Self-Supervised Representation Learning of Semiconductor Wafer Maps

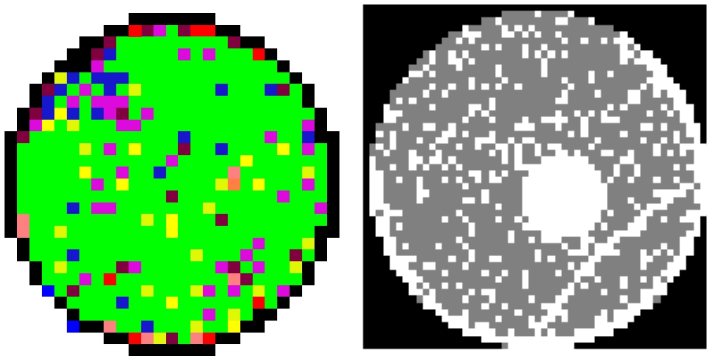
Faris Khan and Hibban Butt

Data splits: https://www.kaggle.com/datasets/mohammedfariskhan/wm811k-clean-subset
GitHub repo:https://github.com/faris-k/fastsiam-wafers

1. Problem Description

1.1 Background Information

Semiconductor device fabrication involves a variety of processing steps to create integrated circuits (chips). After these fabrication processes are complete, semiconductor wafers go through a series of “probe” tests to determine the functionality of the chips on each wafer. The results of these tests are summarized in wafer maps, like the one below.



Typically, there are two types of wafer maps used to visualize probe data. Multi-bin wafer maps, like the one on the left, show where and how a chip fails. In wafer testing, a single probe program involves several electrical tests that check different aspects of device functionality, including device logic, memory, current leakage, short circuits, etc. If a chip fails a certain test, it is assigned a unique code or “bin” corresponding to that test. Wafer map images display these bins as distinct colors. For the image on the left, all the pink pixels would have failed the same test. The second type of wafer map is a “binary” or “pass/fail” wafer map, which simply shows where chips are failing, not *how* they are failing. Use cases of multi-bin vs binary wafer maps depend on device/process maturity and whether they’re being used in product/process development or high-volume manufacturing.

Issues in fabrication processes often lead to distinct “failure shadings” on wafer maps. For example, the binary wafer map shown earlier has a scratch in the bottom-right quadrant of the wafer. It’s quite possible that a surface particle caused the scratch during a chemical mechanical

planarization step. Semiconductor engineers often have several wafer maps pulled up on a screen at once to compare wafer shadings in a wafer “lot” (group of 25 wafers processed together in manufacturing). In a wafer fab, you’ll often hear engineers say something like “A dozen of these wafers in Lot A are showing heavy Bin 10 fallout in the center and near the edge, just like the Lot B wafers from yesterday.”

1.2 Summary of the Issue

Engineers use wafer map images to visually compare failure shadings across wafers, but thousands of wafers may be probed in a day. Even though you may have a different root cause of fabrication issues on two similar-looking wafers, **visual similarity** between wafers is often the first step in a fab engineer’s data mining process.

Failure shadings are unlabeled; there often aren’t any systems in place to automatically assign wafer shadings to every wafer that is probed. There is no way to query a database of wafer maps by particular failure shadings. There is no way to compare visual similarity between wafer maps without using your eyes, since **there is no visual similarity metric** to begin with.

In the earliest papers about machine learning applied to wafer map datasets, the creation of similarity metrics relied on hand-crafted features fed into classical machine learning algorithms (often SVM in conjunction with PCA) [1]. Researchers would compute geometric and statistical features from wafer maps themselves, or at best use simple “classical” computer vision techniques like thresholding to create visual representations of wafer maps. Why hand-craft these features at all, when deep learning can learn strong visual representations from images?

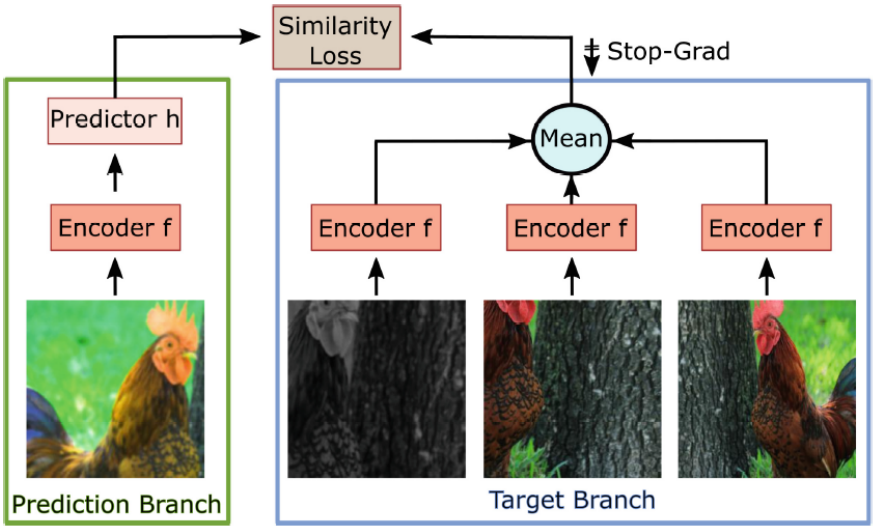
2. Methodology

2.1 FastSiam

Failure shading data is large and unlabeled. There are a variety of downstream image tasks that a fab would want to do using wafer maps, such as image retrieval and object detection. Self-supervised learning has proven to be an excellent option for learning strong, transferable visual representations of images that can be used for a variety of downstream image tasks [2], [3]. Although self-supervised learning in computer vision has shown great promise, its main barrier to entry for the average researcher is that it is far more computationally intensive compared to supervised and semi-supervised techniques. It is common to see frameworks that require long, multi-node training to be competitive with supervised and semi-supervised techniques, effectively excluding a large portion of the computer vision research community from advances in this space.

A recent paper from Pototzky et al. [4] proposes a modification of the SimSiam framework [5] called FastSiam, which promises similar performance for a fraction of the required training time and a much smaller batch size. FastSiam has been reported to perform on-par with other self-supervised methods such as SimSiam, SimCLR [6], and MoCo [7], matching ImageNet

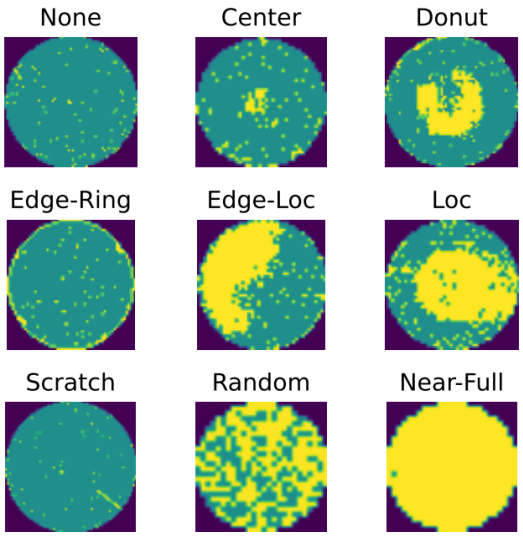
weights up to five times faster. Unfortunately, there was no public implementation of FastSiam at the time of writing, so we implemented FastSiam ourselves using lightly [8], a popular framework for self-supervised learning in PyTorch.



