Profile Analysis, Scalability, and Usability of ivadomed

Kayvon Heravi
University of California, San Diego
kheravi@ucsd.edu

Sri Harsha Shatagopam
University of California, San Diego
sshatagopam@ucsd.edu

ABSTRACT

The advent of deep learning in the medical imaging space brings the following problems of applying machine learning to different domains: the need for robust frameworks that can reliably load data, transform features, train on its data, and handle any post-processing as required. Ivadomed is an end-to-end deep learning pipeline built on PyTorch created for streamlining the creation of models for medical imaging tasks such as segmentation, classification, and detection. Ivadomed can also handle data preprocessing, transformations, and training/testing through simple command line arguments. This project aims to measure the efficacy of ivadomed by performing a profile analysis on the pipeline, as well as evaluating its usability as a package. This is demonstrated by experimenting on the framework with different tasks, models, configurations, and data set sizes in order to both qualitatively document usability and quantitatively measure systems and model-based metrics and performance. According to our qualitative analysis, ivadomed offers a robust package for deep learning professionals with relatively few bugs, though a lack of up-to-date documentation can result in frustration on a user's end for trying to implement currently-depreciated features. Moreover, experiments show that it is not clear-cut where the bottlenecks in ivadomed's pipeline are; it wholly depends on what kind of data set, architecture, and imaging task a user decides to use. In general, training can dominate runtime except when additional data preprocessing is needed, like in the case of 3D segmentation' this can also happen when additional postprocessing is needed, like with binary threshold analysis.

KEYWORDS

Medical imaging, MRI, Ivadomed, Deep learning, CNNs, FiLM

1 INTRODUCTION

Deep learning has been shown to be an effective tool in a variety of domains; in terms of medical imaging, three main tasks in particular perform very well with deep neural networks: segmentation, classification, and detection. Specifically, convolutional neural networks (CNNs) have excelled in these types of computer vision tasks which have enabled researchers to build machine learning systems around such architectures to integrate successive components of an ML pipeline together to deliver accurate and timely results in a well-organized fashion. However, building a pipeline around model-building itself is inadequate for the demands of both research and commercial applications. There must also be robust data loader and preprocessing units that can automatically load and transform a given data set into useful features for the model. Moreover, even after training a model, additional post-processing may be desired, whether that be evaluating a model on a test set, visualizing predictions, or estimating uncertainty. All of these components should be well-connected and automatic without the need for human-in-the-loop action.

There are many candidate machine learning frameworks for biomedical applications, but the chosen system to analyze is ivadomed, an end-to-end deep learning pipeline for medical imaging that focuses mainly on segmentation but also supports object classification and detection. As will be discussed later, ivadomed offers state-of-the-art architectures to analyze medical images, but is this all that is needed for a robust deep learning pipeline? Knowing what bottlenecks, if any, exist within ivadomed's infrastructure can be helpful to improve the project in the future. The goal of this paper is to evaluate ivadomed as a package and deep learning tool both qualitatively and quantitatively at a systems and model-based level. On the latter goal, we aim to analyze ivadomed and the bottlenecks in the pipeline when changing the data set size, model architecture, and task, among other pipeline configuration settings.

In the outline of this paper, first an extensive review of ivadomed and what exactly it offers as a deep learning pipeline is provided, as well as the necessary related literature to understand the domain-specific architectures and features ivadomed includes (Section 2). Secondly, the evaluation of ivadomed both qualitatively and by profiling system usage, runtime, and performance at scale to discover bottlenecks in the system is laid out (Section 3). The experiments used to conduct this analysis and how they were administered are further described (Section 4). Finally, the qualitative observations made about ivadomed during our deep-dive into the pipeline are described (Section 5), as well as the quantitative evaluation of the performance of ivadomed under different parameters (Section 6). This paper does not wholly cover everything that can be analyzed in ivadomed, so we list further work that can be done with this package to perhaps gain greater insight into the efficacy of ivadomed as a deep learning tool (Section 7). The appendix showcases the figures generated from each experiment to create a neater layout for the paper.

2 BACKGROUND

Because the applications of ivadomed are very domain-specific, a brief overview of the related technical concepts in medical imaging analysis and its implementation using deep learning techniques is required to understand the design decisions behind some of ivadomed's more technically advanced features. We can first start by reviewing what functionality ivadomed as a package offers (though only features that are covered in our experimental analysis will be mentioned; additional features would need be analyzed in future works). Next, a more extensive review of the techniques used and their justification in ivadomed can be done.

2.1 Bottlenecks in ML Pipelines

Deep neural networks(DNN) have become popular in recent years as corporations have delved into big data analytics. DNNs require a lot of data to be trained which is great for querying data, but runtime remains an active area of research [10]. Kang et al. has found that

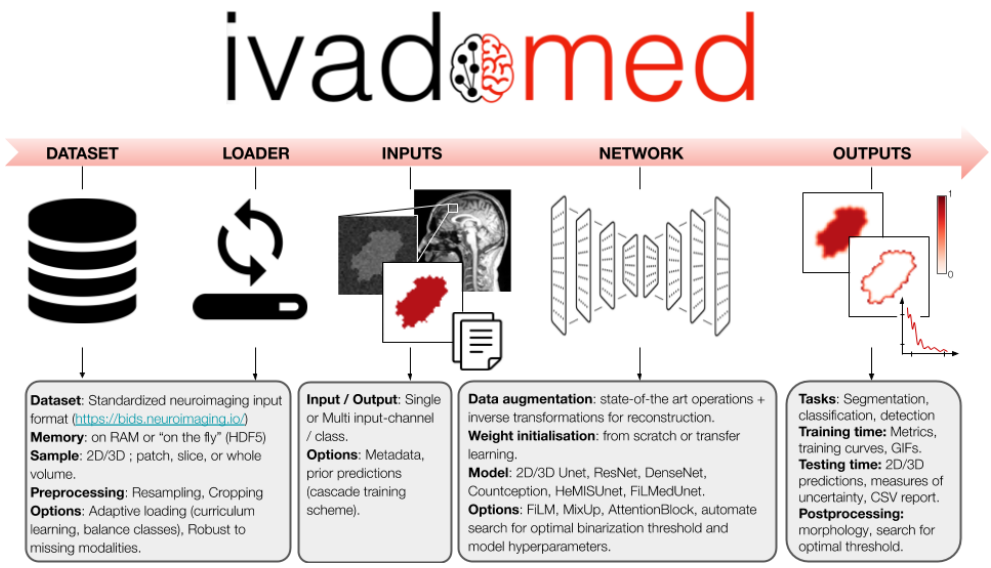

Figure 1: An overview of ivadomed's pipeline components. In our analysis, we simplify this into three sections: data loader and preprocesser, training, and post-processing modules.

the preprocessing of data will be the bottleneck in most visual analytic systems on modern hardware [10]; this is wholly relevant to ivadomed since it is a visual deep learning analysis tool. Anderson et al. also elaborate on the issue of querying through large amounts of data such as images and videos [1], in which they find a similar bottleneck in deep CNNs which can classify objects with high accuracy, but at a slow rate [1]. Ivadomed is a deep learning pipeline that deals with large data and expensive data such as images. As previously dictated, literature hypothesizes that ivadomed would find bottlenecks in runtime at the data preprocessing stage.

2.2 Medical Imaging

Medical resonance imaging (MRI) is a common technique in the medical imaging landscape for diagnostic tests. Essentially, an MRI test exploits the atomic properties of nuclei in hydrogen atoms (a single proton) in the presence of a strong magnetic field by disturbing the alignment of such protons through the periodic sending of radio wave bursts. A signal is emitted as these waves cease and the hydrogen atoms "relax" into their prior alignments, which is measured and transformed from frequencies into voxel (a volumetric pixel in a 3D image) values [15].

Sequences of radio bursts and relaxation periods correspond to the creation of an MRI; using different sequences by varying radio bursts create different MRI images. Common examples are T1 and T2 relaxation times, which can be used as weights to form different MRI images for different diagnostic purposes, such as the analysis of different tissues [15]. The data set used in this paper includes T1, T2, and T2* contrasts for each MRI scan sample.

To automate critical tasks like tumor segmentation or classification of objects in an MRI scan, deep learning has been a technique of interest in the medical imaging landscape. There are problems that come with the use of CNNs for these tasks, though: data set availability and homogeneity is sparse, meaning there is no easy solution for mitigating these problems when training a network (namely, to avoid overfitting to one specific contrast or cluster of subjects) [4]. A robust solution is needed both to process heterogeneous and general data as well as counteracting the absence of desired image characteristics within the data during training.

2.3 Ivadomed

As mentioned previously, ivadomed provides an end-to-end pipeline that, using an input data set and its transformed features, trains specific models and manages any post-processing needed on-the-fly. A simplified view of the pipeline can be seen in Figure 1 [9]. Ivadomed combines all user-set configuration settings for the data set, preprocessing, model hyperparameters, and training parameters through a user-created configuration .json file. From there, a simple command line interface is used to indicate the configuration file, mode of use (training or testing), as well as any post-processing operations. In short, the only component of the pipeline the user has direct control over is the data set, which must be formatted in a specific way for ivadomed to properly process data for its models.

It should be mentioned that in the experiments ran to evaluate ivadomed, the semantic view of the pipeline is simplified into three main modules: the data unit (which loads in data and transforms it into features for the model with data augmentation), the training unit (which trains a predefined architecture on the transformed data), and the post-processing unit (which does any desired post-processing as specified).

2.3.1 BIDS Standard. Ivadomed requires all data sets used in model creation to be formatted according to the Brain Imaging Data Structure (BIDS) standard. In order to standardize the organization of medical images and metadata, the BIDS format was developed. A BIDS data set is organized with a folder for each patient/subject and a .tsv file containing patient metadata in a spreadsheet format.

Within each folder contains the compressed MRI image for multiple contrasts, as well as the metadata associated with each image. A derivatives folder is also used to store the labels for each sample and is formated similarly. Ivadomed states it has support to create a data set from an HDF5 file, though as will be seen in later sections, there is some doubt to this claim. Therefore, in all practical cases, users must create a BIDS-sufficient data set to use ivadomed.

2.3.2 Configuration File. At the crux of ivadomed's platform is its configuration file: a .json file that allows the user to set the exact settings they wish for each the components of the pipeline. There are general system parameters (what command to run, what GPU/CPU to train on, model and data set path, etc.), data loader parameters (data set path, metadata specifications, etc.), data augmentation and feature extraction (data set split, applied transformations, etc.), model parameters (architecture of choice), and post-processing parameters (image operations, uncertainty calculations, evaluation and prediction and visualizations).

Given all this, a user can easily streamline model creation and analysis through the pipeline with a single command linking to a single configuration file. In addition, ivadomed has an extremely simple command line interface that can be simplified to just one argument, as shown below. Additional arguments can be added for further post-processing (such as creating a gif of the training process or performing binary threshold analysis after training).

```
ivadomed -c path/to/config_file.json
```

2.3.3 Post-processing Options. Ivadomed offers some explicit post-processing that can be done after training. One obvious and important example is being able to test a trained model on a test data set. Ivadomed also creates specific command line arguments for training gif creation and threshold analysis. A gif of the trained model applied to a validation subset can be created to show how the model learns every epoch on a particular image slice. In the threshold analysis, a user-specified increment value is used to apply binary thresholding to greyscale 2D images in the validation set. The trained model is applied to these binarized images to uncover the threshold value that results in the best dice loss (which is explained in further detail below).

2.4 Dice Loss

In medical applications, many loss functions often converge to local minima because objects of interest are small against a relatively massive image background. Convolutional networks using standard loss functions therefore fail to completely segment objects without augmenting the images themselves. The dice loss (1) has been shown to alleviate this problem in segmentation tasks, especially in medical imaging applications. Here, p and g represent segmentation pixels/voxels for the prediction and ground truth [12]. The range of the dice loss is in [0, 1] and is to be maximized; in ivadomed, however, the dice loss is actually negated and minimized in a range of [-1, 0].

$$D(p,g) = \frac{2\sum_i^N p_i g_i}{\sum_i^N {p_i}^2 + \sum_i^N {g_i}^2} \tag{1}$$

Table 1: Model architectures and tasks offered by ivadomed.

