUs and 6 V100 GPUs.

3.1. Motivating example

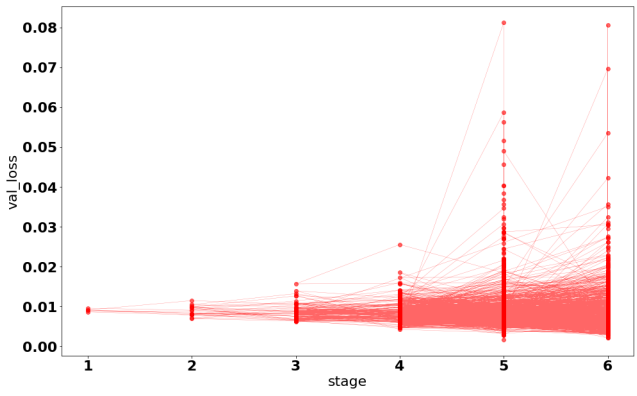

Figure 7: Demonstration of diverse loss behavior across workflow trajectories.

First, we illustrate the problem of running the Sublearning ensemble. An initial run using the workflow above produces the validation loss (`val_loss`) behavior shown in Figure 7.

As shown, a small number of runs on early data sets produce relatively good `val_loss` results 0.01. Then at each subsequent stage as more data is added, and more

Strategy	Description
F-20	Flat-20: Simply train for 20 epochs
F-10	Flat-10: Simply train for 10 epochs
LINE	Linear: Train for a linearly decreasing number of epochs, starting from 20
SQRT	Square root: Train for a number of epochs equal to 20 / sqrt(stage)
EXP2	Exponential: Train for a number of epochs equal to 20 / $2^{(\text{stage}-1)}$

TABLE 1: Epoch strategies used in the Sublearning workflow study.

trajectories are created, some trajectories improve in accuracy, some become less accurate, and some become very inaccurate. Our goal is to find the data subsets that produce the wild inaccuracies as these may indicate problems or features in the underlying data.

We ran the Sublearning workflow in multiple modes, each of which was a different maximum epoch count strategy for each stage. The strategies used are shown in Table 1.

Each strategy was run using the Keras EarlyStopping feature to stop training if the validation loss does not increase over some consecutive number of epochs; we set this value to 3. This is an aggressive setting. This was enabled to allow the workflow to make progress rapidly and move on to the next stage if a particular stage stalled in making progress, on the assumption that errors can be corrected at the next stage while training with additional data.

3.2. Early stopping results

To measure the impact of our early stopping strategy, we recorded how many epochs were actually run. For early stopping results, we first report the number of training runs within each stage that stopped early. This is shown in Figure 8.

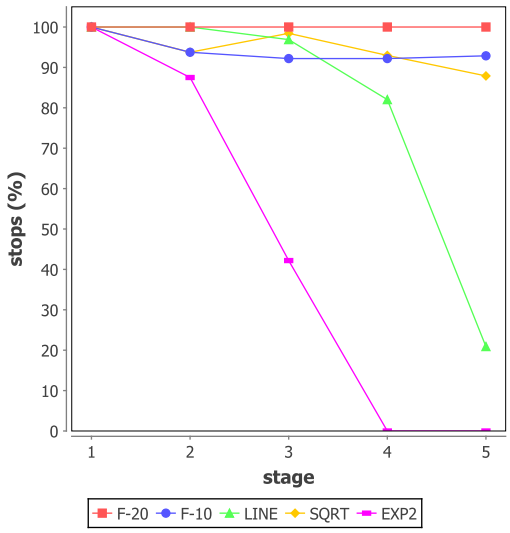

Figure 8: Fraction of early stops by stage.

As shown, all 100% stage 1 training cycles stop early-no training run makes it to the full 20 or 10 epochs. As the stages progress, however, some runs do complete (run to the maximum epoch). In the LINE and EXP2 modes, many or all runs complete and do not stop early, as the maximum epoch is very small. This illustrates that the various epoch strategies do have a strong effect on training time. The Flat approaches generally stop early as 20 or even 10 epochs is more than enough training. The question is whether stopping early affects the overall training error in our workflow structure, which is addressed in the following measurements.