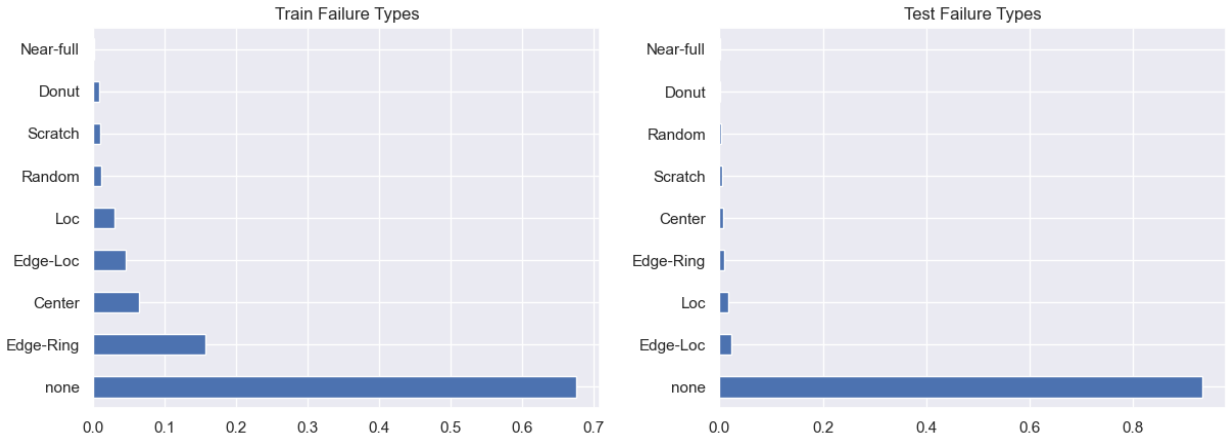
The main difference between FastSiam and SimSiam is that FastSiam uses 4 augmentations per image instead of 2. All four views are passed through the backbone and projection head, the composition of which is denoted f in the image above. Then, only the first view is passed through the predictor head, while the element-wise mean of the remaining three views is used as the target for negative cosine similarity loss (hence the terms “prediction branch” for the first view, and “target branch” for the others since the mean tensor acts as the target). Like in SimSiam, a stop-gradient is applied to prevent representations from collapsing to trivial solutions (same output for all images regardless of image content/true label). Pototzky et al. argue that increasing the number of sampled views speeds up convergence and allows for smaller batch sizes than other self-supervised learning techniques, particularly contrastive learning techniques. Specifically, the more views used for target computation, the smaller the average distance between the target and all other possible views from the distribution of views that exist due to augmentation [4]. Our implementation of FastSiam was benchmarked against the ImageNette dataset [9] (a subset of the ImageNet dataset with just 10 classes), just to see if our implementation worked.

2.2 Data

Our FastSiam implementation was used to learn visual representations of wafer maps in a self-supervised fashion. To do so, we used the WM-811K wafer map dataset, the largest publicly available dataset of real binary wafer maps sourced from semiconductor fabs [1]. Like the name implies, the full dataset has around 811,000 wafer map images at various resolutions and die sizes. However, only about 173,000 of these are labeled. Of these labeled wafers, the original authors have arbitrarily assigned 54,355 as “training” and 118,595 as “test” data. The authors have assigned one of nine labels to each labeled wafer as shown below:

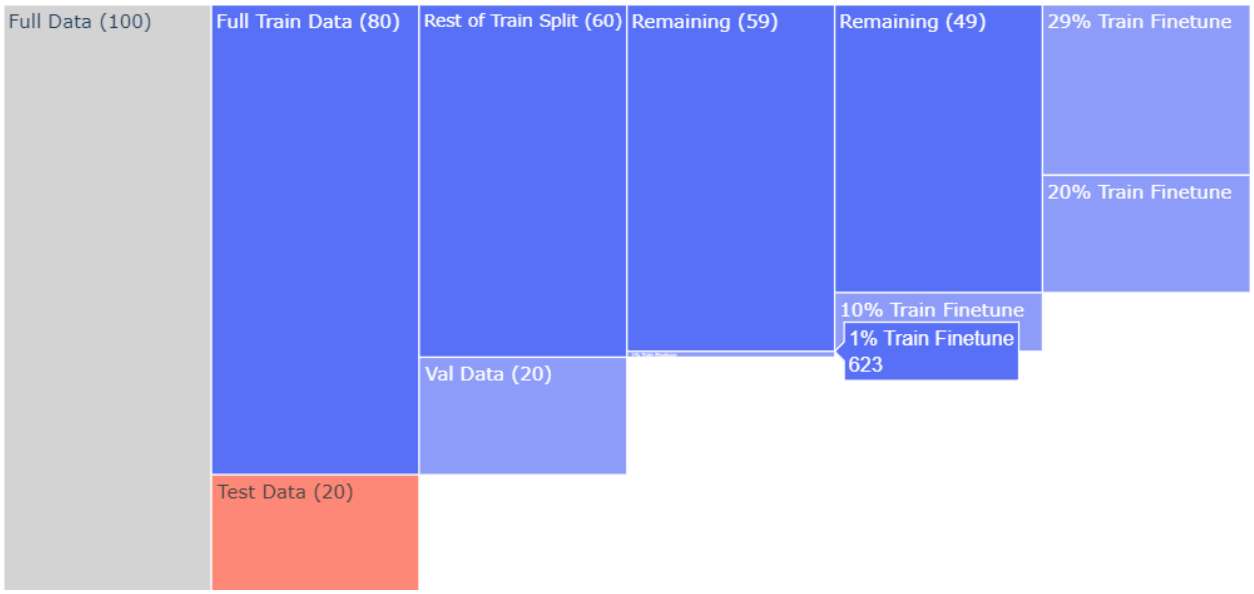


As someone who works in the semiconductor industry, classification isn't necessarily the best task for wafer maps, since there are far more than 9 wafer shading patterns that we see on wafer maps. Moreover, wafer maps are often multi-failure class; a wafer can have a scratch, edge-ring shading, and localized/loc shading all at the same time. Nevertheless, the labeled data can still serve as a foundation for our analysis. Unfortunately, the distributions of classes in the "Training" and "Test" splits from the dataset are completely different. It seems that these splits were somewhat arbitrarily decided by the original authors.

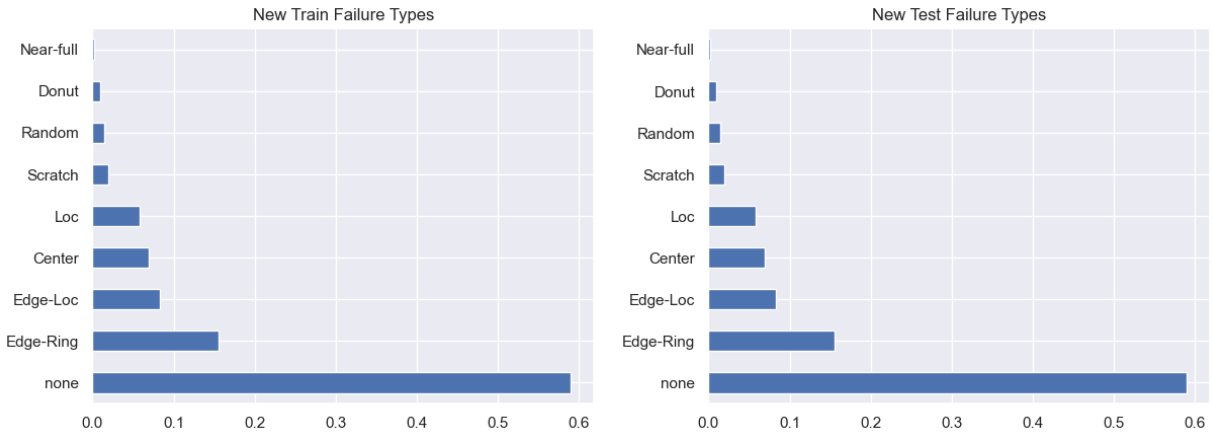


We aggregated the "Training" and "Test" subsets of the original dataset, keeping all of the original "Training" dataset and all of the non-"none" type wafer maps from the "Test" set. From this new aggregate set, we created stratified splits such that every resulting split had the same distribution of failure type categories as the full dataset. Specifically, we created an 80/20 train/test split. From the full training data corresponding to 80% of the data, we created a validation set that was 20% of the full dataset size, and we split the remaining 60% of the data into 1%, 10%, 20%, and 29% splits. This was done for the purpose of fine-tuning a linear classifier on top of a frozen encoder trained in a self-supervised manner, a common

benchmarking technique used in self-supervised learning. The icicle plot below illustrates the breakdown of the data.



Below, we show that the new full training and test datasets indeed have the same distributions due to stratified splitting.



