Master's thesis

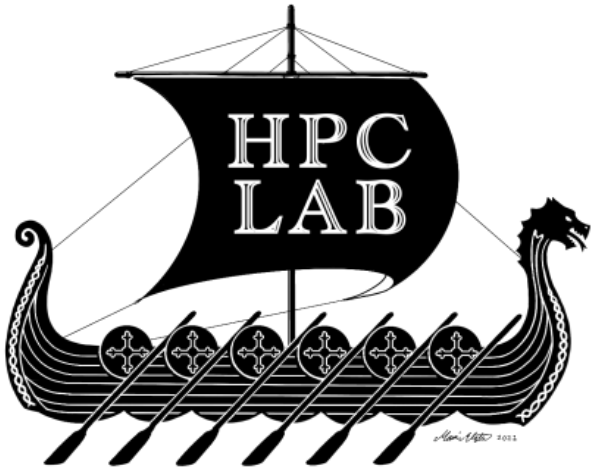
NTNU
Norwegian University of Science and Technology

Lars Bjertnes

Applying Natural-Language-Processing-Based Machine-Learning Techniques to our Large Scale CUDA AutoTuning Dataset

Master's thesis in Computer Science

October 2020



NTNU
Kunnskap for en bedre verden

Lars Bjertnes

Applying Natural-Language-Processing-Based Machine-Learning Techniques to our Large Scale CUDA AutoTuning Dataset

Master's thesis in Computer Science
October 2020

Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Problem description

Machine Learning (ML) is a powerful tool for a variety of analysis-based tasks, but require large datasets to work well. This master thesis will build on the student fall project “CUDA Source Code Dataset”, later reworked and republished at ICA-PAI 2021 as "LS-CAT: A Large-Scale CUDA AutoTuning Dataset" [1]. The LS-CAT project used CUDA codes from the publicly available GIT source code repository. The CUDA source codes selected were parameterized in order to generate a variety of execution-time-based result. With close to 20 000 kernels, the LS-CAT dataset is very well suited for ML-based autotuning tasks. For this this thesis project, the student will use natural language based machine learning, as seen in the earlier related works [2–5].

The main focus of this thesis work will be analyze whether or not an ML-based model can be used to select efficient thread-block sizes to increase kernel performance on our LS-CAT dataset. Additionally, ML attention-mechanisms and the embedding *inst2vec* [3], may also be analyzed with respect to their impact on the ML model’s performance.

Thesis start date: 15. January 2021.
Advisor: Professor Anne C. Elster, IDI
Co-Advisor: Post-Grad Jacob O. Tørring, IDI

Abstract

Autotuning tasks are almost impossible for humans to perform. The abstract relation between hardware parameters and program performance makes setting hardware parameters a far too complex task for any human. Without autotuning, software ends up missing low-level optimizations, resulting in lower performance. Traditionally time-consuming trial and error search methods have been the staple of autotuning. The emergence of machine learning (ML) could diminish these time-consuming searches.

Applying Natural language processing (NLP) based ML methods to source code as a means to perform autotuning-oriented tasks is a growing topic. The earlier projects have, with success, performed a range of different autotuning tasks using multiple source code languages. However, most of the source code data is CPU-oriented, with very little GPU code. Unsatisfied with this, our LS-CAT (Large-Scale CUDA AutoTuning) project used CUDA GPU-based kernels and generated a dataset to perform thread-coarsening. This thesis implements several custom NLP-ML pipelines to evaluate ML-based thread-coarsening using our LS-CAT dataset.

Several model configurations were able to beat both random choice, 0.9400, and the only selecting the largest thread-block (1024), 0.9437. Finally, the best model achieves a score of 0.9483, giving an average performance increase and speedup of 0.49 percent over the largest thread-block.

This project made several discoveries. The implementation of self-attention mechanisms proved beneficial in the learning process by counteracting over-fitting. The choice of underlying methodology is important, with a multi-label method outperforming the rest. Compared to the dataset from [2], our LS-CAT dataset's higher number of thread-coarsening levels gave a false impression of lower performance.

The choice of embedding in earlier works *inst2vec* was unable to parse around half of the CUDA source code LLVM IR tokens, resulting in high data loss. Approaches on how to address this and other ideas for future work are also included in this thesis.

Sammendrag

Autotuning oppgaver er nesten umulige for mennesker å gjennomføre. Den abstrakte relasjonen mellom maskinvare parametere og program ytelse, gjør parameter setting uegnet for hånd. Uten autotuning, mangler programvare low-level optimaliseringer, som resulterer i mindre ytelse. Tid krevende søkemetoder går ofte hånd i hånd med autotuning. Videreføringen av maskin læring (ML) kan minske disse tidskrevende søkeprosessene.

Bruk av naturlig språk prosessering (NLP) basert ML på kildekode, for å gjennomføre autotuning oppgaver er ett voksende emne. Tidligere prosjekter har med suksess utført en rekke ulike autotuning oppgaver med flere typer kildekode språk. Mesteparten av denne kildekoden er relatert til CPU, og lite GPU kode er tilgjengelig. Med vårt LS-CAT prosjekt skapte vi ett datasett bestående av CUDA GPU kode. Denne avhandlingen implementer flere NLP-ML “pipelines” for å evaluere ML-basert «thread-coarsning» på vårt LS-CAT datasett.

Flere modell konfigurasjoner var i stand til å slå både «random choice», 0.940, og kun velge største «thread-block» størrelse (1024), 0.9437. Den beste modellen scoret 0.9483, som gir en gjennomsnittlig ytelse økning på 0.49 prosent over å velge kun den største blokken.

Avhandlingen gjorde flere oppdagelser. Implementeringen av «self-attention» mekanismer virket positivt i læringsprosessen ved å motvirke over-fitting. Valget av underliggende metodologi er viktig, og «multi-label» metoden virket best. Sammenlignet med datasettet fra tidligere forsøk, ga vårt LS-CAT datasetts høyere antall mulige «thread-coarsning» nivåer et falskt inntrykk av lavere ytelse.

Valget av «embedding” I tidligere arbeid «inst2vec» var ute av stand til å tolke rundt halvparten av «CUDA» kilde koden, som resulterte i høyt tap av data. Måter å håndtere dette og andre ideer for fremtidig arbeid er også inkludert i denne avhandlingen.

Acknowledgements

First of all I would like to thank my Supervisor Anne C. Elster and Co-Supervisor Jacob O. Tørring for important feedback along the way. While unfamiliar with some of the technical details of machine-learning, their knowledge of everything else regarding research, writing academic papers, autotuning, and related works, were valuable to say the least.

I would also like to thank my fellow HPC-lab members, Ivar Andreas Helgestad Sandvik, Andreas Hammer, and Maren Wessel-Berg, for giving good feedback on any figure, chart, or graphical element I made or was thinking about making. With special thanks to Andreas for proofreading my thesis.

Lastly, I would like to thank my good friend Erlend Fauchald for being a frequent discussion partner regarding the machine-learning process.

Contents

Figures

Tables

Code Listings

Abbreviations

CPU	=	Central Processing Unit
GPU	=	Graphics Processing Unit
OpenCL	=	Open Compute Language
SIMD	=	Singe Instruction Multiple Data
ALU	=	Arithmetic Logic Unit
BLAS	=	Basic Linear Algebra Subprograms
DSP	=	Digital Signal Processor
CUDA	=	Compute Unified Device Architecture
ML	=	Machine-Learning
NLP	=	Natural language processing
GRU	=	Gated Recurrent Unit
RNN	=	Recurrent Neural Network
PTX	=	Parallel Thread Execution
NVCC	=	NVIDIA CUDA Compiler
IR	=	Intermediate Representation
AST	=	Abstract syntax tree
LLVM	=	Low Level Virtual Machine
LSTM	=	Long Short Term Memory
LR	=	Learning rate
BCE	=	Binary Cross Entropy
MSE	=	Mean Squared Error
LS-CAT	=	Large Scale CUDA AutoTuning
LLVM	=	Low Level Virtual Machine
diff	=	Difference from optimal

Chapter 1

Introduction

As hardware variation and complexity increased, software is increasingly struggling to keep up with the specificity required to have full system utilization. Low hardware utilization created a high gap between actual program performance and theoretical program performance. The cause is a lack of low-level optimizations, which needs hardware taken into account. One solution is having several versions of the same software, to take these details in hardware into account. While creating several different program versions for each system would be possible, the number of systems and knowledge required to make a effective hardware application for all systems made this a challenging task. Instead, the low-level optimizations should be done either by a compiler or an autotuner. The compiler is designed more with compilation speed in mind and not program performance, while the autotuners entire purpose is to increase the end program performance. Traditional autotuners are usually library-based and domain-specific or generative. Some autotuners combine the strength of being both domain-specific and generative. Domain-specific autotuners supply several variations of the same code implementations and create a parameter space representing the variations. Generative autotuners edit source code and change the code using predefined rules. Each potential modification and degree of modification is defined as a parameter space.

There is usually a complex relationship between a specific parameter and the total change in performance. This means that an extensive search through the parameter space is required in autotuning. Searching through all possible legal combinations of parameters is a highly time-consuming process, as for each time search step, the autotuner compiles and executes the program. Then, based on the results, the subsequent search step is performed until an optimal program variation is found. A better alternative would have an autotuner that can set better parameters without searching, compiling, or executing the program.

Machine learning is a method that is well suited in situations where there is an abstract relationship between the data points, and where there is a large enough dataset and enough data processing power [6]. There are different types of machine learning models, but they all conceptually create an internal mathe-

matical model. This model consists of activation, weights, and biases. The weights and biases are adjusted depending on how the model's output value compares to the target value. The model "learns" patterns this way and can be used in a wide range of applications. For example, a machine learning model can skip the intensive searching process needed by autotuners.

Autotuners using source code-based ML methods require a dataset of source codes and program results based on different parameters. Earlier attempts at doing machine-learned autotuning, range in the task performed, source code language, and dataset. Some of these attempts focused on attempting to perform thread-coarsening on an OpenCL dataset created by *end2end-dl* [2]. This dataset is lacking in size and general representation of source code. With that in mind, our LS-CAT project [1] created a CUDA-based dataset. Our LS-CAT dataset consists of CUDA source code kernels and their runtime data. LS-CAT has more thread-block sizes, semi equivalent to thread coarsening level, and significantly more source code samples.

Motivation

The project's primary goal is to apply machine learning and NLP, natural language processing techniques to our LS-CAT, and using ML-model selected thread-block sizes to increase the performance. Additionally, evaluate the impact of both ML-attention-mechanisms and *inst2vec* [3] on the machine-learned model's performance.

Contributions

- **C.1** First implementation of an end-to-end machine learning pipeline, designed for CUDA source code data.
- **C.2** First implementation applying ML NLP techniques to our earlier LS-CAT dataset.
- **C.3** First implementation of self-attention for source code based autotuners
- **C.4** First attempt at using the *inst2vec* embedder with CUDA LLVM IR tokens
- **C.5** First to outperformed both random choice (0.94) and best default option (0.9437) on LS-CAT, with the best model configuration scoring 0.9483.
- **C.6** Findings indicate a generalized learning process, not memorization, implying that CUDA source code has learn-able abstract features.

Outline

The thesis is structured the following way.

- Chapter 2: Presents autotuning and our earlier LS-CAT project.

- Chapter 3: Focuses on the machine learning principles relevant for the thesis, and previous source code based machine learnt autotuners.

- Chapter 4: Describes the entire process of designing the machine learning pipeline, from source code to outputted ML predictions.
- Chapter 5: Results from some key model configurations.
- Chapter 6: Discussion and evaluation of the results.
- Chapter 7: Conclusion drawn from the work in the master thesis with outlines for future work.
- Appendix A: Thesis presented as a poster.
- Appendix B: Our earlier project LS-CAT.
- Appendix C: Parts of the source code used.

Chapter 2

Background and Related Works

To better understand the project, autotuning as a concept and some very relevant papers are introduced in the following sections. Autotuning is hardware-dependent. In this case, the GPU is the hardware part taken into account Section 2.2. The programming language CUDA Section 2.3, can be used to interface with the GPU itself,. Lastly our earlier project LS-CAT is presented, which was created using CUDA kernels.

The Sections 2.1-2.4 are taken from my fall project on LS-CAT, which this thesis is built on.

2.1 Autotuning

Autotuning was a natural response to the increasing diversity in hardware solutions during the 90s. Software companies could no longer feasibly create software designed specifically for all types of hardware. As extensive knowledge of each hardware component would be needed and would require multiple software versions. Making numerous versions would lead to increased project difficulty, and in turn, projects became more time-consuming. Instead, software development neglected the hardware variance, causing varying performance from hardware to hardware. To mitigate this, hardware vendors would, to some extent, create their versions of the software that executed better on their hardware, especially for essential programs such as *BLAS*. The vendor-supplied versions of programs had increased performance, but not all vendors offered this service and would neither optimize all programs. It was not cost-efficient for either the software or hardware manufactures to optimize the code for each combination of software and hardware.

To mitigate the work needed to be done by all parties, software parameters were introduced by the software manufacturers. These parameters can change the way the program runs, the extent to which it uses cache, the order of operations, and more. Each parameter can change the program's performance, accuracy, and run time depending on the hardware. This made it possible to generalize the process of finding a good combination by searching through the parameter space.

However, there was still a pretty high requirement for accurately tuning a program manually, as there would still be a need for understanding the underlying hardware and the effect of each parameter on the run time behavior. Additionally, setting many different parameters, with a high number of possible values for each parameter, results in a time-consuming process. A common way to solve this would be picking some parameters the programmer assumed to work pretty well, testing each of the assumed good combinations and then picking the best. This would still require an experienced programmer and was in no way a guarantee of finding the best parameters.

Instead of using humans for this type of code optimization, the machine would optimize the program by adjusting different parameters. The machine could try out a lot more different parameters by itself compared to a human. It just needed some decent heuristics or sophisticated searching methods to pick suitable parameters efficiently. Before performing autotuning, the machine might also try and find specific hardware information. Hardware information could quickly exclude some parameters. The user might also supply this information. Then the parameters are initialized. After this, the machine compiles, runs the program, and then measures the results. The program keeps a record of that combination of parameter's performance on the hardware. Different varieties are tried by adjusting the parameters, compiling, and running the program until the best one is found. As mentioned above, the method for adjusting parameters can save a lot of time, as just searching through all parameters can be incredibly time-consuming. Some of the earlier autotuning projects *PHiPAC*, that tried to autotune *BLAS*, would, for instance, take several days to complete. Found more clever ways of picking good parameters, and *ATLAS*, one of the first *BLAS* autotuners, performed as well as a vendor-delivered solution, after running for just two hours. These types of classical autotuners usually work well. However, they can have a pretty high run time due to still having to run and compile the program for each parameter adjustment needed to find a good combination.

2.2 GPU

A GPU, graphical processing unit, compared to a CPU, central processing unit, has many more ALUs, or cores responsible for doing simple computations. A modern GPU can have more than ten thousand cores, compared to a modern CPU with around four to eight cores, sometimes more but very rarely above two digits. On the other hand, each core in a GPU has a lower clock speed and less functionality. Making the GPU less suited for tasks with a broad specter of different instructions, a GPU can neither handle interruptions nor other key CPU functionalities. THEREFORE, a GPU is worse than a CPU when it comes to doing complex tasks, such as running an operative system. The advantage of a GPU is that the high core count can quickly process numerous data using identical instructions. A common instruction over large data is called SIMD, single instruction multiple data, and is the key feature of a GPU. Historically this type of computation has been lim-

ited to graphical visualization, hence the name GPU, but modern libraries such as OpenCL and CUDA have made it possible to do other types of tasks on the GPU. Data-intensive tasks that are suited for parallelization can reach a significant increase in performance if they are done on the GPU instead of a CPU. If the problem can be parallelized enough, using a GPU can lead to several hundred times speedup compared to a CPU.

2.3 CUDA

CUDA is NVIDIAs proprietary language that gives the programmer an accessible interface for their GPU lineup. A CUDA function run on the GPU is called a kernel. A kernel is either marked as global if run from the system or device if called from the global kernel. The global kernel needs a block parameter and grid parameter set. The block parameter is a three-dimensional representation of a collection of threads. Each block should be of divisor 32 as a warp is 32 threads large, and a warp executes all thread simultaneously. A block of non 32 divisible would have idle threads. A block can at most run 1024 threads at the same time. Each thread can do one computation at a time. So one CUDA block of size 32x32 can, for instance, do an addition on a 32x32 matrix simultaneously. The 32x32 block seems to be a two-dimensional shape but is represented internally as a 32x32x1 in CUDA, as any dimension not set is set to one. The flexibility in dimensions is predominantly to have more convenient interactions between the threads and data. As data often might be represented as a cube or matrix rather than an array. The optimal number of threads per block is not always 1024, as several smaller blocks would have more unhindered register access, for instance.

However, the grid can fit any number of blocks as long as they are declared but is usually set to be the minimum amount of blocks needed to compute a given input size. To find a grid size given a specific block and input size, divide the input by block size. For instance, an input of 128x128 and a block size of 8x8 would give a grid size of 16x16 blocks. If the input was 127x127 and the block still 8x8 you would need 15.9x15.9 blocks. As this is not possible, the grid would use 16x16 blocks instead.

A critical bottleneck in GPU programming is that the system needs to allocate memory for both the system and the GPU to do data transfers between them. If this transfer takes a longer time than just running the computation on the CPU, there is no point in using the GPU.

2.4 Traditional Autotuning

Modern hardware systems are multilayered and far more complex than they used to be. As a result, compilers are not adequate to accurately make program improvements based on the source code and the hardware system. Instead, autotuners, which range in functionality and problem area, can solve this issue. The

following papers represent some hallmark autotuners that are still important today and introduced some key concepts for later autotuners.

2.4.1 *ATLAS*

ATLAS Auto tuned linear algebra software [7], was one of the first projects to create an autotuned solution for specific software, in this case, *BLAS*. *BLAS*, basic linear algebra subprogram, is a collection of methods for solving matrix-matrix multiplication, matrix-vector multiplication, and vecto- vector multiplication. The matrix-matrix multiplication part is very significant as *BLAS* solved for "blocks" in the matrices and not for cells. This made better use of the cache but requires hardware knowledge for efficient cache usage.

The second version of *ATLAS* optimized the gemm, general matrix-matrix multiplication part of *BLAS*, to run on any hardware that was out at that time. The alternative was creating a hardware-specific solution for each *BLAS* subprogram, which some software vendors did. However, there was in no way an optimized *BLAS* for all types of hardware.

ATLAS aimed to create a general method for finding a good hardware-specific program for the *BLAS* matrix-matrix multiplication. The idea was based on generalizing how the hardware features influenced the performance. Then having a collection of generalized code, the machine could generate a program that efficiently used the hardware by picking the correct parts of the generalized code. Of course, many factors were considered that were influenced by the hardware and would create more parts in the generalized code. Loop unrolling, for instance, could change the way a compiler interacted with the generated code and, at the same time, the amount of loop overhead. In addition, the cache size determined to what extent the matrices could be stored. Other factors are the cache miss rate, the order of floating point operations, the order of looping, and the size of blocks that fit in the matrix to be used.

Some of the values could be supplied by the user or searched for, i.e., the size of L1 cache, but most required running and testing the effects, and this process could take between one to two hours. Nevertheless, the end results were about as good as the vendor-supplied software. So even though the *ATLAS* run time sounds like a lot today, the alternative had a programmer work on creating a specific solution, which would take far longer.

2.4.2 *FFTW3*

FFTW3 [8] performs as well as specialized vendor programs, just like *ATLAS*, and does this automatically, without being tuned to a specific machine. The "planner," as it's called in *FFTW3*, takes a "problem" that is a multi-dimensional loop of multi-dimensional discrete Fourier transforms. The problem is divided into several different parts, called "codelets." Each codelet has different versions of it, each specialized either for a particular case with the type of input or type of hardware. By trying out different combinations of codelets, the optimal combination can be

found,just like with *ATLAS*. This way, the planner can adapt to any given hardware. Unlike *ATLAS* the planner is adapted for a specific input shape and size of matrices, meaning the planner needs to be used multiple times if the system needs discrete Fourier transforms for several matrix shapes. Therefore, the user should decide if time saved after using the planner is worth the planner's execution time.

2.4.3 *OSKI*

Conventional sparse matrix-vector multiply and sparse triangular solvers usually run at 10 percent machine usage, as hardware utilization is low. However, the autotuner *OSKI* [9] provides a library of basic sparse kernels that works as basic building blocks or code selections, similarly to *ATLAS*. *OSKI*s tuning process is given a select machine and matrix, then selecting the data and code structure that increases a user-defined heuristic, often fastest kernel implementation. As the matrix information is needed for the tuning, runtime tuning might therefore be unavoidable. Modern systems include a layered cache structure, this can't be taken into account by i.e. *ATLAS*, but *OSKI* supports multilevel cache tuning. Autotuners can be more time-consuming than the time they save. *OSKI* will only tune for the given matrices, as one tuning can take 40x the time of one regular solver. The number of operations done is also visible to the user to make it easier to decide if the tuning process is worth it. Additionally, *OSKI* stores the matrix patterns so that matrices with the same zero values can use the same *OSKI* routine. There is also the possibility of providing similar patterns, as finding a closely related routine would, in many cases, be faster.

2.4.4 *SPIRAL*

SPIRAL [10] uses a library of DSP functions, and a low-level generator. The DSP functions can be combined to create DSP formulas. While two functions might both be O(n), one might have better cache usage, which leads to a performance disparity. Different functions are combined to create variations of the same formula. The best-performing formula is used in the implementation. After implementing the formula, other low-level optimizations are applied by a generator. As the total amount of possible combinations makes exhaustive search infeasible, *SPIRAL* uses reinforcement learning to speed up the search process. As bonus features, *SPIRAL* supports profiling and user-created functions and transforms. The profiling process makes *SPIRAL* able to use runtime information to increase search efficiency. This type of profiling is usually avoided by compilers, as this would increase compilation time, but since *SPIRAL* is installed once, the profiling cost is negligible.

2.4.5 *Orio*

Compared to the other autotuners that are problem-specific, *Orio* [11] attempts to provide library-based autotuning for general code.

To enable *Orio* tuning, the programmer must annotate the parts of code for tuning. The annotation can be architecture-specific. *Orio* takes these code fragments and generates tuned versions of the same operation. The different versions are then evaluated. Then *Orio* selects the best performing one for production use. Automatic parallelization is also supported using PLuTo. The code generator can create different variations of low-level optimizations, combine various handwritten algorithm optimizations. The user can also add their handwritten algorithm optimizations to *Orio*, and this is the feature that makes *Orio* a general-purpose library-based autotuner. An exhaustive search through all the code variations would be too time-consuming. *Orio* has, therefore, different global search methods and user-definable search restrictions. Then a local search is done. This ensures that the search process is both sufficiently enough and fast. The program tuned by *Orio* outperformed the compiler in all cases. Especially in cases with small problem sizes.