3.3. Number of epochs trained

In this experiment, we report the number of epochs trained by stage. For each run, early stopping was enabled, and we record how many epochs were actually used to do the training before early stopping or the maximum epoch was reached. The average actual epoch count was plotted for each stage in Figure 9.

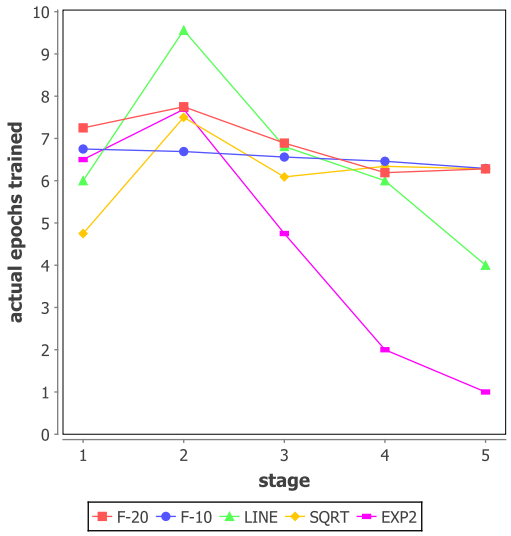

Figure 9: Actual number of epochs trained by stage.

As shown in the figure, there is a range of responses to the maximum epoch strategies. There appears to be a strong tendency for all stages to only require around 6 epochs of training. LINE and EXP2 are constrained to train for a smaller number of epochs at the later stages, so we measure the impact on `val_loss` in the following experiments.

3.4. Validation loss by stage

In this experiment, we consider the change in accuracy under the varying epoch strategies. For each strategy, for each stage, we sorted the `val_loss` results for all trajectories, and report the percentiles 99%, 75%, 50%, 25%, and 10%, where higher percentiles indicate higher accuracy (lower `val_loss`). Results are shown in Figure 10.