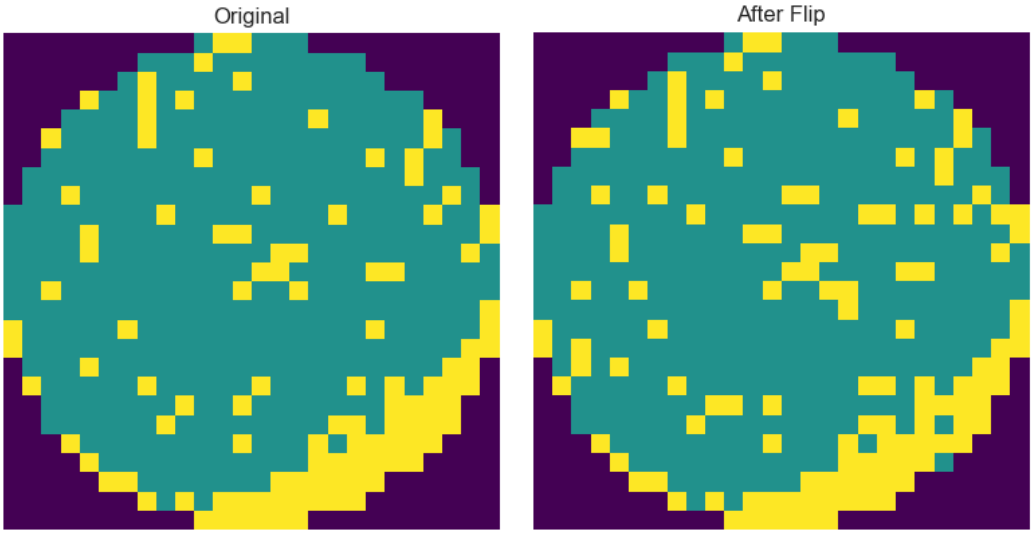
2.3 Image Augmentations for Self-Supervised Learning

Perhaps the most important aspect of self-supervised learning is the choice of image augmentations. Siamese architectures like FastSiam/SimSiam, SimCLR, MoCo, and BYOL [10] all have a strong inductive bias of learning representations of data that are invariant to certain transformations or augmentations. The choice of augmentations depends on the task. Hue et al. [11] have applied SimCLR to the WM-811K dataset in the past, noting a few augmentations that are valid and invalid for wafer map data. Popular augmentations such as gaussian blur and color jitter don't make sense for wafer map images, However, random rotation and random horizontal/vertical flips are valid for wafer maps. A failure shading on a wafer will appear mostly

the same even when reversed, and it's actually quite common for defect patterns with the same root manufacturing cause to appear inverted or slightly rotated on different wafers due to slight variations in manufacturing equipment and process conditions.

Hue et al. use random rotation, random vertical and horizontal flips, and random cropping as their "traditional" augmentation techniques (that is, augmentations that aren't domain-specific). We argue that the choice of random crops doesn't really make sense for wafer maps, since the main defect pattern in a wafer map can be cropped out due to random cropping. The justification of using cropping by Hue et al. is that "it creates new variants of wafer data by extracting a local view of an original wafer map and re-projecting that onto the full wafer," but this isn't really what random cropping does. Projecting a random portion of failing die onto a "blank" wafer can certainly be a valid augmentation technique, but because this isn't random cropping, we opted not to use cropping in our augmentation pipeline.

Regarding domain-specific augmentations, we re-implemented the randomized die noise augmentation proposed by Hue et al., in which every die has a certain probability of being flipped to the opposite sign. That is, a failing die may be flipped to be a pass, and a passing die may be flipped to be a fail. The image below shows our implementation, where the probability of flip is 3%. Note that the difference seems subtle, but for many wafers which have small die sizes and thus a high density of die per wafer, the difference can be much more pronounced.

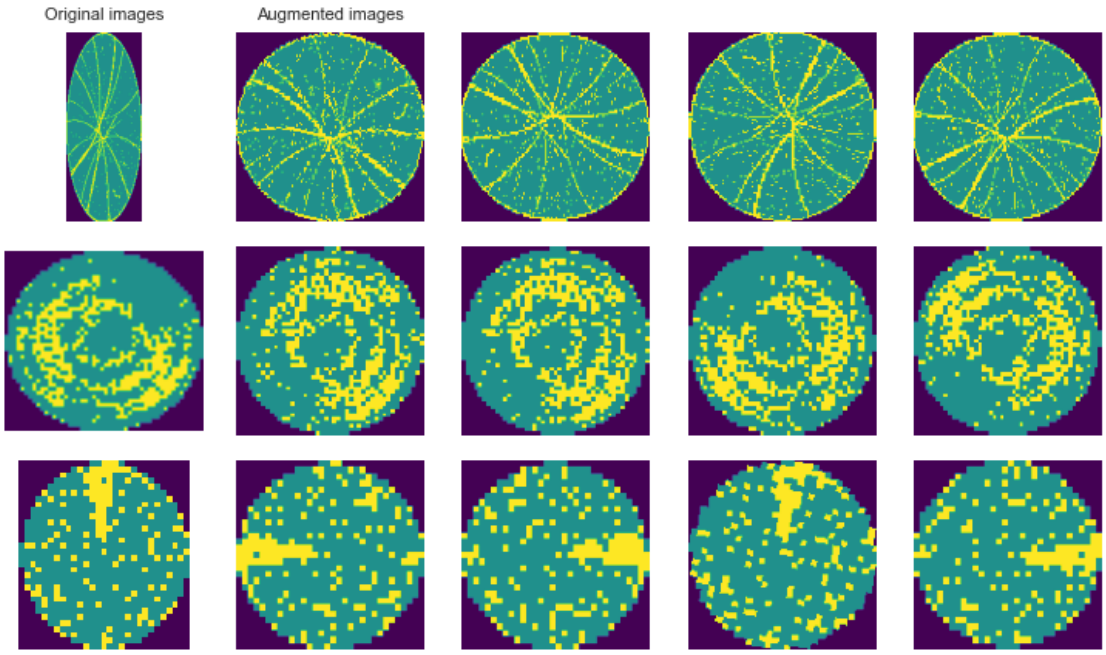


We had also hoped to implement another custom augmentation function that would create a lower die-per-wafer version of a wafer map (imagine moving right to left on the image below). Due to time constraints, we weren't able to implement this technique, but we did conceptualize a base algorithm for this. Given the locations of failing dies (yellow in the images on this page), we take the central cartesian coordinate of that die and project it onto a matrix that has fewer rows and columns (a lower die-per-wafer wafer map). In the new domain, if any of those coordinates fall within the now bigger die, that corresponding die will be marked as a fail.

The outline of all our augmentations is shown below. Note that we start with 2D NumPy arrays that may be non-square, since not all wafers have square chips.

1. Apply random die noise and scale domain from [0, 1, 2] to RGB domain [0, 128, 255]
2. Convert array/tensor to PIL Image for further torchvision transforms
3. Resize to 128 by 128 (now all wafers will appear circular)
4. 50% chance of rotating image by exactly 90 degrees
5. 50% chance of vertically flipping image
6. 50% chance of horizontally flipping image
7. 25% chance of randomly rotating image, this time somewhere between 0 to 90 degrees.
8. Convert single-channel image to 3-channel by repeating the first channel

The image below illustrates these transformations strung together. Worth noting is that we made sure to use nearest-neighbor interpolation for any resizes and rotations, something that is commonly ignored in the literature. Using bicubic interpolation, for example, will result in an image that has a non-discrete domain, but for binary wafer maps, there can only be three possible values for non-wafer area, passing die, and failing die.



4. Results and Discussion

4.1 Evaluation of FastSiam on Imagenette

In order to evaluate our implementation of FastSiam, we decided to run a benchmark on it using the Imagenette dataset, specifically the 160px version of this dataset. We followed the benchmarking suite of lightly (results shown below), in which the model is trained in a self-supervised fashion on the data, and at every step, a kNN classifier uses the raw representations from the model's backbone and tries to fit the representations to the true labels. Note the SimSiam accuracy listed in the table, 0.669 after 200 epochs.