2.5 LS-CAT

Our LS-CAT project [1] had as a goal to increase the amount of GPU source code for machine-learned autotuning. Before LS-CAT, there was little public data of this type available. While earlier works focused on OpenCL GPU code, there were no attempts at creating a CUDA-based dataset. The project used publicly available source code aggregated from GitHub. It reformatted each project into a collection of executable CUDA kernels, which were executed with a range of different thread block sizes and matrix sizes. The CUDA automatic thread block size tool was evaluated but lacked sensitivity to matrix sizes. LS-CAT produced around 19 683 kernels, with 20 thread-block sizes, of which 16 are one dimensional. The increased amount of source code could hopefully make automatic thread coarsening possible.

In the LS-CAT dataset for the T4, the 1024 thread block size was the superior choice and performed on an average of 94.438 percent from the optimal, meaning around a 6 percent speedup can be achieved from always picking the optimal choice. A machine-learned model needs to score higher than 94.438 percent to give any speedup at all.

Chapter 3

Machine Learning and ML-based Autotuners

Machine learning is a complex field consisting of several subsections, each with its specific purpose: transforming data, modeling data, creating internal representations, and analyzing data. The data that the models use is explained in the intermediate representation section. Benchmarking methods are integral to evaluate the model's performance and have their section. The tools used to create most of the machine learning pipeline are presented in the "Python and libraries" section. Lastly, earlier attempts at creating autotuners using source code-based ML methods are presented.

The Section 3.5 was taken from my fall project on LS-CAT, which this thesis is built on.

3.1 Machine Learning

Machine learning consists of an internal mathematical model and a loss function. The internal mathematical model varies in complexity, but most of them can be represented as some combination of weights biases and activation functions. For example, one of the simpler machine learning models is a layered series of linear equations where each equation has its weight and a shared bias.

The model seen in Fig. 3.1, is one of the simplest models possible to create. It has an input vector of size three, a hidden layer of size three, and an output layer of size two. The weights between the vectors are the same as the two common vectors, so the first weight is a three by three matrix, and the second weight is a three by two matrix. This also means that all the computations in this network are computed using matrix multiplications.

Finding the output value by going through the model for the given inputs is called a forward pass. In supervised learning, the output is then compared with the target value for the input value x. This comparison is made using a loss function that tries to find an accurate number for the difference of all output and target

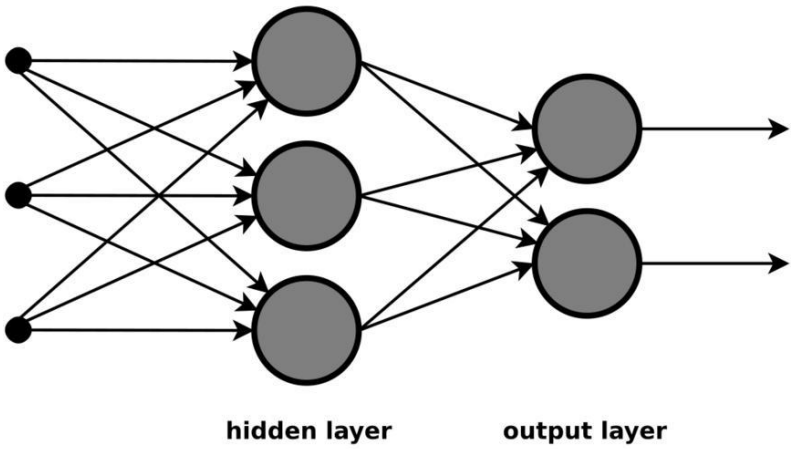


Figure 3.1: Linear layer machine learning model

values at that given training step. Based on the loss score, different weights are adjusted differently based on their gradient, which is calculated per layer using the chain rule. This process is known as backward propagation.

By repeating the process of forward and backward passes, the weights are adjusted to minimize the loss function, making the model output similar values to the target values. Even though the model might score high in this process, there is a chance that the model learned the entire dataset and memorized all the features. This phenomenon is called over-fitting and can be counteracted in several different ways. If the model scores remain low throughout the training phase, the model is under-fitting, either due to lack of complexity, an error, flawed data, or the model is unsuited for that type of work.

Data is the essential factor for doing machine learning. There are some ways to counteract the lack of data, low data quality, or other issues, but having a satisfactory dataset is a critical factor for good results. Inadvertently, this often means that the datasets, not the models themselves, are valuable in machine learning projects.

3.1.1 Activation Functions

Activation functions transform the output of a layer in a machine learning model. Activation functions serve different purposes based on their mathematical properties or computational complexity. One common type of activation function is those who squish the output into a fixed range. The advantage of squishing the output range is that too large of values can cause gradient explosion. Gradient explosion results from too large errors, which creates a gradient too large to adjust the network in any meaningful way. The downside of squishing the output values is the loss of accurate representation of large vs. small values, with some varying solutions to retain a balance between values when down-scaling. The Sigmoid

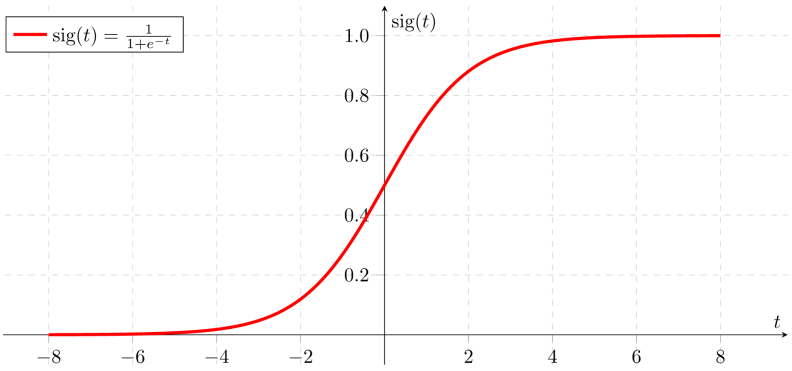


Figure 3.2: Sigmoid activation function

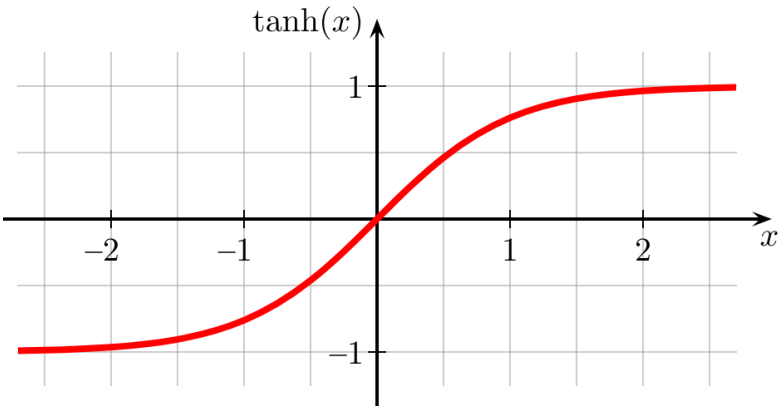


Figure 3.3: Tanh activation function

activation function as seen in Fig. 3.2 uses the Sigmoid function to scale large numbers to the range of zero to one, one issue however is that numbers larger than around six would all be represented equally, and the function is quite steep giving a higher resolution for smaller values.

To mitigate the second effect to some extent the tanh function shown in Fig. 3.3 has a less steep function. The difference between large values would still be lost however.

An alternative to these two functions is the ReLU activation function, which sets negative values to zero and is otherwise an identity function. As a result, around half of the network gets zero activated, which mitigates gradient explosions to some extent while still keeping a better representation of larger weights.

Another important activation function is the Softmax activation function, which scales a vector input into a probability of each element occurring. This activation is commonly used on the last output layer in a classification problem.

$$Softmax = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

$$Softmax(\begin{bmatrix} 3.0 \\ 1.5 \\ 0.5 \\ 0.0 \end{bmatrix}) = \begin{bmatrix} 0.6 \\ 0.3 \\ 0.1 \\ 0.0 \end{bmatrix} \tag{3.1}$$

3.1.2 Learning Tasks

Learning tasks represent the purpose of the machine learning model. Machine learning is used to understand large amounts of data better. The tasks revolve around getting useful aggregated information out of the data. This can include unsupervised tasks such as clustering or supervised tasks like regression or classification, require datasets with labeled data. As this project focuses on labeled data, supervised learning tasks are the most purposeful. Regression is a learning task for data that does not have distinct output values but rather discrete values based on independent variables. For instance, future stock prices are based on previous ones. In comparison, classification is better suited for data with distinct output values, such as finding different animals. The tasks can be separated based on how the output layer is structured and the loss function used.

Classification

Labeled data classification is one of the most common tasks for machine learning. Classification usually has two sub-tasks, multi-target classification and binary classification. Multi-target classification often using cross-entropy loss, and only in a few cases would any other multi-classification loss be used at all. Cross-entropy loss takes the average difference in the probability of the output classes with the target classes.

$$-\sum_{c}^{M} y_c \log(p_c)$$

Where M is the number of classes, p is the vector over predicted probabilities, y is a target binary identifier where the correct class is one and all other zero.

Binary classification is used if the data can be categorized into two classes: anomaly detection. A similar loss function is used binary cross-entropy loss, which has some shortcuts compared to the original loss function.

$$-(y \log(p) + (1-y) \log(1-p))$$

Another version of binary classification is multi-label classification. In this case, samples can belong to more than one class.

Regression

In comparison to the classification, regression is a more complex task as there are several loss functions, which could prove viable in most cases, and there is also the possibility of having multiple time-steps. While the choice of the loss

function is not as apparent as in classification, there are some standard loss types based on the regression requirements, and the starting point of testing regression loss functions should be Mean Squared Error loss. MSE loss is the average of the squared differences between the predicted and actual values. The result is always positive, and a perfect value is zero. The squaring operation gives a higher loss for larger errors, meaning the model is mainly adjusted when making more significant mistakes. This loss function is primarily applicable if the target variable has a Gaussian distribution.

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - p_i)^2$$

Suppose the large error values cause too many adjustments compared to smaller error values. Then, the mean squared logarithmic error can be calculated or MSLE, which adds a log operation before the squaring in the MSE loss.

$$MSLE = \frac{1}{n}\sum_{i}^{n}(\log(y_i + 1) - \log(p_i + 1))^2$$

If the target function has mostly a Gaussian distribution with some higher frequency outliers, the mean absolute error loss can be a better alternative. MAE is the absolute difference between the target and predicted values.

$$MAE = \frac{1}{n}\sum_{i}^{n} \mid y_i - p_i \mid$$

Other than these mainstream regression loss functions, several more specialized loss functions could prove beneficial, but mainly in multi-step regression, which is not relevant for this project.

3.1.3 Embedding

Machine learning models can only take numeric values as input. Therefore, there need to be some transformations in place for any other type of input data to change the data to a numeric representation.

Tokenizing

The simplest type of embedding is a basic numeric representation, where a specific number represents the entirety of an input class. For instance, if the model took farm animals as input, a cow might be represented by a five. However, the issue with this is that relationships between classes are lost and entirely separated, which is an issue if the classes have similarities or dependencies. For instance, the ox might be represented with nine and the chicken eight, making it harder for the model to recognize a stronger connection between five and nine compared to five and eight. In this case, the amount of farm animal classes are most likely low enough that a model could efficiently discover this connection. Still, if instead of

taking farm animals, the model were to take words, the number of classes would be too high for such a simple methodology.

Instead of taking the entire word, one might instead make a numeric representation of each character. However, this would increase the input size drastically, the idea of different words would be lost, and longer words would have more importance as they represented more data than shorter words. Furthermore, this is not really as harsh words can have more semantic value in a sentence than a longer word, making learning harder.

Instead of creating a basic numeric representation of either the words or characters, alternative unsupervised learning methods such as Skip-Gram or chosen bag of words can be used. These methods are designed to create vectors that better represent the words.

Chosen Bag of Words (CBOW)

CBOW uses neighboring words or tokens to find the context and is trained unsupervised by feeding the CBOW model token sequences where one part of the sequence is missing. The unsupervised task is to find the missing sequence. This way, the CBOW model can get some understanding of sentences or sequenced data. After establishing relationships between the tokens, vector representations can be made by feeding the model a token. These vectors are structured such that similar tokens have similar vectors based on context.

Skip-Gram

Skip-gram is a method designed to find the most related words or tokens based on a context. Skip-gram is unsupervised, where a sentence is split and using the original context, the following words are found, which is the other way around in comparison with CBOW. While skip-gram training is more complex and takes more time, it is also associated with better performance and results than CBOW, especially for infrequent data points.

N-GRAMS

While both of these methods can establish contextual relationships, direct semantic relationships can pose a challenge. Words with minor syntactic differences like "word" and "words" are different tokens and used in slightly different semantic circumstances. Instead of representing the words as tokens, can s split the words into their N-Grams. N-Grams are a collection of character combinations. For example, the 2-Gram of "words" would be "wo-or-rd-ds". This way, each token or word can be represented as their unique collection of n-grams. This also means that syntactically similar words have similar N-Grams and is less of an issue for the CBOW or Skip-Gram model.

3.1.4 Recurrent Neural Network (RNN)

Recurrent neural networks are a subsection of machine learning models that capture sequenced temporal information using a different variation of directed graphs. This directed graph-based representation has the unique feature of taking a series of vectors as input of varying length, with each vector being able to express a different time step. Additionally, an RNN can store previous time step information as a secondary input to these directed graphs so that a model can establish time or sequence-based dependencies in the data across time steps.

The main issue with RNNs was the vanishing gradient problem, which occurs when the gradient is too small to impact weight adjustment. The vanishing gradient problem is usually associated with deeper networks, as the chain rule works per layer, and the further away from the output layer, the smaller the gradient becomes. Other vanishing gradient causes are the choice of activation functions, for instance, the Sigmoid activation function. The Long Short Term Memory, LSTM cell, was developed to solve these issues in RNNs.

LSTM

The LSTM cell includes Constant Error Carousel (CEC) units, which handles the vanishing gradient problem. Represented as C_t in Fig. 3.4 the CEC is combined with inputs from the "input" and "forget" gate before being used in the "output" gate function. The "forget" gate and the CEC are combined with a multiplication operation. This serves as a filter to completely ignore values that are regarded as unimportant. As the CEC is not reliant on any other activation function directly, and the term is only multiplied with the "forget" gate, backpropagation of C_t for C_{t1} causes no vanishing gradient issue.

Performance-wise LSTM cells have shown significant performance in a range of different applications, i.e. computer-vision [12], and LSTM cells remain one of the most common modules included in deep learning networks that handle sequenced information.

GRU

The GRU is considered a more simplistic version of the LSTM, where the forget and input gate are replaced by an update gate. This makes both forward and backward pass less time-consuming, and the GRU cell is, therefore, faster than an LSTM cell. The downside is that GRUs have lower accuracy or network performance in most cases compared to LSTM cells, and if training time is not an issue, LSTM is the preferred choice over GRUs.

3.1.5 Attention Mechanisms

LSTMs or other types of RNN techniques can be used to establish the sequential relationship between time steps. However, connections between different parts

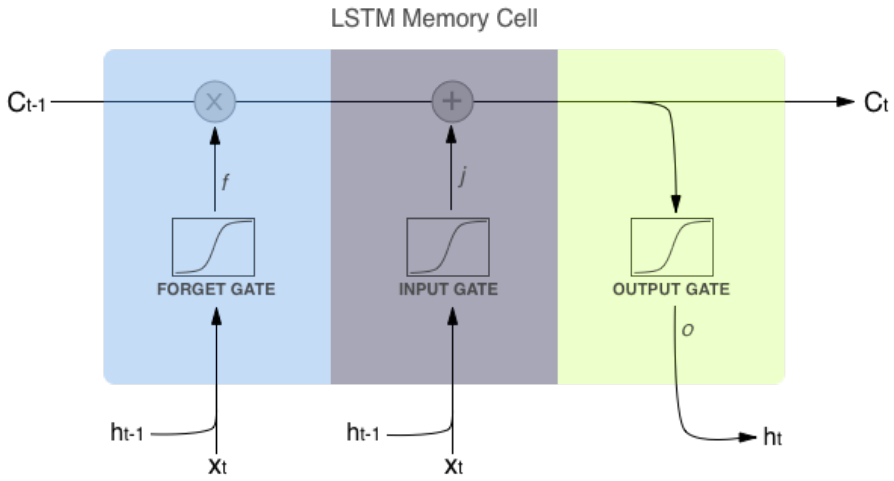


Figure 3.4: Common LSTM cell implementation

of the sequence, or if multiple sequences are used, the relationships between all the sequenced data are less clear. Attention mechanisms aim to establish clearer cross sequence connections and the relative relationships between data points. The calculations look at how one data point influences the other data points and how the rest of the data points influences that single data point. Self-attention is computing the attention score of a single data-stream.

These attention mechanisms are often used in conjuncture with encoder-decoder frameworks. Encoder decoder frameworks consist of an encoder that creates an abstract representation of the input vector and a decoder that decodes the abstraction to output vectors. One of the main advantages of encoder-decoder frameworks is more flexibility for data size transformations. For instance, sequences with unequal lengths can be compared by encoding decoding the input to the target length.

Bahdanau Attention

Bahdanau [13] attention is one of the earlier attempts at creating an attention mechanism. Encoder decoder frameworks usually compress the encoder output, which results in loss of data quality. Bahdanau attention aims to pick the relevant part of the encoder output and keep it, so the relevant parts are not lost to compression. This is done using a "soft search" functionality, which enables the model to pick what part of the source input is most relevant, the corresponding encoder outputs are kept.

Luong Attention

Luong attention [14] is similar to Bahdanau, with most of the differences being smaller mathematical choices. However, the essential difference between these two methods is that Luong's attention reuses the attention from the previous step, as Luong believed that attention history is vital.

Multiheaded Attention

Multiheaded attention [15] uses the "Scaled dot product attention" formula.

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

The formula works by taking the softmax of the dot product between the queries and keys then multiplied with the value, which results in a matrix where each element signifies the influence of other values on that element and vice versa. This mechanism is applied to the data for each attention head, with the output being concatenated. The advantage of multiple attention heads is that they in parallel capture information from differently positioned subspaces. However, one attention head would likely be incapable of this due to the averaging effect applied by the "Scaled dot product attention".

This attention mechanism is suited for computing self-attention.

3.1.6 Optimizers

Backpropagation computation is determined by choice of algorithm. The most "basic" option is gradient descent. Gradient descent computes the loss for the entire training dataset, which is very slow and memory intensive. Newer methods aim to improve upon gradient descent. These algorithms vary in accuracy, speed, and amount of parameters. In some cases, specific optimizers might be an ideal choice. However, there is usually no need to use more than two of the optimizers [16].

3.1.7 SGD With Momentum

Stochastic gradient descent uses random sampling instead of using the entire training set in comparison with gradient descent. This has a few benefits besides being significantly less memory intensive. Sampling makes it possible to update the gradient more frequently, and this should make training faster if the dataset training samples have overlapping or similar data points. Gradient descent converges to the minimum of the parameter space. SGD's fluctuates due to sampling, which causes a "jumping" effect, making it possible to discover a better local minimum potentially. However, this does make convergence to the exact minimum a lot harder, as SGD "overshoots". This can be solved by lowering the learning rate while training. Adam Adaptive Moment Estimation, [17], is an optimizer that finds

adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients, Adam also keeps an exponentially decaying average of past gradients, similar to momentum. However, whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which prefers flat minima in the error surface. Nevertheless, Adam has been shown to works well in practice and compares favorably to other adaptive learning-method algorithms.

3.1.8 Hyperparameters

Hyperparameters are some of the key configuration features that have a different impact on network performance. However, correctly setting hyperparameters can be challenging as there is often not a clear-cut case of what values the different parameters should be set to. Often, the network needs to train to judge the choice of hyperparameters.

Learning Rate (LR)

The learning rate, LR, determines the rate at which the weights are adjusted by the gradient [18]. Using a higher learning rate means that the network "learns" features faster. The downside of using a high learning rate is that smaller key features can be missed. This occurs when the step distance between the current weight and the minima for that weight is smaller than the learning rate, which would cause the backpropagation to over-adjust the weight. If the learning rate remains the same, the weight would be continuously adjusted to miss the minima by over-adjustment for each training cycle. This can manifest as a static loss. On the other side picking a low value for learning rate would lead to a longer learning time. The model would, however, learn diffuse features more efficiently. The downside is the potential issue of getting stuck in local minima. Any weight adjustment would be too small to escape the local minima, making finding the global minima impossible. Therefore, the learning rate hyperparameter is often considered the most important one, and there are some usually good default picks. Still, good learning rate values are very model and dataset dependent.

Batch Size

The batch size determines the number of samples to be used for each training cycle. The advantage of having a larger batch size is added generality. On the other hand, the model becomes less sensitive to outliers, this does increase the memory requirement for training, and the model might be too reliant on the mathematical advantage of using averages for predictions and not finding and learning individual key features [19]. At the same time, a smaller batch size would have a lower memory cost and diminishing the value of the average. The model might not learn the average key features and over-focus on some of the sample's extreme individual key features. This can be observed as high loss oscillation. This means

that using a too large or too small batch size could significantly limit the learning potential of a neural network.

Momentum

As a network reaches the minima training speed slows down, as the error gradient lessens, smaller and smaller "steps" are taken each training cycle. This phenomenon can be fixed by adding momentum to the optimizer. The momentum serves as a directional bias and increases the step length for each training cycle with an additional constant distance.

3.1.9 Changing Settings While Training

While some combinations of optimizers and hyperparameters might prove to be a decent choice in the early stages of network training, there is a case for adjusting these as the training goes on. The main reason for this is that minute features can be more challenging to catch with a fast network using Adam and a high learning rate. By stopping the network and lowering the network learning rate, and changing the optimizer to SGD, the network has a high chance of finding smaller features, boosting overall network performance. In addition, changing the optimizer and learning rate at a certain point would make it possible to combine both speeds from Adam and accuracy from SGD.

3.2 Intermediate Representation

The intermediate representation is used to compiler source code agnostic, which means that the compiler can compile the source code regardless of language as long as the source language can be turned into an intermediate representation. Modern compilers can be imagined as a three-part pipeline, consisting of a front, middle and back end. The transformation from source code to intermediate representation is viewed as the front-end part of the compilation process. This is followed by an optimization process and then a back-end process to turn it into an object or assembly code. The front end is fully responsible for creating accurate translations that retain all the source code information. Depending on the compiler pipeline used, several source codes can be transformed into the exact intermediate representation.

The Clang compiler can transform C-based source code and derivatives such as CUDA to an intermediate representation, seen in Fig. 3.5, which in turn can be used by the LLVM back end [20]. This transformed source code is then outputted on the format of Clang LLVM IR, with some sub-variations, i.e., CUDA LLVM IR.

```
C
| int a = create_a();
| a = a+2;

IR
| %a = call i32 @create_a()
| %a = add i32 %a, 2
```

Figure 3.5: Intermediate representation translation

3.3 Benchmarking

To benchmark and evaluate the performance of the machine learning model is done by looking at several factors. Most apparent is the learning, training, and validation loss, which, as mentioned above, is a measurement of the model's error based on the loss function. The loss is not, however, necessarily directly indicative of real-world performance. There are alternative measures that could be more indicative of the model's efficiency. Accuracy is the measurement of correct guesses compared to wrong guesses and is often used in classification tasks. Accuracy, however, does not consider the usage of thresholds or to what extent the model should be confident for activation or prediction to count. If the predictions have a particular limit, the concepts of precision and recall are beneficial. Both of these concepts are based on the idea of true positives, false positives, false negatives, and true negatives [21] illustrated in Fig. 3.6.