Task	Architecture
Classification	ResNet DenseNet
Keypoint Detection	Countception
Segmentation	UNet FiLMedUNet HeMISUNet 3DModifiedUNet

2.5 Architectures

Ivadomed offers multiple state-of-the-art architectures for training, though they are limited by task, as shown in Table 1. For classification, two specific models are available: an 18-layer ResNet and 121-layer DenseNet model. It is not clear why ivadomed's creators chose to support these specific variants of each architecture, but they are both taken from the standard implementations offered by PyTorch. A Countception model is included for keypoint detection, but that task is not considered in our analysis, so this architecture was not investigated further. The UNet class of models is used exclusively for the segmentation task, and requires more explanation as the different variants are more technically involved.

2.5.1 FiLMedUNet. Convolutional architectures struggle to fully adapt to all types of MRI contrasts, in which the use of different MRI sequence parameters can result in distinct differences for images of the same scan (Figure 2 [6]. Feature-wise Linear Modulation (FiLM) layers condition a network to learn from relevant metadata in order to help the network generalize to these types of differences. This conditioning is a simple affine transformation on the previous layer's features as shown in (2), where x is the feature vector of the previous layer and z is the metadata information input. γ and β vectors are generated and optimized by the FiLM generator, a multi-layer perceptron network [14].

$$FiLM(x) = \gamma(z) \odot x + \beta(z) \tag{2}$$

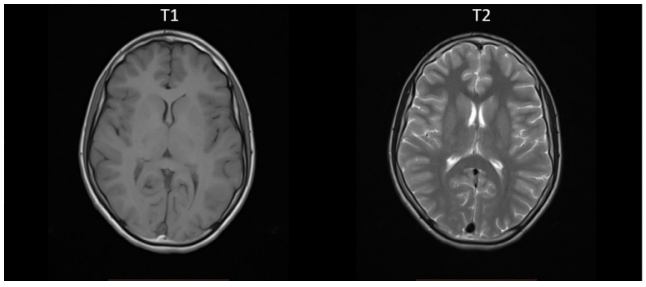

Figure 2: An example of the visual differences between MRI scans with different contrasts (T1 and T2).

In FiLMedUNet, the metadata used (z) is MRI contrast metadata, which includes contrasts such as T1, T2, and T2* among others.

In theory, the inclusion of FiLM layers should help a CNN to differentiate between contrasts to improve accuracy in predictions, though this is not guaranteed. Past experiments have shown that normal UNet implementation fare just as well in maximizing dice loss compared to a UNet architecture modified with FiLM layers [17].

2.5.2 HeMISUNet. It is often unreasonable to expect data sets of MRI images to include all types of contrasts for all samples; some may be missing or be entirely different between patients. Hence, utilizing strategies that can train models to generalize despite these missing modalities is desired. Hetero-Modal Image Segmentation (HeMIS) is a technique to mitigate the detrimental effects of missing modalities by embedding each modality (whether it is present or not for a sample) in convolutional layers; these layers are combined via the calculation of layer-wide statistics (mean and variance) and then further combined and put through convolutional layers. This embedding and abstraction model manages to achieve state-of-the-art performance in the presence of missing contrasts [5]; in ivadomed, the HeMIS technique is adapted into a UNet architecture. It should be noted that HeMISUNet only accounts for T1, T2, T2S contrasts [8].

2.5.3 3DModifiedUNet. As the name implies, this architecture is inspired by UNet to instead process 3D volumes of images rather than 2D image slices. This presents a more intuitive training process, as MRI scans are already three-dimensional stacks, and in recent experiments it has been proven that this model achieves state-of-the-art performance on a tumor segmentation data set [7]. Ivadomed's modification of this architecture adds the optional inclusion of attention gates (AG) to the decoder process of the UNet [8]. AGs learn to narrow the model's focus onto relevant regions of interest in the image, leading to better model performance [13].

3 TASK

The task of this paper is to do a profile analysis of ivadomed and test the usability and scalability of ivadomed. One of the main tasks at hand is to find bottlenecks in key areas such as GPU/CPU utilization, runtime performance, loss, data preprocessing, and data loading. In the background research of this project, we found that most major bottlenecks are in the data preprocessing and data loading components of the project [10] as mentioned in Section 2.1. The goal of profile analysis of ivadomed is to find said bottlenecks and determine if these results are similar to what is presented in literature. In terms of usability, we ask if ivadomed is a good tool for deep learning professionals and if the use of the package is streamlined such that ouputs are automatically orgnaized, easy to find, and semantically clear. Hopefully, the results of this paper can be used to help ivadomed improve the efficacy of their pipeline.

4 APPROACH

The path taken to investigate the questions laid out in the previous section indicated two different analyses needs to be taken: a series of qualitative observations of how usable and convenient of a tool ivadomed is to deep learning professionals in terms of medical imaging analysis, and a more concrete evaluation of ivadomed in terms of system usage and model performance across different configurations, tasks, architectures, and data set sizes.

4.1 Experimental Plan

To chronicle ivadomed's performance and discover possible bottlenecks in the system, we ran a profile analysis to collect system and model-specific statistics. In general, we were interested in seeing how GPU/CPU utilization and GPU/CPU memory usage changes for different models and data sets, as well as seeing how these choices affect the runtimes for each pipeline component. The statistics collected fall into two categories:

(1) System-wide statistics for each pipeline component in the system (data loader and preprocessor, training, and post-processing units).
   (a) GPU utilization
   (b) CPU utilization
   (c) GPU memory utilization
   (d) CPU memory utilization
   (e) Time per pipeline component
(2) Fine-grained statistics within the training and validation loops of the training unit per mini-batch. Previous statistics in **(1)(a)-(d)** are included as well as the following listed below.
   (a) Time to train/validate per mini-batch
   (b) Loss per mini-batch

4.1.1 Experiments. We decided to evaluate different features, architectures, tasks, and hyperparameters based on what ivadomed advertised in their original paper and in their webpage. Experiments were repeated three times and their results averaged. Experiments were also evaluated using ivadomed version 2.8.0. This is by no means an exhaustive analysis of ivadomed as a system, and as described in Section 6, more work can be done to possibly discover additional bottlenecks. Nevertheless, the experiments we did carry out are listed below.

(1) Comparison of Segmentation Models
   Here we are evaluating how different segmentation archtiectures offered by ivadomed work at the systems level by evaluating GPU/CPU utilization as well as runtime for each model. We can also compare how these models perform on a given data set by analyzing fine-grained training/validation statistics per mini-batch. The architectures test are **UNet, FiLMedUNet, and 3DModifiedUNet**.
(2) Comparison of Classification Models
   Similar to the above experiment, we use the same test for available classification models: **ResNet18 and DenseNet121** (out of brevity we will simply refer to these as ResNet and DenseNet from now on).
(3) System Performance Under Scaled Data
   We wanted to evaluate the scalability of ivadomed by using the previous architectures on data sets at different scales. We specifically test three multipliers of our original data set: 0.5, 1.0, and 10.0. We test three differeent architectures at these scales: **UNet, 3DModifiedUNet, and ResNet**.
(4) Comparison of Post-processing Inputs in Binary Threshold Analysis

In order to compare the impact of post-processing operations on the pipeline, we decided to test two increment values for the threshold analysis mentioned in Section 2.3.3: **0.1 and 0.01**; ordinary UNet models with the same hyperparameters were used in both tests.

(5) Comparison of Transfer Learned Models
Naive assumptions may posit that the use of transfer learned models may lessen the computation and resource load on the system; we use this experiment to see if that is true. An ordinary UNet model is used as a baseline, and we use transfer learning to retrain the baseline based on different fractions of layers to be trained (i.e. layers that are not frozen): **0.25, 0.50, and 0.75**.

(6) Comparison of Batch Sizes
We wanted to evaluate the effect model hyperparameters have on the pipeline; though this can be expanded further for different hyperparameters, a good starting point would be to vary the batch size on an ordinary UNet model. The batch sizes tested are **8, 16, and 64**.

To actually implement these experiments and generate relevant statistics and plots, we created a series of scripts that calls ivadomed on the current experiment configuration in a new sub-process then collects the statistics mentioned in Section 4.1 in .csv files for the entire system, training loop, and validation loop. Experiments are repeated three times and the scores in the logs are averaged. Once statistics are collected, a series of plots for both system-wide and training-specific metrics are created and saved along with the log files to a folder. This method ensures less human error and an easy testing platform for finding bottlenecks.

4.2 Data Set

We used a BIDS-formatted example data set offered by ivadomed, data_example_spinegeneric, and a subset of the larger Spine Generic Public Database. This data set includes spinal cord MRI volumes for ten patients with T1, T2, and T2* contrasts and metadata available for each subject. Data augmentation is used to create more samples during training, including resampling, center crop, random affine transformations, and elastic transformations. Across all experiments a train/validation/test data set split of 60/20/20 was used. In order to scale the data set as needed for experiment (3), a script was made to automatically copy an existing subject folder and change the names of the folder and files within to "trick" ivadomed into thinking the copy-pasted folders corresponded to new samples; this process was repeated for the derivatives folder to similarly copy-paste the labels. This was done only to create the 10.0 multiplier data set (100 subjects). To create the 0.5 multiplier data set (five subjects), half of the samples were simply deleted from the original data set of ten subjects.

4.3 Model Hyperparameters

For all experiments, the model hyperparameters usually stayed the same. All experiments ran for 5 epochs with a batch size of 18 (the default value in the example configuration file), except for experiment (6) which of course varies the batch size. 3DModifiedUNet models run with a batch size of 1 in order to get the same number of mini-batches as the UNet and FiLMedUNet models during training and plotting. For classification models, a CyclicLR learning rate (LR) scheduler was used, whereas for segmentation models, a CosineAnnealingLR scheduler was instead used, both with an initial learning rate of 0.001. FiLMedUNet has one FiLM layer applied to the UNet model. For classification models, the log loss was used, whereas for segmentation models, the dice loss was used.

4.4 Experimental Setups

We tested all experiments on two types of machines: a single machine (SM) with an Intel i7 CPU, GTX 1070, and 16GB RAM, as well as on a server instance of UCSD's Data Science/Machine Learning Platform (DSMLP); this particular instance was equipped with eight Intel Xeon CPUs, a GTX 2080 Ti, and 16 GB RAM. We found in our experiments that there are differences between experiments when using both environments, which are discussed further in Section 6.9. It is important to note that because the GPU on the SM setup did not have enough memory to process a batch size of 64, experiment (6) was only done on the DSMLP instance. Otherwise, all other experiments were completed on both setups. For experiments (1)-(5), only the plots generated on the SM setup are used as a result of the aforementioned experimental differences between setups.

5 QUALITATIVE EVALUATION

As outlined below, we found that ivadomed is a robust tool for deep learning professionals wanting to apply convolutional models to traditional imaging tasks on MRI data sets. There are few bugs/errors that persist with ivadomed, and none affect the model to a great degree. However, one of the most important tools when using a package for the first time is having easy-to-read, comprehensive, and up-to-date documentation, which ivadomed unfortunately does not excel at.

5.1 Usability

In terms of usability, ivadomed offers a very simple command line interface to call the pipeline on a configuration file as mentioned previously in Section 2.3.2. It offers additional commands to override a configuration file (for example, using '–test' to test on an already trained model pointed to within a configuration file), as well as additional post-processing arguments for further analysis (threshold analysis to optimize dice loss) and visualization (creating a gif of the model being trained and applied to a validation image Figure 3 as well as an ROC plot for the threshold analysis). Pipeline outputs are plentiful, relevant, and neatly organized in an output folder; these include a final ONNX model, training logs, evaluation metrics, an updated configuration file, etc. There is also easy TensorBoard functionality added to visualize loss curves and learning rate after training a model.

Ivadomed has multiple pre-trained models available already, though this highlights the fact that the pipeline is limited to the architectures offered by ivadomed. For example, without changing the source code, it would not be possible to use a different version of ResNet or DenseNet for a classification task. This is not necessarily a detriment however, as compared to similar projects in ivadomed's domain, the amount of architectures and configuration options per architecture offered is actually much more comprehensive than alternatives like monai and delira.