Model	Batch Size	Epochs	KNN Test Accuracy	Time	Peak GPU Usage
BarlowTwins	256	200	0.587	86.2 Min	4.0 GByte
BYOL	256	200	0.619	88.6 Min	4.3 GByte
DCL (*)	256	200	0.762	53.3 Min	4.3 GByte
DCLW (*)	256	200	0.755	53.7 Min	4.3 GByte
DINO (Res18)	256	200	0.736	86.5 Min	4.1 GByte
MSN (ViT-S)	256	200	0.741	92.7 Min	16.3 GByte
Moco	256	200	0.727	87.3 Min	4.3 GByte
NNCLR	256	200	0.726	86.8 Min	4.2 GByte
SimCLR	256	200	0.771	82.2 Min	3.9 GByte
SimSiam	256	200	0.669	78.6 Min	3.9 GByte
SMoG	128	200	0.698	220.9 Min	14.3 GByte
SwaV	256	200	0.748	77.6 Min	4.0 GByte

Unfortunately, when we evaluated FastSiam using the same batch size of 256, this maxed out the GPU quite quickly. We used an RTX 3080 with 10GB of GDDR6X memory where the official Imagenette results pictured above were produced using a multi-node setup with multiple Nvidia Tesla V100, a card with 32GB of memory each which costs significantly more, which made it challenging to match the same batch size. So, we let it run for 200 epochs but with a far more manageable batch size of 16. Nevertheless, we ended with a 67.5% accuracy (even slightly beating out SimSiam at 66.9%).

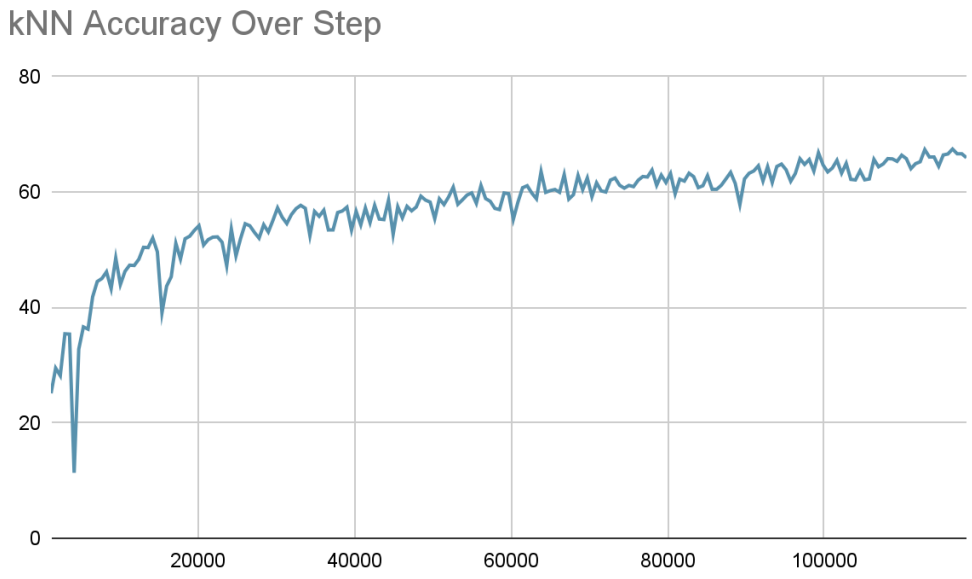
Model	Batch Size	Epochs	KNN Test Accuracy	Time	Peak GPU Usage
FastSiam	16	200	0.675	717.9 Min	4.7 GByte

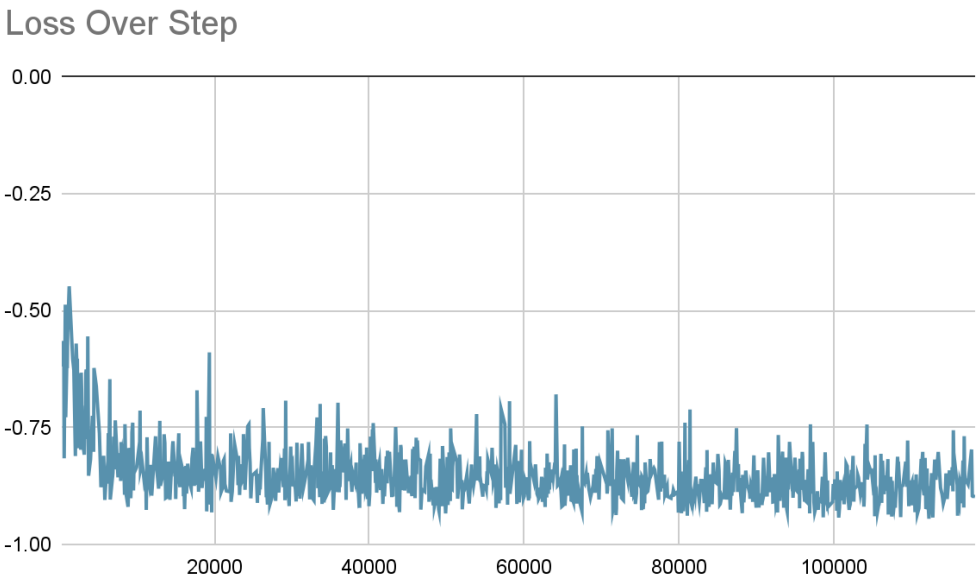
It is challenging to showcase the "fast" in FastSiam relative to SimSiam when we are so limited in our GPU resources to the point it takes just over 700 minutes, but via this means of evaluation, we are still able to show it performs at least as well as SimSiam. If evaluation conditions were as practical as the ones the official results were produced in, we are confident we could have demonstrated its faster nature in addition to its competitive accuracy.

Overall, our kNN accuracy tended to increase per step, although there were some heavy dips down at first. This seems to be consistent with SimSiam; the maintainers of the lightly repository note that SimSiam struggles at learning representations for the first few epochs, but quickly shoots up in accuracy to be on par with models like SimCLR and MoCo [12]. Like SimSiam, our

FastSiam implementation quickly began to learn after the first few epochs, but this was followed by more dips. But these dips became far less frequent and to a far smaller magnitude to the point that they seemed to fall within a reasonable margin of error.

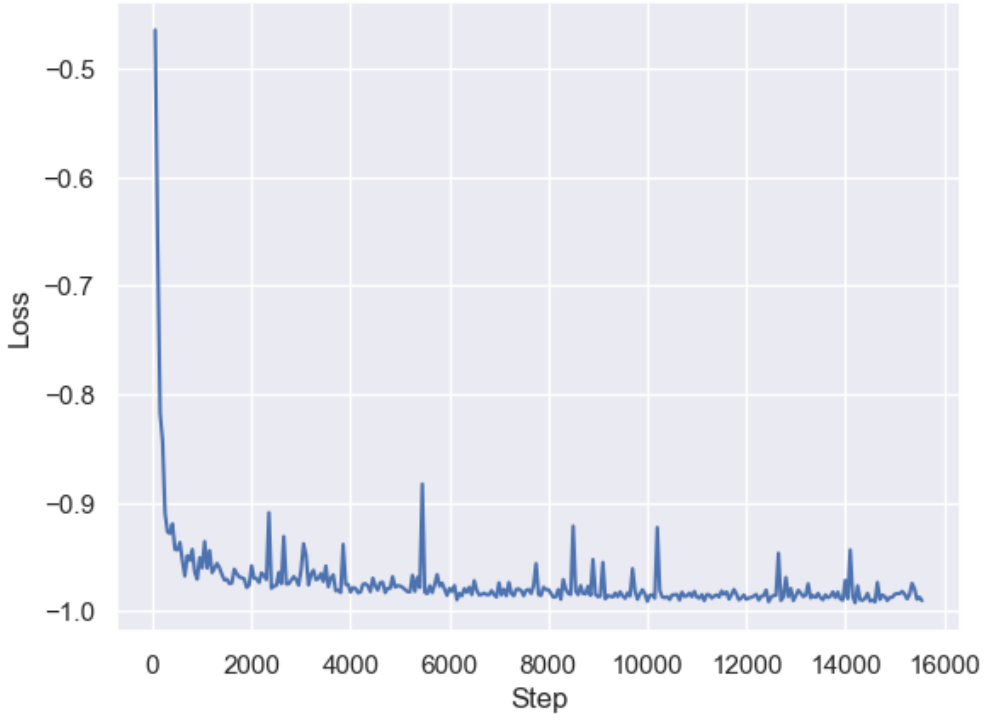
Loss also seemed to be a little high for the first short 10000 steps or so, but then simmered down to the -0.8 range for the rest of the training process. The minimum for negative cosine similarity is -1.0. In short, our benchmark shows that our implementation of FastSiam is valid, managing to match SimSiam with a batch size of just 16.