	True condition	False condition
Positive prediction	True positive	False positive Type 1 error
Negative prediction	False negative Type 2 error	True negative

Figure 3.6: Classification errors

Precision can be defined as true positives divided by the sum of true positives and false positives. At the same time, recall is defined by the true positives divided by the sum of true positives and false negatives. Thus, these two measurements can be combined into a precision-recall curve.

Simplified, this curve showcases accuracy based on how confident the classification is. This can be used to great advantage if the cost of an error in relation to correct is known. For example, making errors has a higher cost than the ben-

efit of making a correct guess one could lower recall and secondhand increase the model's performance. However, this would only work if there is a correlation between precision and recall, which is not always the case.

Other than accuracy, precision-recall, and loss, custom special case scoring systems can be used. For example, in our LS-CAT dataset, there is for each kernel matrix combination an optimal choice. Still, subsequent choices can be described as a deviation from the optimal choice. Evaluating the model based on the average amount of variation can be easily compared with random choice and largest block.

3.4 Python and Libraries

Python is a framework that comes with extensive libraries capable of adding several functionalities to the framework. Including GPU efficient machine learning libraries like Tensorflow or Pytorch that are flexible and make modular modeling possible. Tensorflow is a google developed library that gives the user access to several machine learning tools and functionalities. Pytorch is developed by Facebook and is functionality-wise and performance-wise very similar to Tensorflow. Tensorflow has a static computational graph, while Pytorch has a dynamic computational graph. The data loaders in Pytorch are, therefore, better suited to handle variable-length sequences found in RNNs. Data prepossessing can also be handled by python libraries such as Pandas. Pandas give the user efficient and high-level interfaces with CSV or other tabular text data, such as SQL queries or their own boolean data searches. The library *FastText* developed by Facebook is capable of performing skip-gram, CBOW, and creation of n-grams to generate embeds.

3.5 Autotuners Using Source Code Based ML Methods

Variations in source code are proven to have a significant impact on performance due to hardware variation. Therefore, machine learning methods could combine oversight over hardware and source code to use machine learning to make calculations or forecasting. This would avoid having to change and compile then execute source code iteratively. The following papers present some attempts at using source code-based ML techniques for autotuning.

3.5.1 *end2end-dl/deeptune*

The motivation for *end2end-dl* [2] was replacing the manually crafted heuristics, as manually creating features can be very difficult and time-consuming, instead of having the machine-learned model formulate its heuristics based on source code.

The dataset consists of a handful of individual kernels, executed with different hardware systems and either on the GPU or CPU and with differing amounts of thread coarsening. In total, there are 17 programs with 6 separate thread coarsening factors for each hardware platform. The source code is stored as raw source

code but is pre-processed, discarding unnecessary information, then each line code is turned into a sequence of tokens, where the and turned into embedding by the model. The model itself is a combination of LSTM cells, which are good at capturing sequenced information and is often used for learning language features in a machine learning context.

They showed that their solution could outperform machine-learned methods relying on human-designed heuristics and features by 14 percent. For predicting if a kernel should be run on a GPU or CPU, a type of binary classification. Thread coarsening, which *deeptune* treated as a multi-classification problem and their competitor as several binary classifications. Both models scored pretty low when doing thread coarsening on the NVIDIA GPU, most likely due to the small number of training samples.

3.5.2 *NCC*

NCC [3] tries to create a general machine-learned model to do classification and regression on source code of different origin languages. However, NLP does not take into account the nature of code structure, and *NCC* proposes their embedding *inst2vec* to take the specificities of source code into account. The general tasks they want to accomplish are device mapping, algorithm classification, run-time prediction, and thread coarsening.

By making their model use the IR, intermediate representation, the model should work on languages that can create an IR without creating extra language support themselves. This also made it possible to use source code from a lot of different sources. For example, the code embedding *inst2vec* uses a skip-gram method on the IR lines. This is then combined with the contextual flow graph, which represents how different parts of the code are depended on each other. *NCC* uses then a series of RNN, recurrent neural network, for the model itself.

The setup *inst2vec NCC* scored higher than other models for algorithm classification, at around 95 percent accuracy. The other models. They also outperformed *deeptune* on doing both device mapping and thread coarsening, using the same dataset. *inst2vec NCC* showed that a combination of using both the dependencies or flow and the sequenced nature of code could yield better results than focusing on just one aspect.

3.5.3 *CDFG*

Instead of focusing on the sequenced part of source code learning, *CDFG* [4] makes a case for relying more on graph representation. It is pretty similar to *NCC*, but *NCC* while sort of including graph representation did not use graph-based machine learning techniques. The goals of *CDFG* are device mapping and thread coarsening.

The model takes the embedded source code and a code representation of the AST, abstract syntax tree. The AST is a graph representing how all the parts of the code are dependent on each other, *CDFG* also labels all the connections so

that they are not interchanged. Passing this through a series of GRUs, which are similar to LSTMs, a graph neural network does the final calculations.

Compared to the other models, they compared with *NCC* and *deeptune*, *CDFG* performed slightly better on device mapping and thread coarsening. But when the dataset was split into groups based on the origin of the code, *CDFG* significantly outperformed all the other models in device mapping. In contrast, the other models could not beat random selection.

3.5.4 *ProGraML*

Representing source code as graphs proved beneficially in past attempts, *ProGraML* [5] tries to further push this by using three different graphs derived from source code. With the goals being device mapping, thread-coarsening, and algorithm classification.

The three graphs are control flow, data flow, and call flow. The control flow is the order statements are executed based on branching and sequence. The data flow is a data dependency graph. Finally, the call flow is just a graph to connect the origin instruction jump and the destination or the connection between called functions and where they were called. This combined graph representation is made by using IR that has been normalized using *inst2vec*. Then by using graph-based machine learning, a prediction can be made.

ProGraML did not compare itself with *CDFG*, but with both *deeptune* and *NCC*. In conclusion, *ProGraML* achieved the best results when compared with the others in device mapping and algorithm classification.

Chapter 4

Applying NLP-ML Techniques to our LS-CAT Dataset

Any machine learning process consists of several steps. Each step is dependent on the previous and is therefore done in a chronological, often iterative process. The three first sections detail transforming raw data into ML readable data. In the first section, the raw source code data is transformed to an intermediate representation (IR). To transform the IR to numerical data, two different methods *inst2vec* and are tested in the Sections 4.2 and 4.3. The machine learning pipeline itself is outlined in Section 4.4 and consists of several subsections. An intermediate evaluation of *inst2vec* is done in Section 4.5. Lastly the two practical methods binary classification and regression is described in Sections 4.6 and 4.7.

4.1 Source Code to Intermediate Representation

While pure source code could be used more or less as-is for machine learning, with some small conversions to numeric values, using the IR, intermediate representation, as described in Section 3.2 would be far superior. In addition, the intermediate representation has many advantages when generalizing the code. Firstly the variable names are all standardized to simple register references, which reduces the variance between each kernel and should make actual distinctions more easily identifiable. The second advantage of using IR is that the machine-learned model would be source code independent and utilize all source codes that could be transformed to the same IR.

CUDA has its intermediate representation, *PTX*, that is only used internally in the NVCC compiler. With the correct settings, the *PTX* file can be ejected, seen in List 4.1.

Code listing 4.1: *PTX* generation

```
nvcc source -PTX -o path.PTX
```

In the related works section, most of the previous attempts made use of the *inst2vec* pipeline. This pipeline requires Clang LLVM IR, which CUDA can be compiled into. However, compiler linking can be a tedious process, especially when using unfamiliar modules. The Clang CUDA guide was therefore followed closely [22]. This guide, however, was not enough to seamlessly create IR from CUDA, probably due to some version or path issues, and some modifications had to be made. In the end, the following setup found in List 4.2 was used to create all IR from the CUDA source code.

Code listing 4.2: LLVM IR generation

```
clang++-10 path source.cu --cuda-gpu-arch=sm_50 -pthread \
--cuda-path=/usr/local/cuda-10.1 -std=c++11 -fno-exceptions \
-stdlib=libstdc++ -S -emit-llvm -O3
```

Every CUDA source code kernel was transformed into IR with names corresponding to their location using a bash script. By going through each folder and sub-folder containing the kernels, they can then be compiled and named following the format parent folder underscore sub-folder. The location is the unique identifier in the LS-CAT dataset, as each IR has its path as a name. Thus, minimal effort is required to access its run time data, only some minor formatting. Unfortunately, some kernels were unable to be transformed into IR. In total, 19540 out of 20257 were transformed or 96%. This should, therefore, not be an issue. The combined size of all the IR files is around 72 MB.

4.2 The *inst2vec* Pipeline and *NCC*

The *inst2vec* pipeline is used in all the later papers [4] [5], and creates embedding based upon LLVM IR, and is trained using a range of different source codes. Since the data embedding is one of the first steps taken in an NLP context, the efficiency of this *inst2vec* needs to be evaluated as early as possible.

The *NCC* project comes with a pre-trained *inst2vec* model for the embedding process, but it is possible to train the embedder with custom data. Using custom data and training a new *inst2vec* model was not relevant in this project due to two reasons. First, all the data should be conserved and not exposed to the model to avoid any form of preemptive over-fitting that could occur. Second, the training itself is a time-consuming process. This would delay all further development and drastically halt any project progress.

While the *inst2vec* was not trained on the CUDA LLVM IR codes, it was still capable of embedding parts of the code. Around 55 percent of code was not turned into embeds. In comparison, the OpenCL kernels used initially had approximately 12-13 percent of non-embedding code. While the missing data posed a potential issue, machine-learned models can perform with missing data in a lot of cases, and missing data problems are their well-studied sub-field of study in machine learning [23]. It was neither apparent what kind of data was missing or how vital these statements might be. The *NCC* model was adapted to perform classification

on the dataset, as this was regarded as one of the more straightforward tasks to complete and evaluate.

The classifications are split into finding an optimal block and deciding whether or not thread-block size 1024 was a "good" block size given a kernel and matrix size. In addition, our LS-CAT dataset needs to have some transformations applied to create the learning tasks. Data related to the T4 system was used in both cases, as this data should be more stable.

For the optimal block a grouping operation is applied to the data on the kernel and matrix combination. This results in more columns as every thread-block is represented as a column header instead of stored in the row data.

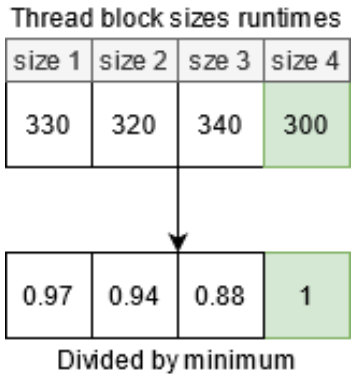


Figure 4.1: Transform to find relative performance

To then find the optimal choice in each combination is picking the lowest runtime number. This can indirectly be used to find the binary classification labels, as the runtime of the 1024 thread-block can be compared with the lowest number. This comparison is made by dividing the smallest number by the 1024 thread-block such that one would indicate that 1024 is the optimal thread-block size. The transformation is seen in Fig. 4.1. Next is deciding at what rate the binary classification labels should be set and labeled either as "good" or "bad" choices.

This "good" factor was set to be above 0.99 percent of the optimal, and the "bad" was set to be lower than 0.925 percent. This leaves a grey zone, but this task is primarily for checking the learnability of the model given this type of problem and dataset, and the amount of bad and good data is very close. The few extra data samples were cut out to create a balanced dataset. The distribution of "good" and "bad" is visualized as a histogram in Fig. 4.2.

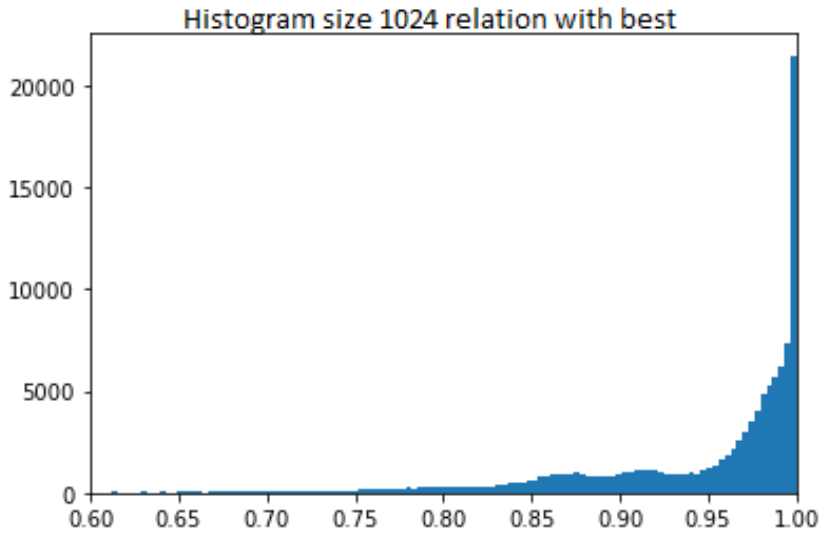


Figure 4.2: Distribution of 1024 thread-block size performance compared to optimal

To make the *NCC* model capable with the runtime data and classification labels from the LS-CAT dataset, some modifications had to be made. First, the data loader had to be replaced with loading both the LS-CAT runtime data and IR of the kernels. Second, the *NCC* network itself was tweaked to work as a binary classifier, with the LS-CAT data as an input.

The final results of the *NCC* model ended up being around 60 percent in binary classification.

4.3 *FastText* Embedding

To evaluate if the *inst2vec* data loss did indeed pose a bigger problem than the potential benefit of adopting the pipeline, independent text embedding had to be tested out.

For this purpose, the tool *FastText* was chosen. *FastText*, developed by Facebook, has some key features that make it stand out. It has the flexibility of creating custom length vectors based on input—the ability to use both the training methods skip-gram and a chosen bag of word. There are also additional settings, such as learning rate, that can be easily modified. This offers a lot of options as different vector representations of the kernels could be tested out. The perhaps most notable feature to *FastText* is that unseen data can also be represented by the vectorization process, which was an issue with *inst2vec*. This new unseen data would vectorize where text with similar semantic and syntactic values turns into similar vectors.

A custom text prepossessing pipeline was created to generate *FastText* training data. This could be reused to create a vector representation of any LLVM IR image.

Each line of code was subjected to the following steps from the List 4.3.

Code listing 4.3: Data pre-processing regex steps.

```
processed_lines = []
for l in lines:
  l=re.sub(r'\(.*?\)', "", l) #Squishes statements between parentheses
  l=re.sub('\(', '␣',l) #Removes parentheses
  l=re.sub('\)', '␣',l) #Removes parentheses
  l=re.sub(',', '␣',l) #Removes commas
  l=re.sub('␣+', '␣',l) #Removes extra white spaces
  processed_lines.append(l)
```

In the original pipeline, some more steps were present. However, such as removing digits, initial testing showed that keeping the digits was a better option.

For the data training file that *FastText* requires to create an embedding model, all the LLVM IR are read and line by line added to a single file. This file is then fed into the *FastText* training function, and the finished trained model is stored for later use.

To create vector embeds from the LLVM IR, a file is opened. Each token is turned into a vector appended to a list, turned into a two-dimensional Numpy array, and stored in a separate sub-folder with a name indicating which LLVM IR it was made from. The training process is significantly faster by storing all the embeds as the generation process is only done once. However, this does come with a potential downside, as the embedding layer is completely "frozen" for the entire training sequence. The embedding is therefore unable to "learn". In this case, the last layers and parameters should be able to take the weight of adjustments needed, as shown in [24].

4.4 ML Pipeline

The new embeds created using *FastText* lacked a model to compare it to the *NCC* model itself, which is mostly just LSTM cells. To give a fair comparison, a bare LSTM network was used. To use the model, a Pytorch ML pipeline was designed. This pipeline is used to load the embedded data, feed the data to a model, use training and evaluation epochs, store results, save trained model weights, and other auxiliary functionalities.

The pipeline has some important parts that can be broken down into a trainer, data-loader, configuration, data, the *FastText* embedder, model, CSV data preprocessor, and utilities. This is set up so that different versions of the same part can be used depending on the task being done. The trainer determines which version of the various components are used, and the trainer being used is chosen by the main file being used.

4.4.1 Embedder

This is the embedder from the Section 4.3. The embedder is only used by the pipeline when there are any changes to the embedding process.

4.4.2 CSV Data Pre-Processing

The CSV data pre-processing is used to give additional depth to the data, annotate or transform it. This is done in Jupyter notebook. Jupyter can store program execution progress and keeps track of any object. This was helpful as some of the dataset transformations were time-consuming and frequent oversight over the data was required to ensure everything was correct. The specific steps taken to create the first anomaly and classification data sets are as follows.

4.4.3 Data

The data that the pipeline uses is organized using the following folder structure as seen in Fig. 4.3.

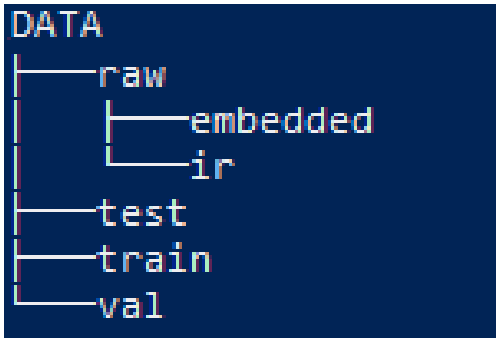


Figure 4.3: Data folder structure

The sub-folder "raw" contains two more folders, an "embedded" and an "ir" folder. The intermediate representations derived from the clang compiler are stored in the "ir" folder. The "embedded" folder holds the embedded versions of the corresponding IR. Additionally, CSV has the entry for all the runtimes given a kernel matrix and block is stored in the "raw" folder.

After the required dataset pre-processing, the runtime dataset is split and divided into the test, train, and validation folders.

The CSV datasets for the learning tasks are similar, but they have each their own set of unique columns, as shown in the Tables 4.1 and 4.2.

The binary classification data contains the runtime for the largest thread block (1024), associated with the number 15 as its block number 15. The path column directly correlates to the kernel being used, the first number is the project number, and the second number is the kernel number. The time is the execution time for that kernel and matrix. The diff_15 number is the difference between this runtime and the optimal for that matrix kernel combination. The flag column is either zero or one, depending on whether the diff_15 is lower than 0.925 or larger than 0.99.

The regression data contains the combination of kernel matrix and the runtime for each thread-block size; this is a subset of columns as the entire set goes up to 15. The label column indicates the optimal thread block for any given matrix and

	path	matrix_id	time_15	diff_15	flag
53622	step2/9345/4/main.cu	3	2998.956667	0.881539	1
38493	step2/553/10/main.cu	5	2665.463333	0.965539	1
61731	step2/121/124/main.cu	4	2634.127500	0.991567	0
48612	step2/8184/22/main.cu	3	3042.326667	0.855970	1
47092	step2/787/0/main.cu	4	2785.083333	0.926795	1

Table 4.1: Binary classification CSV data

	path	matrix_id	time_0	time_1	time_2	time_3	time_4	lable
48377	step2/2052/1/main.cu	1	3241.193333	3211.866667	3248.693333	3236.333333	3248.683333	8
90566	step2/7633/227/main.cu	1	2698.206667	2546.920000	2575.806667	2561.413333	3844.313333	11
99255	step2/8562/13/main.cu	5	2638.133333	2706.956667	2856.470000	2990.040000	3281.983333	0
81988	step2/6427/4/main.cu	5	2953.103333	2928.710000	3453.006667	2686.743333	2702.713333	3

Table 4.2: Regression CSV data subset

kernel.

4.4.4 Configuration

The configuration file stores key hyperparameters, detailed in the Section 3.1.8. The embedder settings and dataset information are also stored in the configuration. As *Jupyter* notebook scripts handle the dataset modifications, the only viable dataset setting is selecting which dataset to use. Whether or not to use the embedder to recreate the data, the embedder input/output dimension can be changed in the embedder part of the configuration file. All normal hyperparameters can be set in the trainer configuration part, including the number of training epochs and if any previously saved weights should be loaded. The modular design makes it possible to use more than one configuration file. Still, compared to having multiple model or training files, the added complexity would not be worth just remembering or storing settings information.

4.4.5 Data-Loader

The data-loaders' purpose is to load the saved data created from the embedder and feed it to the trainer conveniently. This is done using a function to create a data loader object, a custom dataset class, and a class to enable potential GPU loading or device loading.

The function to create a data loader takes information from the configuration file, information from the trainer about what part of the dataset should be loaded,

and what device to use. By combining information from the trainer and configuration, the correct dataset CSV file can be loaded. This file is loaded as a pandas data frame and is used to generate a dataset object. This dataset object is then used by the Pytorch "DataLoader" class and the batch size from the configuration. This "DataLoader" object can return batched samples from the dataset, with each batch having the size set by the configuration. As a default, the "DataLoader" returns CPU localized samples, but for greater performance, data samples should be stored and returned on the GPU. The basic "DataLoader" can be encapsulated by another class to enable potential GPU loading. This new class uses moving data samples to the available device and the Python yield generator to create a GPU data loader.

The custom dataset class contains some simple functionalities such as storing the pandas data frame, returning the length, and item retrieval. The characteristics for item retrieval differ based on the task being performed, and this information will be better detailed in Sections 4.6 and 4.7.

4.4.6 Utilities

There are some shared utility functions for both of the different learning tasks, either relating to loading information onto the correct device or GPU and storing and loading trained weights. In addition, the scoring metric is also a utility function but is keyed to the task being performed and not detailed in this section.

The device loading functions are used primarily by the trainer and data-loader, with the explicit purpose of moving data to the GPU if possible. To determine if a CUDA capable GPU device is available, the Pytorch method "torch.Cuda.is_available" should be used. After doing this, loading CPU data to the GPU is done using the Pytorch ".to" method and setting the device parameter. This method is available to Pytorch tensors, and any data that should be on the GPU requires to be transformed to a Pytorch tensor.

Managing trained weights are done using a saving function and two different loading functions, one for the last training epoch and one for the current best epoch. As the model is trained and evaluated, the saving function gets called by the trainer with a boolean to determine if it's the current best performing one. If the current model is the best one, an additional copy is stored with the designation "best", then a copy is stored with a name corresponding with the global training step. To clear up storage, any previous non best saves are deleted. A list containing the names of all the saved weights is also updated with the training step taken. The function to load the best weights uses the static name-space, and the function to load the last weight uses the list as mentioned above to find the latest saved weights.

4.4.7 Trainer

The trainer uses the data loader's data and uses a target function and the configuration hyperparameters to perform machine learning training and evaluation.

This is done using a training class with some helper functions for training, testing, evaluation, loading, saving, and initializing data.

