Classifying Speedup of Loop Optimizations

CSC 461 - Final Project Report - Ryan Brooks

github.com/ryanbrooks816/CSC461_FinalProject

wandb.ai/ryanbrooks-uni/MLFinal/overview

Introduction

This research addresses a critical challenge in the domain of high-performance computing and compiler optimization: classifying the performance impact of specific loop optimizations. By understanding whether a program experiences a speedup or slowdown under certain loop transformations, developers can make informed decisions to improve computational performance. This project focuses on enhancing the classification accuracy of these impacts, that is to *classify the speedup/slowdown of compiler optimizations related to loops in source code*.

It first required adopting to an existing codebase as part of a research project. I learned about the existing models, architecture, data representation, embeddings, training, and other challenges. The relevant parts will be more concretely discussed when crucial to the parts I wired on.

Problem Setup

We're working with embeddings of source code–representations of the code in a very high vector space generated by an LLM. We have embeddings from many LLMs, but I worked with Coderank. The LLM is passed the reference and transformed code to create embeddings for.

The code we're working with comes from many different benchmarking programs. Each benchmark has a series of applications. The applications themselves range in size and task. For example, some deal with cache locality, others process communication or model problems dealing with fluid dynamics. We then target a loop in that application. A series of transformations are applied to that loop from 5 possible ones, each with varying parameters, including:

- **Interchange**: Specifies the new order of the loops after interchange. This rearranges nested loops to improve data locality or parallelism.
- **Tiling**: Indicates the sequence of tiles or blocks for iterating over the data. This improves cache reuse.
- **Unrolling**: Specifies the sequence of unrolled iterations. This reduces loop control overhead.
- **Distribution**: Determines the sequence of loop body splitting. This separates independent computations into distinct loops to enable parallel execution or better cache utilization.
- **Unrolljam**: Specifies how the loop nesting is handled during unroll-and-jam. It combines unrolling with fusing parts of the inner loop into the outer loop for better performance.

These each control how the loop is interpreted, such as the order of execution or memory access patterns, and can take arguments that control the loop bounds, iteration setups, or granularity of execution. There are many combinations of transformations that could be applied to any given loop. All of these combinations make up the "loop group" (for that application's loop).

The Dataset

The dataset contains all of these transformed groups, each are data points. The dataset also has the IDs for all the embeddings, information about how the code was optimized, and the final speedup value–how much faster (or slower) the code now performs with the transformations applied. This is the target value., which is bucketized into 4 classes.

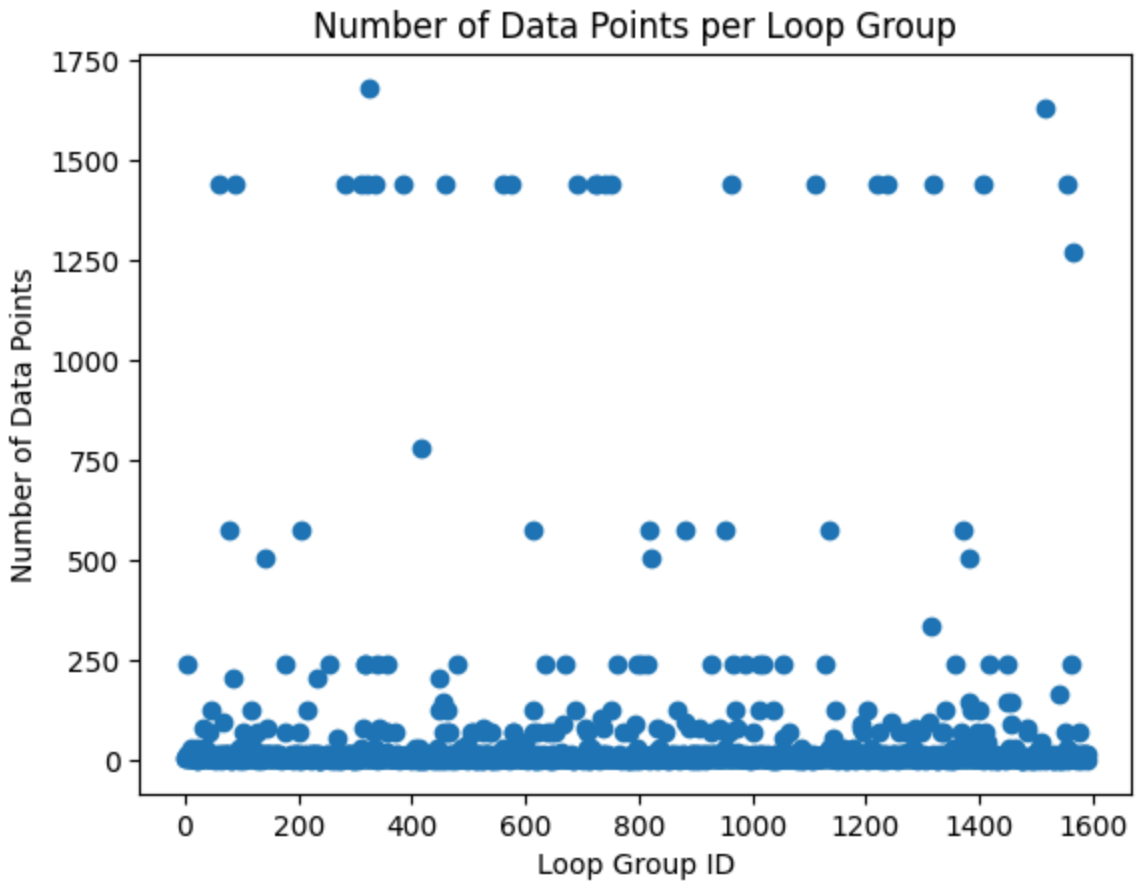
- HSL (< 0.7 slowdown) 40,281 data points, 58.87%
- SL (>= 0.7 and < 1 slowdown) 13.356 data points, 19.52%
- SP (> 1 and < 1.5 speedup) 11,914 data points, 17.41%
- HSP (>= 1.5 speedup) 2,859 data points, 4.18%

The dataset has 68,427 total data points across 1591 loop groups.

Problem 1: Since the dataset is highly skewed toward slow downs, it's easily possible to overfit these data points.