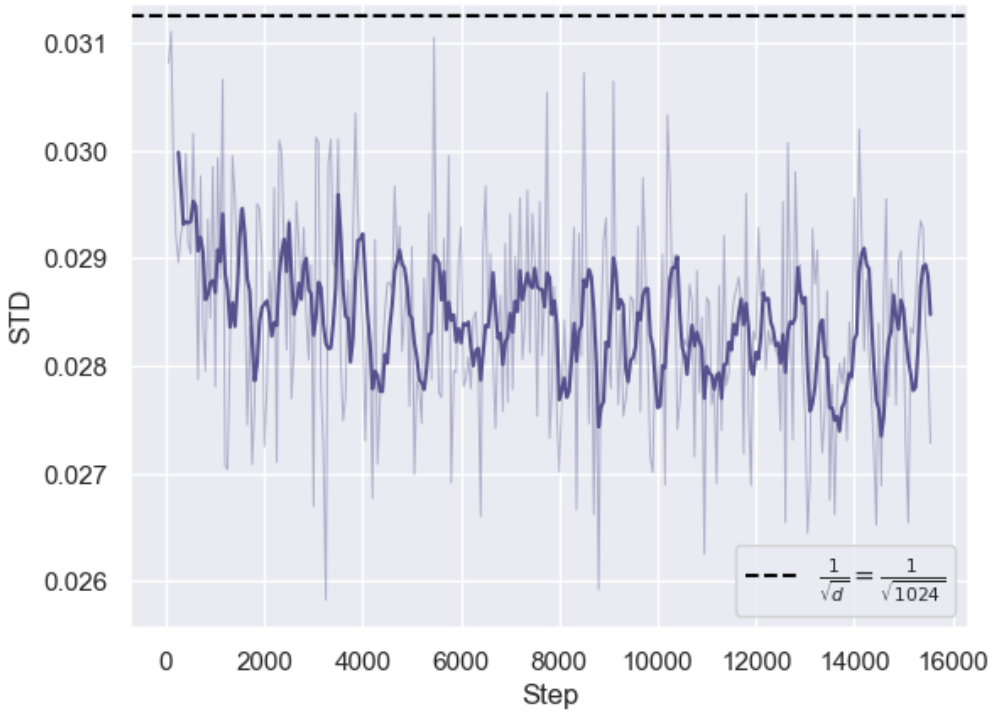
4.2 Self-Supervised Pre-training of FastSiam on WM811K

Our FastSiam setup for this portion used a modified ResNet-18 backbone from lightly, which differs from torchvision in that it uses 3x3 convolutions at the beginning instead of 7x7 to make the model faster and run better on smaller input image resolutions. For the projection head of this FastSiam model, we used 1024 dimensions (for ImageNet, the projection MLP in SimSiam has 3 layers, each 2048-dimension), and for the prediction head, we used 1024 dimensions for its input and output and 256 for its hidden layer bottleneck (for ImageNet, SimSiam uses 2048 dimensions and 512 dimensions, respectively). We used SGD with momentum, with a base learning rate of 0.06 and no additional learning rate scheduling. We trained this model on our full training dataset (49,798 images) for just 10 epochs on an RTX 3080 Ti (12GB). In the FastSiam paper, all results are reported at 10 epochs and 25 epochs, and while we could have gone to 25 epochs here as well, the model had already shown very strong training as shown by its loss curve below.



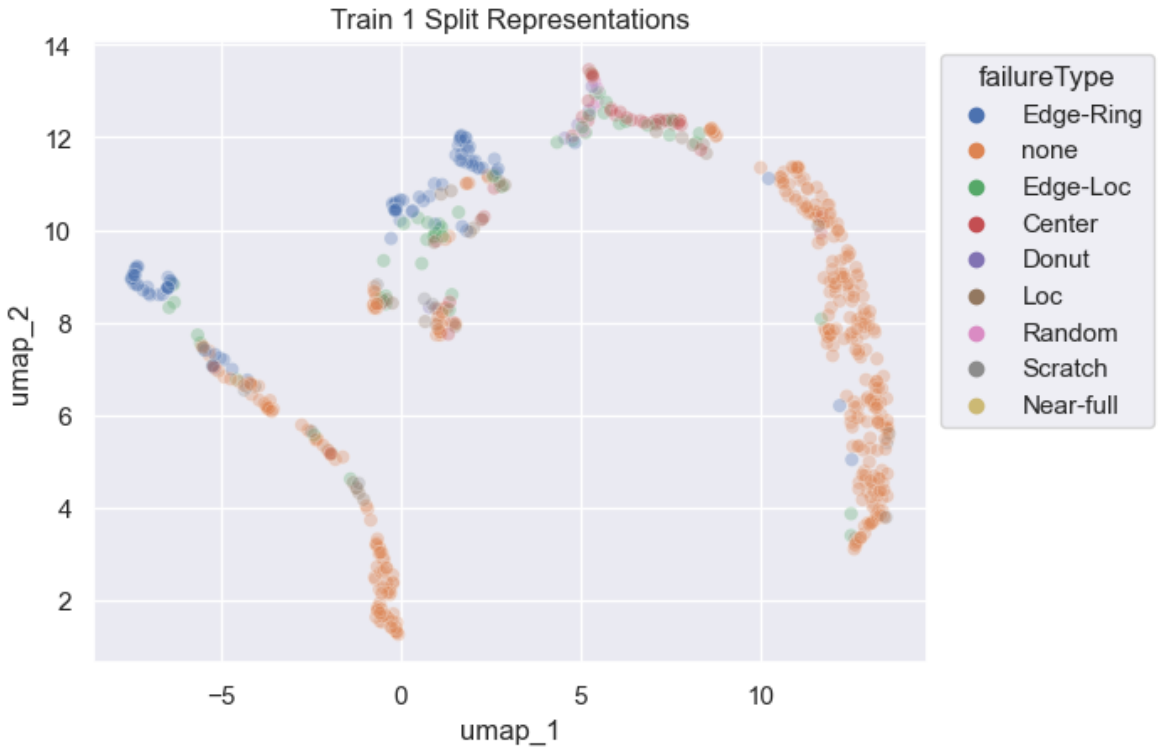
To ensure that the model had not learned trivial/degenerate solutions in which all representations were the same, we used a method from the SimSiam paper, in which the standard deviation of the L2-normalized representation was logged at every step. For SimSiam, collapsed representations will have an STD of zero, while strong training will hover near a value of 1 / sqrt(d), where d is the size of the representation. Confusingly, Chen et al. write that they monitor this metric for z, which technically refers to the output of the projection head, not the representation (raw output from the ResNet backbone after global average pooling and flattening). Following this notation, we monitored this metric for our projection head outputs, and for FastSiam we see that this value hovers slightly lower than 1 / sqrt(d) in our case. This may