The data initializing and loading is done when creating the trainer object. The configuration settings and a name for keeping track of the setup are taken as input parameters first as they are required for later steps. The Pytorch optimizer is created using either SGD or Adam and the configuration. The loss criterion is designated, which for binary classification is BCEWithLogitsLoss. A model is created using the chosen model file. The training test and validation datasets are loaded. Then based on the configuration, saved weights can be loaded. Several python collections ordered dictionary objects are created to organize and keep track of all the results for any training or evaluation being done. There are ordered dictionaries for training loss, test loss, validation loss, test, and validation accuracy in the basic solution. Additionally, a timer is initialized to keep track of the training speed.

The validation and evaluation are very similar but are called at different intervals during the training cycle and use their respective datasets. First, the model is set to evaluation mode, as this avoids any weight adjustments and need for backward passes, then the relevant performance metric function gets called based on the learning task being done. The result from this performance metric is then stored in the corresponding ordered dictionary. Finally, the model is put back into training mode. Then the console prints the results, epoch and training step count, and the number of batches being calculated per second. The batches per the second number are calculated using the total training step count and total time spent.

The primary function of the trainer class is the training function. It is here that the model performs forward and backward passes calculate training loss, and calls any potential evaluation to be done after set intervals. The trainer class uses a double looping structure to repeat all the needed steps and finishes by storing the ordered dictionaries. This trainer function is illustrated in Fig. 4.4.

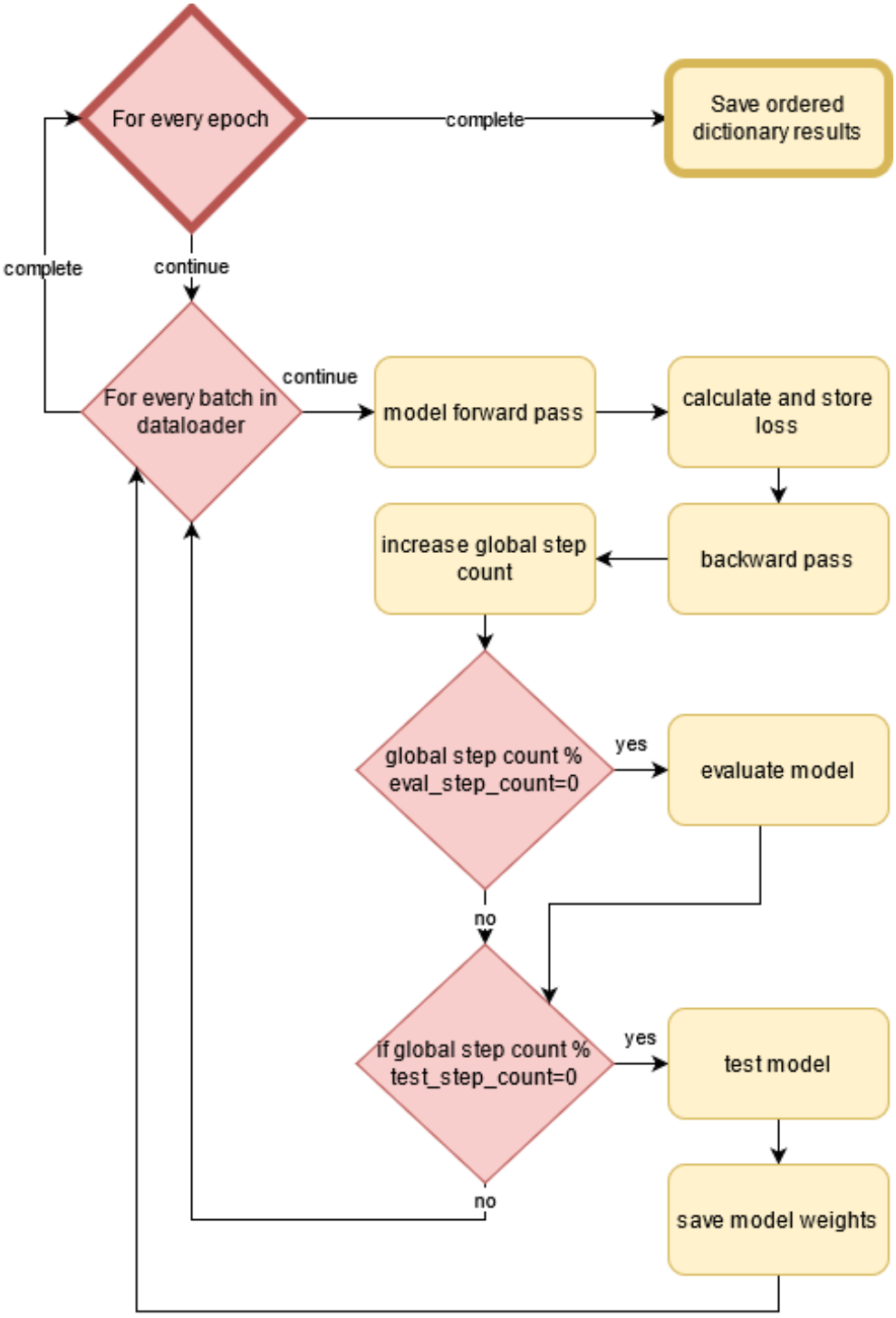


Figure 4.4: The entirety of the training process found in the trainer class

4.4.8 Model

The *NCC* model, which was tested out previously, is LSTM based, and this new Pytorch model should also be based on LSTM modules as this would make a better comparison between the embedding techniques instead of the later parts of the model.

The models are based on the standard model classes used in Pytorch. This means that there is an initialization function and a forward pass function. The initialize function is used to declare any variables, Pytorch machine learning structures, activation functions, and weight initializing. The basic network requires only the Pytorch LSTM module, Linear layers, activation function, and embedding layers.

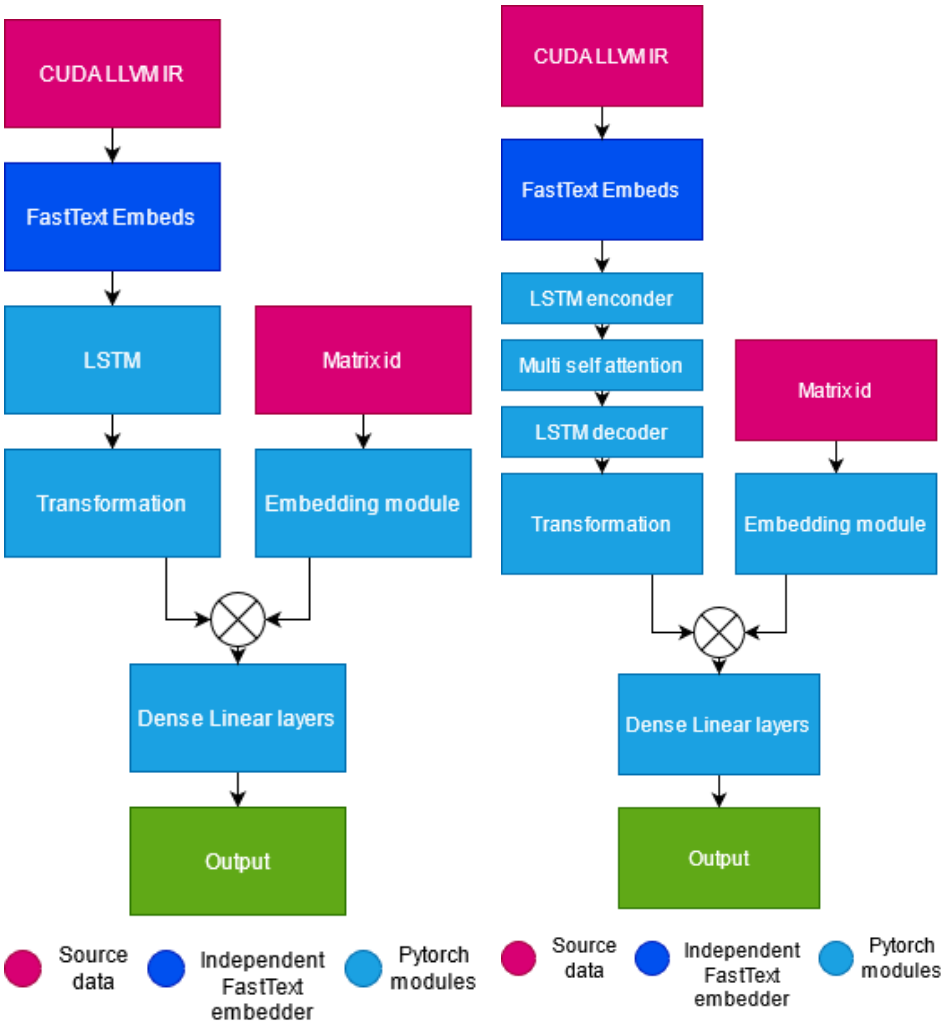
The LSTM module is based on the LSTM cell described in Section 3.1.4, and is a parallel combination of these cells, with adjustable input size, sequence length, and hidden dimension.

The embedding layer is a Pytorch improved version of one-hot-encoding. This module transforms a vector of integers into a two-dimensional matrix that can represent the distinct integers as separate entities and not a discrete series. Any particular auxiliary information, such as the type of matrix, can be passed through an embedding module.

The linear or dense layer can be used to combine the output of the LSTM module and embedder, and at the same time, shape the dimensions to fit the target dimensions.

The model itself, shown in Fig. 4.5a, is structured such that the *FastText* embeds are fed directly into the LSTM module. However, the outputted tensor has three dimensions, and both the embedder and dense linear layer works with two dimensions. To solve this conflict, the outputted tensor is transposed such that the batch dimension is first, same as in the linear layer, then a view transform is applied, which changes the three-dimension (batch, x, y) tensor into a (batch, x*y) tensor. After the transformation, this new tensor is then combined with the matrix information using the embedding module. The resulting tensor goes through a series of linear layers, with each layer reducing the number of neurons until the final layer reduces the tensor to the target shape.

Another more advanced model was also made, an LSTM Encoder-Decoder with self-attention, the attention mechanism is described in Section 3.1.5. The new model is illustrated in Fig. 4.5b.



(a) The basic LSTM model setup

(b) The LS-CAT self multi attention LSTM network

Figure 4.5: The design for the two different core ML models

4.5 Evaluation of *FastText*

To evaluate the embeds created using *FastText*, the same learning tasks used in combination with the *inst2vec* pipeline and *NCC* model had to be repeated. Overall this *FastText* and basic LSTM model outperformed the *inst2vec* and *NCC* model by around 7 percent which is quite significant. The most probable cause is the data loss caused by the *inst2vec* pipeline. Fixing this pipeline error would be quite hard as extensive knowledge would be required. Even the *inst2vec NCC* project has data loss on its thread-coarsening task. This would likely indicate that fixing this part of the pipeline is indeed a difficult task to overtake.

Instead of working on the faults in the *inst2vec* pipeline, attempting to refine the model, *FastText* embedding, and this new machine learning pipeline, would likely be more beneficial. More practical learning tasks to get potential marketable results would have to be designed.

The results for this model indicated a strong advantage over the *inst2vec* + *NCC* model solution, which had an accuracy of 0.604 on the same dataset. The basic model + *FastText* embedding achieved an accuracy of 0.67. This shows that the *inst2vec* data-loss gives quite a disadvantage, as the models themselves should perform similarly. The practical learning task models are therefore using *FastText* embedding, and not *inst2vec*.

4.6 Binary Classification Model

There are some key reasons why a binary classification model could be useful for thread-coarsening. Firstly the error cost is comparatively low to other classification scenarios, as a non-optimal thread-block would still have a performance that's not zero. Secondly, using the largest thread-block size is best, but has some scenarios where this is not the case. Another thread-block size might significantly outperform block size 1024 in its worst cases and could be used as a second alternative. Switching between these two cases could give good results depending on the accuracy, precision, recall values of the model and yield a performance increase compared to relying solely on the best block size. Theoretical results can also be computed beforehand. It should be possible to find this second thread-block for the binary classification model using just these preemptive calculations.

The first step is calculating different scenarios of model performance and what results can be achieved with that performance. To evaluate this methodology, a good second thread-block option needs to be located, as the good first option needs a good second option in the case of errors. Using Pandas, the dataset is split up in any given way, and all the different thread-blocks can be compared in different accuracy scenarios. In this case, the perceived accuracy and potentially the threshold are crucial to determining a good and bad scenario for block size 1024. Earlier attempts at creating a basic model gave an accuracy of 0.67 percent, with the threshold being 0.925 and 0.99 for the 1024 block size. This can be used for testing purposes without actually creating the model beforehand to see what

type of performance can be expected. However, this type of threshold makes an obvious "grey-zone" between 0.925-0.99, and this needs to be taken into account to get accurate performance results.

A script was made to test out some theoretical scenarios. This script divides the dataset based on the threshold conditions given. Then the parts were designated "good" and "bad". Using these two sets, the mean of all the other thread-blocks for that set can be calculated. Combining the mean values for the 1024 thread-block size at each threshold with the different mean values makes it possible to estimate probable performance.

The binary classifier has two choices, either the 1024 or another block size. If the classifier had a hundred percent accuracy, the performance would be the mean performance of the 1024 block size in the "good" set and another block size mean performance in the "bad" set.

$$perf = (good_{1024} + bad_{block})/2$$

As the classifier will not have a hundred percent accuracy, the wrong prediction cost must be taken into account. Which is the mean of the 1024 block size in the "bad" set and the mean of another block in the "good" set. Thus, using an accuracy variable, the cost of good guesses and bad guesses can be calculated.

$$perf = acc * (good_{1024} + bad_{block})/2 + (1 - acc) * (bad_{1024} + good_{block})/2$$

The following heat-map of thread block sizes performance in the above scenarios given the thresholds of 0.9759 and 0.976, as this threshold avoids any grey-zone and results in a near perfect split. The resulting performance for the accuracies: 0.55, 0.6, 0.65 and 0.67, are represented as a heat-map in Fig. 4.6.

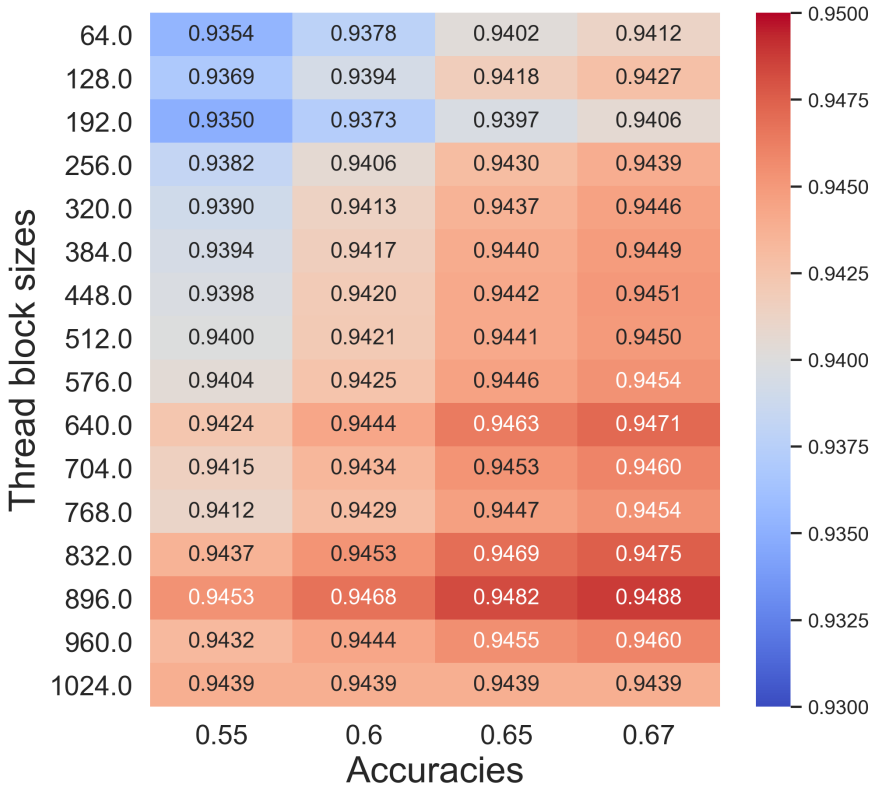


Figure 4.6: Pre-calculated performance given accuracy levels

The 896 thread-block sizes showed themselves to be the best possible choice, followed by 832 and 640. Therefore, the binary classifier decides to use either the thread-block size 1024 or 896 in a given scenario.

The dataset used for this task has the following columns and value formats, as seen in Table 4.3.

	path	matrix_id	time_13	time_15	diff_13	diff_15	flag
53622	step2/9345/4/main.cu	3	3022.586667	2998.956667	0.874647	0.881539	1
38493	step2/553/10/main.cu	5	2612.203333	2665.463333	0.985226	0.965539	1
61731	step2/121/124/main.cu	4	2668.517500	2634.127500	0.978789	0.991567	0
48612	step2/8184/22/main.cu	3	3054.410000	3042.326667	0.852584	0.855970	1
47092	step2/787/0/main.cu	4	2594.370000	2785.083333	0.994924	0.926795	1

Table 4.3: Binary classification task

The diff_13 is the thread-block size 896 performance concerning the optimal, and diff_15 is the relative performance optimal for the thread-block size 1024.

The binary classifiers data loaders item retrieval function loads the Numpy

array associated with the path, matrix ID, diff_13 diff_15 as a combined vector, and the flag target value. The binary classifiers trainer uses this data by feeding the model and calculates the BCE loss using the outputted value and the flag.

For each evaluation and test call, the model's accuracy and the performance of its block choice are measured. This performance figure is made using the diff_13 diff_15 vector combination and selecting the effect of the model choice for each combination. This logic is illustrated in Fig. 4.7.

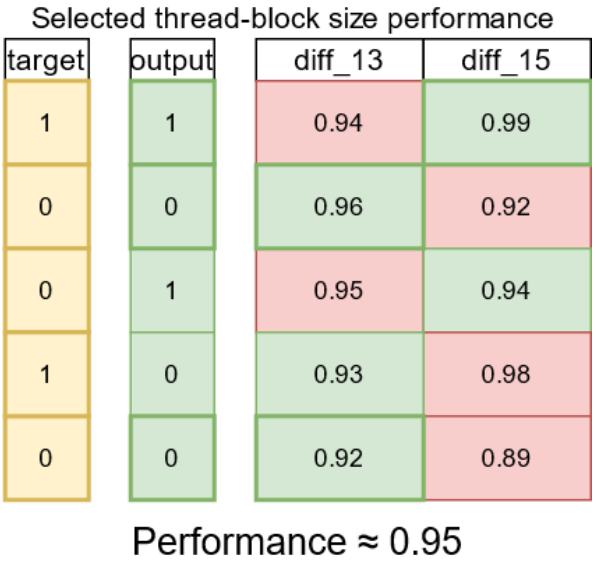


Figure 4.7: Binary classification evaluation

In this case, the thread-block size 1024 or diff_15 is selected each time the output is one, marked with green. The accuracy of the model in this example was 60 percent, and the performance 0.95. This is an improvement of the average 1024 thread-block size performance.

To find a network configuration that worked. A lot of adjustments had to be made to the model. The learning rate, batch size, and dense layer dimensions were found by repeated adjustments. The learning rate should have a value between 0.00001-0.0001. A learning rate decay function was also applied, this concept is described in Section 3.1.9. The decay function adjusts the higher-end learning rate down to the lower-end learning rate at a constant pace. The batch size could vary between 128-256, and this is common in NLP tasks [25]. The dense layer dimensions ended up being 5-10 linear layers and a hidden size of 500. Additionally, stacked LSTM modules were tried out with little success.

Classification and binary classification allows for using a threshold value. This value can impact when the output is counted as one or a zero. Any value above 0.5 is calculated as a one in binary classification, and anything less as zero. A script was designed to find the impact of a changing threshold and graph the precision and recall curve for the model. This would make it possible to adjust for error cost

and avoid potential bad predictions.

4.7 Regression Oriented Models

As the score of each thread-block can be expressed as discrete numbers and not just an optimal choice, means that regression techniques can be used. The model can be inputted a matrix IR combination and output a series of discrete numbers, one for each thread-block. A prediction would be picking the optimal value, as that number would be linked directly to a specific thread-block size. Alternatively, the model could output a single digit for the input of matrix, IR, and thread-block. This setup would, however, require multiple predictions and results in comparison for finding the optimal thread-block size. The multiple predictions model is shown in Fig. 4.8.

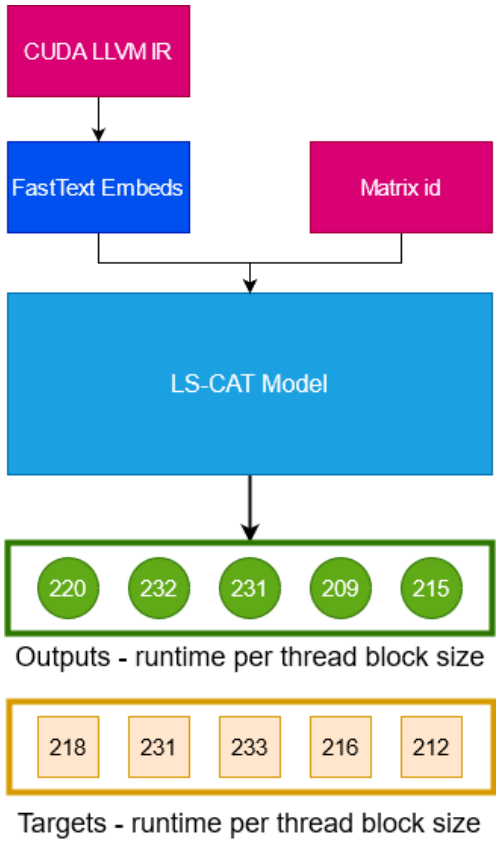


Figure 4.8: The regression output and targets

In this case, the model has only five thread-blocks, for illustration purposes. Each thread-block option is represented by its green node and represents a different amount of thread-coarsening. Each of the thread-block nodes has its output and target value. These output values are compared with the target values using

a loss function. As the model improves and the difference between the targets and outputs diminishes, it should be possible to either get the best or a good thread-block by picking, in this case, the lowest runtime number. In this example, the lowest runtime outputted by the model is 209, which corresponds to the fourth thread-block, while the real optimal would have been thread-block number five. Overall the model was not too far off with its prediction, and avoided the worst outcomes, and performed better than the average choice. If the model can prevent enough bad thread-block outcomes and pick better or the best thread-block, it would have a clear advantage over random choice and could outperform selecting the most frequently good option (block size 1024).

The relationship between the numbers matters more than the real numeric output value. For example, if the model above, instead of predicting 220, 232, 231, 209, 215 and instead predicted 318, 331, 333, 316, 312, the model would have a significantly higher loss, but the indirect prediction of picking the optimal thread-block would have been correct.

The outputted values are pretty high, and machine learning models tend to like values between 0 and 1 as this can avoid phenomena such as gradient explosion. Therefore, it might be necessary to transform the target values, using a transformation function to make the regression task possible. In this case, the Softmax activation function can be used to keep the relationship between the numbers, which is vital in this learning task. Alternatively, the relationship between the numbers could also be used directly to transform the values, for instance, division by max. Transformations such as the Sigmoid activation would also alter the values to the range of 0 to 1, but at the loss of fine resolution, and is therefore not applicable.