Problem 2: The sizes of the benchmark applications and loop groups have a very high variance—some loop groups and applications dominate the dataset.

For example, the NPB benchmark I'll be using later, it had applications with over 10,000 samples (across over 50 loop groups) and others had only 400 samples, or even just 30 and containing loops from only 2 groups. This presents problems that would account to the biggest hindrance of improving the model accuracy, the goal of this project.



The above chart shows how imbalanced the loop groups are.

- Max number of data points per loop group: 1680
- Min number of data points per loop group: 1
- Median number of data points per loop group: 4.0
- Mean number of data points per loop group: 43.009
- Mode number of data points per loop group: 4
- Standard deviation of the number of data points per loop group: 188.422

The Model

We're using a standard MLP containing a series of linear layers then a normalization layer (either batch, layer, or none), a ReLU activation function, and a dropout layer at a variable percentage. The first layer

always has an input size of 768 based on the dimensionality of the embeddings I was working with and output size of whatever is defined for the latent embedding dimension (usually 256). The output layer always of course has 4 classes for the different speedups.

There's also a data loader responsible for filtering the loop groups and performing the train-test-split. Embeddings are retrieved based on the selected loop groups. Employs a stratified split to help combat the variance issues.

The main idea of stratification is to ensure that subgroups of data are represented proportionally in training, validation and test splits. We'll see later how certain loop groups dominate the dataset. There's 3 different stratification methods (or going random), each aiming to help balance the different loop groups. The first metho, binary stratification, marks a loop group as good or bad depending if a loop with a speedup > 1 is present; it's only concerned that in the train-test-split good and bad groups are both represented. The next stratification method is clustered, which uses K-means clustering to take the 4 percentiles of the speedup and clusters them into groups 50 times. The 4 resulting groups are used to stratify the train-test-split. Finally, majority stratification, where the dominant class in each loop group determines its label.

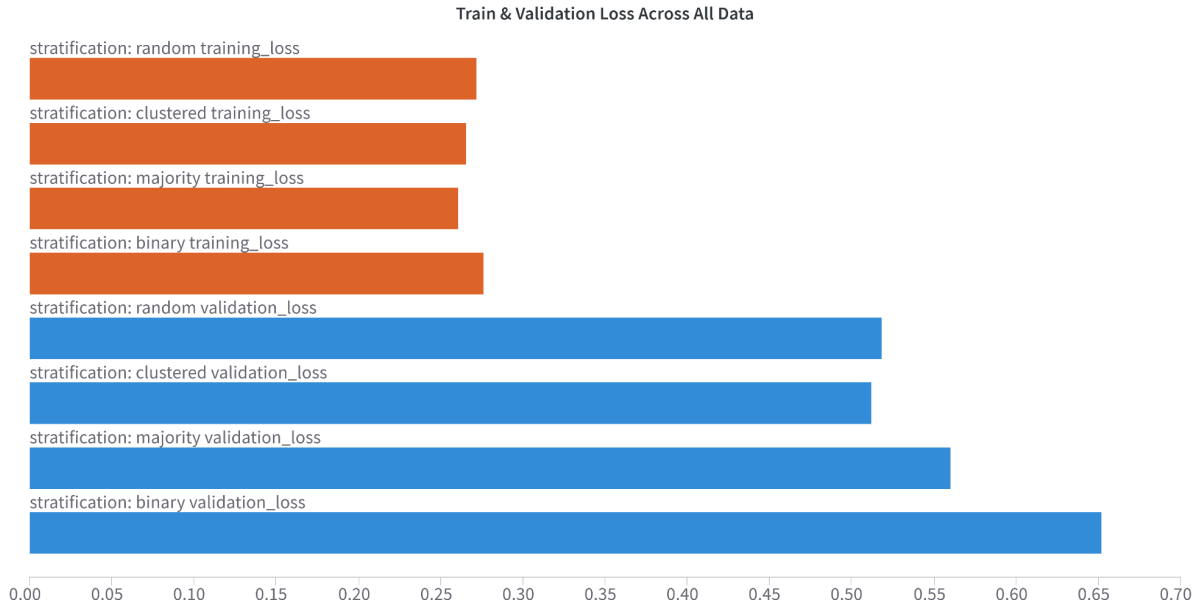
There's also methods for handling training and evaluation. I would not need to modify these. My main contributions to the codebase involved expanding on the my-models notebook, where I'd perform all my work through various iterations. You'll find all the code on the GitHub repository in the my-models file as well as the my-models file used for past experiments. The `past_experiments` folder contains previous visions of the notebook for my own reference. The most updated code is in the main my-models notebook.

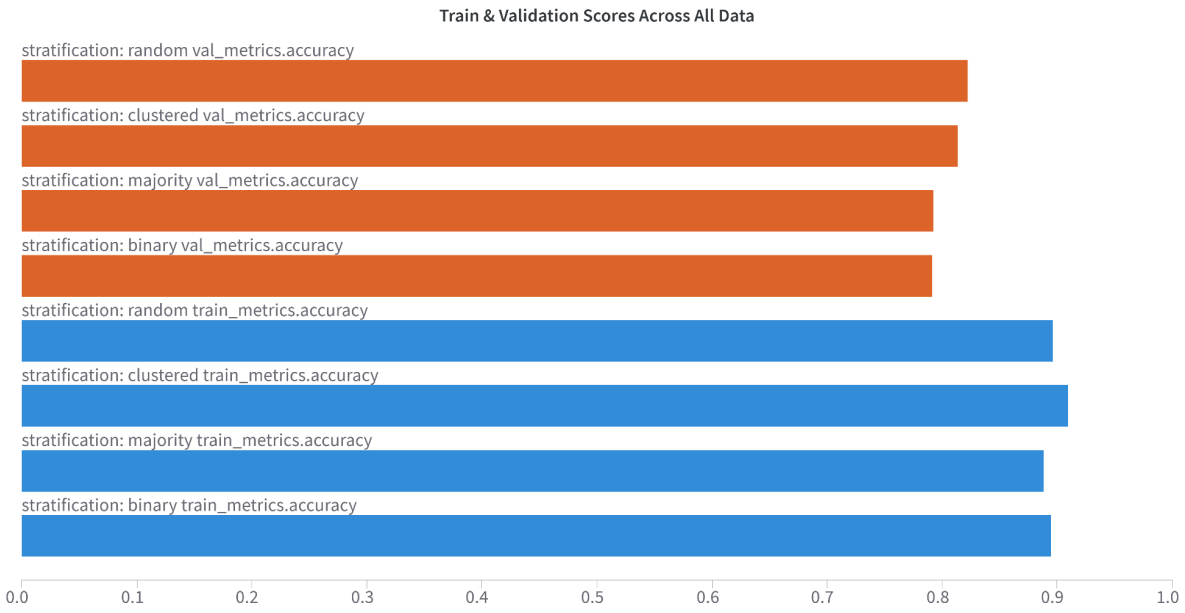
I worked on creating my own class to conduct experiments. It would use the existing functions to handle initializing the model, training, evaluation, and model configuration. The class would also handle Weights & Biases integration and running sweeps.