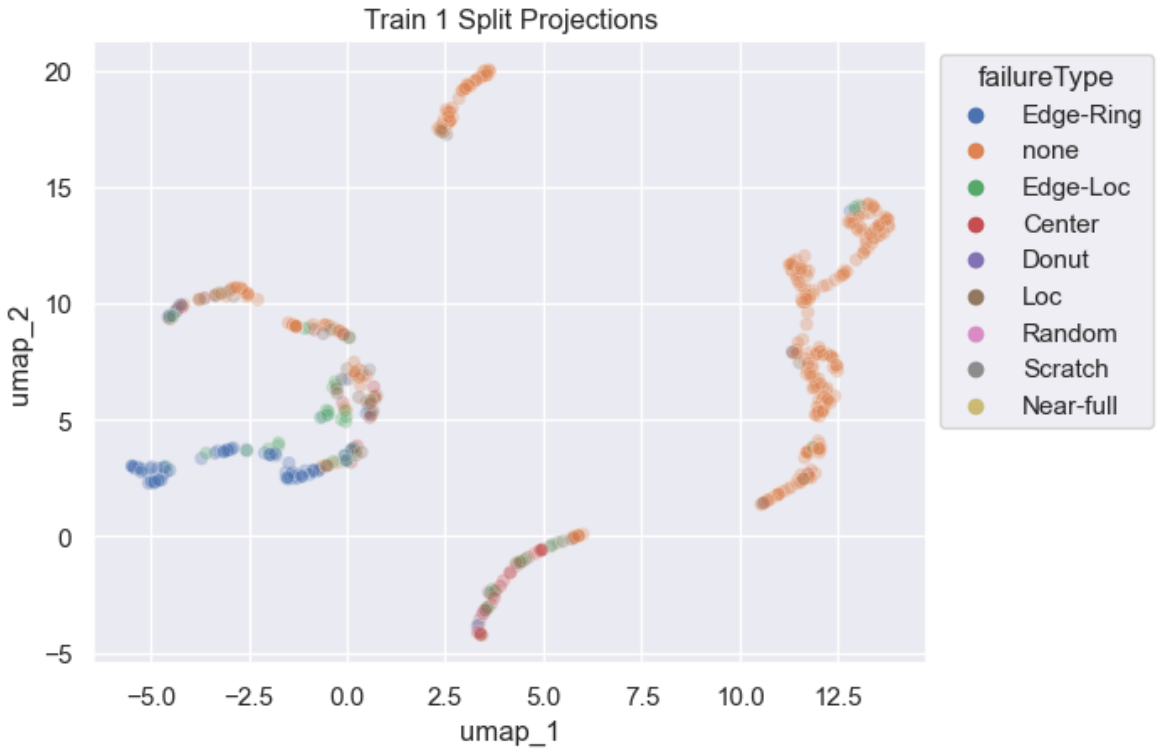
be a typo on the author's part or a misunderstanding on our part, but in any case, the plot below shows that the representations (really, the projected representations) have a non-zero standard deviation somewhat close to 1 / sqrt(d), so the model has not collapsed and after 10 epochs, the representations should be good.

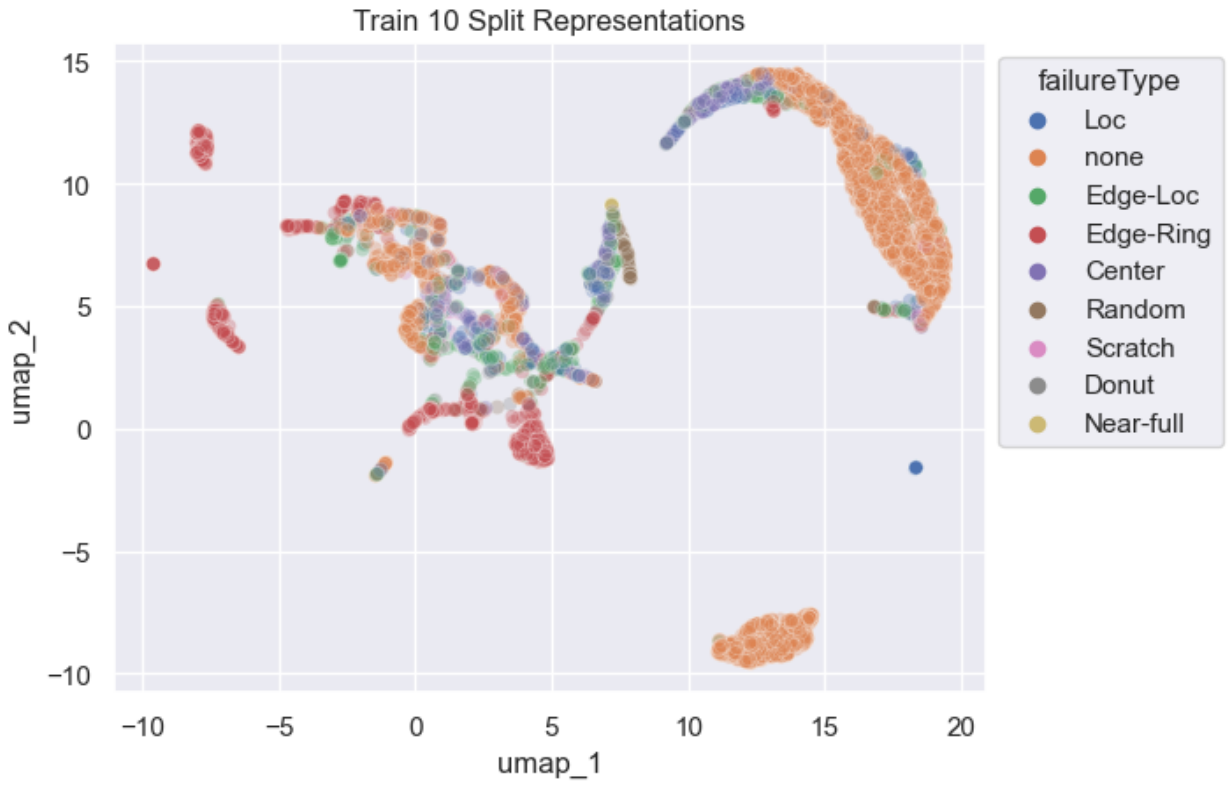


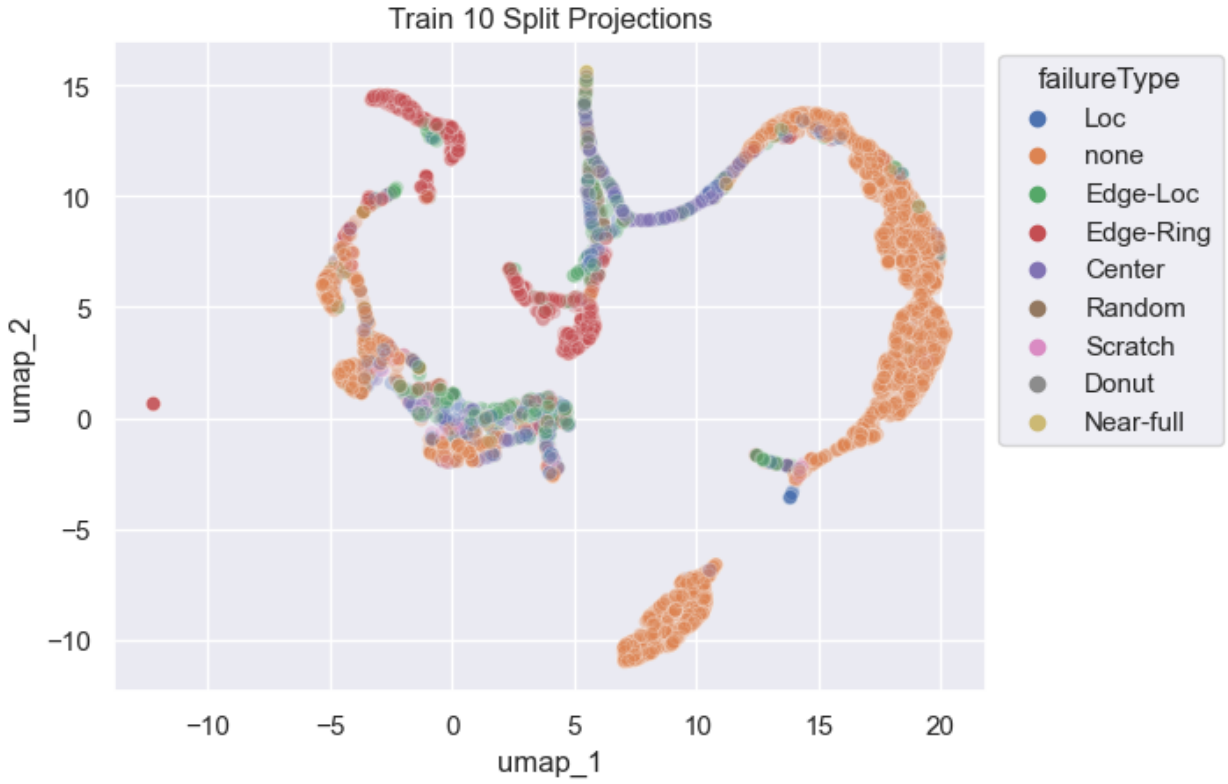
4.3 Comparing to a Fully Supervised Baseline