Two target transformations were designed, and can be seen in Listings 4.4 and 4.5.

Code listing 4.4: First target transformation

```
def target_transform(Y):
  Y = Y.min()/Y
  return Y
```

Code listing 4.5: Second target transformation Softmax

```
def softmax_transform(Y,constant):
  Y = Y/Y.max()
  Y = np.exp(Y * constant)
  Y /= np.sum(Y)
  Y /= Y.max()
  return Y
```

	path	matrix_id	time_0	time_1	time_2	time_3	time_4	lable
48377	step2/2052/1/main.cu	1	3241.193333	3211.866667	3248.693333	3236.333333	3248.683333	8
90566	step2/7633/227/main.cu	1	2698.206667	2546.920000	2575.806667	2561.413333	3844.313333	11
99255	step2/8562/13/main.cu	5	2638.133333	2706.956667	2856.470000	2990.040000	3281.983333	0
81988	step2/6427/4/main.cu	5	2953.103333	2928.710000	3453.006667	2686.743333	2702.713333	3

Table 4.4: Regression subset without all 15 thread-block ids

The data loader item retrieval for the regression works similar to that of the binary classifier. However, instead of a flag value, all the runtimes are combined to a single vector and returned in its place. The label column represents the optimal thread-block size id.

We used the exact model setups as in the binary classification, with changes to the number of output neurons. Regression with no target transformation was attempted using the dataset as is, shown in Table 4.4. Neither L1 or MSE loss proved able to beat random choice, as the model could not distinguish the finite differences between such large numbers or their abstract relationship. Target transformations had to be applied, in this case, the optimal time divided by time, seen in List 4.4. Using the transformation the alternative dataset was generated, shown in Table 4.5.

	path	matrix_id	time_0	time_1	time_2	time_3	time_4	diff_0	diff_1	diff_2	diff_3	diff_4	lable
48377	step2/2052 /1/main.cu	1	3241	3211	3248	3236	3248	0.980	0.989	0.977	0.981	0.977	8
90566	step2/7633 /227/main.cu	1	2698	2546	2575	2561	3844	0.939	0.995	0.984	0.990	0.659	11
99255	step2/8562 /13/main.cu	5	2638	2706	2856	2990	3281	1.000	0.974	0.923	0.882	0.803	0
81988	step2/6427 /4/main.cu	5	2953	2928	3453	2686	2702	0.909	0.917	0.778	1.000	0.994	3

Table 4.5: Regression subset with transformations without all 15 thread-block ids

To accommodate for this change, the data loader item retrieval was modified by using the diff values instead of runtimes, and an optional Softmax transformation was added as described in List 4.5, which can be applied to the diff target vector with variation in the exponent constant.

The target values are now all floating in the range of 0-1, which makes it possible to utilize BCE loss, not in the sense of binary cross-entropy loss, but rather multi-label loss. Each thread-block size can be described as being a deviation from the optimal or partly being the optimal class. The LS-CAT regression task can therefore use this loss function and the other regression functions either one by one or as a hybridization.

The regression task measures the performance the same way as the binary classification, by letting the output layer select a potential thread-block size and

evaluating the average performance of the choices made. This process is illustrated in Fig. 4.9.

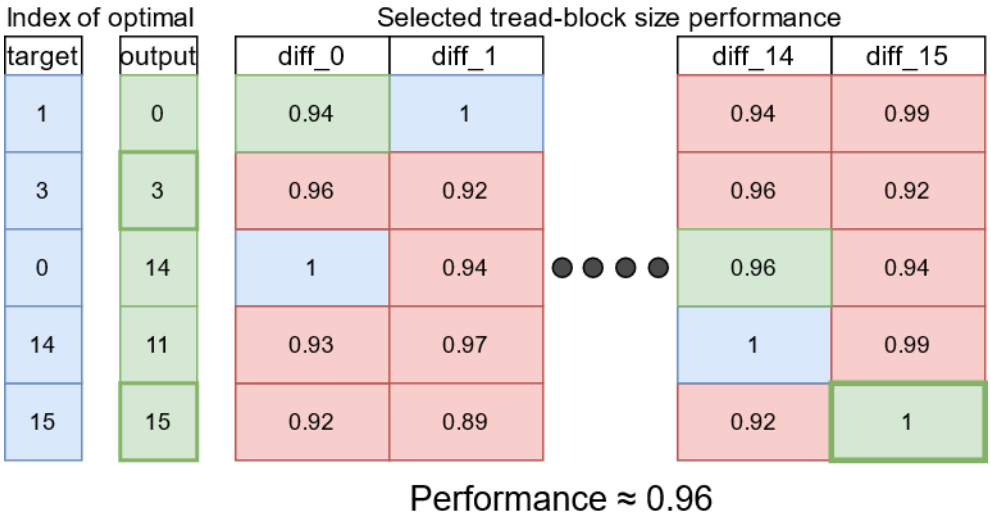


Figure 4.9: Evaluating the performance of the regression model

Selected thread-block sizes are marked as green, the overlap with the optimal thread-block size is marked with a border, non selected optimal sizes are marked blue. As the model is not explicitly trained to identify the best block size but rather good options, measuring accuracy is not essential.

Chapter 5

Results and Benchmarks

Since there were several variations in methodology to perform the same task and variations in models, each combination should have its independent results. These models have some difference in metrics but can then be directly compared with each other using the shared "performance" metric, as described in Section 4.6. First, the binary classifier results are presented. Then the regression-based model results are presented. Lastly, a cross-method comparison is performed.

5.1 Binary Classifier

The binary classifier's primary results are the accuracy and real performance of the model. Further, the impact of adjusting the threshold variable can also be displayed as graphs. Finally, the precision-recall curve can give insight into the models learning capabilities.

5.1.1 LS-CAT LSTM Model

The LSTM network reached minimum test loss after 40 thousand training steps. After that, accuracy continued to increase until around 100 thousand training steps and getting a maximum value of approximately 60.05. The potential performance increased drastically during the first few training steps, so quick that the first result beat the option of always using the 1024 block size, then it remained stable at around 0.9452 with a single peak of about 0.9458, shown in Graph 5.1.

Using the threshold variable in the case of the LSTM network, did not increase the performance as the default value of 0.5 scored the highest, shown in Graph 5.2.

5.1.2 LS-CAT LSTM Self-Attention

The self-attention LSTM network reached minimum test loss and the highest accuracy after around 50 thousand training steps. The accuracy ended up being somewhat lower than the pure LSTM network and hovering around 59.8 percent.

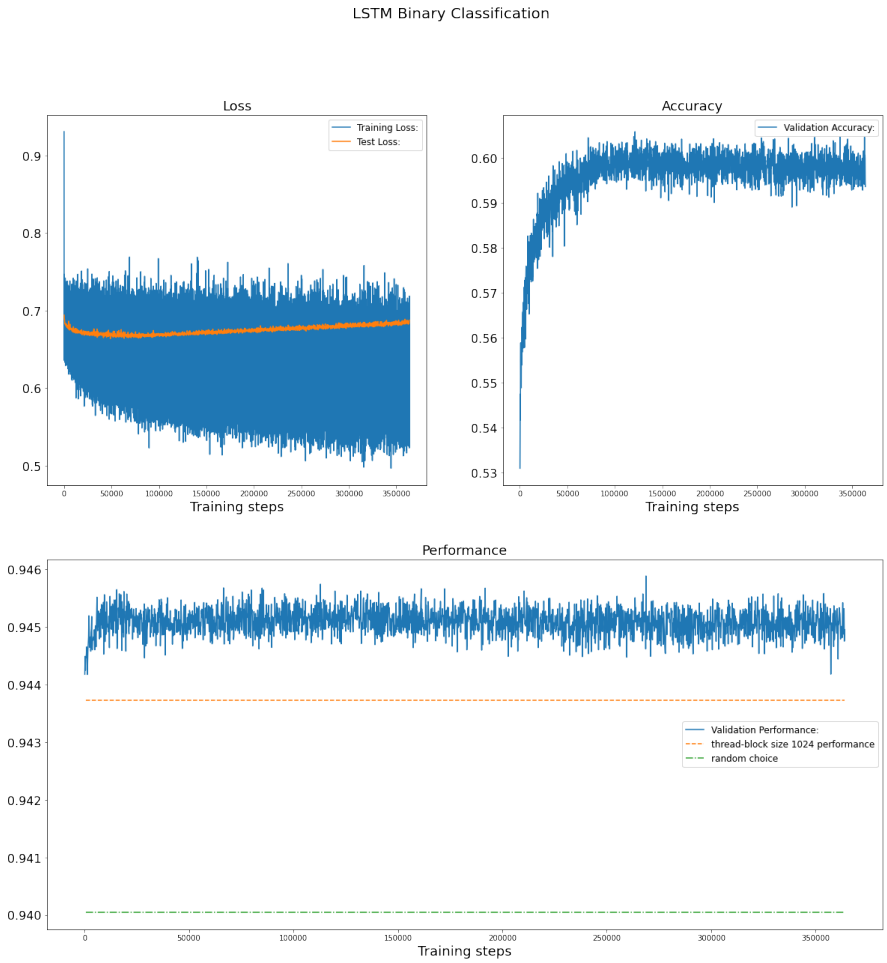


Figure 5.1: The binary classification LSTM model results

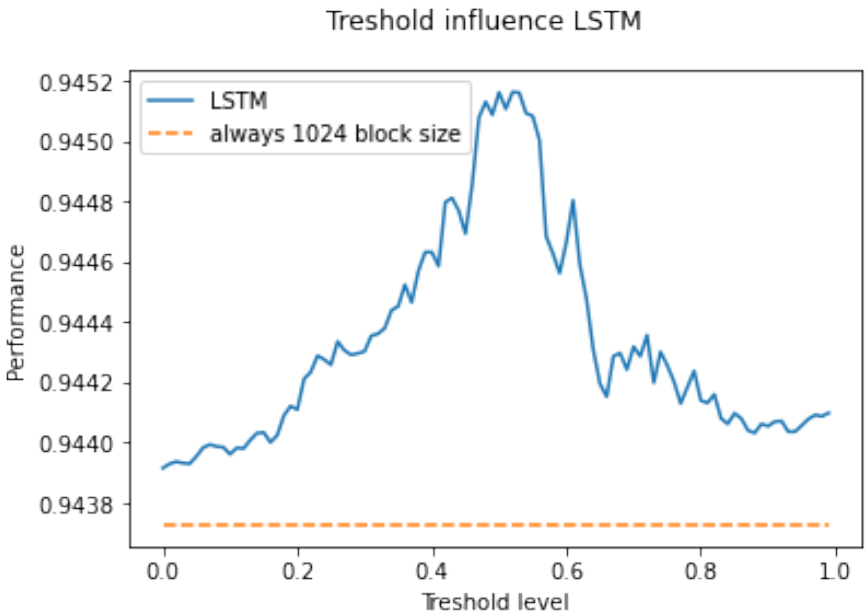


Figure 5.2: The threshold effect on the binary classification LSTM model results

The potential performance increased slower than the pure LSTM network, and the stopping point overlaps with the training loss and accuracy at 50 thousand steps. All of these results are in the Graph 5.3.

In this case using another threshold variable than the default would lead to an increase in performance, and a threshold of 0.38 could potentially boost the performance of this network. The threshold is graphed out in Fig. 5.4.

The precision recall curve, as seen in Fig. 5.5. Was very similar for both of the models, and in both cases there is a correlation with higher precision to lower recall, disregarding the small precision fluctuation at the lower recall values. By plotting the results together the pure LSTM based network has higher accuracy, but the potential performance is very similar for both of the models. The results are compared in the Fig. 5.6.

5.1.3 LS-CAT LSTM Self-Attention

5.2 Regression Results

The results of the regression task only include the loss and potential performance the model would have caused. The initial regression models used both MSE and L1 loss. Due to the transformation, found in the List 4.4. BCE-loss can compute multi-label loss. While strictly not a regression type loss, it operates with discrete targets.

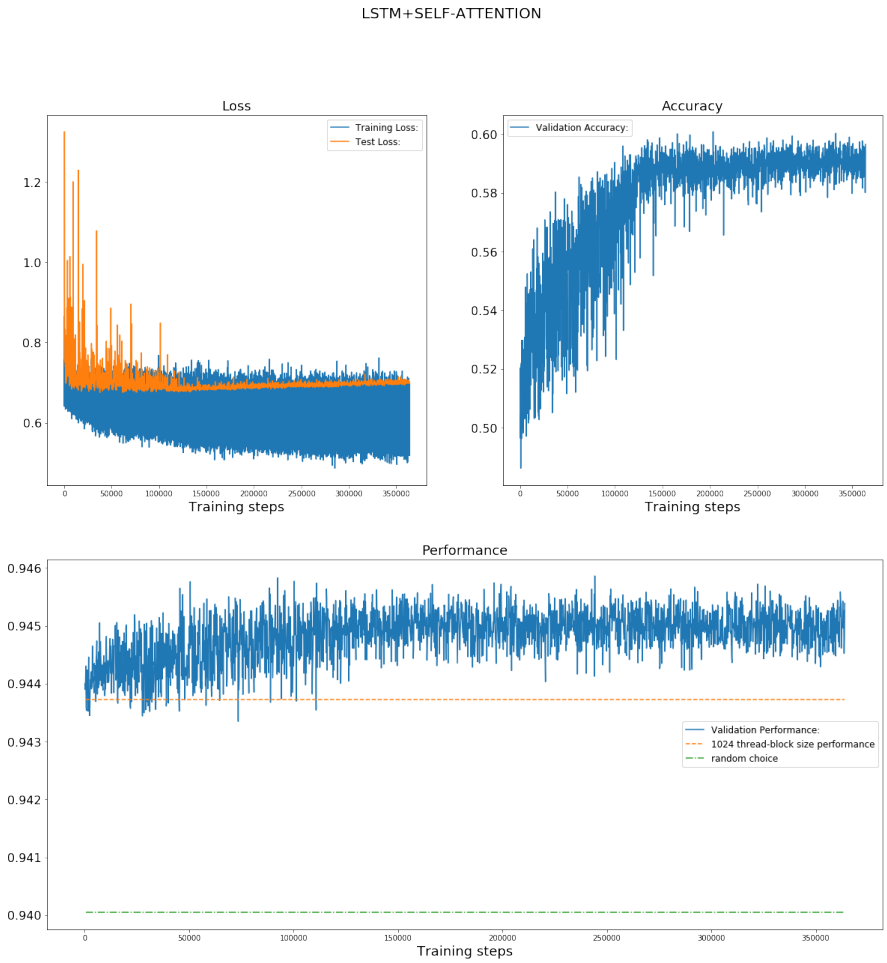


Figure 5.3: The binary classification LSTM model with self-attention results

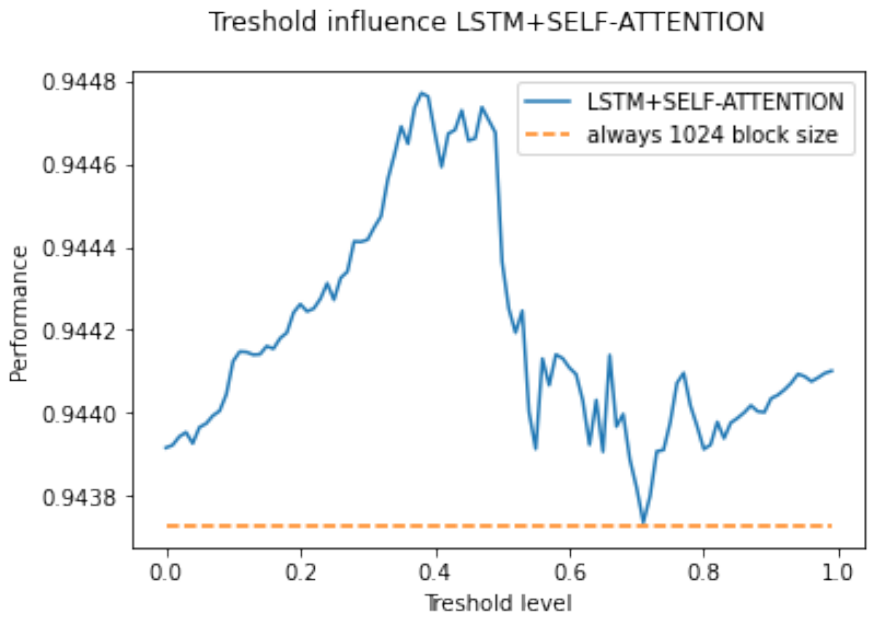


Figure 5.4: The threshold effect on the binary classification LSTM model with self-attention results

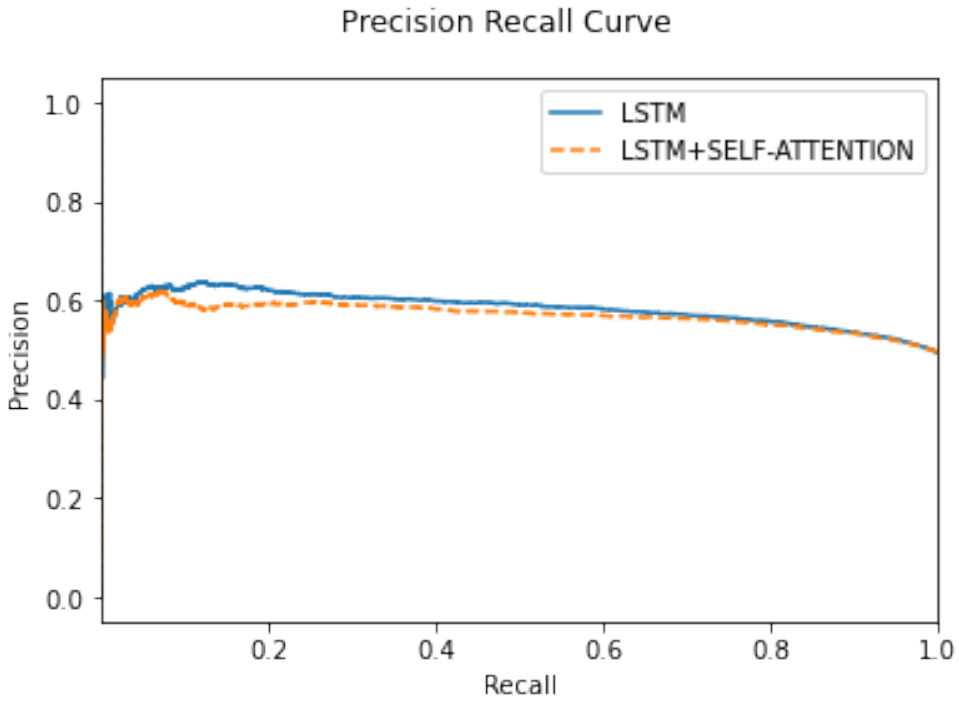


Figure 5.5: The precision recall curve of the LSTM and LSTM Self-Attention models

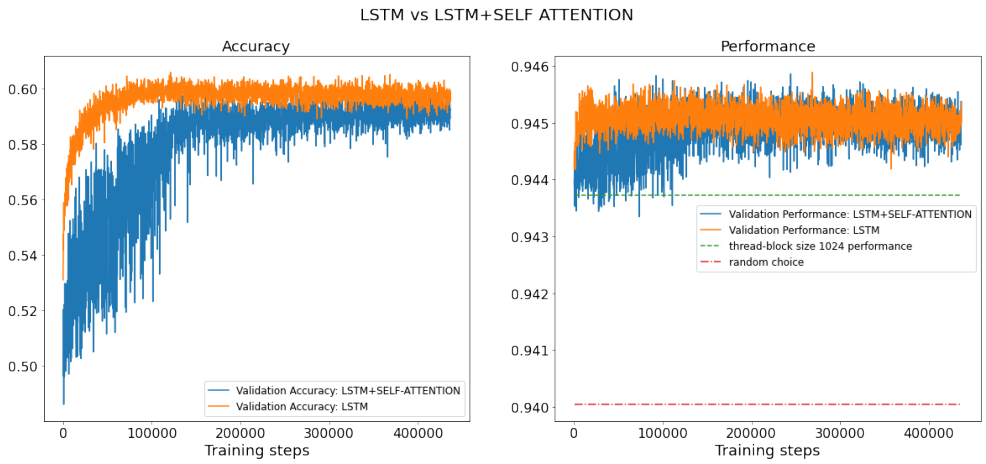


Figure 5.6: Comparison of the binary classification LSTM and LSTM with self-attention

5.2.1 LS-CAT LSTM Model

First MSE, L1, and BCE loss are each applied to the transformed target values, as loss criterions in the training process. Displayed in the Graphs 5.7-5.9.

The L1 loss has a noticeable performance lack, shown in Graph 5.7. The only loss function with no over-fitting was the BCE loss function, shown in Graph 5.8. Due to its shortcomings, the L1 loss was dropped in favor of MSE and BCE loss. All of the graphs have a noticeable dip at specific points, likely caused due to the low difference between the target values. To counteract this dipping effect, an additional target transform is applied. Using an adjusted Softmax function, described in List 4.5, with an increased exponent, the difference between the target values should increase, and this might increase the model's ability to distinguish them.

Using the same LSTM-network, and applying the Softmax transformation, from List 4.5, before training. However, in this case, the loss drops rapidly before 50 thousand training steps but keeps decreasing slightly as the training continues. As a result, the performance curve lacks a significant downward spike. Instead, the curve keeps a slight increase over time after 50 thousand training steps, shown in Graphs 5.10 and 5.11.

5.2.2 LS-CAT LSTM Self-Attention

Using the transformation described in List 4.5, the LSTM and self-attention network is retrained using both BCE and MSE loss.

The model with the included Self-Attention and MSE loss has very similar characteristics to the pure LSTM model, only reaching a lower performance and somewhat higher loss, seen in the Graph 5.12.