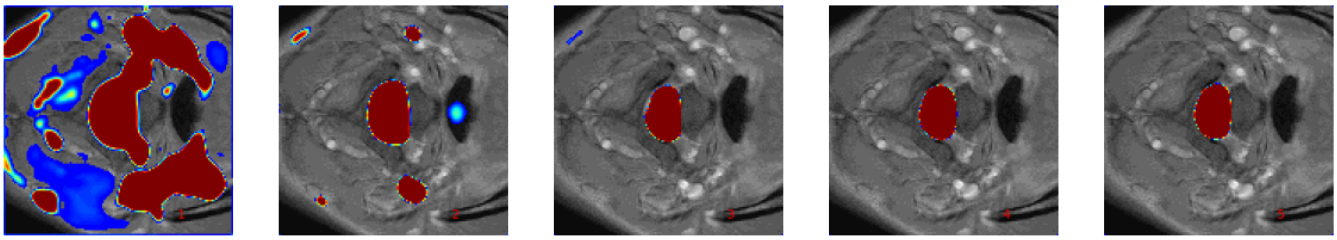

Figure 3: Frames of a training gif for a model that ran for five epochs.

Further functionality is offered in terms of a series of scripts that help automate busywork while using ivadomed, which include downloading data sets, visualizing data augmentation on an image, comparing multiple models through plot visualizations, etc. These scripts were not used much in our experimentation, so further evaluation is left for future work. In terms of the data packages offered by ivadomed (through their ivadomed_download_data script), there are only three (the rest are the aforementioned pre-trained models) that are used for tutorials and unit/functional testing. This could be seen as a detriment as on our end, it was difficult to find external BIDS data sets that would satisfy all of ivadomed's requirements.

5.2 Bugs

There were few bugs that actually caused the ivadomed module to exit out of execution, so overall the package is quite error-free. We found that during the training gif creation in `training.py`, there are some errors in actually naming the gif file, so in our experiments we just changed the code to give the file a dummy name. Another error occurred when evaluating 3DModifedUNet in which we discovered that the validation loss per epoch was always reporting `nan`; the source of the error is in `transforms.py`, in which the NormalizeInstance() function outputs `nan` when the standard deviation of a sample is zero. As a quick fix, we simply added a small positive value to the standard deviation when it was zero to avoid the error.

In terms of bugs in implementing features from ivadomed, we were not able to include HeMISUNet (and by extension, the testing of ivadomed's support for the HDF5 file system format); there were multiple errors in trying to use the architecture in the pipeline, and we eventually found out that both HeMISUNet and HDF5 support were currently being depreciated from ivadomed, a fact we were only able to deduce from the project's GitHub pull requests page.

5.3 Documentation

We found ivadomed's documentation to be very well-written in an organized fashion as sections for installation, usage, and tutorials are easy to find and read. Moreover, there are multiple sections that go into great technical detail not only for the features ivadomed offers, but also the more technical aspects of their platform, such as the configuration file, type of data that can be used with ivadomed, and others. In this regard, ivadomed has very good documentation; however, some sections still remain out-of-date (namely that their home page advertises ivadomed's inclusion of HeMISUNet and HDF5 support) which led to much frustration on our end when trying to implement already deprecated features into our experiments.

6 EXPERIMENTAL RESULTS

For each experiment, a series of four plots are produced: one that showcases system-wide statistics per-architecture for each pipeline component, one that shows per-component runtime for each architecture, and a final two plots that shows per-mini batch statistics for both the training and validation loops within the training unit. Because not all experimental results were relevant or wholly interesting, we decided to show the most pertinent results for each experiment. Given the data shown, we deduce any potential bottlenecks or other relevant analysis related to the experiment. Note that the figures mentioned below are present in the appendix.

6.1 Comparison of Segmentation Models

The UNet variants tested can be compared in a variety of ways: note that from Figure 5, FiLMedUNet achieves the highest GPU and CPU utilization for the training unit, and the 3DModifiedUNet takes the lowest utilization. This is expected as the former architecture literally has additional FiLM layers appended to the network. Moreover, note that because 3DModifiedUNet iterates of 3D volumes instead of 2D image slices like for UNet and FiLMedUNet, ivadomed compiles the given 2D image samples within its train and validation sets and combines them to form three-dimensional stacks of images. Because of this, the new samples that 3DModifiedUNet iterates over is therefore smaller compared to the other two architectures. Because there are less samples (and therefore mini-batches), utilization is lowered.

Figure 6 also validates these results by recording the fine-grained GPU and CPU utilization statistics per mini-batch during the training loop. It can be seen that once again, FiLMedUNet has the highest utilization and 3DModifiedUNet the lowest. Note that there are five major dips consistent with each architecture; this corresponds to the number of epochs each network trained for. Because utilization effectively "restarts" after every epoch, ivadomed's models are utilizing the CPU and GPU well as there is no real delay after a new epoch to nearly fully utilize the GPU/CPU.

The runtimes per pipeline component can also be analyzed: we can see that from Figure 7 that although total runtime per architecture is roughly equal, there are key differences in the amount of time each pipeline unit takes up. Overall, post-processing takes up very little time, which makes sense, as there are barely any computations not related to file I/O and logging. Between UNet and

FiLMedUNet, it can be seen that the data unit takes nearly the same amount of time, but training time is somewhat less for the latter architecture. This may seem contradictory, as FiLMedUNet is a technically more complex network, but the conditioning applied to the network via FiLM layers could potentially lessen training time. Significant differences can be found when comparing to 3DModifiedUNet, which doubles its data unit time yet lessens its training time. Again, this is consistent with combining samples into 3D volumes which takes significantly more preprocessing, and with training over less samples/mini-batches overall which combined, lead to a smaller training time.

To validate the experiments ran, it is also worth looking at how the loss changes per mini-batch, as this will give some insight into whether these models actually learn anything. Figure 8 shows that these models do actually learn in under five epochs, which is promising under our profile analysis as our results may be generalizable. Moreover, 3DModifiedUNet is much better than its 2D counterparts; this can be attributed to having a lower batch size as well as being able to learn more from rich 3D volume structures compared to 2D images. Consistent with previous results in [17], FiLMedUNet performs no better than regular UNet when optimizing Dice loss.

6.2 Comparison of Classification Models

In comparing ResNet and DenseNet models for the classification task in Figure 9, it can be seen that though they have similar GPU memory utilization for data and post-processing operations, there is a significant difference while training in that DenseNet requires higher GPU memory utilization on the same task. This is consistent with Zhang et al.'s explanation in [18] that the concatenations between layers that is inherent in DenseNet's architecture are reliant on dense GPU memory use to carry out such operations. Compared to ResNet, this effect can be visualized. Moreover, in Figure 10, the effects of using more GPU memory for those concatenations can be seen to increase training time by a significant portion. It is not clear why DenseNet's post-processing time is much higher than ResNet's, as no additional post-processing is done for either model.

It may be interesting to note that despite the higher utilization DenseNet requires, it does not result in a more generalizeable network in all cases. As seen in Figure 11, though ResNet is able to converge in terms of the training and validation loss within five epochs, DenseNet struggles to compare; the validation loss does not even converge to begin with. This stark difference between performance may not indicate anything of signficance; after all, experiments were only run for five epochs. It is possible DenseNet can achieve better performance under different hyperparameter configurations. Most likely, this result implies that the data set used is simply better utilized by ResNet.

6.3 System Performance Under Scaled Data

When comparing how different architectures perform and utilize resources under scaled data, it is most pertinent to discuss runtimes, as in our other collected data for this experiment, results were as expected. Visualizing bottlenecks is most clearly shown when analyzing runtimes per architecture per data set. For example, Figure 12 shows that for normal 2D UNet, the training unit is clearly the bottleneck as data scales up. Though the data unit sees increases in time for each scale factor, it is not as significant compared to the training time increases. Postprocessing time stays relatively constant throughout scale factors, however.

3DModifiedUnet, however, showcases quite different results. Figure 13 shows that the bottleneck in this case is actually the data unit and not the training process. Based on what has been seen in the segmentation experiment, these results intuitively make sense. As the data set scales up, more collections of 2D image samples need to concatenated into 3D volumes, which can eat up data preprocessing time. Moreover, because this stacking process will result in fewer samples overall, training time will be relatively lessened.

In some cases, however, it is not immediately clear what the bottleneck in the pipeline actually is. When evaluating the ResNet model in Figure 14 at different scale factors, both the training and data unit times increase in similar proportion. Without more data (by scaling the data set even further to 100 subjects, perhaps), the bottleneck in runtime cannot be properly discovered. It is possible to predict that the data unit is likely to be the bottleneck by looking at the trend of increases, but there are simply too few data points to build a constructive analysis when testing with ResNet.

6.4 Comparison of Post-processing Inputs in Binary Threshold Analysis

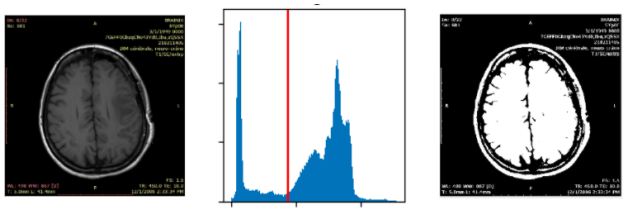

Figure 4: An example of how ivadomed applies binary thresholding to a greyscale image.

Binary thresholding is a method of segmenting images into two choices (Figure 4). As mentioned in Section 2.3.3, ivadomed takes a user-specified increment value and applies binary thresholding to a greyscale 2D image from the validation data set. The trained model will be applied to binarized medical images to acquire a threshold value that results in the best dice loss. Dice loss is is widely used in medical image segmentation tasks to address the data imbalance problem [19]. The data preprocessing, training, and post processing all take the same amount of time and the overall runtime is impacted by the increment size as shown by Figure 15. In comparison of the two graphs, a ten times increase of increment size led to approximately eight times increase in runtime. The accuracy will be better with small increments but in this case, an increment step of 0.1 is close to the minimum needed to have what could be similar loss. Ivadomed will do an exhaustive search for the threshold, thus leading to a possible bottleneck as runtime is drastically affected by the increment size. One possible way to combat this bottleneck would be to optimize this search such as testing with search methods such as random search.

6.5 Comparison of Transfer Learned Models

Transfer learning is a machine learning method that improves itself by freezing layers and training on more relevant ones, i.e a transfer of learning. Ivadomed utilizes transfer learning for mitigating imbalance issues in class sampling. In Figure 16, we see that GPU utilization of UNet is uniform across different layers of retraining. As the layers that are retrained increase, so does the GPU utilization. In Figure 17, the runtimes are similar for the different amount of layers trained which differs from the understanding of the previous figure. The possible reasoning is that ivadomed loads the transfer learning settings separately but will still run for the same number of epochs as the configuration file specifies, thus leading to the result of similar runtimes across the different proportions of layers trained.

6.6 Comparison of Batch Sizes

In comparing batch sizes, we see expected results of GPU and CPU utilization between batch sizes in Figure 18. As batch size increases, it makes sense as to the GPU and CPU utilization would rise to accommodate for the amount of samples per batch increases. This uniformity of results extends to Figure 19 where runtime decreases as the batch size increases; this makes sense: as the batch size increases, the number of mini-batches decrease along the model's training time because the model is literally working on fewer batches (the reverse is also true). As shown, ivadomed's GPU and CPU utilization along with runtime will not be affected drastically by changing the hyperparameter of batch sizes and will run similar to how tuning batch size generally works in deep learning pipelines.

6.7 Differences Between Experimental Setups

An area of interest lies in the GPU utilization between the single machine and DSMLP in Figure 20. DSMLP's GPU utilization is around 8 times less than the SM. We hypothesize that this difference is correlated with DSMLP being virtualized. Thus GPU utilization is lower due to a "split" of possible memory as the memory is on a shared server. Whereas the SM will have a higher utilization of GPU as it is not being shared among different projects. The outcome will vary machine to machine, which highlights a potential problem in profiling these kinds of pipelines across different machines. At the very least, machines that run these profiles should be similar in architecture (i.e. not a single machine versus a distributed setting). Note that despite this difference in the scale of utilization statistics, the runtime results are actually quite similar, as shown in Figure 21. Furthermore, Figure 22 shows that the loss between models among the two setups are not very different; the SM manages to get better utilization which may have led to better loss.