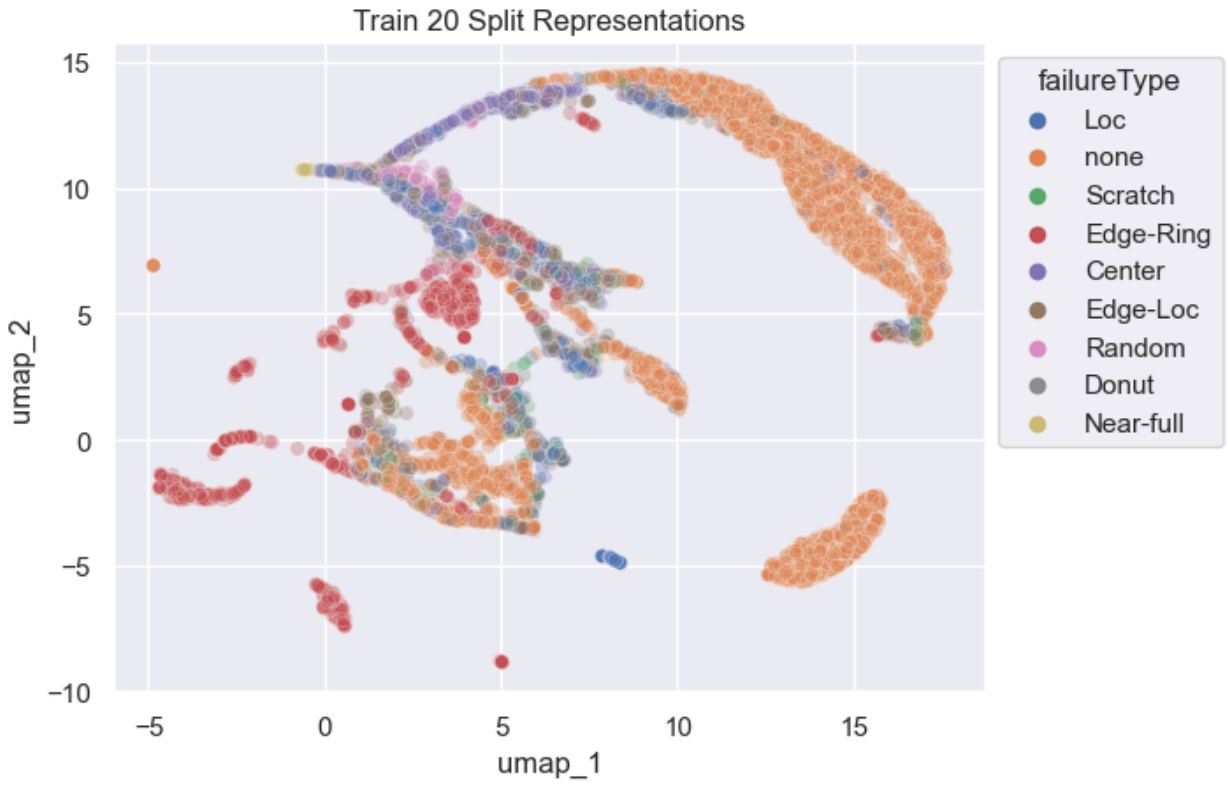
Before getting into more benchmark metrics, we decided to visualize the frozen features from our pre-trained model on our test splits mentioned earlier in the report. Below, we show UMAP embeddings of the representations colored by ground-truth labels, and we can see that FastSiam has enabled strong representations of these wafer maps to be learned in just 10 epochs. Tensorboard has an embedding projector that enables interactive visualization and exploration of embeddings; on our end this link doesn't work, but check our GitHub repository at https://github.com/faris-k/fastsiam-wafers to check for updates on this if interested: https://projector.tensorflow.org/?config=https://gist.githubusercontent.com/faris-k/658a3bc7eda3010804eb886ecb93744d/raw/bdfc40ac587a8e70b085684b27dd372a77b18588/fiastsiam_1_config.json. The static views for the UMAP embeddings for representations and projections are shown on the next few pages. The outputs from the prediction head look much the same as the that from the projector head, so we omit this (our jupyter notebooks show them all though).

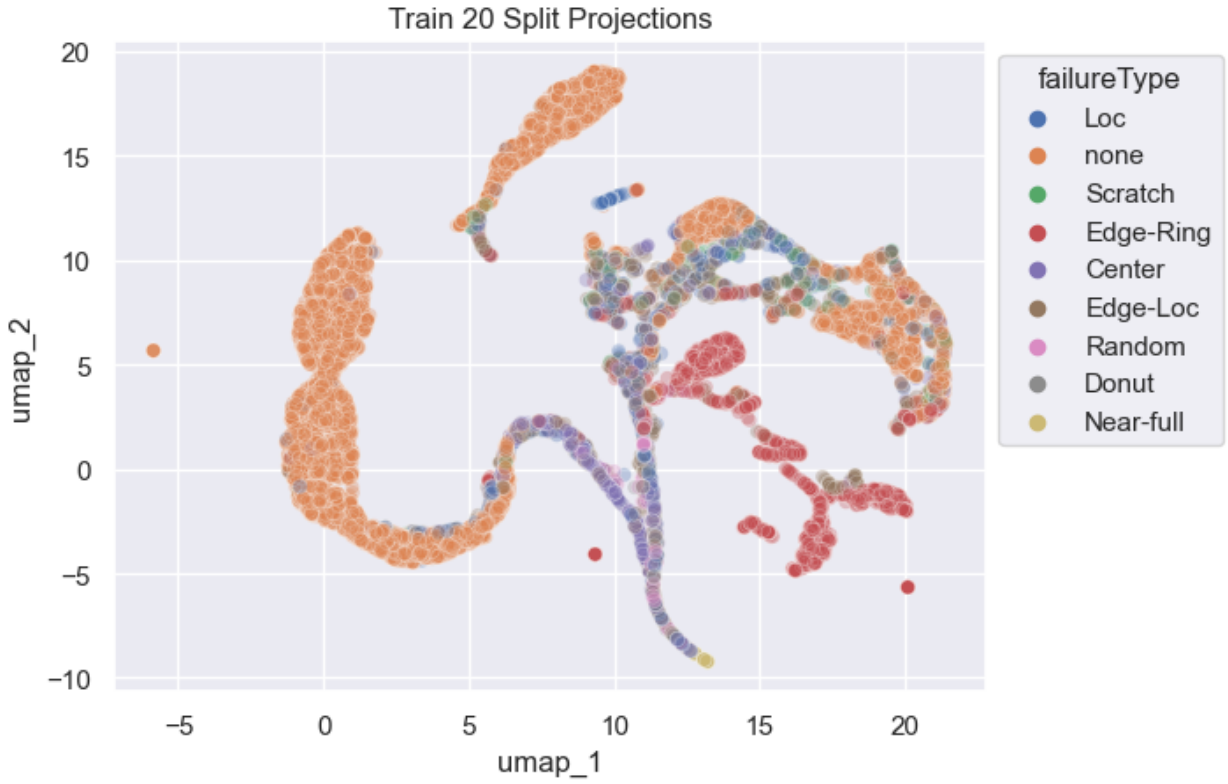

Train 1 Split Representations
umap_2
umap_1
failureType
Edge-Ring
none
Edge-Loc
Center
Donut
Loc
Random
Scratch
Near-full


Train 1 Split Projections
umap_2
umap_1
failureType
Edge-Ring
none
Edge-Loc
Center
Donut
Loc
Random
Scratch
Near-full


Train 10 Split Representations
failureType
Loc
none
Edge-Loc
Edge-Ring
Center
Random
Scratch
Donut
Near-full
umap_2
umap_1
15
10
5
0
−5
−10
−10
−5
0
5
10
15
20


Train 10 Split Projections
failureType
Loc
none
Edge-Loc
Edge-Ring
Center
Random
Scratch
Donut
Near-full
umap_2
umap_1
15
10
5
0
−5
−10
−10
−5
0
5
10
15
20


Train 20 Split Representations
umap_2
umap_1
15
10
5
0
−5
−10
−5
0
5
10
15
failureType
Loc
none
Scratch
Edge-Ring
Center
Edge-Loc
Random
Donut
Near-full


Train 20 Split Projections
umap_2
umap_1
20
15
10
5
0
−5
−10
−5
0
5
10
15
20
failureType
Loc
none
Scratch
Edge-Ring
Center
Edge-Loc
Random
Donut
Near-full

In general, it seems like the projections form tighter clusters than the representations. It seems this would imply that finetuning using the projections would be better for a wafer classification task than the representations, but several papers have noted that the representations straight from the backbone of the encoder generally provide better, more transferable features for downstream tasks.