Using BCE loss created more differences between the models than MSE. Disregarding the early loss oscillation, the loss diminishes quickly until 100 thou-

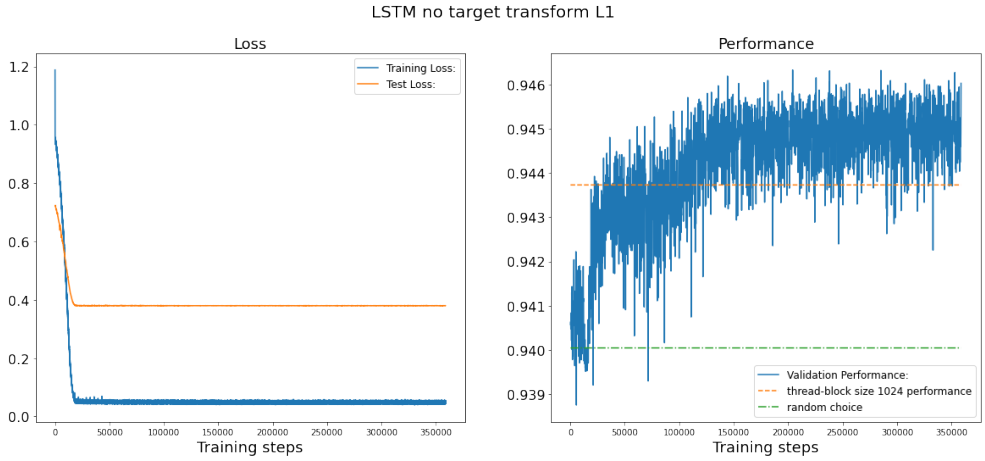


Figure 5.7: The regression LSTM model with no target transform and L1 loss

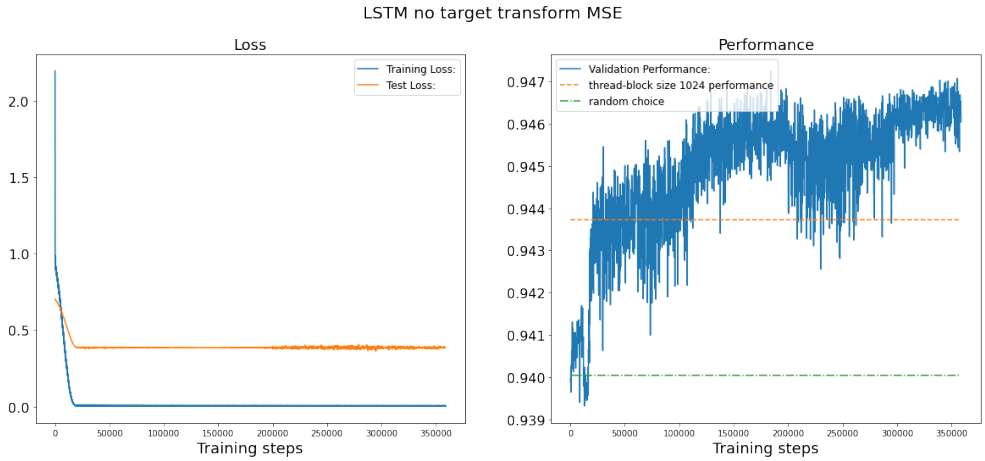


Figure 5.8: The regression LSTM model with no target transform and MSE loss

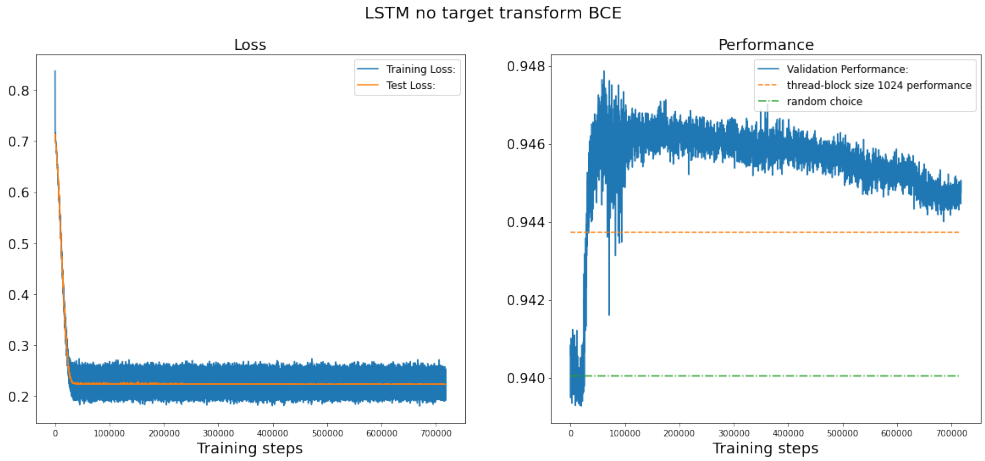


Figure 5.9: The regression LSTM model with no target transform and BCE loss

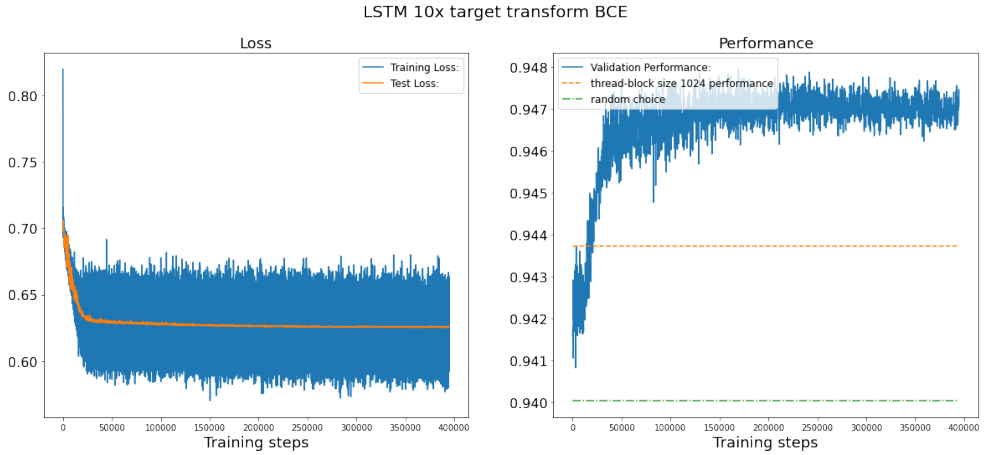


Figure 5.10: The MSE LSTM model with a 10x transform

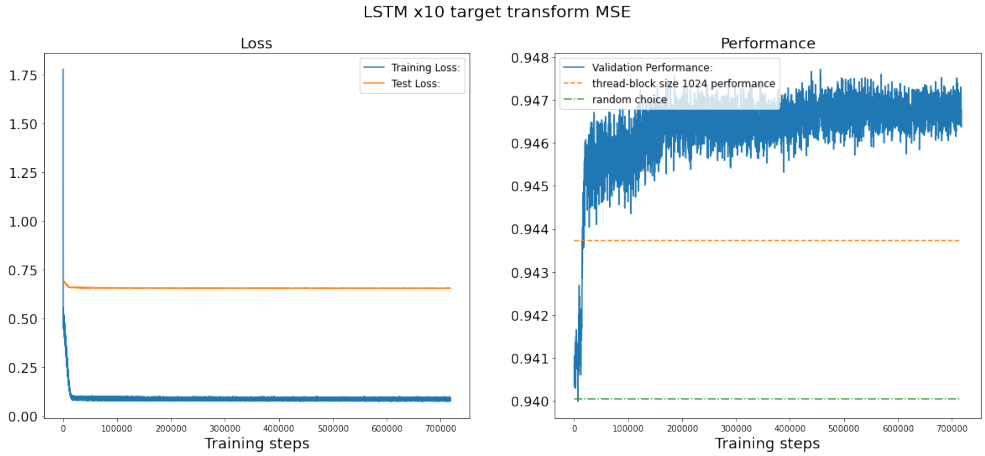


Figure 5.11: The MSE LSTM model with a 10x transform

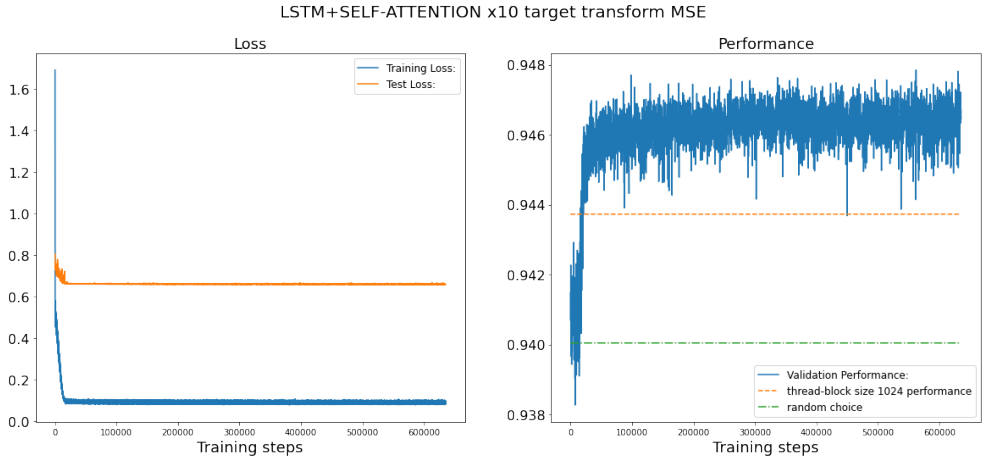


Figure 5.12: The MSE LSTM model with self-attention and a 10x transform

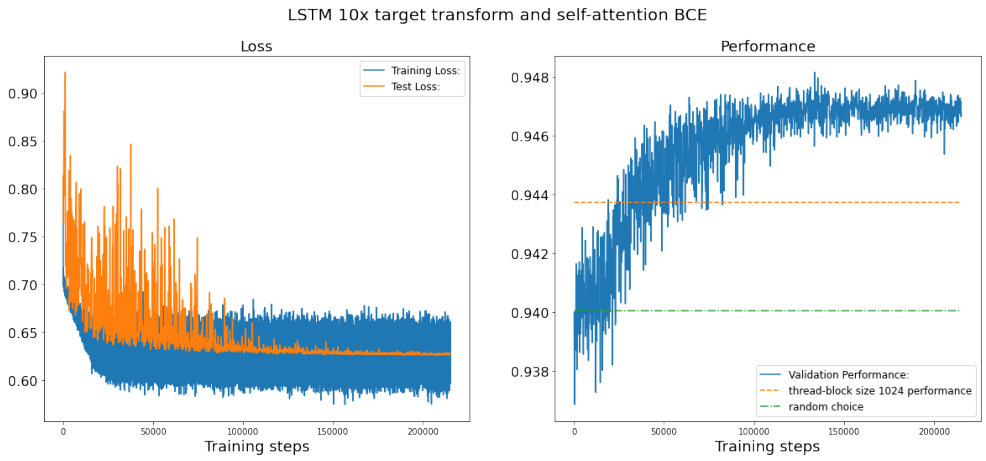


Figure 5.13: The BCE LSTM model with self-attention and a 10x transform

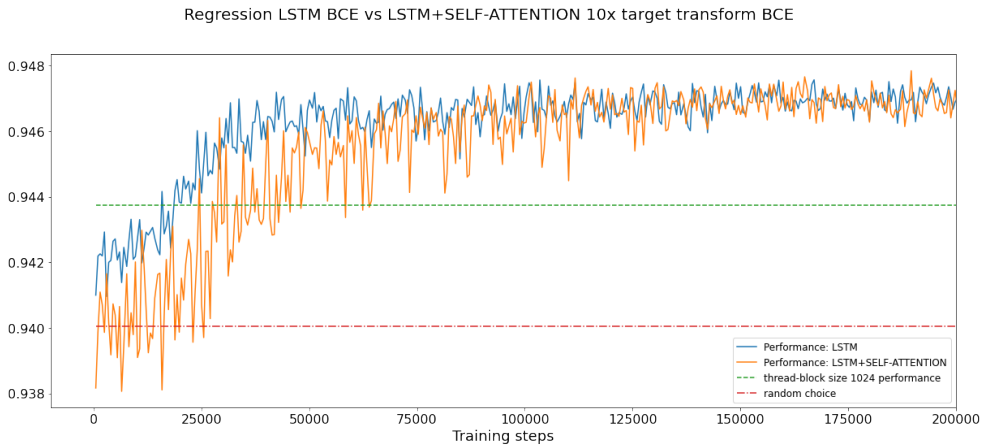


Figure 5.14: Comparison of the LSTM and LSTM with self-attention models using BCE

sand training steps, and then the decrease is less pronounced. The performance increase overlaps with the loss decrease and tapers off at around 100 thousand training steps, illustrated in Graph 5.13.

Comparing the regression task performance of the LSTM and LSTM with self-attention, highlights how similarly they performed after 100 thousand training steps. The pure LSTM network has a greater initial increase in performance, and the self-attention network had a much higher degree of oscillation. The BCE loss function creates the greatest difference between the two models. Shown in the Graphs 5.14 and 5.15.

At almost all points during the training BCE outperformed MSE, with the exception of the very beginning, seen in Graphs 4.3 5.17.

Comparing the regression performance using BCE and binary classification tasks, the superiority of the regression task in both the self-attention and pure

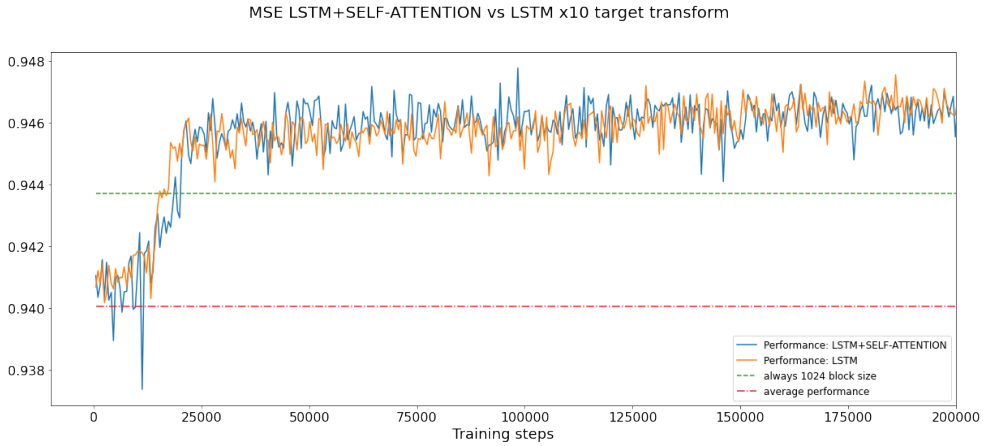


Figure 5.15: Comparison of the LSTM and LSTM with self-attention models using MSE

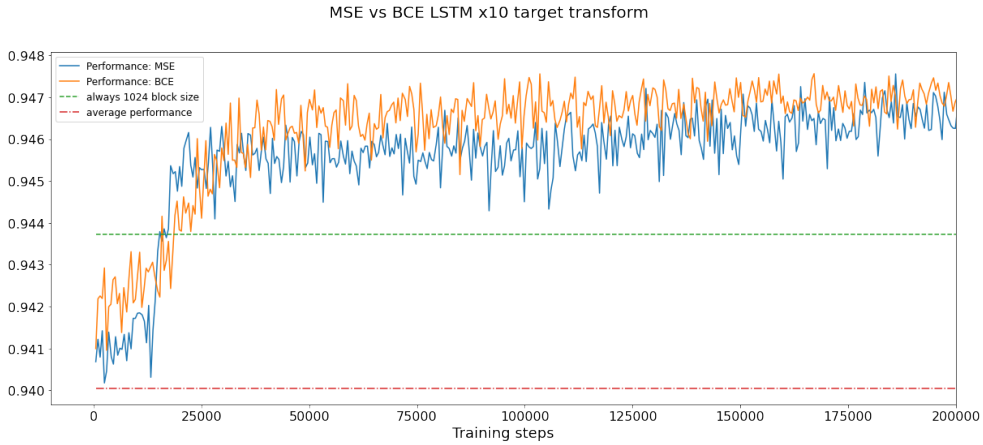


Figure 5.16: Comparison of the LSTM models using MSE or BCE

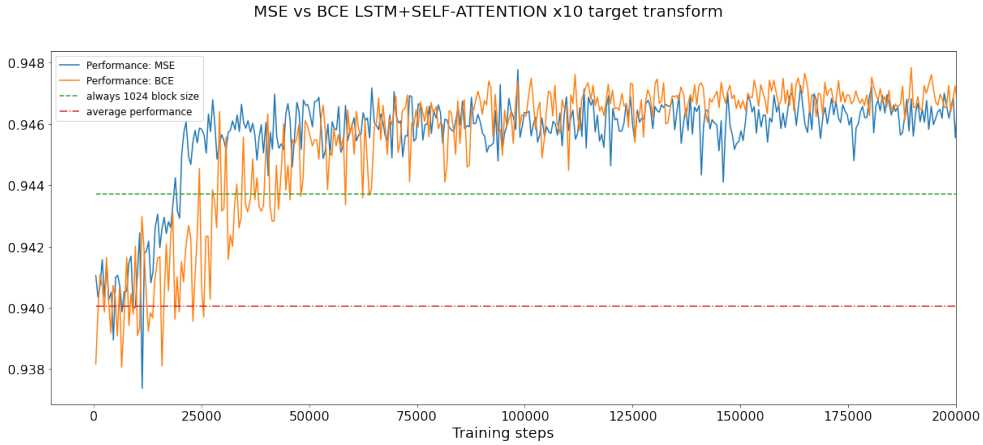


Figure 5.17: Comparison of the LSTM and self-attention models using MSE or BCE

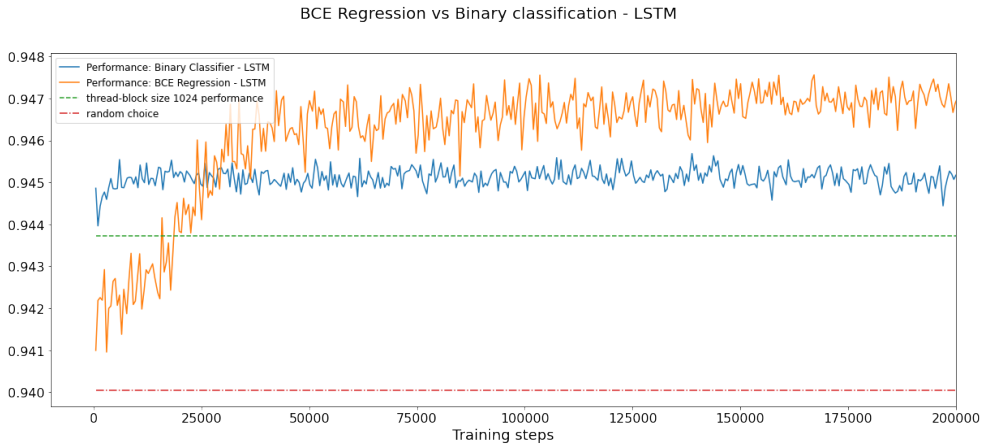


Figure 5.18: Comparison of the BCE regression LSTM binary classification LSTM

LSTM models is displayed. In both cases the regression task outperformed the binary classification somewhere between 25-50 thousand training steps. As shown in Graphs 5.18 and 5.19.

Overall the best configuration was the multi-label loss with a twelve exponent, self-attention, and 256 batch size. The best configuration results are shown in the Graph 5.20. Scoring a performance of 0.9483, this exact performance is an outlier and only case of the self-attention networks performing better than those without.

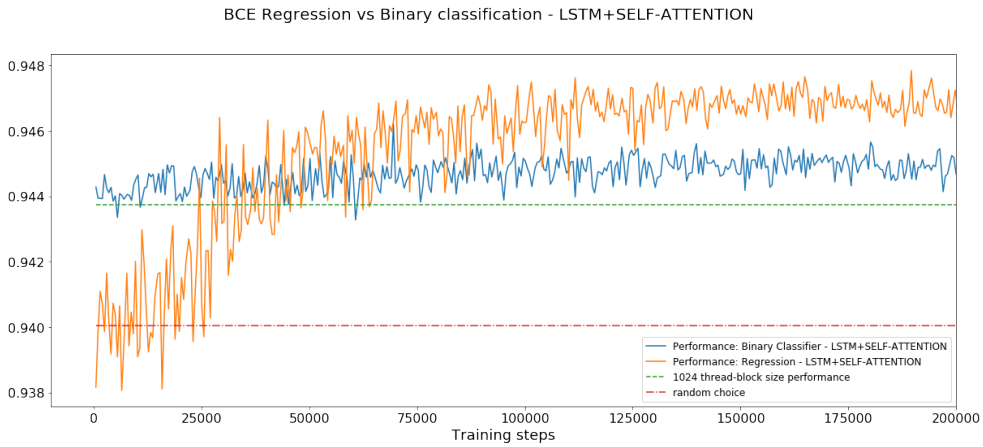


Figure 5.19: Comparison of the BCE regression LSTM+self-attention and binary classification LSTM+self-attention

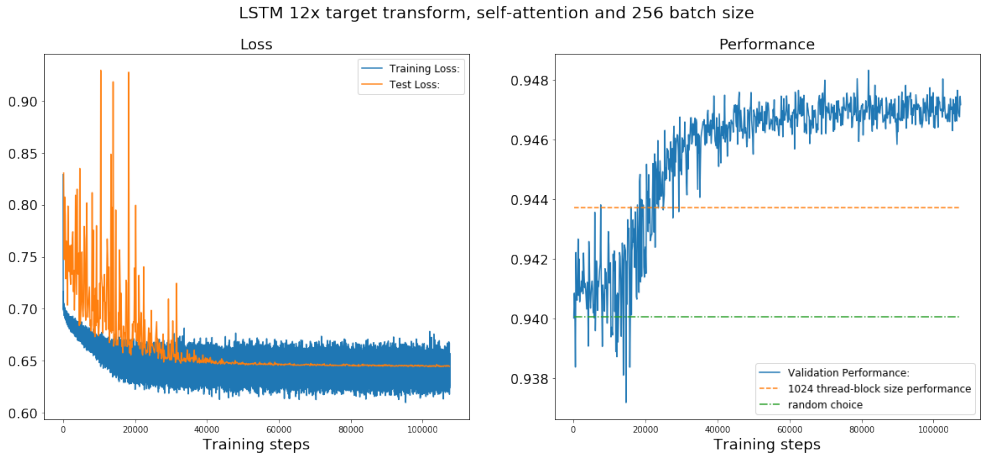


Figure 5.20: Best single point performing model

Chapter 6

Discussion

During the thesis we were able to discover several vital findings. In this chapter we will present these findings, including the choice of embedding, evaluation of the model's results, and evaluation of our LS-CAT dataset.

6.1 Embedding

The choice of embedding method and embedding sophistication proved to have a significant impact on overall performance in all learning tasks, [3]. Due to the LS-CAT dataset consisting of CUDA LLVM IR, the *inst2vec* was unable to translate a sufficient amount of tokens. More than half of the IR were lost during the embedding process using the *inst2vec* embedder. Therefore, the sophisticated *inst2vec* embedder had to be replaced with a simpler skip-gram-based embedder. The alternative embedder outperformed the *inst2vec* embedder, most likely due to the loss-less embedding process. If the *inst2vec* embedder were to be modified to handle more CUDA LLVM IR tokens and have the same token loss rate as in its original case of around 12 percent, *inst2vec* might perform better than the simpler skip-gram *FastText* embedder.

There is at least one key reason why *inst2vec* still might not outperform simple skip-gram. Compared to typical source code and program flow. Kernels are designed with linear execution in mind. Due to the cost of advanced program behavior in GPU programming. The main advantage of the *inst2vec* is clarifying more advanced program behavior for a machine-learned model. The level of complexity innate in the kernels and the ability of *inst2vec* to accurately represent this complexity would be essential factors to take into account. If the level of complexity is low in the kernels, direct linear representation might be enough. However, suppose the kernels have a high degree of instruction jumps and other non-linear program behavior or complex instruction calls. In that case, basic linear representation might not be sufficient, and more complex representation such as *inst2vec* should be preferable. An analysis of the complexity of the kernel could potentially be performed using unsupervised clustering.