I would perform a series of tasks to study model performance on the validation set under different conditions and database splits with the end goal being to monitor how different subsets of data affect the model's ability to fit to the training data, generalize to validation data, and variance received.

Initial Testing

To start, I ran some tests with the model options already in place without any filtering by benchmark or application. I tried each form of stratification. My instructions were to focus on random and majority primarily, but I also ran 4 runs of binary and clustered to get an idea of how each rank. Below are my results from the 16 runs.





We can see that the accuracy of training was between 88% and 90% depending on stratification and the validation was between 82% and 79%. The best performing runs on validation happen to be random, with clustered not too far behind.

My hyperparameter configuration used:

- Batch size: 2048
- Dropout: 0.2
- Emb latent dim: 256
- Learning rate: 0.001
- Number of hidden layers: 3
- Normalization: Batch
- Weight decay: 0.001

All 4 classes were represented in each based on the unique target values. This could not be true if a subset of the data used in a run doesn't have loop groups that don't represent the high speed up class for example. All runs were an 80-20 split.

For reference, the stratified distribution of loop groups for each of these runs is the following:

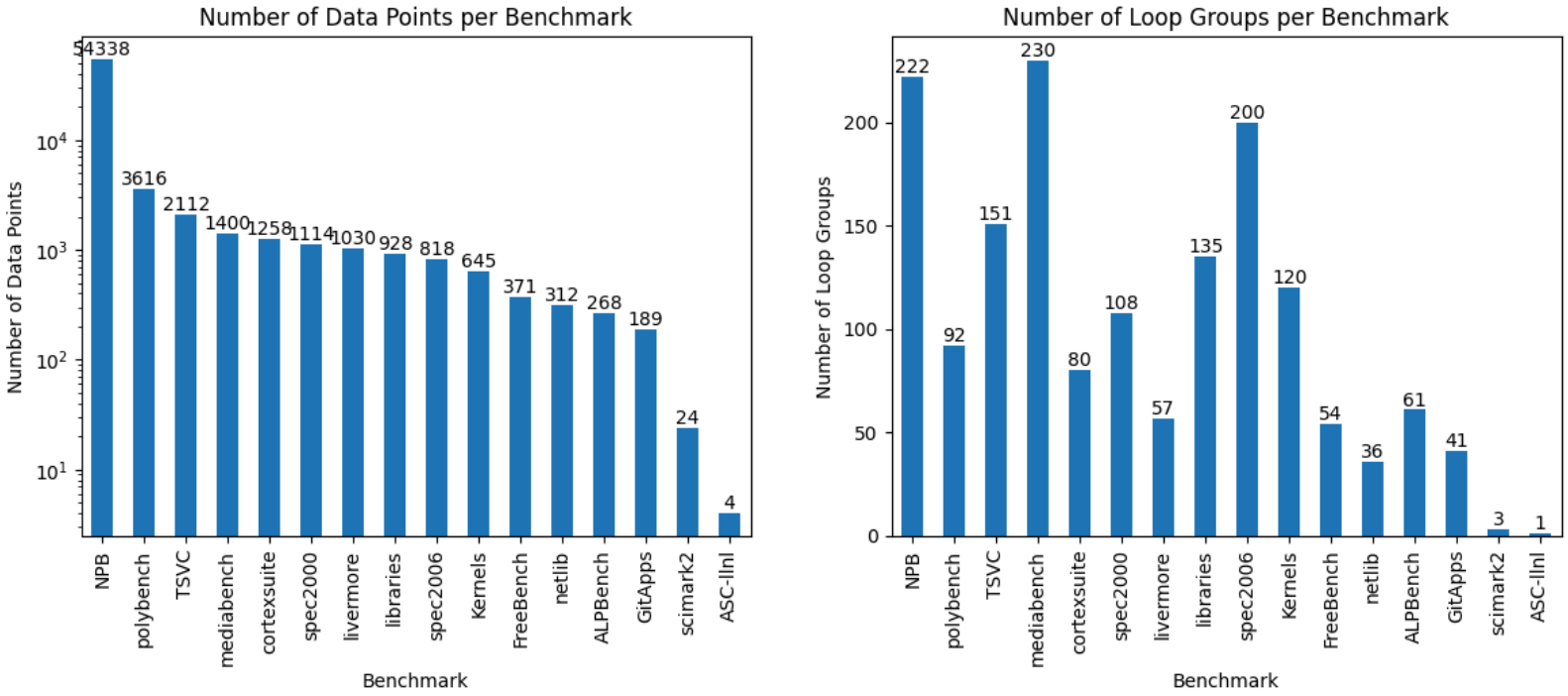
	Total	HSP	SP	SL	HSL
Majority*	1587	5	525	595	416
Clustered	1587	9	143	382	1053
Binary**	1587	1106 Good		481 Bad	

* determined by the largest class in the loop group.
** determined if the loop group has at least 1 loop with a speedup > 1.