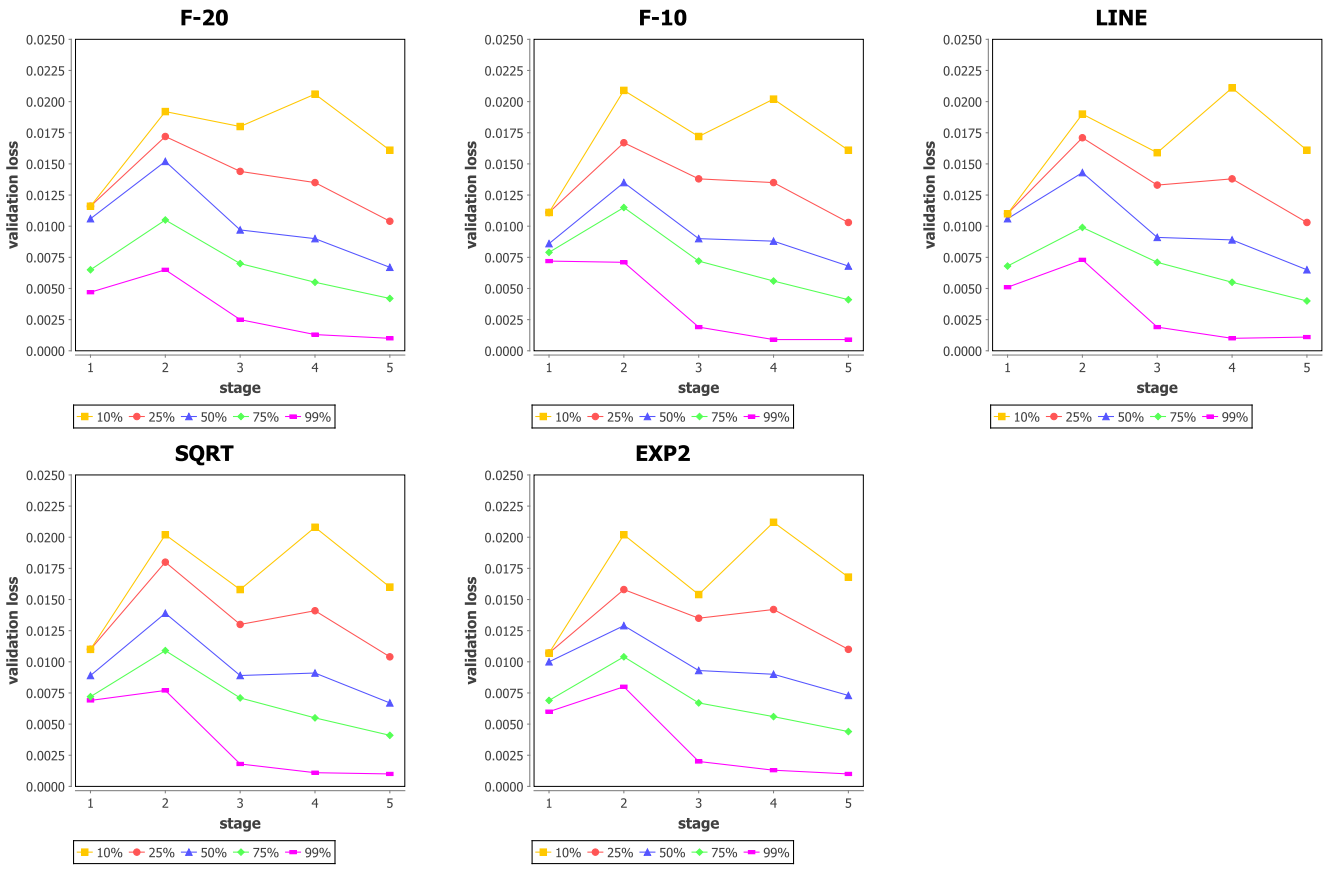

Figure 10: Validation loss results by epoch strategy and workflow stage.

As shown in the table, `val_loss` is relatively stable across the epochs strategies. Typical accuracies for each percentile level continue to increase as new data is provided from stage 2 to the final stage. This indicates that generally speaking there is additional learning to be done at each stage, but aggressively reducing the number of epochs used at later stages does not cause a significant increase in `val_loss`. This indicates that generally speaking the additional data is assimilated well without inducing errors in the NN.

3.5. Training cost to achieve best accuracy

In this experiment, we investigate the overall results of the training workflow. We selected the most accurate stage 5 runs from each workflow. We report the wall clock time in training hours used to run that trajectory (the final node and its parents, not all nodes in the workflow). We also report the cumulative training steps reported by TensorFlow; the number of steps is based on the size of the training data (which increases by stage as the Sublearning workflow increases the data set size). We also report the validation loss.

As shown in Table 2, the aggressive EXP2 epoch reduction strategy produces very good `val_loss` results, and by saving on epochs in the later stages, provides a reduction in the training time and steps needed. This justifies the use of aggressively limiting training time.

Mode	Training hours (per trajectory)	Training steps (cumulative)	Validation loss
F-20	2.21	165,340	0.000785
F-10	2.17	66,188	0.000655
LINE	2.49	66,188	0.000650
SQRT	2.65	148,923	0.000652
EXP2	1.46	16,534	0.000568

TABLE 2: Statistics on training time, steps, and resulting loss.

3.6. Finding outlier nodes

In this experiment, we apply the previously vetted workflow modes to the problem of finding data subsets that decrease the NN accuracy, measured by increasing `val_loss`. For each epoch strategy, within each workflow stage, we counted the number of nodes that increased in `val_loss` with respect to the previous stage. The results are shown in Figure 11.

As shown, nearly all stage 2 results show an increase in `val_loss`, an interesting result. This indicates that the

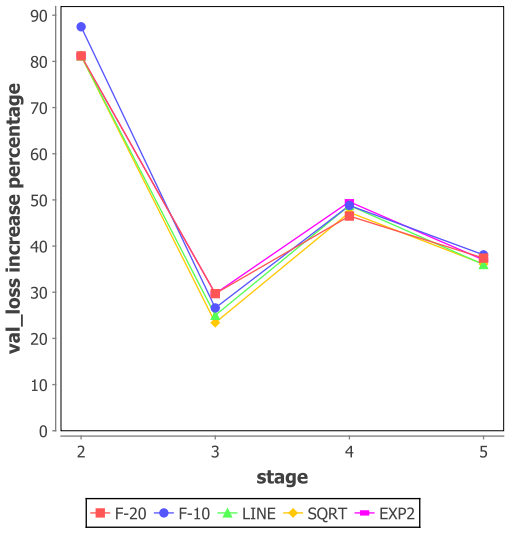

Figure 11: Fraction of subsets that causes an increase in validation loss by stage.

large addition of stage 2 data generally reduces accuracy, and that much more training is needed after this stage. By the final stage, however, the increase rate has leveled off to the range 36.0% - 38.1% for all modes.