To accommodate for the CUDA LLVM IR tokens, the *inst2vec* embedder has to be modified, or another complex embedder could be created using the same principles as those used in the *inst2vec*.

6.2 LS-CAT ML Models Results

Overall both the binary classification and regression-based tasks outperformed the choice of relying solely on the thread-block size 1024. Regression-based methods outperformed the binary classifier. The second model consisting of a self-attention module between the LSTM encoder decoders performed somewhat worse than the pure LSTM model. However, the semi adversarial behavior of the encoder-decoder attention, where adjustments in the encoder or attention increase the training difficulty for the decoder, increased the generality of the model. In the case of the self-attention with BCE, multi-label loss, the Graph 5.13 show these three qualities: The adversity effect is apparent in the loss graph. The test loss is the average of the training loss. The validation performance has a high degree of overlap between the training loss training step-wise. Combined, all of these factors would indicate an actual learning process and not over-fitting. While not scoring higher than the pure LSTM model, machine learning has to create general model solutions to complex problems, and this was displayed to a much higher degree in the self-attention-based model.

In comparison, the binary classification task showed signs of gradual over-fitting seen in both Graphs 5.1 and 5.3, as the test loss graph diverges upwards from the training loss, but remaining within the range of the training loss oscillation. The threshold graphs did indicate a weak correlation between the recall and precision, which can't give a definite answer about the learning process. Stronger correlations would likely signal a learning process, and no correlation would conversely indicate memorization. The threshold influence graphs give insight into some of the issues with this exact methodology. In this case, the easier to predict samples does not significantly impact the performance, meaning the difference between the two block is low when the model has high enough accuracy to more frequently pick the correctly labeled size.

The regression method using MSE loss seen in Graph 5.9, displayed the highest amount of over-fitting, and the validation training loss hoovered high above the training loss.

6.3 LS-CAT ML Model Architecture Variations

Several variations in model dimensions, combinations of loss types were tried out. At the same time, increased model complexity can lead to an increase in model performance, at the cost of training performance, and potential over-fitting. Except for the dense output layers, the input size determined directly or indirectly all the other layer's parameter count. Due to the model structure and inclusion

of self-attention, and having the model without self-attention being as similar as possible. Input sizes larger than 80 incurred a significant training speed reduction, and input sizes above 240 would be unfeasible time-wise. No increase in performance was observed at an embedder input size larger than 40. If the models did not use this self-attention model, the input size would not dictate all the dimensions, and different values for sizes, could be tested.

Model depth is the number of layers the model uses in its internal structure. The depth is one way to increase a model's ability to represent abstract relationships between the data and internal data structures. The models relied on 5-10 linear dense layers. Any more layers showed no increase in the model's performance. This was also the case for an increase in the amount of LSTM layers or amount of stacking. The likely reason these potential improvements did nothing was the increased distance between the target parameters and the crucial first layer of LSTM cells. The relation between these parts would be a lot more abstract with the added model depth.

By combining the cross-entropy loss with a regression loss, the model should also be punished for not predicting the best thread-block size. Unfortunately, this did not work, and any configuration or variation in the models using cross-entropy loss resulted in a performance score around picking the average kernel. Therefore, the multi-target classification was judged not to be a feasible solution to the selection of adequate thread-block sizes.

6.4 Evaluation of our LS-CAT Dataset

To evaluate our LS-CAT dataset, one could start looking at the performance of the models or compare the results achieved at LS-CAT with the results from the [2] OpenCL dataset. However, comparing the results from each dataset would be pretty unfair due to the difference in select-able thread-block sizes or thread-coarsening levels. The earlier works dataset has only six levels, compared to our LS-CAT dataset, sixteen levels.

Fewer possible choices result in a higher degree of distinction between the choices, and this distinction might both make thread-coarsening appear more lucrative and also easier to perform. As an example, we can compare four levels vs. eight levels of thread-coarsening, seen in Fig. 6.1.

Four levels

0.94	0.96	0.98	1

Eight levels

0.94	0.95	0.96	0.97	0.97	0.98	0.99	1

Figure 6.1: Comparison of four and eight levels of thread-coarsening - Displayed as difference from optimal

In this case, the average performance of the thread-coarsening levels is the same at 97 percent and has the same median of 97 percent. There is less variation for each thread-coarsening level at the example with eight levels, making them harder to distinguish from each other than four levels. As a test, the LS-CAT dataset was reduced from sixteen levels to only eight levels. The consequence was an immediate jump in perceived performance, displayed in the Graph 6.2, using only a prototype model.

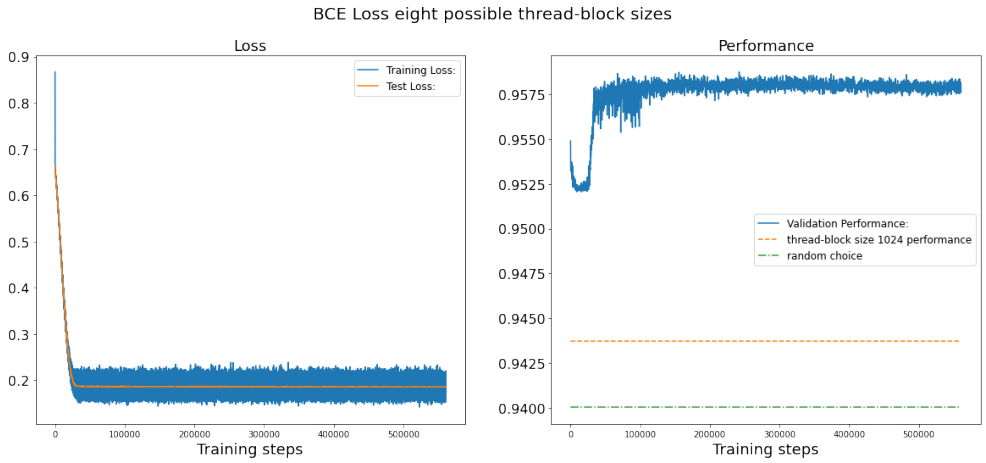


Figure 6.2: LS-CAT with only eight thread-block sizes

This reduction in choices of thread-block sizes gives also an impression of higher performance. The process of going from sixteen to eight thread-block sizes ends up removing a lot of the best performing scenarios, which falsely increases the overall average performance. The different results from each dataset can therefore not be compared fairly.

For these reasons, our LS-CAT dataset can not be judged by making a simple model result comparison with this earlier dataset. Instead, seeing how the models performed on our LS-CAT, compared to random choice, and relying solely on the best block size would be better. There was enough semantic information in the

kernel IR, for a model (0.9483) to outperform both random choices (0.94) and only select the largest thread-block size (1024) (0.9437). Several models gave results indicating strongly that a learning process and not memorization was performed. Also, a model relying on only the matrix run-time and name gave results similar to random choice. The presence of this semantic information implies both that the LS-CAT dataset is valid and that a kernel's source code has sufficiently distinct information that makes abstract learning tasks possible to perform. Taken this into account the LS-CAT dataset works.

Chapter 7

Conclusions and Future Work

Performance tuning tasks are hardly possible for humans to perform by hand. However without tuning, software ends up lacking key low-level optimizations, causing a drop in overall performance. Furthermore, current autotuners rely on extensive search processes, which can end up being more time-consuming than time-saving. As shown in this thesis, machine learning can reduce or almost nullify the search process, and in turn, create better autotuners. New source code-based ML autotuners have performed a range of different tasks but require large datasets. With this in mind, we created a code base and dataset generator LS-CAT. The first dataset can be used for finding thread-block sizes for a kernel matrix combination. This thesis applied natural language processing and machine learning techniques to our LS-CAT dataset.

The goal of having a model select thread-block sizes to increase the performance was met. Two main methods and two main models proved to beat the performance associated with the largest thread-block size. With the best configuration scoring 0.9483, an increase of 0.49 percent over the largest block. The multi-label loss-based regression proved to be the most efficient, followed by more normal regression and then binary classification. While the self-attention-based models gave a less overall performance. The network's self adversarial properties increased the learning difficulty and increased the generality, which is very desired in machine learning models.

The self-attention model with multi-label loss indicated more strongly a learning process than the other models. Regression using MSE or L1 loss and the binary classification resulted in over-fitting. Overall the results for the different models indicated that the semantic information in the source code was enough to learn the abstract relationship between relative performance, source code, thread-block size, and matrix size. This would strengthen earlier claims that source code can be used to learn abstract program features. Even if earlier attempts used a very small dataset, it's entirely possible that their model did learn enough features and did not over-fit or memorize the data.

The embedding process of *inst2vec*, which was deemed significant in earlier projects, had issues with data loss when transforming the intermediate represen-

tation to numeric data. The CUDA-based LLVM IR had around five times the data loss as the OpenCL LLVM IR, which the earlier projects utilized. The solution to this was implementing skip-gram-based methods using the tool *FastText*, as this is a fully lossless process. A prototype learning task of binary classification was designed as a test. In this test *inst2vec* was outperformed by the more uncomplicated skip-gram method, likely due to the data loss. The potential benefit of a more complex embedding process that does not also include high data loss remains, therefore, unexplored on our LS-CAT.

Future Work

Ideally, an embedding process more adapted to source code and handling a broader range of tokens should be developed. This sophisticated embedder could be fairly tested in comparison with lossless skip-gram. This could be done potentially by improving upon the *inst2vec* pipeline and its interpretation of LLVM IR tokens. *inst2vec's* biggest issue revolved around data-loss. There are several ways to mitigate data loss. For instance, improve the handling for unseen tokens, rather than just dropping them. Or instead, increasing the amount of seen tokens during training would directly diminish data loss. As a last alternative, have a custom user-defined parser for the unseen tokens. This user-defined parser would avoid the potential issue and not put too much work on the developers. There are probably more special cases than just CUDA Clang LLVM IR tokens not being recognized by their system, as the developers were fine with a twelve percent loss associated with the OpenCL data.

This project focused solely on the use case of finding thread-block sizes. The base LS-CAT source code and result generator can be used for a range of different scenarios. For example, LS-CAT could create datasets for other tasks such as heterogeneous device mapping, or measurements such as energy consumption, instead of just time and thread-block sizes. Machine learned models are partly problem agnostic and more dependent on either type of data or specific dataset, so the models mentioned in this paper would work on other learning tasks.

Bibliography

[1] L. Bjertnes, J. O. Tørring, and C. A. Elster, "Ls-cat: A large-scale cuda autotuning dataset," *IEEE International Conference on Applied Artificial Intelligence (ICAPAI 2021)*, vol. 31, May 2021.

[2] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-End Deep Learning of Optimization Heuristics," en, in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Portland, OR: IEEE, Sep. 2017, pp. 219–232, ISBN: 978-1-5090-6764-0. DOI: 10.1109/PACT.2017.24. [Online]. Available: http://ieeexplore.ieee.org/document/8091247/ (visited on 10/31/2020).

[3] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural Code Comprehension: A Learnable Representation of Code Semantics," en, *arXiv:1806.07336 [cs, stat]*, Nov. 2018, arXiv: 1806.07336. [Online]. Available: http://arxiv.org/abs/1806.07336 (visited on 10/31/2020).

[4] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon, "Compiler-based graph representations for deep learning models of code," en, in *Proceedings of the 29th International Conference on Compiler Construction*, San Diego CA USA: ACM, Feb. 2020, pp. 201–211, ISBN: 978-1-4503-7120-9. DOI: 10.1145/3377555.3377894. [Online]. Available: https://dl.acm.org/doi/10.1145/3377555.3377894 (visited on 10/31/2020).

[5] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "ProGraML: Graph-based Deep Learning for Program Optimization and Analysis," en, *arXiv:2003.10536 [cs, stat]*, Mar. 2020, arXiv: 2003.10536. [Online]. Available: http://arxiv.org/abs/2003.10536 (visited on 10/31/2020).

[6] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, Aug. 2019, ISSN: 0360-0300. DOI: 10.1145/3320060. [Online]. Available: https://doi.org/10.1145/3320060.

[7] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Nov. 1998, pp. 38–38. DOI: 10.1109/SC.1998.10004.

[8] M. Frigo and S. Johnson, "The Design and Implementation of FFTW3," en, *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005, ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840301. [Online]. Available: http://ieeexplore.ieee.org/document/1386650/ (visited on 12/08/2020).

[9] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," en, *Journal of Physics: Conference Series*, vol. 16, pp. 521–530, Jan. 2005, ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/16/1/071. [Online]. Available: https://iopscience.iop.org/article/10.1088/1742-6596/16/1/071 (visited on 12/08/2020).

[10] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," en, *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, Feb. 2005, ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840306. [Online]. Available: http://ieeexplore.ieee.org/document/1386651/ (visited on 12/08/2020).

[11] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," en, in *2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, Italy: IEEE, May 2009, pp. 1–11, ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161004. [Online]. Available: http://ieeexplore.ieee.org/document/5161004/ (visited on 12/08/2020).

[12] Q. Liu, F. Zhou, R. Hang, and X. Yuan, "Bidirectional-convolutional lstm based spectral-spatial feature learning for hyperspectral image classification," *Remote Sensing*, vol. 9, no. 12, 2017. [Online]. Available: https://www.mdpi.com/2072-4292/9/12/1330.

[13] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," *arXiv:1409.0473 [cs, stat]*, May 2016, arXiv: 1409.0473. [Online]. Available: http://arxiv.org/abs/1409.0473 (visited on 05/25/2021).

[14] M.-T. Luong, H. Pham, and C. D. Manning, "Effective Approaches to Attention-based Neural Machine Translation," *arXiv:1508.04025 [cs]*, Sep. 2015, arXiv: 1508.04025. [Online]. Available: http://arxiv.org/abs/1508.04025 (visited on 05/25/2021).

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *arXiv:1706.03762 [cs]*, Dec. 2017, arXiv: 1706.03762. [Online]. Available: http://arxiv.org/abs/1706.03762 (visited on 05/25/2021).

[16] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, *On empirical comparisons of optimizers for deep learning*, 2019. arXiv: 1910.05446 [cs.LG].

[17] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Jan. 2017, arXiv: 1412.6980. [Online]. Available: http://arxiv.org/abs/1412.6980 (visited on 05/25/2021).

[18] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, vol. 1, no. 4, pp. 295–307, 1988, ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(88)90003-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0893608088900032.

[19] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't Decay the Learning Rate, Increase the Batch Size," *arXiv:1711.00489 [cs, stat]*, Feb. 2018, arXiv: 1711.00489. [Online]. Available: http://arxiv.org/abs/1711.00489 (visited on 05/25/2021).

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[21] J. Davis and M. Goadrich, "The Relationship between Precision-Recall and ROC Curves," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06, event-place: Pittsburgh, Pennsylvania, USA, New York, NY, USA: Association for Computing Machinery, 2006, pp. 233–240, ISBN: 1-59593-383-2. DOI: 10.1145/1143844.1143874. [Online]. Available: https://doi.org/10.1145/1143844.1143874.

[22] *Compiling CUDA with clang*. [Online]. Available: https://llvm.org/docs/CompileCudaWithLLVM.html (visited on 01/24/2021).

[23] P. J. García-Laencina, J.-L. Sancho-Gómez, and A. R. Figueiras-Vidal, "Pattern classification with missing data: A review," *Neural Computing and Applications*, vol. 19, no. 2, pp. 263–282, Mar. 2010, ISSN: 1433-3058. DOI: 10.1007/s00521-009-0295-6. [Online]. Available: https://doi.org/10.1007/s00521-009-0295-6.

[24] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks, and G.-Y. Wei, *Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference*, 2021. arXiv: 2011.14203 [cs.AR].

[25] M. Neishi, J. Sakuma, S. Tohda, S. Ishiwatari, N. Yoshinaga, and M. Toyoda, "A bag of useful tricks for practical neural machine translation: Embedding layer initialization and large batch size," in *Proceedings of the 4th Workshop on Asian Translation (WAT2017)*, 2017, pp. 99–109.

Appendix A

Poster

Applying NLP ML techniques to our Large-Scale CUDA AutoTuning dataset

NTNU
Norwegian University of Science and Technology

-Master thesis Lars Bjertnes-

Supervisor Anne C. Elster,
Co supervisor Jacob O. Tørring

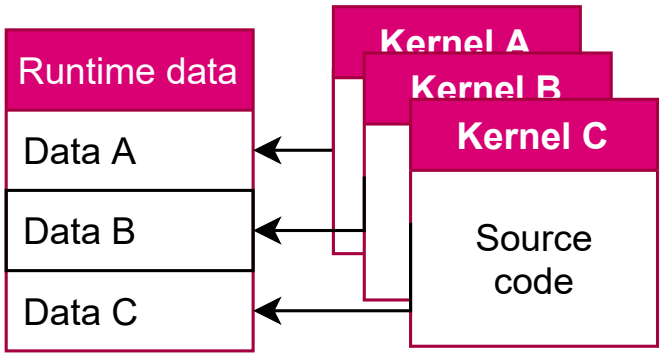
HPC LAB

Autotuning tasks are almost impossible for humans to perform. The relation between hardware parameters, and program performance is far too complex, for any human to set. Without auto-tuning, software ends up missing low-level optimizations, resulting in lower performance. Autotuning with ML-NLP based methods uses source code to perform autotuning oriented tasks [2-4]. There is little GPU source code data for these tasks. The LS-CAT (Large-Scale CUDA AutoTuning) project uses CUDA GPU based kernels, and generated a dataset with the goal of performing thread-coarsening. This new project implements several ML pipelines to perform thread-coarsning on LS-CAT

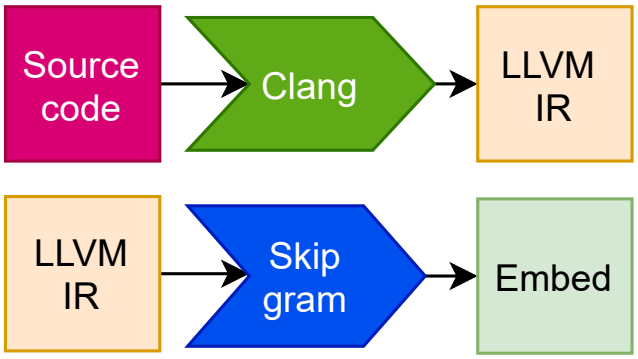
Summary

Implemented an end-to-end machine learning pipeline, designed for CUDA source code data.
First implementation applying ML NLP techniques on our earlier LS-CAT dataset
Findings indicate a generalized learning process, not memorization, implying that CUDA source code has learn-able abstract features.

Beat both random choice (0.94) and best default option (0.9437), scoring 0.9483.

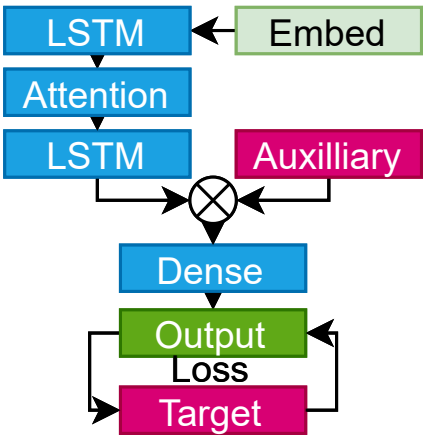


LS-CAT consists of CUDA kernels and their runtime information



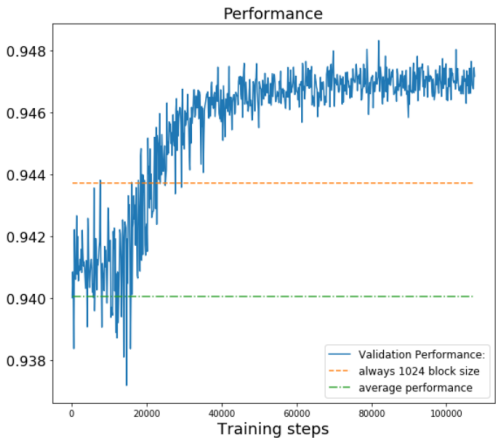
The source code is transformed using Clang and Skip-gram

1. "LS-CAT"



ML model using the embedded and auxiliary information to predict optimal thread-block sizes.

2. "Embedding"



Several different methods and model variations, outperformed both average case and the average best block size

3. "Model"

4. "Results"

Future work

Adding sophisticated embedding:
The "inst2vec" [2] embedder proved unable to fully parse CUDA Clang LLVM IR tokens, resulting in high data-loss. Either fixing the embedders fault or creating a new more complex source code oriented embedder would give a potential advantage over skip-gram which is developed for plain text.

Creating more learning tasks:
Currently LS-CAT has only thread-block sizes and runtimes. This could be expanded upon by adding more measurements, adding more parameters that are modifed, or adding device mapping.

References:

[1] Lars Bjertnes, Jacob O. Tørring and Anne C. Elster, "LS-CAT: A Large-Scale CUDA AutoTuning Dataset" (2021). IEEE *International Conference on Applied Artificial Intelligence* (ICAPAI 2021).

[2]T. Ben-Nun, A. S. Jakobovits and T. Hoefler, 'Neural Code Comprehension:A Learnable Representation of Code Seman, Nov. 2018

[3] Cummins, Chris and Petoumenos, Pavlos and Wang, Zheng and Leather,Hugh, "End-to-End Deep Learning of Optimization Heuristics,", IEEE,September 2017

[4]Brauckmann, Alexander and Goens, Andr´es and Ertel, Sebastian andCastrillon, Jeronimo, "Compiler-based graph representations for deeplearning models of code,", ACM, February 2020

Appendix B

Our LS-CAT paper

LS-CAT: A Large-Scale CUDA AutoTuning Dataset

Lars Bjertnes, Jacob O. Tørring, Anne C. Elster
Department of Computer Science
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
larbje@stud.ntnu.no, jacob.torring@ntnu.no, elster@ntnu.no

***Abstract*—The effectiveness of Machine Learning (ML) methods depend on access to large suitable datasets. In this article, we present how we build the LS-CAT (Large-Scale CUDA AutoTuning) dataset sourced from GitHub for the purpose of training NLP-based ML models. Our dataset includes 19 683 CUDA kernels focused on linear algebra. In addition to the CUDA codes, our LS-CAT dataset contains 5 028 536 associated runtimes, with different combinations of kernels, block sizes and matrix sizes. The runtime are GPU benchmarks on both Nvidia GTX 980 and Nvidia T4 systems. This information creates a foundation upon which NLP-based models can find correlations between source-code features and optimal choice of thread block sizes.**

There are several results that can be drawn out of our LS-CAT database. E.g., our experimental results show that an optimal choice in thread block size can gain an average of 6% for the average case. We thus also analyze how much performance increase can be achieved in general, finding that in 10% of the cases more than 20% performance increase can be achieved by using the optimal block. A description of current and future work is also included.

***Index Terms*—CUDA codes, Autotuning, NLP, Machine Learning (ML), dataset, GitHub**

I. Introduction

To get full system usage, software implementations have to target each system. However, With the increase in hardware variations, the interface between software and hardware is becoming more complex. However, by parameterizing the software, this interface could be configurable. By tuning the parameters, the software could more efficiently interact with the hardware, resulting in increased performance. The challenge is that there are a lot of different parameters, that each usually have certain specific limits and legal values. Setting good parameters for a program requires high competence, while also being a time consuming process.