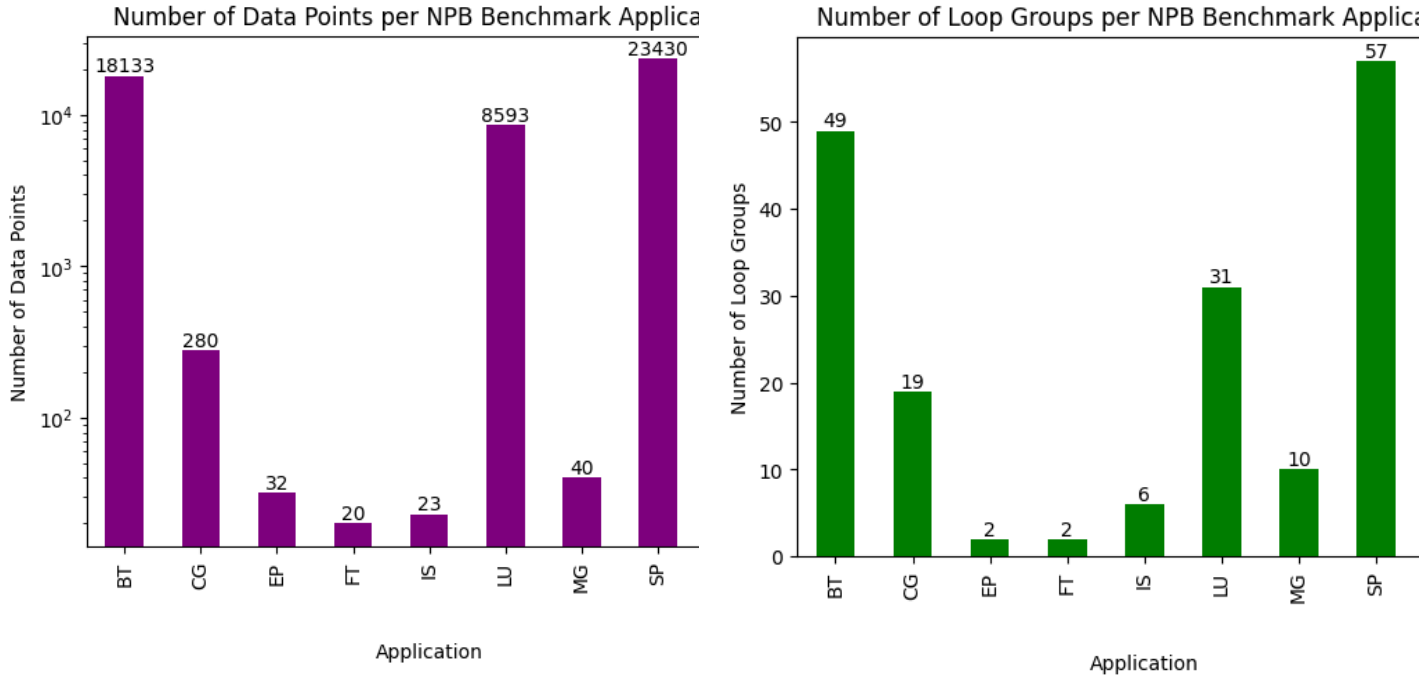
From here, given the dominant nature of certain applications in the dataset, one of my first goals would be to narrow the amount of data we're working with—both in benchmark and application levels. My goals being to observe how the models behave with the goals mentioned previously.

LOOCV

To narrow down the dataset to help stabilize it, I selected the largest benchmark, NAS Parallel Benchmark (NPB).



Figures 4 & 5



Figures 6 & 7

NPB has data for 9 applications. 'BT', 'LU', 'UA-SNU', 'CG', 'SP', 'MG', 'FT', 'IS', 'EP'. NPB accounts for 222 / 1591 loop groups and 54,338 / 68,427 data points, so around only 14% of the loop groups but 79.4% of the dataset, which means we expect very large groups. Each application varies in size and purpose. Understanding them was crucial to understand how well the model will generalize to this smaller subset of data.

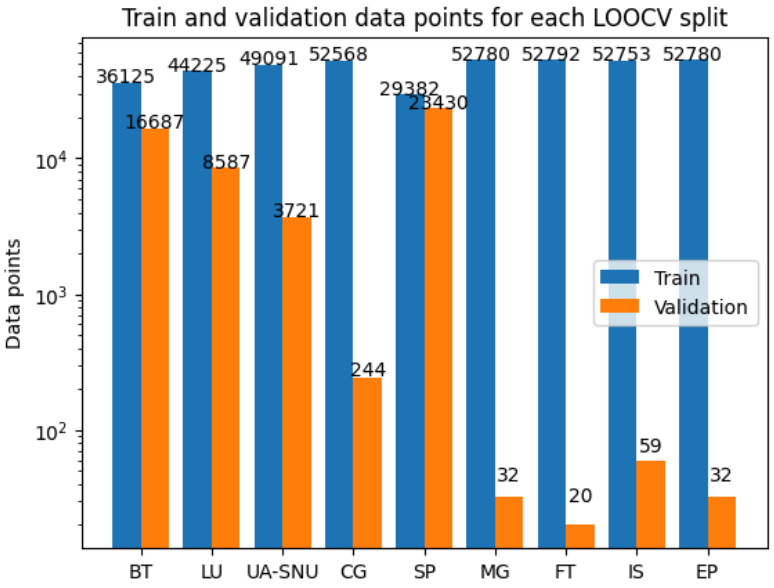
By name, NPB contains benchmarks regarding different parallel computing experiments. Here's a brief overview of what each application does and its purpose.