7 FUTURE WORKS

This project experiments on some of the functionality that ivadomed offers. In future works, a more in depth analysis can be done on techincal functionalities not covered in this paper. For example, ivadomed can employ epistemic or aleatoric uncertainty measures at test time; profiling this feature would be helpful to indicate further bottlenecks in the post-processing unit. Furthermore, we did not use Mixup for data augmentation, and we expect the data unit to increase in time as a result if it is used. We also did not include CountCeption in our architecture analysis as we did not use keypoints detection as a task. All of these techniques can be further analyzed to gain greater insight into potential bottlenecks in the ivadomed infrastructure.

8 CONCLUSION

After analysing ivadomed, the key qualitative takeaways are that ivadomed is a robust pipeline tool for deep learning professionals. It is well written, but the documentation is out of date. The API configurations are extensive but limited by available architectures. Earlier in this paper, literature had hypothesized the bottlenecks would be found in data preprocessing [10]. From the experimental takeaways, the bottlenecks found in ivadomed are not clearly found in one area; these bottlenecks are hence dependant on the type of architecture, task, and data set a user decides to use.

REFERENCES

[1] Michael R Anderson, Michael Cafarella, German Ros, and Thomas F Wenisch. 2019. Physical representation-based predicate optimization for a visual analytics database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1466–1477.

[2] Özgün Çiçek, Ahmed Abdulkadir, Soeren S Lienkamp, Thomas Brox, and Olaf Ronneberger. 2016. 3D U-Net: learning dense volumetric segmentation from sparse annotation. In *International conference on medical image computing and computer-assisted intervention*. Springer, 424–432.

[3] Reuben Dorent, Samuel Joutard, Marc Modat, Sébastien Ourselin, and Tom Vercauteren. 2019. Hetero-modal variational encoder-decoder for joint modality completion and segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 74–82.

[4] Charley Gros, Andreanne Lemay, Olivier Vincent, Lucas Rouhier, Anthime Bucquet, Joseph Paul Cohen, and Julien Cohen-Adad. 2020. ivadomed: A Medical Imaging Deep Learning Toolbox. *arXiv preprint arXiv:2010.09984* (2020).

[5] Mohammad Havaei, Nicolas Guizard, Nicolas Chapados, and Yoshua Bengio. 2016. Hemis: Hetero-modal image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 469–477.

[6] Ryan Hird. 2021. *The Basics of MRI Interpretation.* Retrieved December 1, 2021 from https://geekymedics.com/the-basics-of-mri-interpretation/

[7] Fabian Isensee, Philipp Kickingereder, Wolfgang Wick, Martin Bendszus, and Klaus H Maier-Hein. 2017. Brain tumor segmentation and radiomics survival prediction: Contribution to the brats 2017 challenge. In *International MICCAI Brainlesion Workshop*. Springer, 287–297.

[8] Ivadomed. 2020. *Architectures - ivadomed documentation.* Retrieved November 20, 2021 from https://ivadomed.org/architectures.html

[9] Ivadomed. 2020. *Home - ivadomed documentation.* Retrieved November 20, 2021 from https://ivadomed.org/index.html

[10] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. 2020. Jointly optimizing preprocessing and inference for DNN-based visual analytics. *arXiv preprint arXiv:2007.13005* (2020).

[11] Alexander Selvikvåg Lundervold and Arvid Lundervold. 2019. An overview of deep learning in medical imaging focusing on MRI. *Zeitschrift für Medizinische Physik* 29, 2 (2019), 102–127.

[12] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. 2016. V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 fourth international conference on 3D vision (3DV)*. IEEE, 565–571.

[13] Ozan Oktay, Jo Schlemper, Loic Le Folgoc, Matthew Lee, Mattias Heinrich, Kazunari Misawa, Kensaku Mori, Steven McDonagh, Nils Y Hammerla, Bernhard Kainz, et al. 2018. Attention u-net: Learning where to look for the pancreas. *arXiv preprint arXiv:1804.03999* (2018).

[14] Ethan Perez, Florian Strub, Harm De Vries, Vincent Dumoulin, and Aaron Courville. 2018. Film: Visual reasoning with a general conditioning layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[15] David C Preston. 2016. *Magnetic Resonance Imaging (MRI) of the Brain and Spine: Basics.* Retrieved December 1, 2021 from https://case.edu/med/neurology/NR/MRI%20Basics.htm

[16] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

[17] Olivier Vincent, Charley Gros, Joseph Paul Cohen, and Julien Cohen-Adad. 2020. Automatic segmentation of spinal multiple sclerosis lesions: How to generalize across MRI contrasts? *arXiv preprint arXiv:2003.04377* (2020).

[18] Chaoning Zhang, Philipp Benz, Dawit Mureja Argaw, Seokju Lee, Junsik Kim, Francois Rameau, Jean-Charles Bazin, and In So Kweon. 2021. Resnet or densenet? introducing dense shortcuts to resnet. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 3550–3559.

[19] Rongjian Zhao, Buyue Qian, Xianli Zhang, Yang Li, Rong Wei, Yang Liu, and Yinggang Pan. 2020. Rethinking Dice Loss for Medical Image Segmentation. In *2020 IEEE International Conference on Data Mining (ICDM)*. 851–860. https://doi.org/10.1109/ICDM50108.2020.00094

A EXPERIMENTAL PLOTS

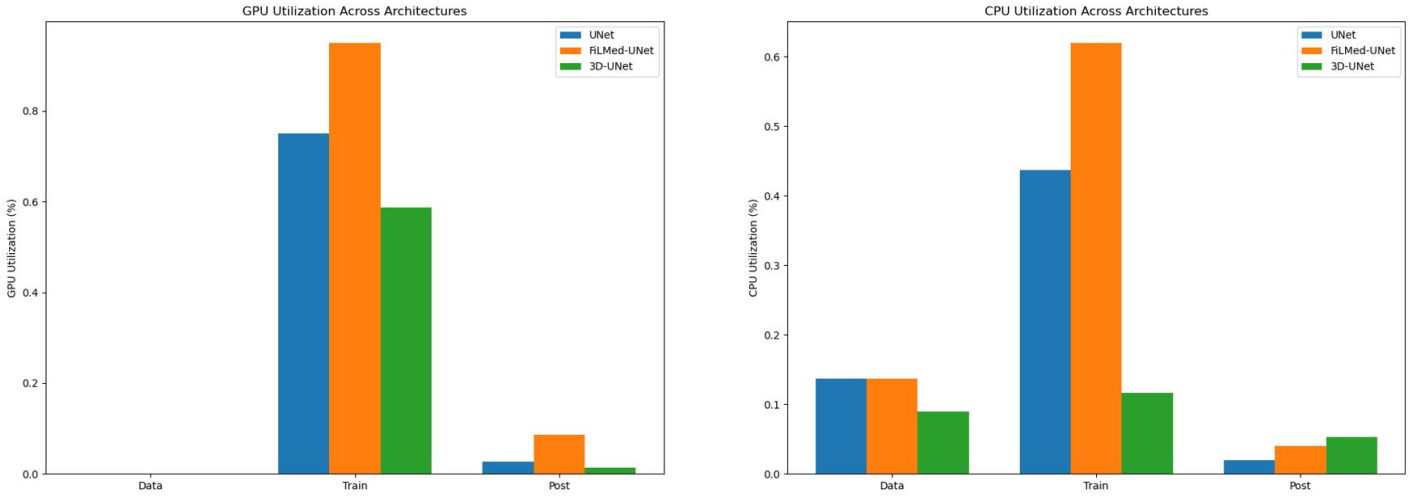

Figure 5: GPU and CPU utilization across UNet variants.

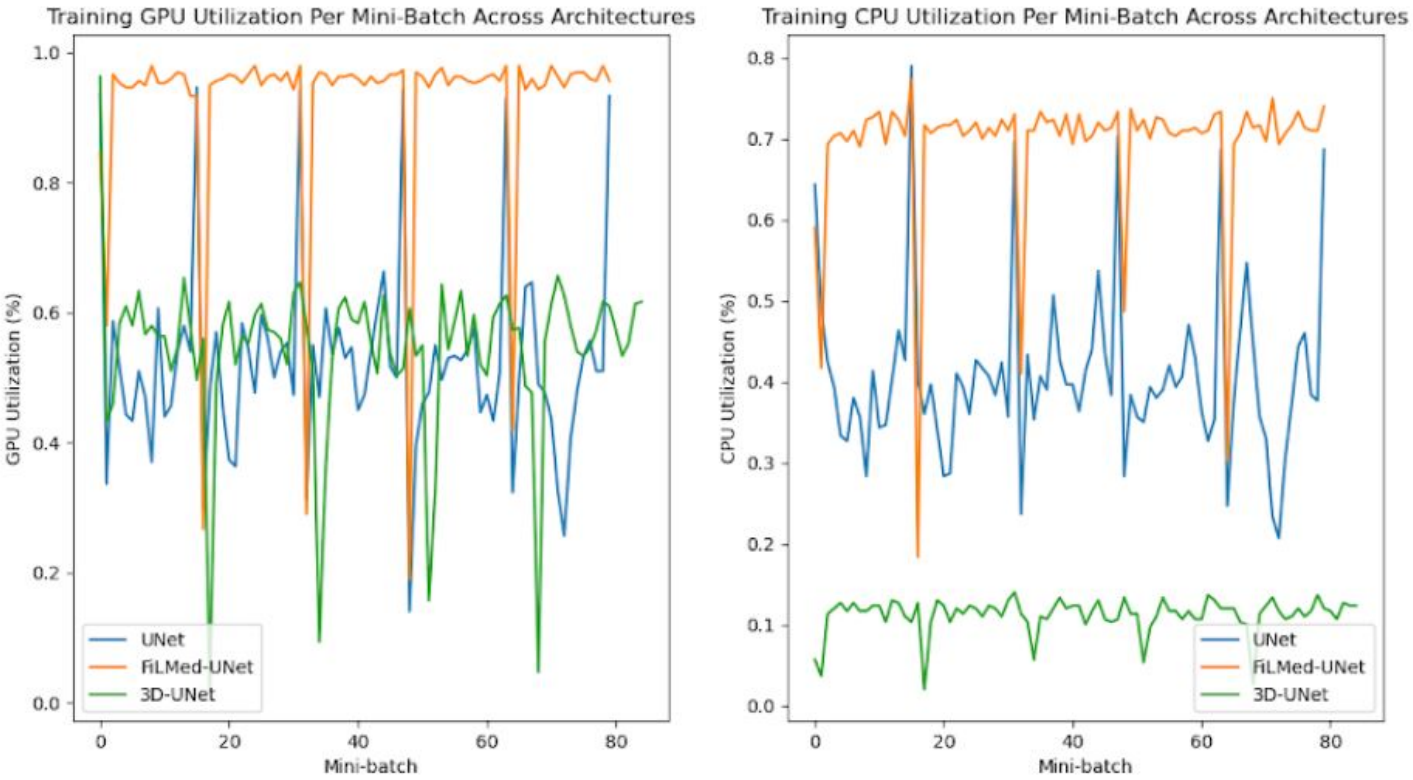

Figure 6: Training GPU and CPU utilization across UNet variants.