3.7. Measuring the level of inaccuracy

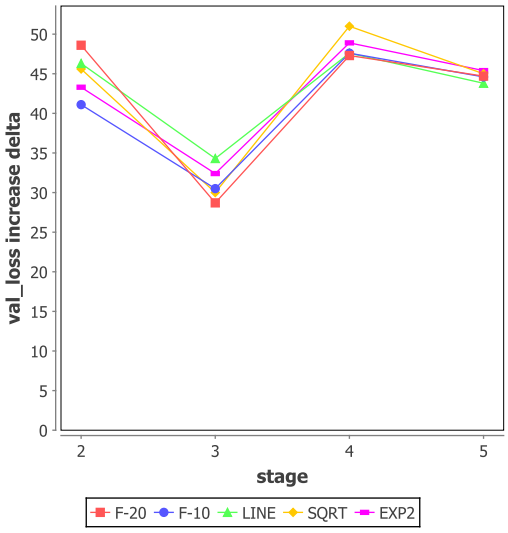

Figure 12: "Increase delta" for validation loss by stage.

In this experiment, we desire to measure the amount of inaccuracy (not just the count). For the workflow nodes that increased in `val_loss`, we report the average amount of increase with respect to the average `val_loss` for that workflow stage, here called the "increase delta". Results are shown in Figure 12.

As shown, the average increase is under 50%, indicating that `val_loss` increases are generally modest when they occur. Of particular interest is the F-20 epoch strategy, which produces the highest increase delta at stage 2, but the lowest at stage 3. Thus it appears that running for more epochs allows the NNs to recover accuracy after the large data addition at stage 2.

3.8. Finding the largest error increases

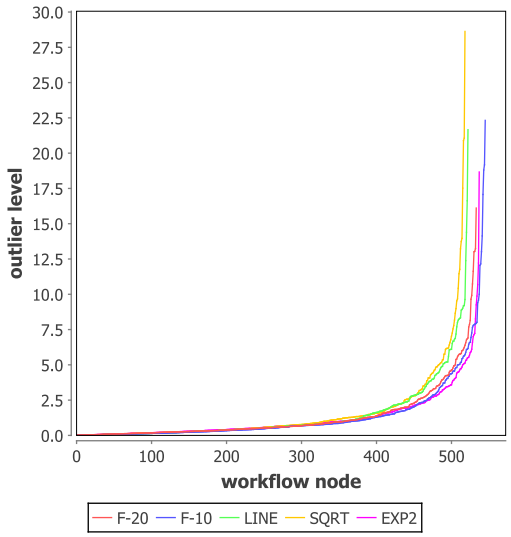

Figure 13: "Outlier level" for validation loss by stage.

In our last experiment, we desire to use the results above to find the workflow nodes with the greatest `val_loss` increases (reductions in accuracy). For each workflow node that increases in `val_loss`, we sorted that node with respect to that workflow mode. The node represents the value of its `val_loss` increase divided by the `val_loss` for the parent of that node, here called the "outlier level". We then plotted the nodes cumulatively to demonstrate the distribution of these increases. The result is shown in Figure 13.

As shown, most runs have a relatively low outlier level. Of the 6,820 training runs across the 5 workflow modes, only 899 have an outlier level greater than 1 (13.2%) and 489 have an outlier level greater than 2 (7.2%). This greatly reduces the amount of work that needs to be done to find interesting data subsets.

In theory, one could continue this process until only a single training data record is removed from each model, thus isolating individual records for their error effect. In practice, we have run the cancer data set used here to stage 6, but the data becomes sparse at this stage and does not provide results that are generally interpretable.