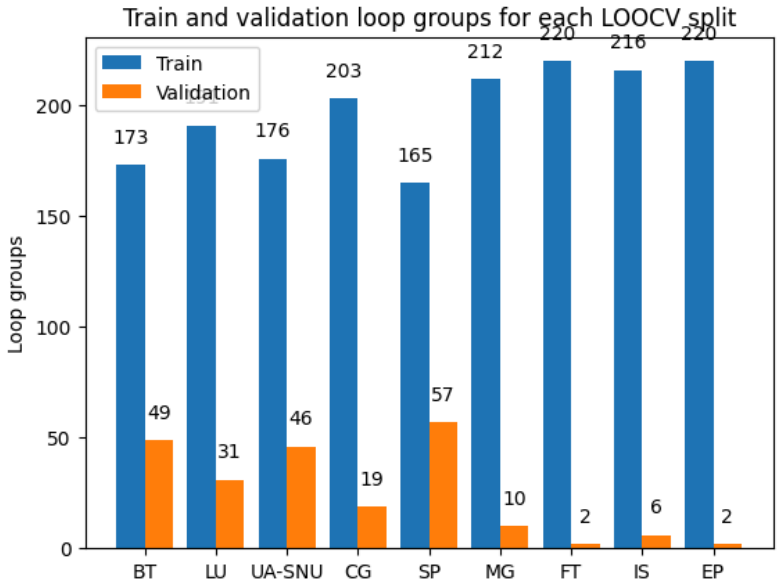
- IS (Integer Sort): Sorts large sets of integers using bucket sort, stressing non-contiguous memory patterns. Targets: Cache performance and irregular memory access optimization.
- EP (Embarrassingly Parallel): Evaluates computational performance with minimal communication overhead by generating random numbers and computes their statistical properties. Targets: Independent, compute-heavy tasks with high parallelization potential.
- CG (Conjugate Gradient): What It Does: Solves sparse linear systems iteratively using sparse matrix-vector multiplications. Targets: Sparse data handling and irregular scientific computations.
- MG (Multi-Grid): Solves partial differential equations using the multigrid method, requiring both long- and short-range communication. Target: Multi-scale problems and memory-intensive applications and hierarchical communication patterns.
- FT (Fast Fourier Transform): Performs 3D FFTs with extensive all-to-all communication across processes. Targets: Global communication and data-intensive high-performance computing.
- BT (Block Tridiagonal Solver): Solves block tridiagonal linear equations for fluid flow simulations, with sequential block dependencies. Targets: Structured computation and memory-access optimization with moderate inter-process communication.
- SP (Scalar Penta-diagonal Solver): Solves scalar penta-diagonal systems, dominated by floating-point operations. Targets: Computation-heavy tasks with predictable memory access.

- LU (Lower-Upper Gauss-Seidel Solver): Solves linear systems using iterative LU decomposition with alternating computational and communication phases. Targets: Structured grid-based simulations with communication in iterative workloads.
- UA (Unstructured Adaptive Mesh): Simulates adaptive mesh refinement for unstructured grids, involving dynamic load balancing and irregular memory patterns. Target Area: Unstructured computations and dynamic problem-solving.

While I was not familiar with a lot of these techniques, I got the general idea how the applications differed from each other, which helped me gain domain knowledge and make certain predictions and explain why certain applications might not generalize well with each other (or future benchmarks I research),

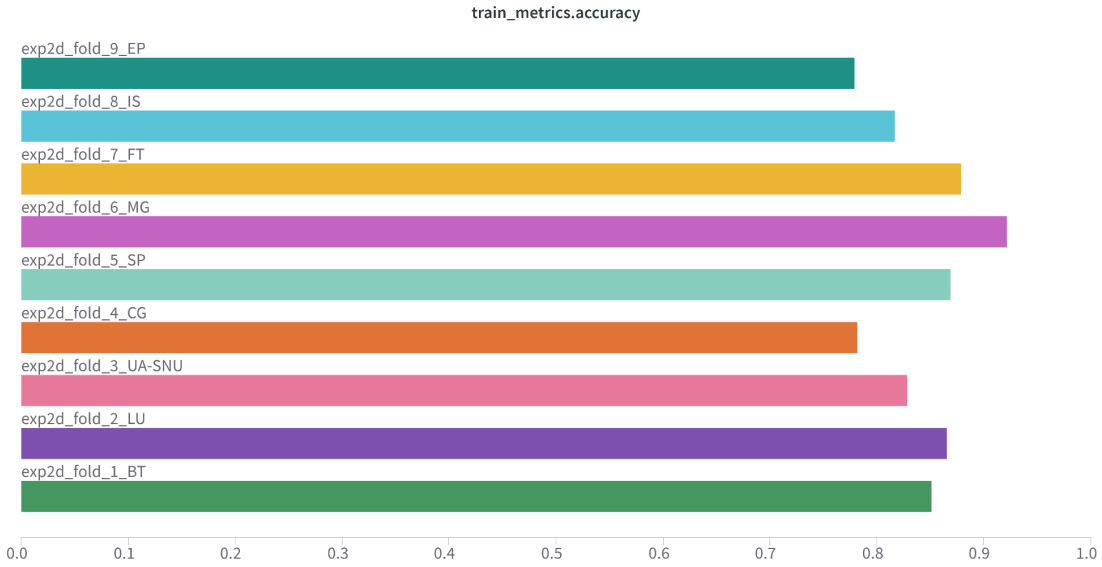
Now that we understand the makeup of the NPB benchmark and the nature of each application, we can begin to understand the results of the LOOCV. It was implemented by creating a new data loader function in the data.py file responsible for calling a filter function that can be passed any number of filters compatible with tha Pandas data frame. It takes in the benchmark you want to filter for and loops over each application, creating the folds, leaving one application for the validation set. It returns a list of data loaders objects plus the name of the validation application for plotting and debug purposes. Finally, I expanded the functionality of my Experiment Runner class to handle LOOCV testing.