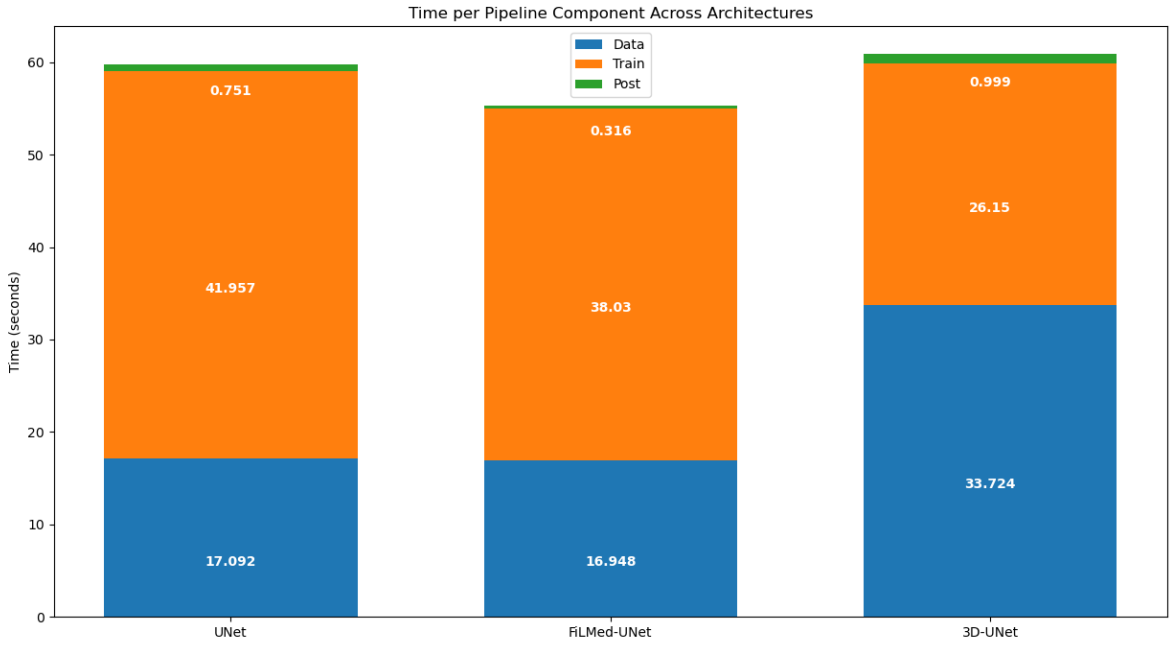

Figure 7: Runtime analysis across UNet variants.

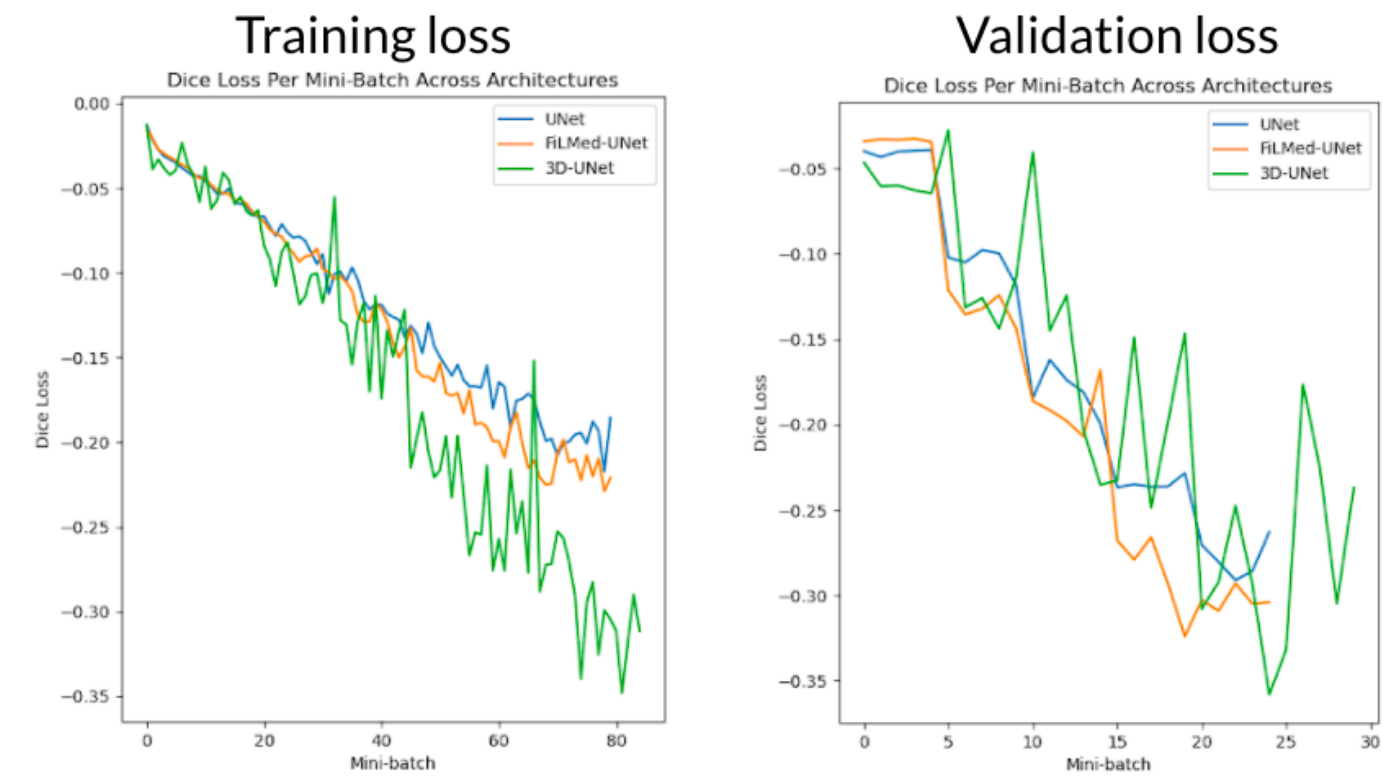

Figure 8: Training and validation dice loss per mini batch across UNet variants.

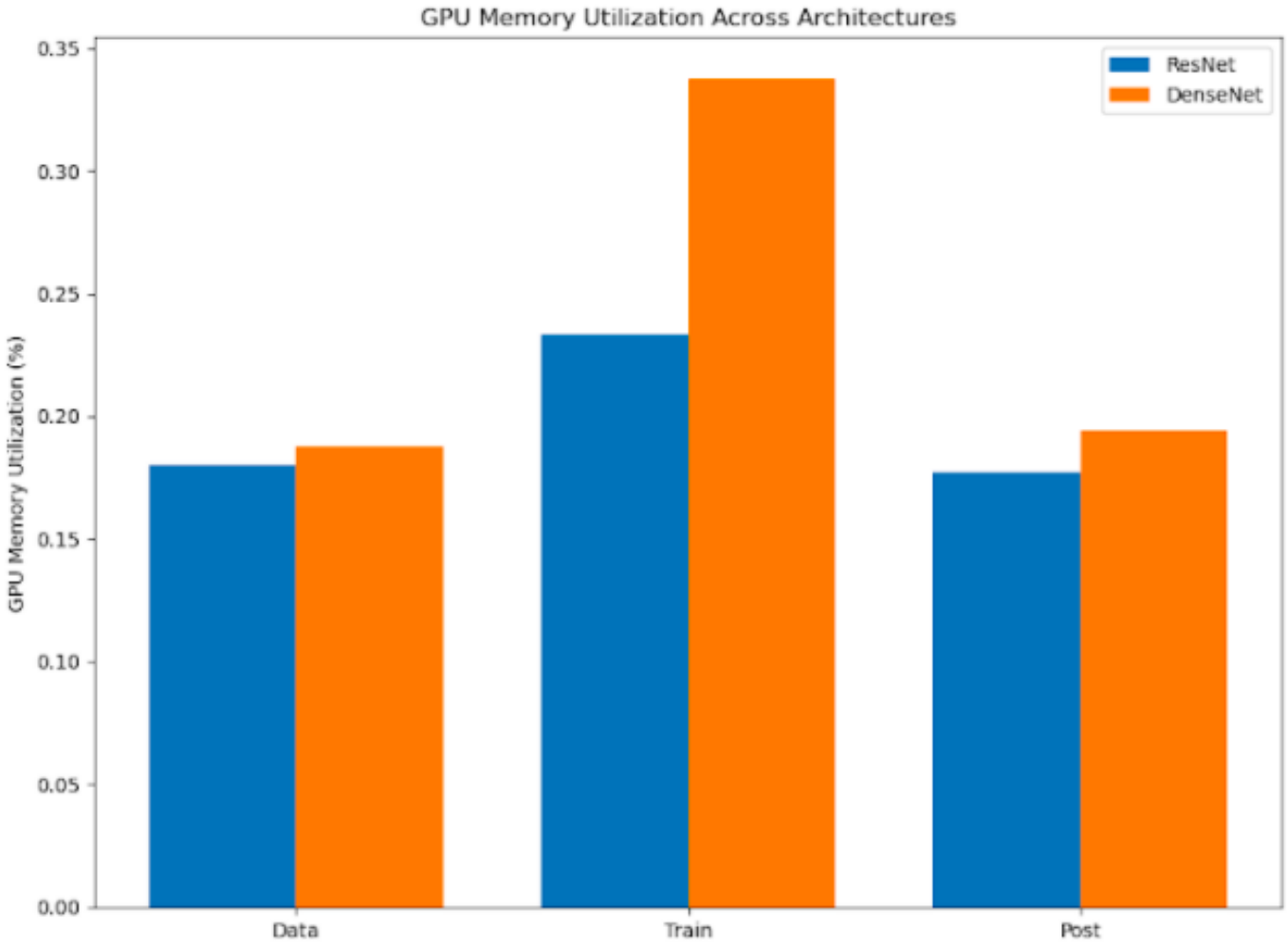

Figure 9: GPU memory utilization of ResNet and DenseNet.

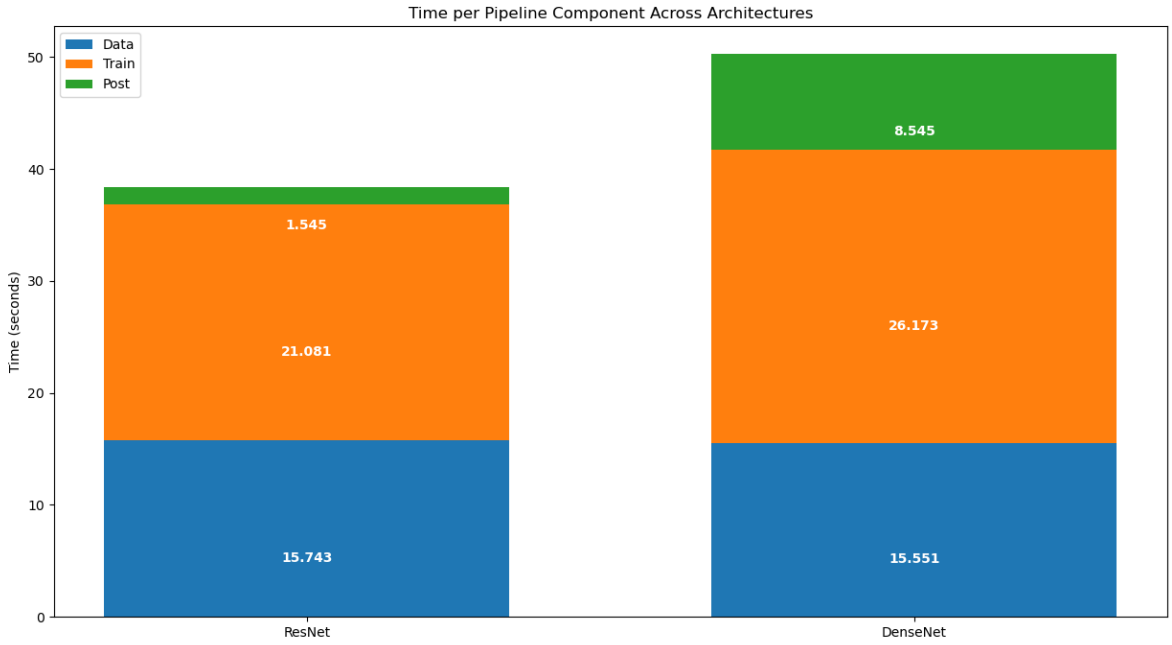

Figure 10: Runtimes for ResNet and DenseNet.