We compare the performance of our FastSiam features against a similar architecture trained end-to-end in a fully supervised fashion. Specifically, we just use ResNet-18 for a 9-class problem. The fully supervised model is trained normally on each of the training splits corresponding to 1%, 10%, and 20% of the available labeled data. For our FastSiam features, we use the frozen representations, scale them with scikit-learn's StandardScaler, then use the scaled features to train a shallow linear layer (512 hidden dimensions to 9-dimension output) in a supervised fashion on the respective training subsets. We repeat this using the frozen projector head outputs to see if the representations are indeed better for downstream tasks than the projections. Below, we display the multiclass accuracy, area under the receiver operating curve (AUROC), and macro-F1 score. For context, the 1% split has just 623 samples, the 10% has 6225 samples, and the 20% split has 12,449 samples.

Model	**Fully Supervised ResNet-18**			**FastSiam ResNet-18 Representations**			**FastSiam Projector Head Outputs**		
Data Split	1%	10%	20%	1%	10%	20%	1%	10%	20%
Acc	0.458	0.605	0.764	**0.663**	**0.753**	**0.786**	0.568	0.679	0.749
AUROC	0.908	0.959	**0.989**	**0.956**	**0.987**	**0.989**	0.927	0.978	0.985
F1	0.445	0.547	0.740	**0.658**	**0.755**	**0.792**	0.538	0.682	0.749

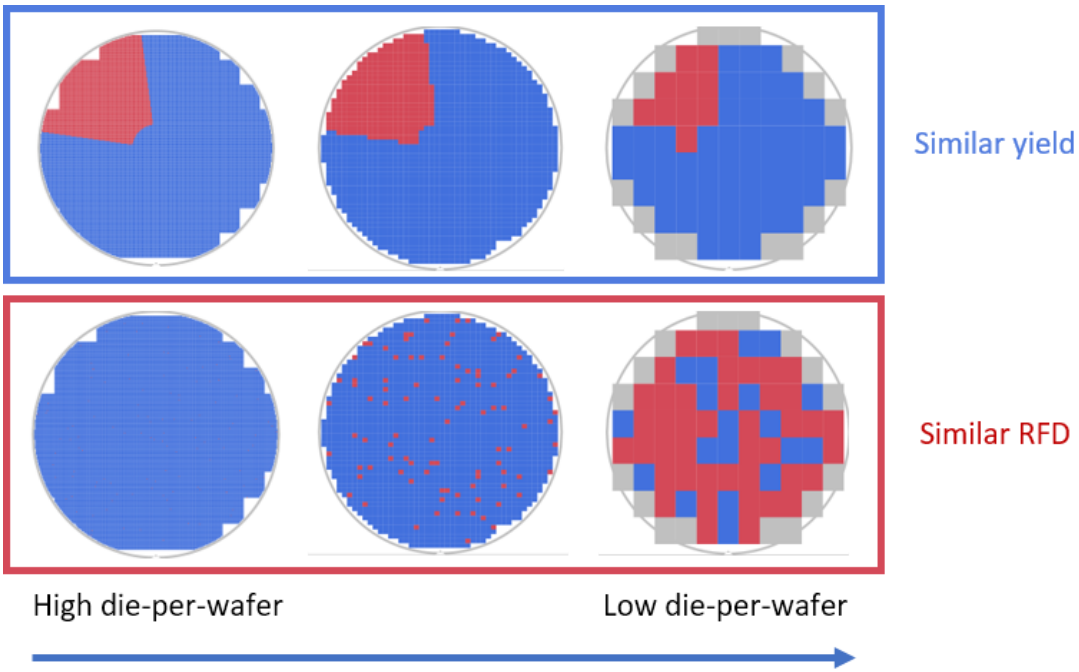
Self-supervised pretraining far outperforms fully-supervised training on all of these splits, and the representations are indeed better than the projector head outputs. With more labeled data at 20%, the gaps close, but in a real-world scenario like semiconductor manufacturing, data is large and unlabeled. This strongly demonstrates the efficacy of self-supervised pretraining on a real-world problem.

5. Conclusions

FastSiam proved to be a very effective self-supervised learning framework for our problem. It produced strong visual representations with no hand-labeled data and a low batch size, allowing pretraining to be completed on consumer-grade hardware within 24 hours. Unfortunately, we had originally planned to do something closer to an image retrieval problem and embedding exploration for our project, but because our tensorboard embedding explorer wasn't able to work, we went with an image classification comparison against a fully-supervised baseline instead. Self-supervised learning allows an embedding/metric space to be created for unlabeled

data, with representations that are as good as the augmentations used in the learning process. For future work, we hope to have even more domain-specific augmentations like die-per-wafer transformations, and a better version of the “random cropping” used by Hue et al. that actually displaces a random portion of a segmented failure shading onto a whole “blank” wafer. While we haven’t investigated this thoroughly, the representations produced by our self-supervised training probably aren’t invariant to die-per-wafer transformations. What that means is that the representation of a high die-per-wafer wafer map with a fixed shading pattern (say, a scratch or an edge-ring shading) would probably be substantially dissimilar to a representation of a low die-per-wafer wafer map with the root cause issue.

Consider the images in the bottom row of this figure. Visually, they appear substantially different, probably to the point of tricking even our model since none of our augmentations account for die-per-wafer variation invariance. While this wasn’t much of a problem for the WM-811K dataset, in a real-world scenario, it might be. Nevertheless, self-supervised learning offers a promising approach to tackling big data problems in semiconductor manufacturing.



6. Works Cited

[1] M.-J. Wu, J.-S. R. Jang, and J.-L. Chen, “Wafer Map Failure Pattern Recognition and Similarity Ranking for Large-Scale Data Sets,” *IEEE Transactions on Semiconductor Manufacturing*, vol. 28, no. 1, pp. 1–12, Feb. 2015, doi: 10.1109/TSM.2014.2364237.

[2] M. Caron *et al.*, “Emerging Properties in Self-Supervised Vision Transformers.” arXiv, May 24, 2021. doi: 10.48550/arXiv.2104.14294.

[3] T. Chen, S. Kornblith, K. Swersky, M. Norouzi, and G. Hinton, “Big Self-Supervised Models are Strong Semi-Supervised Learners.” arXiv, Oct. 25, 2020. doi: 10.48550/arXiv.2006.10029.

[4] D. Pototzky, A. Sultan, and L. Schmidt-Thieme, “FastSiam: Resource-Efficient Self-supervised Learning on a Single GPU,” in *Pattern Recognition*, Cham, 2022, pp. 53–67. doi: 10.1007/978-3-031-16788-1_4.

[5] X. Chen and K. He, “Exploring Simple Siamese Representation Learning.” arXiv, Nov. 20, 2020. doi: 10.48550/arXiv.2011.10566.

[6] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, “A Simple Framework for Contrastive Learning of Visual Representations.” arXiv, Jun. 30, 2020. doi: 10.48550/arXiv.2002.05709.

[7] X. Chen, H. Fan, R. Girshick, and K. He, “Improved Baselines with Momentum Contrastive Learning.” arXiv, Mar. 09, 2020. doi: 10.48550/arXiv.2003.04297.

[8] “lightly-ai/lightly: A python library for self-supervised learning on images.” https://github.com/lightly-ai/lightly (accessed Dec. 16, 2022).

[9] *Imagenette*. fast.ai, 2022. Accessed: Dec. 16, 2022. [Online]. Available: https://github.com/fastai/imagenette

[10] J.-B. Grill *et al.*, “Bootstrap your own latent: A new approach to self-supervised Learning.” arXiv, Sep. 10, 2020. doi: 10.48550/arXiv.2006.07733.

[11] H. Hu, C. He, and P. Li, “Semi-supervised Wafer Map Pattern Recognition using Domain-Specific Data Augmentation and Contrastive Learning,” in *2021 IEEE International Test Conference (ITC)*, Oct. 2021, pp. 113–122. doi: 10.1109/ITC50571.2021.00019.

[12] “Benchmarks — lightly 1.2.38 documentation.” https://docs.lightly.ai/self-supervised-learning/getting_started/benchmarks.html (accessed Dec. 16, 2022).