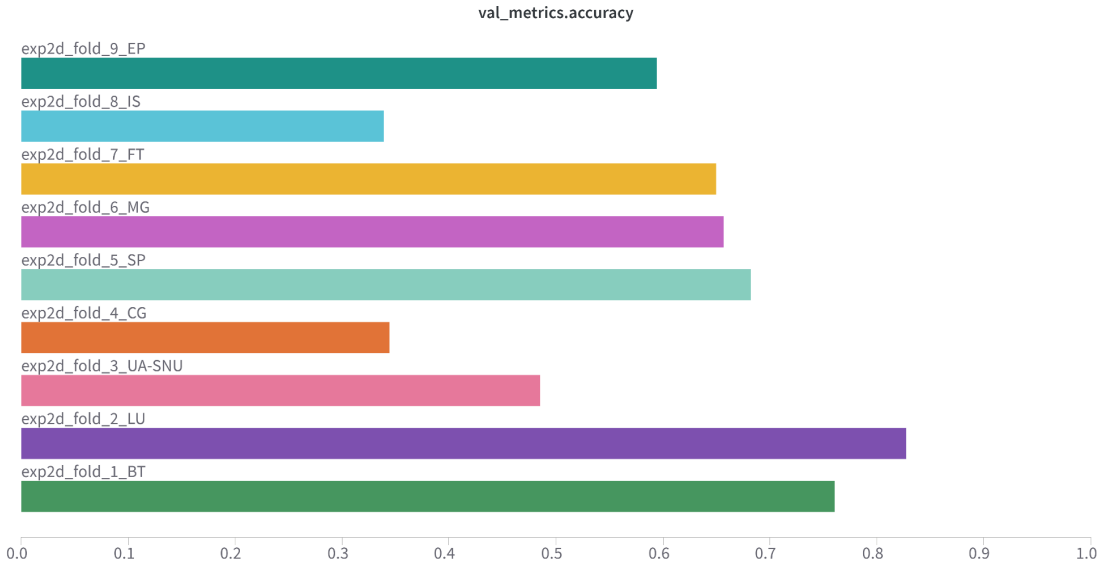




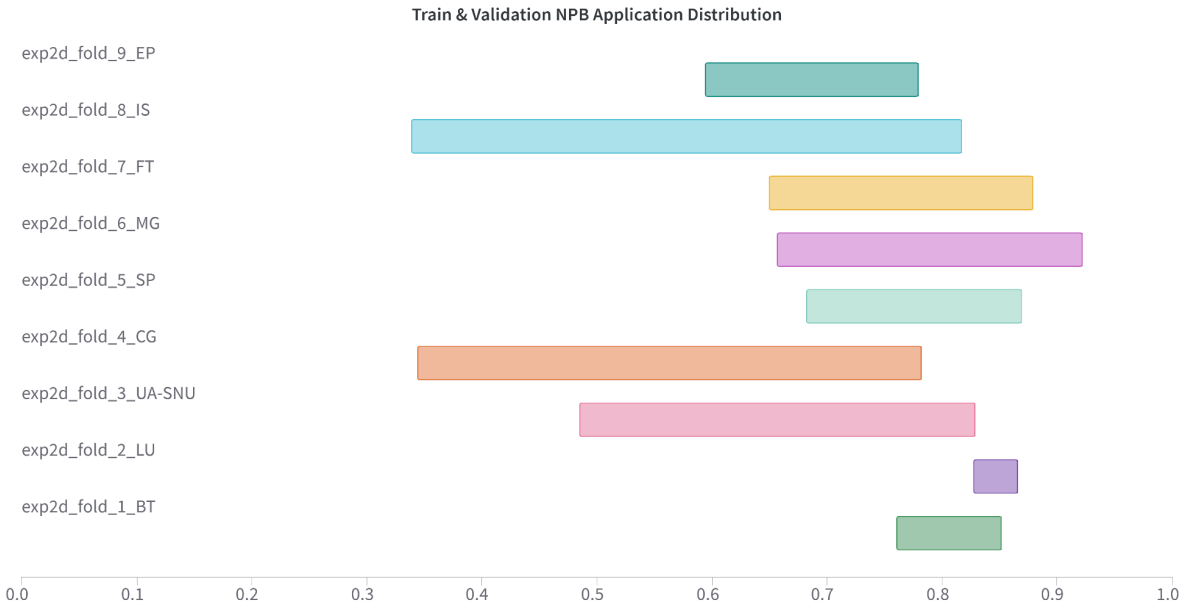
These charts show the distribution of data points and loop groups of each fold. As expected by the dataset metrics, the number of data points in each loop group is significant. Plus, the distribution of loop groups is not that bad, running from 165 to 220. However, a big problem is displayed where the validation set is working with amounts of data, that in most cases, is far less than the training data or desired 80-20 split. All of the folds, naturally, are similar in size in training data, but then you have over 50k training data points and only 20-30 in a few applications.

My predictions at this point were it would do better than on the whole dataset but validation accuracy would suffer a lot. The size difference is the main factor, but we must also consider the different nature of the applications (and as we introduce more benchmarks of different applications) how the validation accuracy will work with different types of applications; for example, if we train on 9 applications for vectorization benchmarks and then test on an application for floating point calculation—we don't expect it to generalize well.





We once again have the results from runs in W&B. We can see that the training accuracy did improve overall. The validation accuracy, however, was very mixed. We did a bit better on the LU application than previous experiments at 85% accuracy. Then we have other applications which did very badly indeed—all ones where the validation sets were far too small.



The above box chart shows the range of the training and validation accuracies. As expected, the larger blocks all correspond with the folds that had the greatest disparity in training and validation data amounts.

Finishing up this experiment, we made modular improvements to understanding how the model works and where the generalization issues arise. It would now make sense to observe the performance of a simpler version of the problem once more.

SP Application Testing

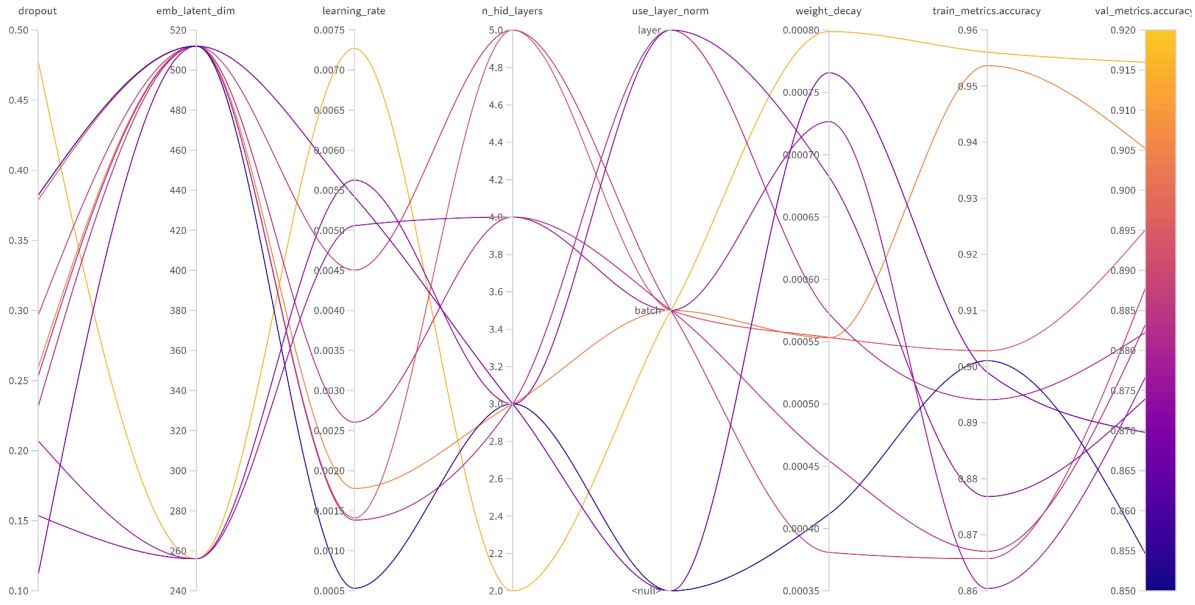
For this final experiment, I chose the largest NPB application, SP; we would hope now that the model is able to fit and generalize quite well. Assuming so, we can tune hyperparameters with a bunch of sweeps through W&B.