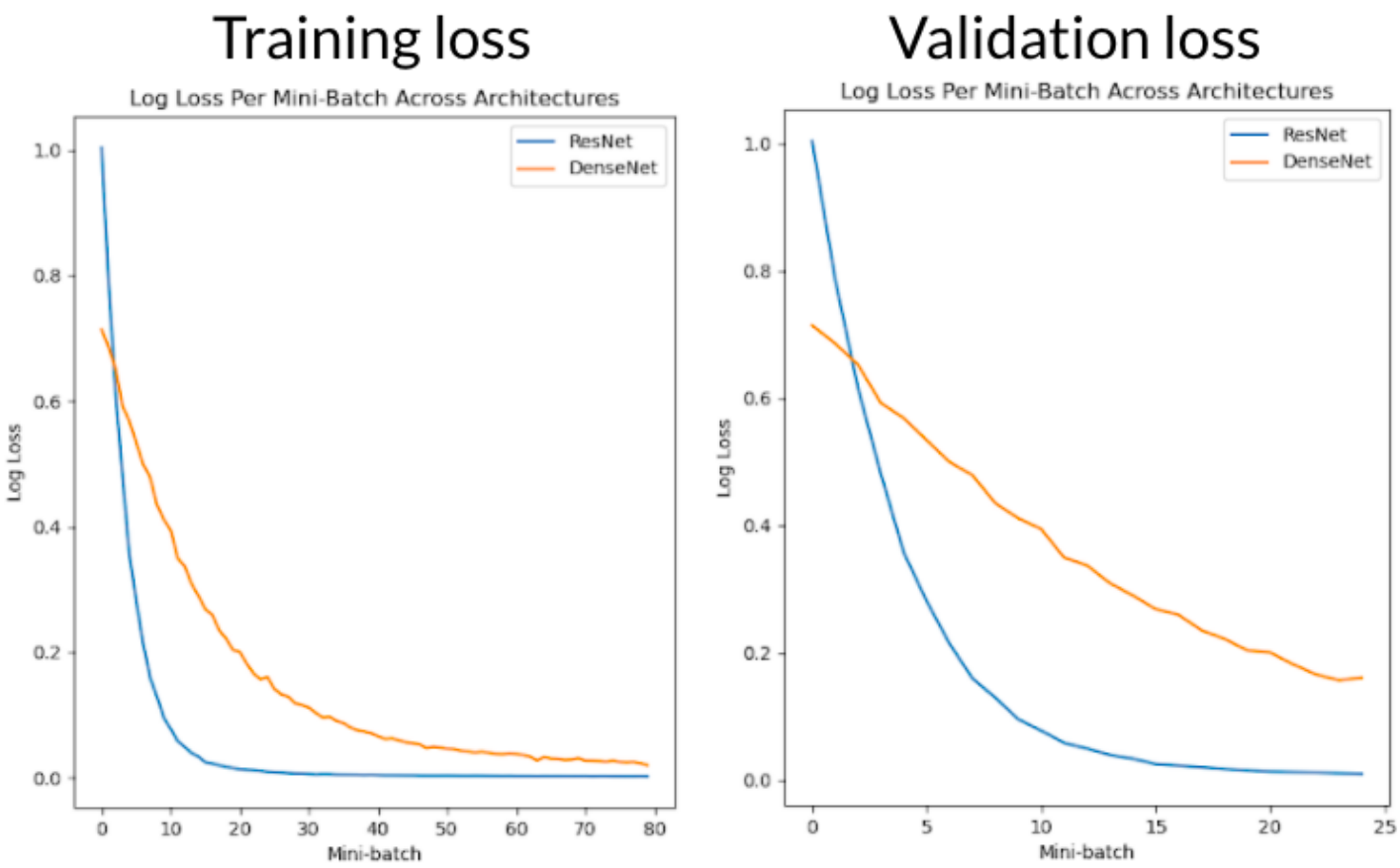

Figure 11: Training and Validation log loss of ResNet and DenseNet

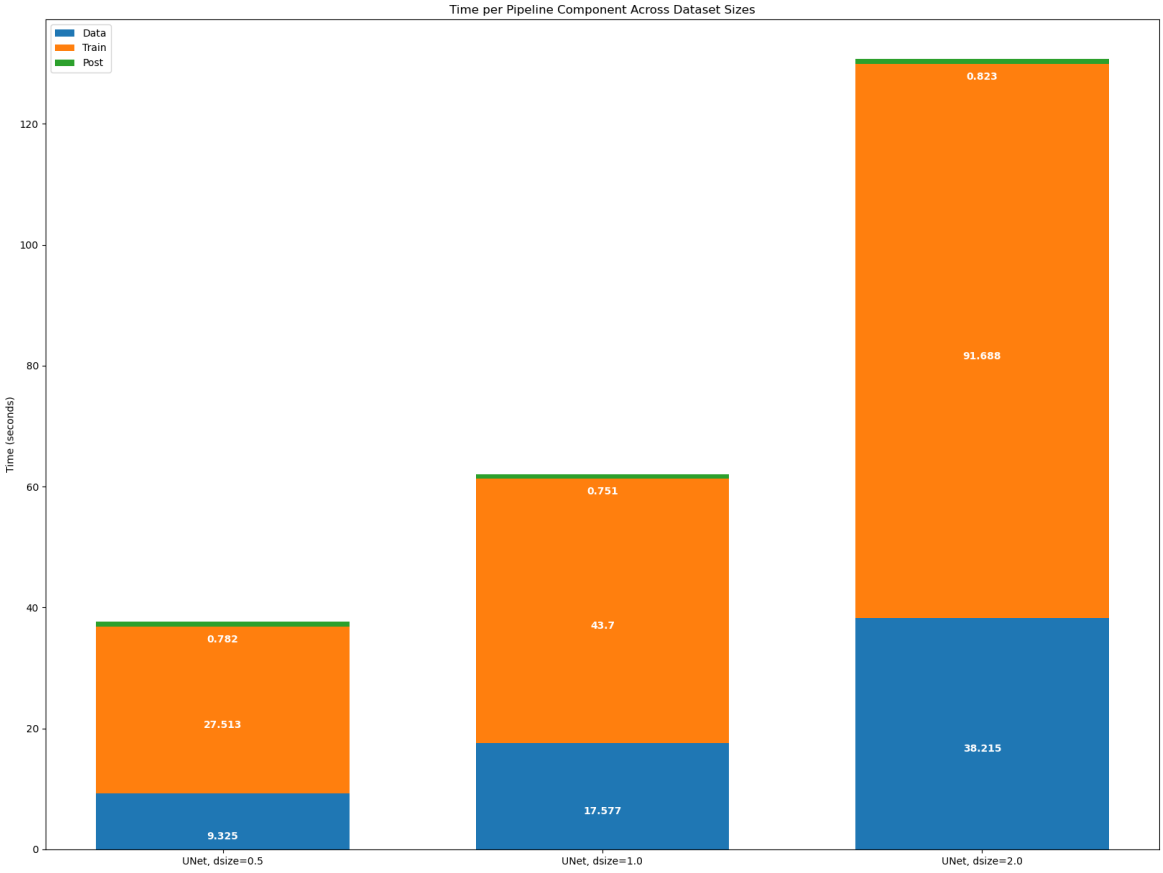

Figure 12: Runtimes for UNet for different data set scale factors: 0.5, 1.0, and 10.0

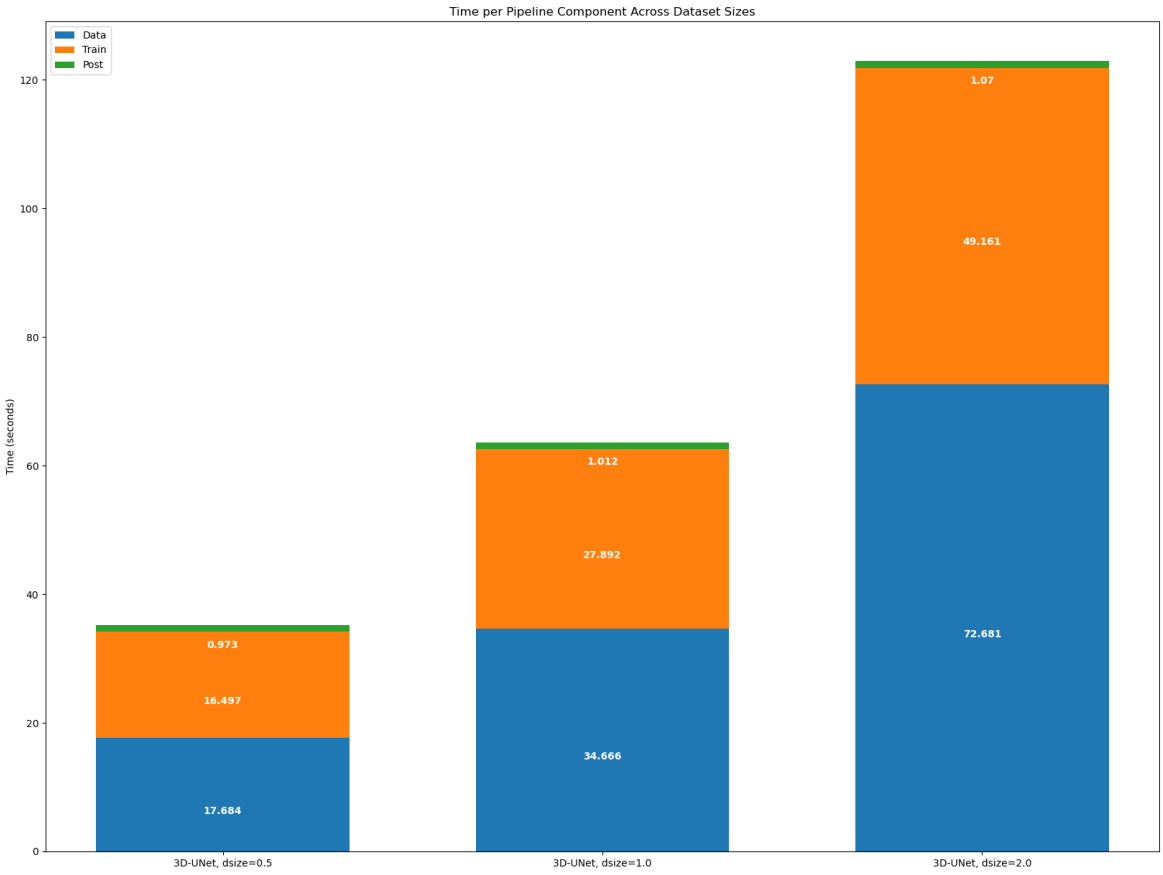

Figure 13: Runtimes for 3DModifiedUNet for different data set scale factors: 0.5, 1.0, and 10.0

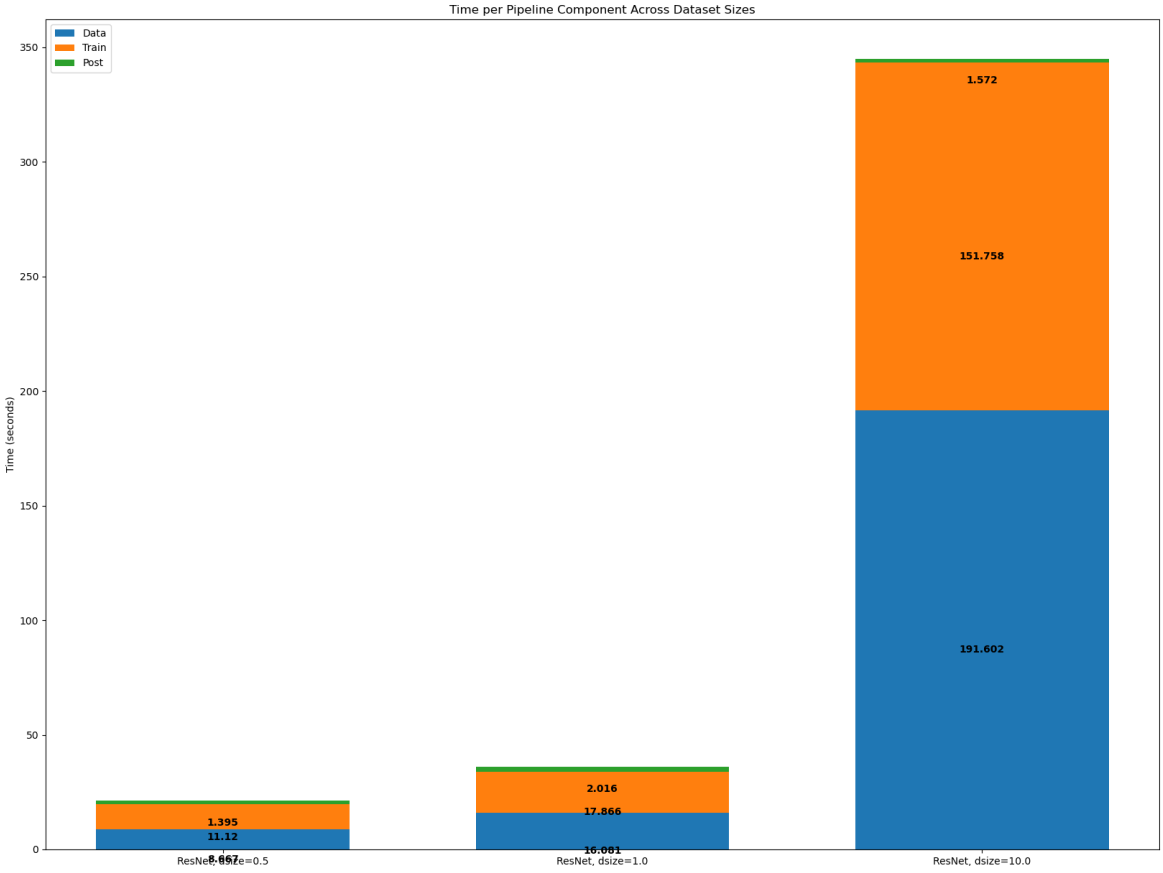

Figure 14: Runtimes for ResNet for different data set scale factors: 0.5, 1.0, and 10.0