3.9. Workflow performance

As a demonstration of the utilization of the Summit system under this workflow, Figure 14 shows the load produced

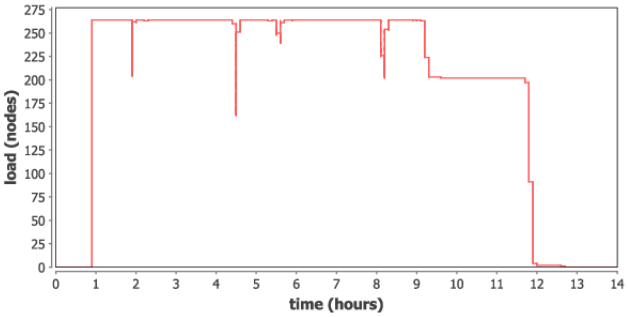

Figure 14: Compute load over time on Summit.

by a typical sublearning workflow run. This shows a 264-node run toward the end of a workflow. As shown, the workflow system rapidly fills the allocation with model training execution, although some gaps are visible as training tasks exit and are replaced. Toward the end, as the workflow runs out of work to perform, utilization drops. Swift/T workflow features could be used to detect the load drop and exit early, but these were not enabled for this run.

A typical workflow campaign involves multiple restarts as wall-clock limits are encountered. The CANDLE/Supervisor restart capabilities (§2.9) are used to restart workflows efficiently at the epoch checkpoint level, minimizing wasted GPU time.

4. Conclusion

Finding erroneous or interesting samples in large data sets is an important problem in deep learning, and is critically important in biomedicine, both to prevent inaccurate results and to accelerate discovery. In this paper, we showed how large ensembles of NN training runs can be used to collect statistics about training that can be used to gain insight into the underlying data. In this paper, we described a new workflow structure to address this problem and the implementation of the workflow in some detail. We showed how our solution fits into CANDLE/Supervisor, a framework for rapidly developing deep learning workflows for cancer and other applications, and applied the technique to a cancer benchmark, Uno, that predicts tumor dose response across a range of studies. The approach isolates a small number of cell lines that contributed the most to introducing error in the NN training and could be prioritized for further study. Additionally, our experiments demonstrated that aggressive approaches to epoch strategies produce useful results.

In future work, we plan to apply this approach to other scientific problems in which outliers or other erroneous data could impact training. We also plan to attempt more aggressive, dynamic search trajectories through smaller piecemeal subsets of the training data to accelerate the discovery of anomalous data.

5. Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *International Conference on Data Mining*, 2008.

[2] Sahand Hariri, Matias Carrasco Kind, and Robert J. Brunner. Extended isolation forest. 2018.

[3] Prasanna Balaprakash, Romain Egele, Misha Salim, Stefan Wild, Venkatram Vishwanath, Fangfang Xia, Tom Brettin, and Rick Stevens. Scalable reinforcement-learning-based neural architecture search for cancer deep learning research. In *Proc. SC*, 2019.

[4] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. DeepFreeze: Towards scalable asynchronous checkpointing of deep learning models. In *CGrid'20: 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, pages 172–181, Melbourne, Australia, 2020.

[5] Bogdan Nicolae, Justin M. Wozniak, Matthieu Dorier, and Franck Cappello. DeepClone: Lightweight state replication of deep learning models for data parallel training. In *CLUSTER'20: The 2020 IEEE International Conference on Cluster Computing*, Kobe, Japan, 2020.

[6] Bogdan Nicolae. DataStates: Towards lightweight data models for deep learning. In *SMC'20: The 2020 Smoky Mountains Computational Sciences and Engineering Conference*, Nashville, United States, 2020.

[7] Mark D Wilkinson et al. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, 3(160018), 2016.

[8] Justin M. Wozniak, Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, and Ian T. Foster. Swift/T: Scalable data flow programming for distributed-memory task-parallel applications. In *Proc. CCGrid*, 2013.

[9] Timothy G. Armstrong, Justin M. Wozniak, Michael Wilde, and Ian T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *Proc. SC*, 2014.

[10] Justin M. Wozniak, Philip E. Davis, Tong Shu, Jonathan Ozik, Nicholson Collier, Manish Parashar, Ian Foster, Thomas Brettin, and Rick Stevens. Scaling deep learning for cancer with advanced workflow storage integration. In *Proc. Machine Learning in High Performance Computing Environments (MLHPC) at SC*, 2018.

[11] Francois Chollet et al. Keras. https://github.com/fchollet/keras, 2015. https://github.com/fchollet/keras.