I first ran 10 runs of random stratification and majority stratification without tuning any other parameters to start. It allowed me to get a good idea of how well the current model performs.

My approach for selecting hyperparameters consisted of first modifying my Experiment Runner class to handle sweeps in W&B, then I did random sweeps over a range of different hyperparameters, including:

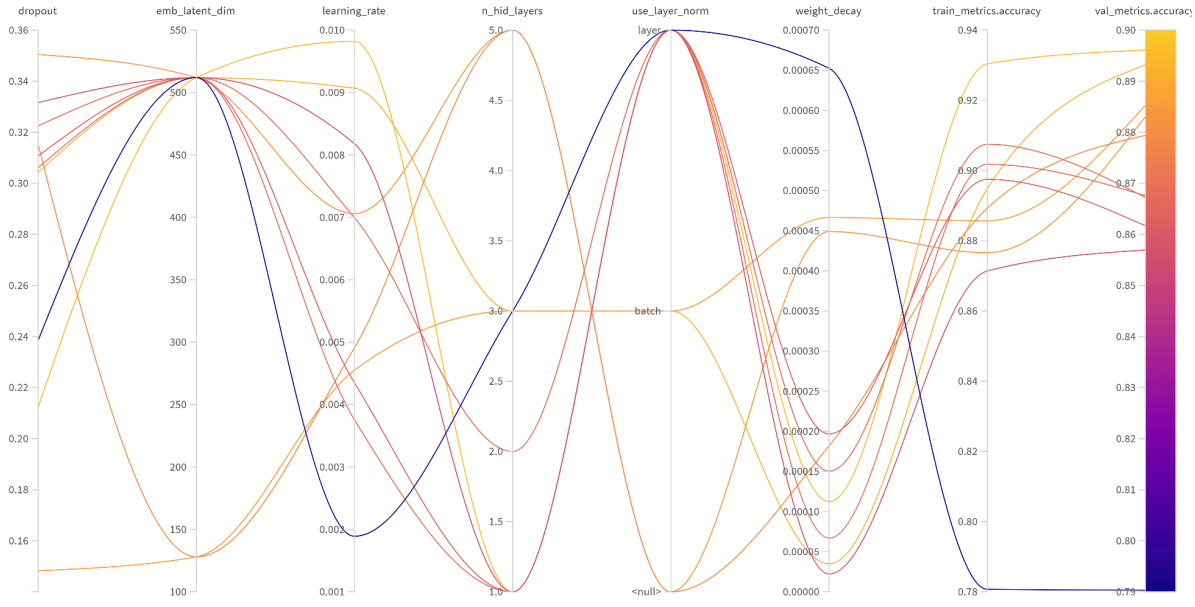
- Number of hidden layers between 1 and 5
- Embedding latent dimension 128, 256, or 512
- Dropout between 0.1 and 0.5
- Learning rate between 1e-5 and 1e-2
- Layer normalization either bach, layer or none
- Weight decay from 1e-5 to 1e-3

The following results are from the majority stratification run. The training accuracy ranged from 86% to 95%, based on the configuration, and the validation accuracy from 85% to 92%, and there was very little overfitting. At most there was about a 4% difference, while most runs only had about 1% difference in accuracy.



The following chart shows how the configuration of hyperparameters led to the accuracy achieved. There is quite a bit of variation. In general, no configuration, however, tanked the model's performance. One thing I did notice is how in general (and in future runs) not using a normalization method in general performed worse than runs with normalization. This makes sense of course because normalization allows faster convergence by reducing sensitivity to noise, regularizes the model to minimize overfitting, and keeps activations within a stale range to avoid the value growing too large and exploding or too small and vanishing.

These are the results from the run with random stratification run. The training accuracy ranged from 79% to 90%, based on the configuration, and the validation accuracy from 79% to 89%. Similar level of overfitting as the previous experiment. From this we can conclude in general the majority stratification performs better than random. Perhaps in the future it might be worthwhile to look into clustering stratification since it also performed well in my first experiment.



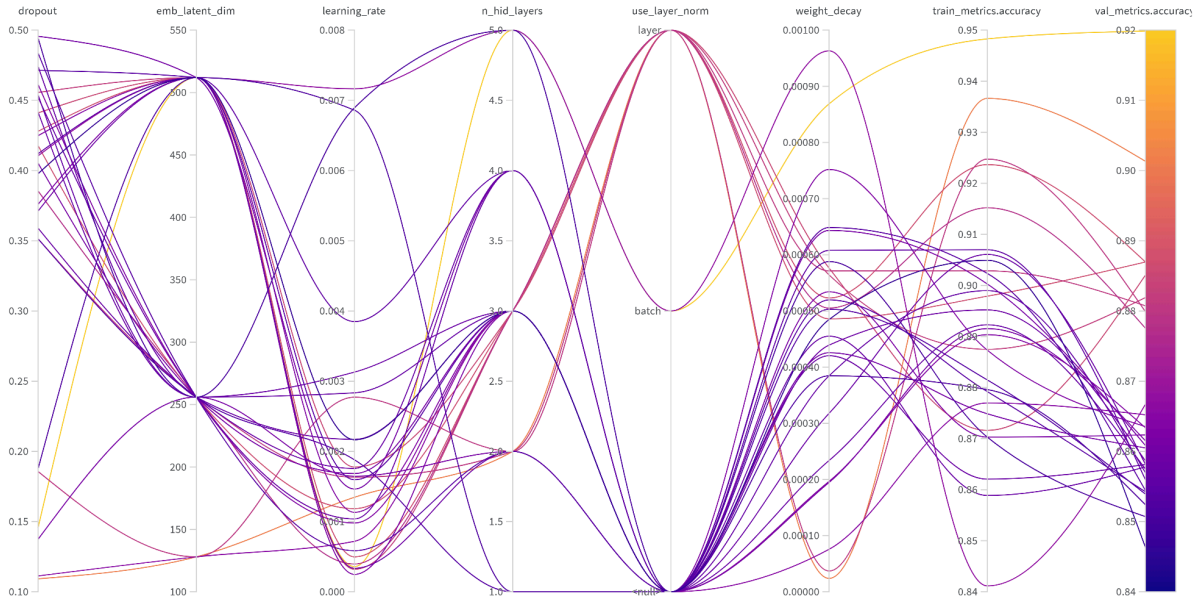
From this experiment, I concluded that in the future I would want to test with a broader range of learning rates and weight decay. From this, it looks like the run with 0.00065 weight decay performed the worst of the 20 runs, but what if there are weight decay values > 0.001 that perform better given that's what the model was set with in the beginning.