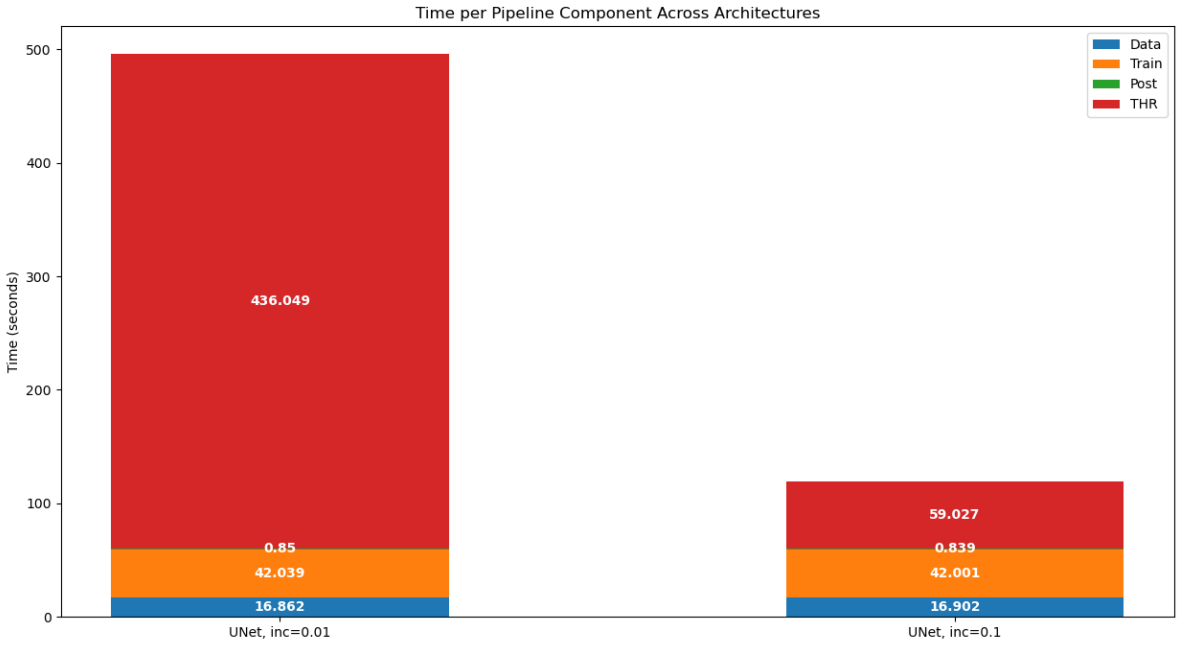

Figure 15: Runtime analysis of binary thresholding depending on increment sizes 0.01 and 0.1.

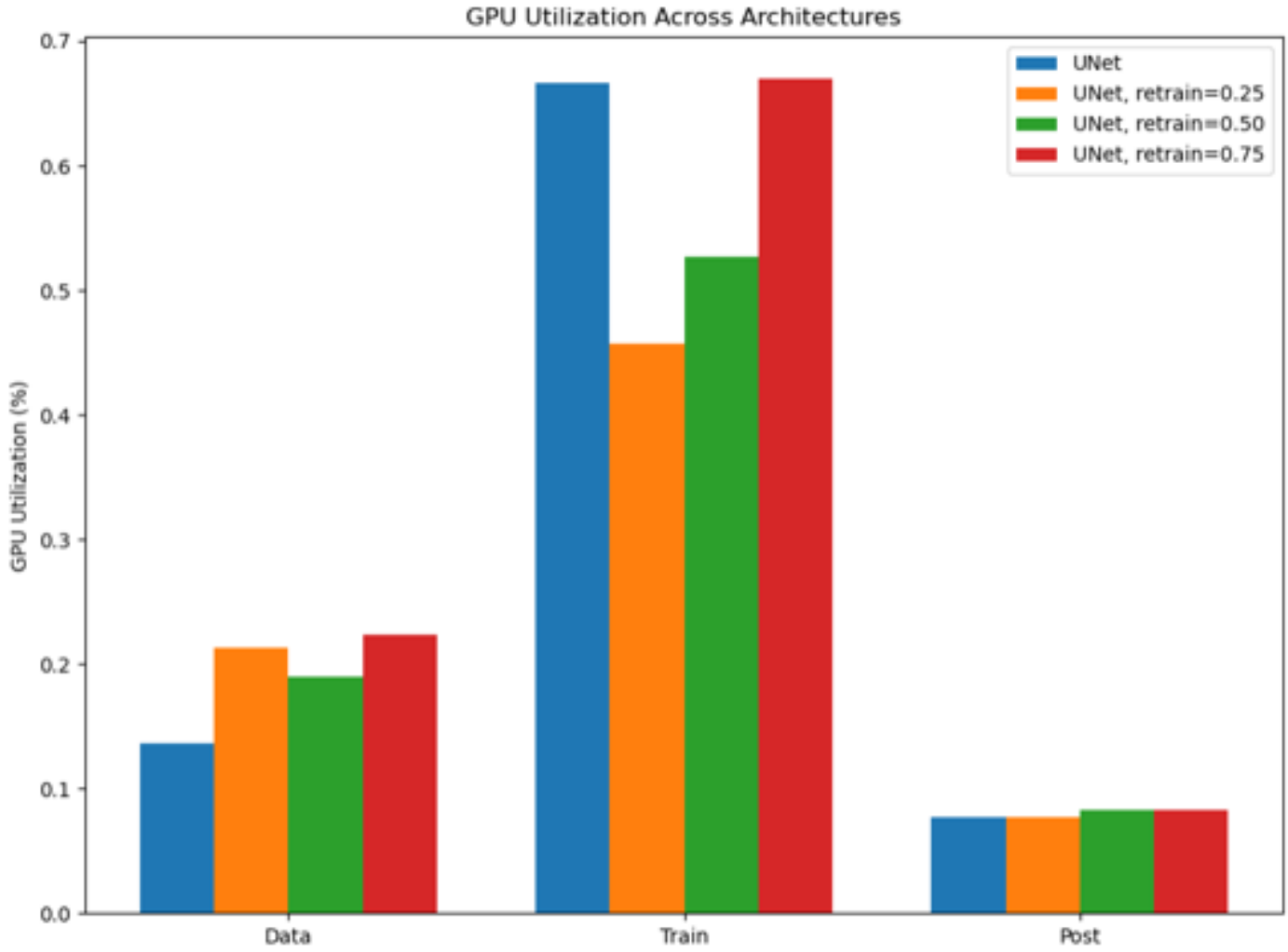

Figure 16: GPU utilization of transfer learning depending on proportion of layers trained: 0.25, 0.5, and 0.75.

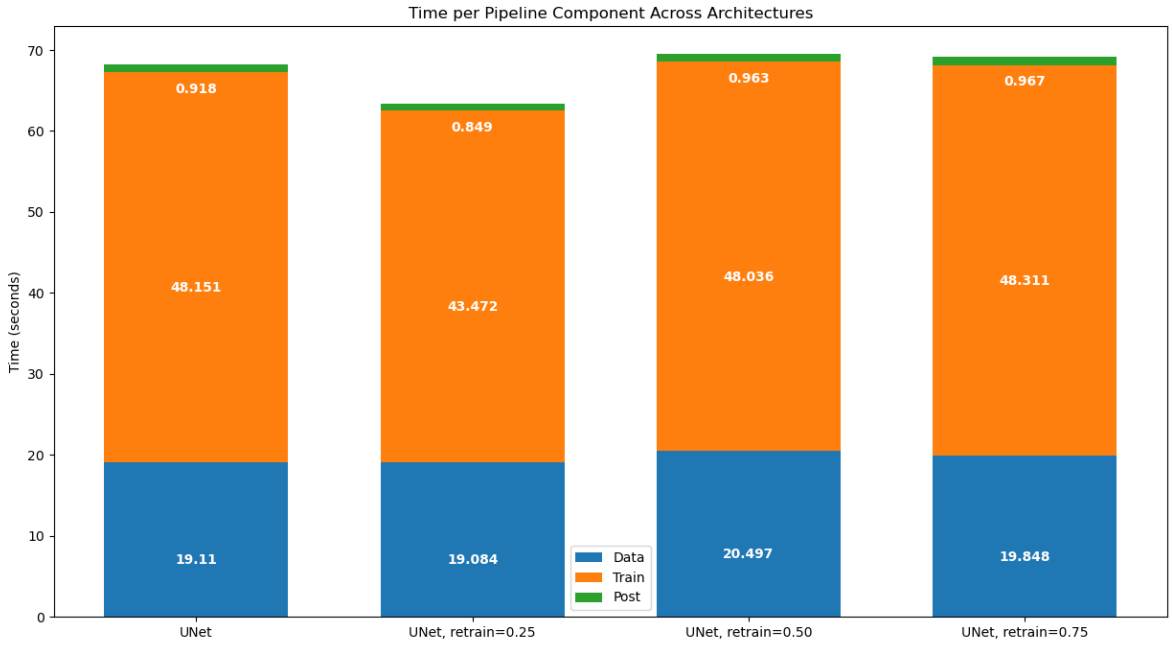

Figure 17: Runtime analysis of transfer learning depending on amount of layers retrained.

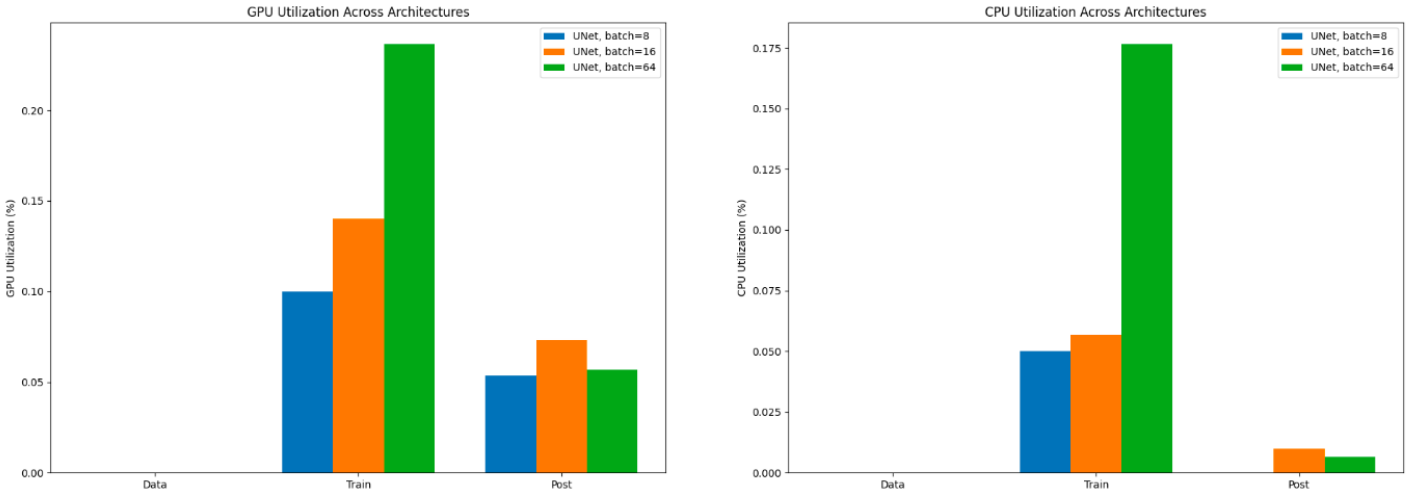

Figure 18: GPU and CPU utilization depending on batch sizes used on UNet.