Another thing I noticed in this run is that it was reported that weight decay had an importance value of 0.58 and a positive correlation of 0.671, while in the previous run with majority stratification, it also marked 0.45 importance but with a negative correlation of -0.588. These are the parameters that impacted the validation loss the most. As for the different correlations, these were only 10 runs and random values were chosen in the ranges defined above, so it's entirely possible values in the range were a bit too large or small for a majority of these runs, leading to these results.

Lastly, I wanted to run 2 sweeps that were larger than just 10 runs to make sure I wasn't making conclusions without enough runs to support them. The first run had 30 runs. I used majority classification and the same range of hyperparameters as before.

** Note that the scale here starts at 0.7 for size reasons.*

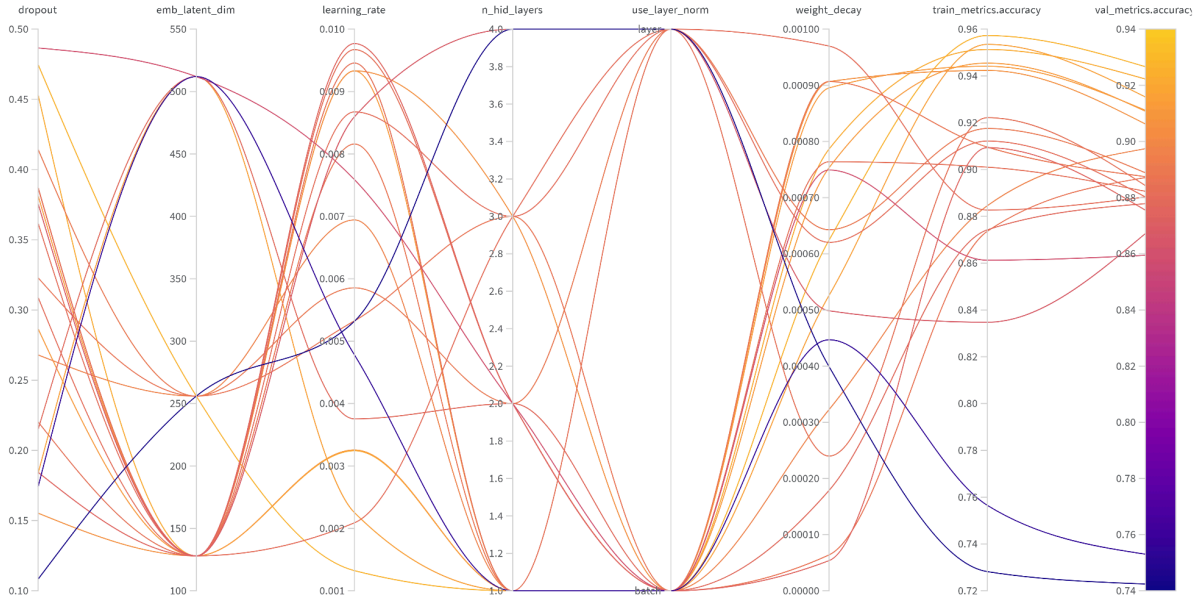
The training accuracy ranged from 85% to 95%, based on the configuration, and the validation accuracy from 85% to 92%. The above plot shows the range the validation scores differed from the training for each run,



From this chart, I could definitely conclude that without normalization runs were performing worse, so I only allowed it to be batch or normalization for the last experiment of 20 runs. This run also posed two possibilities 1) the emb_latent_dim is too high and should stick to 256, 2) the learning rate and weight decay values still need more tweaking.

For my final run, the goal here was to see if removing the possibility of no normalization led to clearer trends in hyperparameters. So far, I only saw minor improvements in accuracy, so I figured my efforts would best be spent broadening the subset of data I was working with once more now that I've explored the results of a stabilized one.

There were 20 runs and this turned out to be my most promising run yet. The training accuracy ranged from 96% to 72% and validation from 93% to 74%. Yes, in some cases the validation sets did perform better than training, though this was only for the low performing runs. We had the best runs, overfitting was about 3%.



Despite the broad range of scores, many more of the runs performed well. Sticking to normalization helped a lot, but I also noticed 4 hidden layers might be too many again. 2 or 3 seems like the best bet. I also saw a lot of runs perform well when given an embedding latent dim of 128. I think it's time to remove the 512 option in favor of 128 or 256.

Next Steps

Given this research is ongoing I was thinking about doing a 3-fold validation on BT, SP, and LU applications given their relative nature and better splits than others in NPB. I also would of course want to build up to working with more benchmarks again while monitoring changes that could be made to

improve to improve the model or dataset itself, such as changing the representation of the embeddings loops with only a few transformations aren't represented well—a very sparse binary representation.

Conclusion

Through this project I was presented with a new codebase that aimed to solve the difficult problem of classifying the speedup of compiler optimizations on loops. It required me to adapt in a short amount of time and start performing my own experiments, understanding my results, and brainstorming new approaches to further understand the dataset and make improvements to the model. The dataset itself was also a challenge, it required adapting in a real-world scenario where the data isn't nice to work with and there isn't a clear solution. I had to try multiple experiments to gradually improve the performance of the model, giving me real world experience in how to solve challenging but practical problems in machine learning. I believe the experience will help me in future research into this topic and my future in machine learning in general.