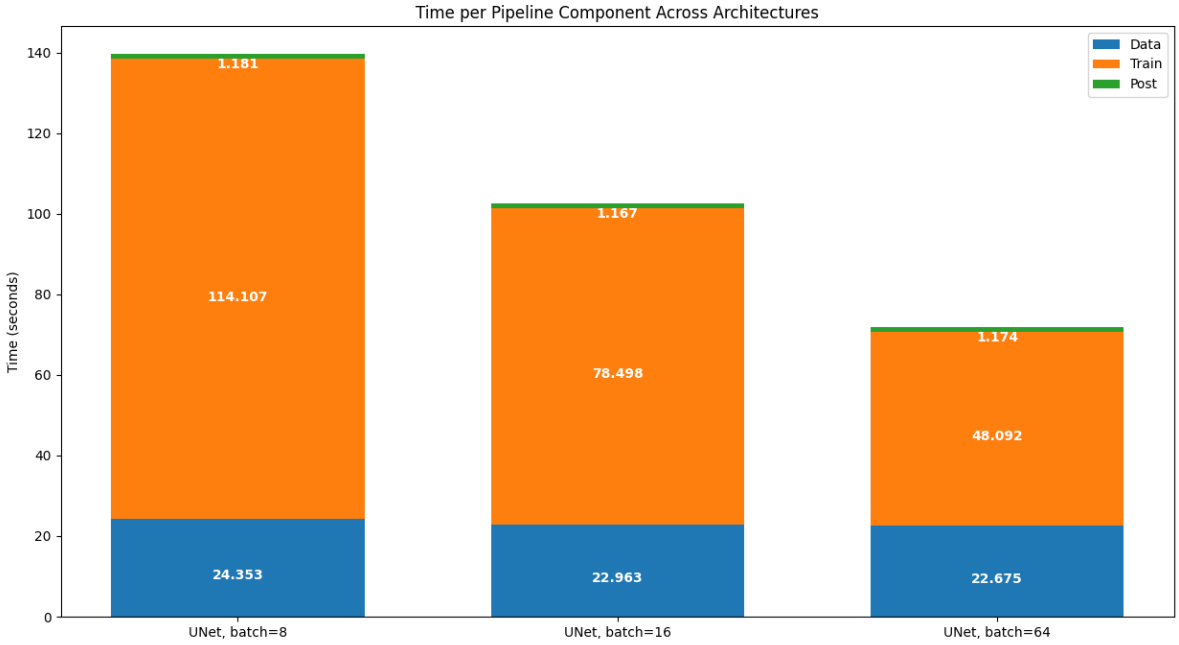

Figure 19: Runtime analysis depending on batch sizes used on UNet.

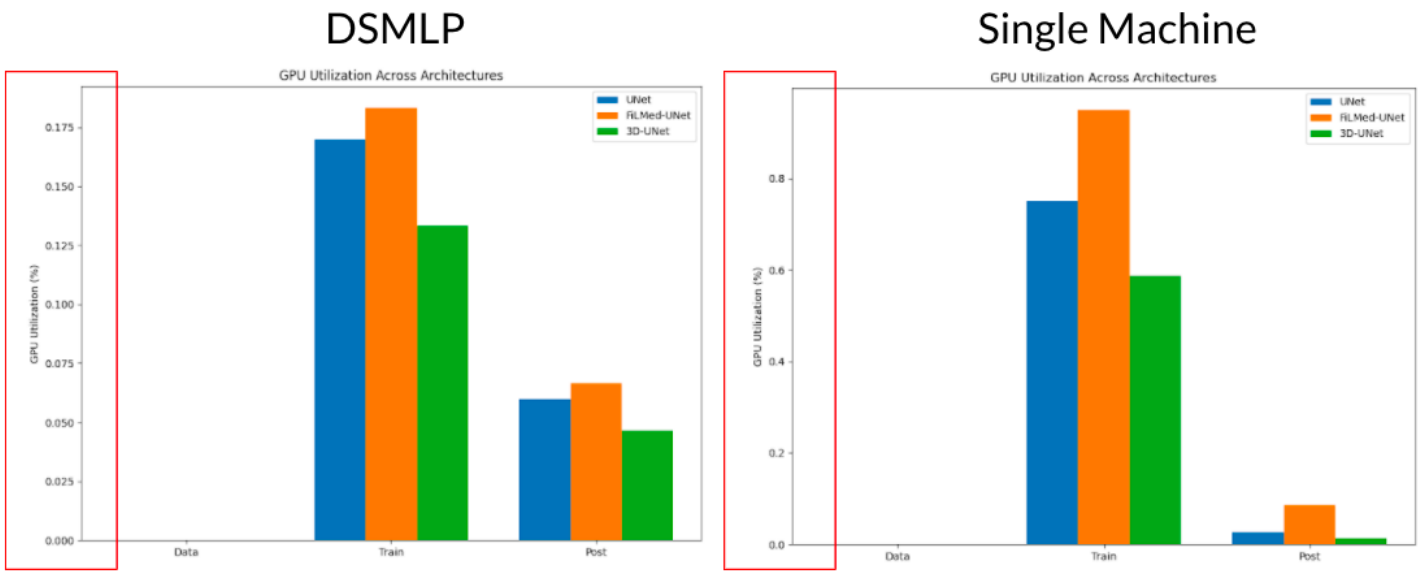

Figure 20: DSMLP and Single Machine GPU utilization across UNet variants.

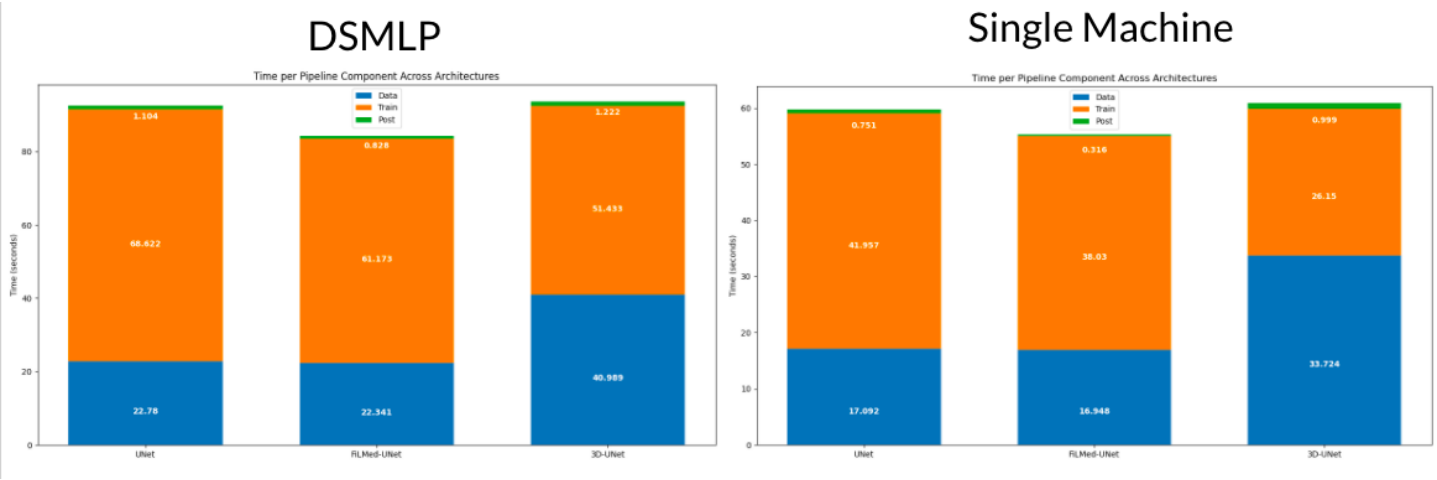

Figure 21: DSMLP and Single Machine runtimes across UNet variants.

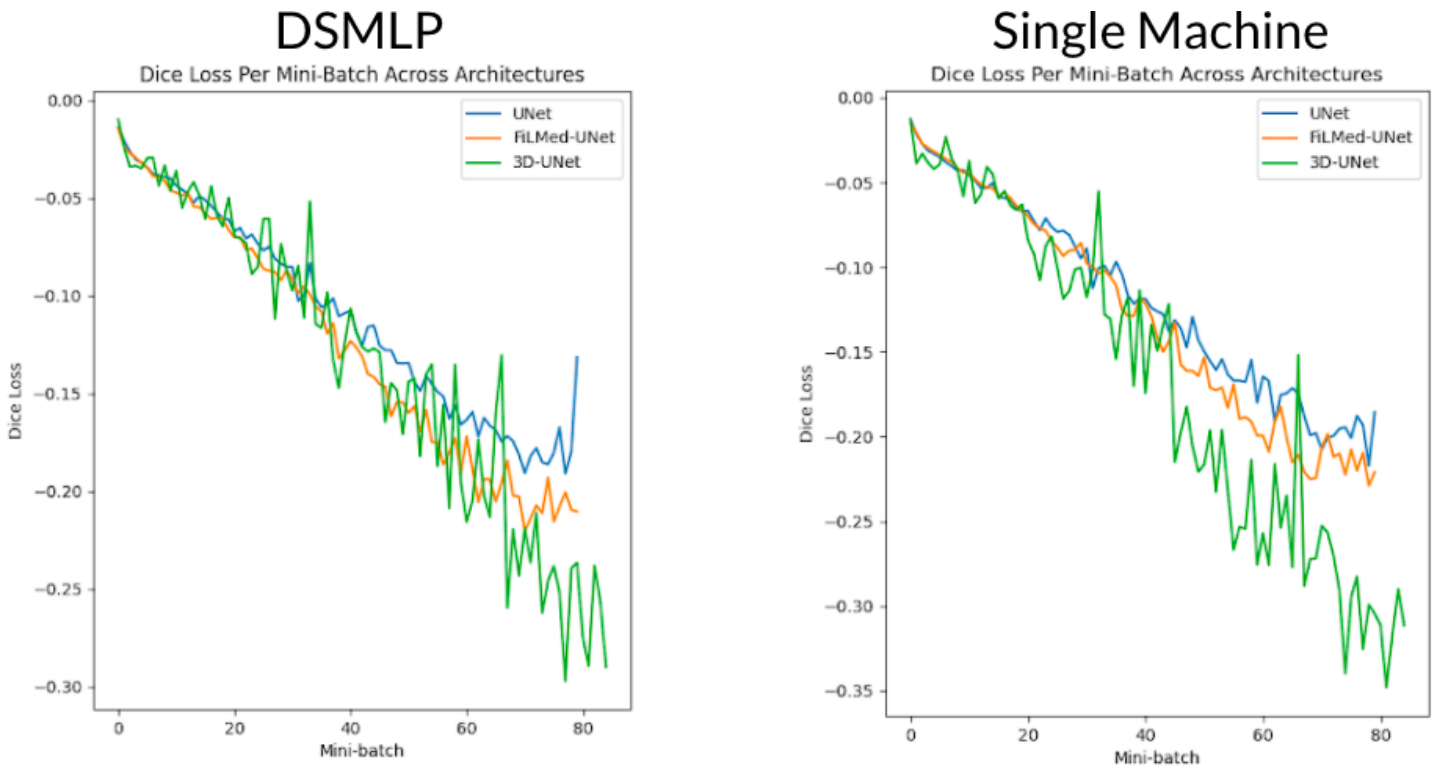

Figure 22: DSMLP and Single Machine dice loss across UNet variants.