Autotuning attempts to solve this problem by computerizing the parameter adjustments. The system adjusts each parameter to some extent, then compiles, executes and measures the program. By comparing different results, the optimal combination of parameters can be found.

There is usually a complex relation between a specific parameter and the total change in performance. A naive search through the parameter space is therefore typically not optimal. Since searching through all possible legal combinations of parameters is also a highly time consuming process, some methods have been developed [1]–[3] to search through the parameters more efficiently.

However, these methods are still reliant on compiling and executing the program to do gradual adjustments, which still takes a lot of time. A better alternative would be to have an autotuner that can find good parameters without compiling or executing the program.

A dataset consisting of the results for each legal combination of hardware systems, and all other information that can change the performance or the run-time, could be used to find the absolute optimal combination of parameters for any given configuration. Unfortunately creating such a dataset would, as mentioned above, take incredibly long time, and the amount of data required would make the dataset huge. However, one could create a smaller dataset, that has a good enough coverage of the parameter space. This dataset could be used to find good parameters without compiling and executing everything, if there were a method to use this smaller dataset that is almost as good as having the entire dataset. Machine Learning (ML) is well suited in situations where there is enough data, and there is an unclear relationship between the start and end point. There are several machine learning models, but they all conceptually create an internal mathematical model typically consisting of activation's, weights, and biases. The weights and biases are adjusted depending on how the models output value compares to the target value. The model "learns" patterns this way, and can be used in a wide range of applications.

ML-assisted autotuning implementations have mostly focused on only the parameters and hardware information, and mostly using source code meta features [4]. However, the entirety of the source code plays a role in the software and should be taken into consideration.

ML-based attempts at using the whole source code for autotuning include finding the run time of parallel programs [5], device mapping [6], or multiple tasks [7]–[10]. These attempts have usually focused on the programming languages C, C++ and OpenCL. OpenCL is a programming language that makes it possible to utilize the GPU for more general purpose programming. CUDA is designed specifically for Nvidia GPUs. The upside is that CUDA has a higher performance compared to OpenCL [11].

The earlier attempts at using source code to do ML-based autotuning on OpenCL, while getting good results, have limited themselves to a low number of distinct source codes. A lot of the data is sourced from libraries, which might not be representative of most written code. In this paper, we will, however, present how we generated a larger CUDA dataset

sourced from a collection of independent developers.

II. Background

A. GPU and CUDA

GPUs have high core count that can process a lot of data simultaneously. CUDA, which is C++ based targets Nvidia GPUs. CUDA functions that run on the GPU are called kernels, and are either marked as global if run from the host system, or device if called from the global kernel. The global kernels take blocks of threads issued over a grid. The block parameter is a three dimensional representation of a collection of threads. Each block should be divisible by 32, which is known as the warp size. A warp executes all threads simultaneously. A block can at most run 1024 threads at the same time. The optimal number of threads per block is not always 1024, as several smaller blocks would have more unhindered register access, for instance.

B. Machine learning and NLP

Machine learning relies on two important concepts, the forward and backward pass. The forward pass is done by iterating over mathematical functions using an input parameter. In supervised learning, the output is then compared with the target value for the input. This comparison is done using a loss function that tries to find an accurate number for the difference of all output and target values at that given training step. This loss gives the model an adjustment target. A backward pass is then done by adjusting its internal weights and biases.

By repeating the process of forward and backward passes, the weights are adjusted to minimize the loss function. This, in turn, achieves outputs with similar values to the target values.

As datatypes fed into a machine learning model have to be represented numerically, source code can't without any processing be fed directly into the model.

Natural language processing (NLP), is focused on how to best represent text as numerical vectors. By using NLP techniques, source code can be transformed into distinct vectors, which can in turn be used for machine learning.

III. Related works

A. end2end-dl/deeptune

Manually crafted heuristics are usually not ideal and can be challenging to create. Deeptune by Cummins et al. [7] therefore used an end-to-end ML-based model to create heuristics automatically. The dataset itself consists of a handful of unique OpenCL kernels, executed with different hardware systems on the GPU or CPU, and with varying number of threads. The source code is stored as raw code, but is pre-processed, discarding unnecessary information. Each line of code is then turned into a sequence of tokens, and turned into embeddings by the model.

B. NCC

By using some of the methods from NLP, combined with code dependencies, NCC [9] tries to create an embedded representation of code based on LLVM IR or an intermediate representation of the code. Since it uses LLVM IR, the model should work on languages that can be compiled with LLVM. The embedder "inst2vec" can therefore be trained on a larger general purpose dataset, consisting of library sources. NCC then train on a smaller dataset that has a specific task, and the OpenCL dataset from DeepTune is reused for the same tasks.

C. CDFG

CDFG [10] uses graph-based ML-techniques, unlike DeepTune and NCC, who focus on the sequenced part of source code learning. CDFG focuses on device mapping and thread coarsening using the same DeepTune and inst2vec datasets.

One significant change made to the dataset is the addition of an abstract syntax tree (AST). The AST is a graph representation of how the code parts depend on each other. CDFG also labels all the connections so that they are not interchanged.

D. ProGraML

ProGraML [8] further build upon using graph representations, by using three different graphs derived from source code. The goals are device mapping and algorithm classification, on the same DeepTune dataset.

The three graphs used are control flow, data flow, and call flow. The control flow graph represents the order the statements are executed in based on their sequence and branching. The data flow graph is a data dependency graph. The call flow graph connects the original instruction jumps and the destinations, or the connection between called functions and where they were called from. This combined graph representation is made by using IR that has been normalized using inst2vec.

ProGraML does not compare itself with CDFG, but with both DeepTune and NCC. Here ProGraML achieved the best results when compared with the others in device mapping, and algorithm classification.

E. Public source code datasets and GH Archive

To make our large structured source code dataset from GitHub we use GH Archive, which is a public archive of GitHub audit logs from the period of 2011 to now. Each log includes the JSON-encoded events reported by the GitHub API. The entire GH Archive is also available as a public dataset on Google BigQuery, enabling SQL searches on the dataset.

IV. Data generation pipeline

A. Source code gathering

The GH Archive dataset consist of several fields, the important ones in this case are the repository URL and payload. The payload contains meta data, and for instance the repository languages that have been used. To find which of the repositories that have a connection with CUDA or is CUDA related an SQL query to match the word CUDA can

be used. In total there were 18534 unique repositories with the keyword CUDA.

There are several different ways to get the source code from GitHub. GitHub has multiple APIs that can be used by for instance Python. It is also possible to script the GitHub cloning function. The last way to retrieve the data is by downloading a repository as a zip file, which can also be automated by Python. Both methods were evaluated and timed, to find the faster one. Downloading as a zip using Python proved to be around twice as fast as cloning.

The script that downloads the repositories creates a folder structure that corresponds to the indices in the repository URL file. Repositories from GitHub more often than not contains unneeded file types, by filtering out files based on file ending. Each repository were left with just C, C++, CUDA files and headers for the different languages.

The total amount of downloaded repositories were lower than the amount of URLs, which is explained by either the repositories being changed from public to private or that the repositories have been removed. In total 16247 repositories were downloaded successfully.

B. Combining and executing code

This section describes how we formatted and made the CUDA source code runnable.

To get any type of performance results from a repository, one might naively try to compile and execute the repository. This, however, is not a simple process at all – especially if this process needs to be automated. The first part of compiling the repository is very dependent on a good Makefile or something similar, and even then any reliance on an external library would halt the automation significantly.

Out of a hundred random repositories, only three managed to compile ”out of the box”. Even if all the repositories compiled flawlessly, an even bigger problem arises. How would one measure the performance, and the impact of modifications, on a random program? One might measure the entire program execution, or each CUDA kernel, but this would require to know specific program characteristics especially if the program had any kind of user interference.

The solution to both of these issues were to find, isolate, and build each CUDA global function. Then each CUDA kernel could be run on its own, and measured accurately, with different modifications. The scripting for this process was done in Python, due to its ease of interacting with folders, files, and text. This is a multi-step process which can be summarized as follows:

- Identify all global and device functions in repository.
- For each global function, find all relevant includes and device functions.
- Store global function as new file in new sub-folder.
- For each include any iterative dependency is found and added to the sub-folder.
- Find, format and store all global input parameters as list of tuples with names and types.
- Lastly, a generator uses the parameter information and naming to create a main file that can initialize all the needed variables.
- This main file can then issue a call to the global function.

C. Sampling and restructuring

Even though the kernels are built, they still need to run, and additional work is needed to identify working kernels. With extra measures taken to fix broken kernels. To find out if a kernel has been properly isolated and can also be executed, a script is responsible for restructuring and compiling all the isolated kernels. With Python sub-processes, Python can execute any Linux command, including using the NVCC compiler and executing the kernels.

This process of repeatedly fixing and compiling in Fig. 1 increased the amount of working kernels in the trial phase from around 22% to 29%.

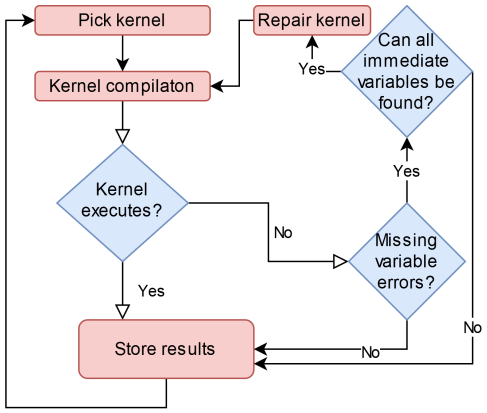


Fig. 1: Restructuring and sampling

After the time consuming process of compiling and fixing the potential kernels was done, 20 251 isolated kernels were compiling and could be executed as is. Additionally since all the errors were stored, potential improvements in the process of isolating the kernels could be identified by looking through the error messages.

To get the results a new script was made which purpose was to modify the variables belonging to the kernel, compiling and executing, reading the outputted results and storing them.

D. Benchmarking

Our experiments showed significant variations in run time for the same program. These were likely caused by external uncontrollable hardware and system processes. Bad results could make a machine learning model learn the wrong features. To mitigate some of these discrepancies, the official NVIDIA benchmark guide was followed [13].

This guide is divided into two parts, locking hardware GPU and CPU clock speeds, and preheating the GPU. To preheat the GPU the program has to be executed once before any measurement is done on the actual program execution. This preheating ensures that the GPU spends minimal time going

from an idle to active state, as this transition would impact run time. Similarly locking the hardware clock speed ensures that each program execution is stable without any boosting. Both of these changes decreases performance, but increases the accuracy of run time readings. On top of this the kernel itself is executed 1000 times, to minimize the variation of single run times.

The NVIDIA guide improved the accuracy slightly, but there would still be some run time outliers large enough to impact the coherency of the result data. One way of tackling this is running the program several times, and aggregating the result again, in a way that makes the outputted result the most stable. Five different simple methods were tested out, by creating a dataset of 100 000 run times, then randomly selecting ten and aggregating, by repeating this process the variations after aggregation can be measured. Out of the different aggregation methods tested the median performed the best with a variation of around one percent.

V. Experimental Setup

We did initial tests of our dataset using a desktop Nvidia GeForce GTX 980 card based on Nvidia´s Maxwell architecture with 2048 CUDA Cores and 4GB of memory. We also used a newer system with 20 Nvidia T4s based on Nvidia´s Turing architecture with 2560 cores and 16GB memory each.

A. Choice of parameters

Of all the different parameters that could be tested the most reasonable ones to test, were matrix sizes and thread block sizes. This makes it also possible to compare with past OpenCL projects [7]–[10].

To find parameter values that would be both reasonable to pick, and not too exhaustive as this would drastically increase the run-time, both the CUDA guidelines and similar projects were taken into consideration.

The official CUDA guidelines, suggests using thread block sizes divisible by 32 and as large as possible. The largest block is 1024, and all blocks should be 32 divisible, leaving 32 different block sizes to be tested out, which is quite high.

Lim et al. [14] used the blocks in range of 0-1024 which were divisible by 64, this should be enough to achieve good results while also halving the amount of search space required. Additionally the 2D blocks of sizes (8,8) (16,16), (24,24) and (32,32) were also tested. To find a varying set of matrix dimensions some inspiration was taken from the Intel guide ”measure the performance of matrix multiplications by dimensions” [15]. In the end there were seven matrix sizes and twenty thread block sizes for a total of 140 combinations.

B. Kernel executor

The last step is the execution of all the runnable kernels, with all the configurations needed. As the kernels were filtered into the ones that could execute and those that could not, the next step was modifying and setting variables, so that results could be extracted for every parameter combination.

The variable values were set mainly by their name, as the name-space proved to be a significant indicator for what kind of values they were supposed to take, w for width for instance or n for total size. The variables are split into either being set to the width, height, size, or 1 as some variables such as stride should remain static and within a reasonable range. Most, around 90% of the variables, were covered by either width, height, or size or any other common name-space used in scientific applications, like k, the rest were also just set to one.

The script decides first what type of template to use, and compiles the kernel using that template. After compilation the output executable can be executed with the Python sub-process, and each result is stored in a Pandas dataframe with the parameter combination, path, function name and result.

An important note is that when using the Python sub-process module the Linux timeout functionality should be used. The original reason it was implemented was to timeout kernel executions that were ”hanging” and taking significantly longer time than other kernels. This does in effect filters out some of the kernels that might take too long to execute, and a potential ratio of how many kernel runs can be achieved vs the total time cost. For two seconds of timeout around one third of the kernels had enough time to execute, and at 120 seconds timeout only 1 in 500 did not execute, as a lower reasonable value, 30 seconds of timeout was used for the final multi GPU run. While two seconds as a test, were used on the GTX 980 system. A full execution on the GTX 980 took approximately 97 hours.

The cudaOccupancyMaxPotentialBlockSize is Nvidias own solution to find the optimal thread block size for a kernel, and was tested to see if it did indeed find the best size. These results could also be used in comparison with the final product, to do a comparative evaluation of a potential machine learnt model and this API. The cudaOccupancyMaxPotentialBlock-Size API took significant less amount of time to execute, but was stopped halfway as the results indicated no result difference across matrix size.

The next step was executing this script with the same source data on our Nvidia Tesla T4 system. The T4 system has 20 GPUs. By using cudaSetDevice, the additional GPUs could run the script in parallel.

VI. Results

A. GTX 980 System

After the first full execution on the GTX 980 system we generated a dataset with 2140796 rows, and 97% non NaN data.

We can see from Fig. 2 the downward trend in execution time caused by increase in thread block size, which both substantiate the claims by Nvidia regarding picking larger thread block sizes being the way to go, and that the results that were achieved were realistic.

Now if the average was the best indicator and the variance between kernels thread block performance was low, then just picking a large thread block would always be the optimal choice. However if the graph of just one kernel is shown Fig. 3, this no longer seems to be the case.

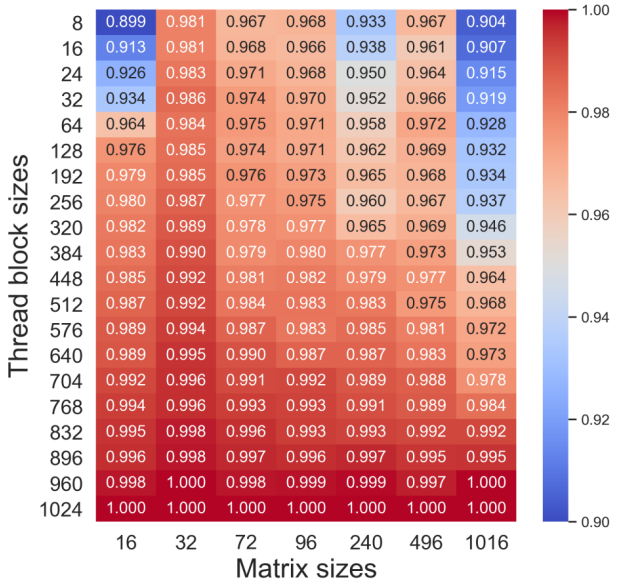


Fig. 2: GTX 980 average performance of thread block sizes on matrix sizes

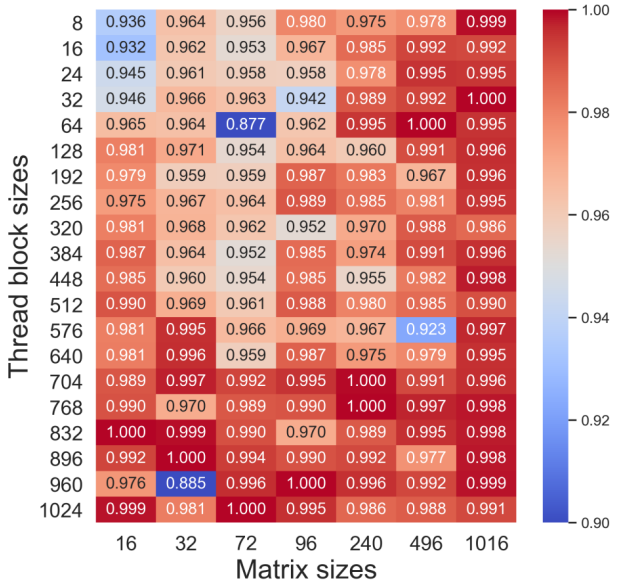


Fig. 3: GTX 980 performance of thread block sizes on matrix sizes on the kernel euclidean_kernel

In this case the best performing thread block was not the largest one, and this was also true for 83% of the kernels. The largest block did perform on average of 98.7% of the best block, but in around 1% of the kernels this ranges from 40 to 85%, which would signify a large performance increase, if the better block size was used instead.

B. T4 System

There was some small changes made to the matrix sizes, and an increase in timeout factor which did increase the amount of kernels with a run-time to 19683.

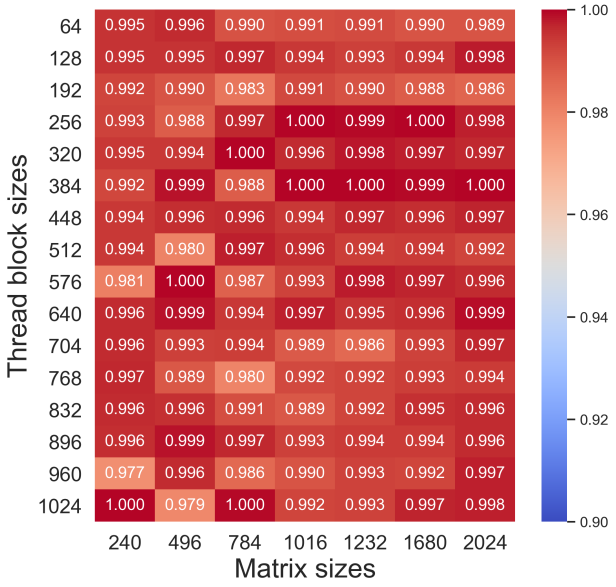


Fig. 4: T4 Average performance of thread block sizes on matrix sizes

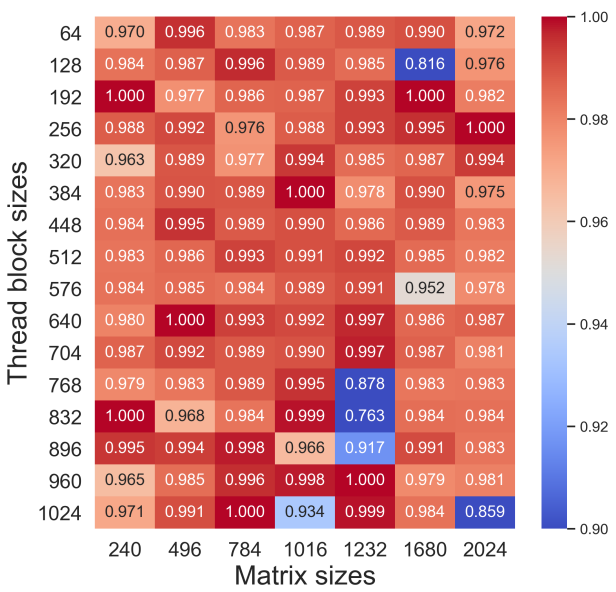


Fig. 5: T4 performance of thread block sizes on matrix sizes on the kernel euclidean_kernel

For the average kernel thread performance for each matrix size, there was quite a difference compared to GTX 980 Fig. 4.

No longer is the average 1024 the best in all the average cases, which might either be due to increase in the amount of kernels which were recorded, or the change in hardware.

Results from the same single kernel as in GTX 980, Fig. 5.

It becomes quite evident that 1024 is not the perfect size in all cases, and a closer look shows that the 1024 block size

has a 86% performance compared to the best block. In 12% of the cases a 1024 block would perform less than 85% in comparison with the optimal block.

VII. Discussion

Compared to the other related works which are based upon Cummins et al. [7] original dataset, our LS-CAT dataset is significantly larger. We present over 19 683 kernels compared to the Cummins et al. dataset of 256 kernels [7]. The amount of variance in our dataset's programming style should also be impactful.

The range of different "topics" covered by the sheer difference in kernel amount would also be of help to more realistically evaluate the efficiency of machine learning on code, as unsupervised topic modeling could be used to see if some kernel topics are easier to evaluate than others.

To actually check the quality of the LS-CAT dataset for machine learning purposes, the dataset has to be used with a thread coarsening task in future works. If the machine learnt model performs well, the dataset is of sufficient quality. If the model performs poorly then either the dataset, or the methodology of using kernel code to find thread blocks is insufficient.

If the LS-CAT dataset is proven to be insufficient, for whatever reason, there is enough modularity in the data processing pipeline and enough error information, from stored logs, to reconfigure the process to either increase data volume or quality. The public LS-CAT dataset will be updated in this case.

Another key finding was that 1024 proved to be a reasonable thread size for most cases, it might therefore be easier to identify the kernels for which this is not the case, than to find the best block for each kernel.

VIII. Conclusions and Future Work

In machine learning (ML), access to large enough datasets are very important for how well the resulting models perform. Previous work on using ML for autotuning CUDA codes, have had mixed results due to the lack of access to suitable code databases.

In this paper, we described how we generated a large-scale real-world dataset of CUDA kernels (LS-CAT[1]) for the purpose of training NLP-based ML-models for autotuning. The kernels were constructed by using source codes from GitHub via the GH Archive project [12].

We successfully downloaded 16 247 projects, out of the 18 534 projects that GH Archive showed as available, after pruning old and non-existent ones. Out of these, 20 251 runnable kernels were generated and compiled, and out of them again, 19 683 have results that we could use as a database of runable CUDA codes.

In addition to the CUDA codes, our LS-CAT dataset contains 5 028 536 associated runtimes (including both GTX 980 and T4 results), with different combinations of kernels, block sizes and matrix sizes.

[1]Available through https://www.ntnu.edu/idi/hpc-lab/

Our experimental results coincided with what NVIDIA themselves have found, that increase in thread block size is usually enough, however, this is only true for the average case.

The results also indicate that always picking the optimal block over the largest, would net a 6% performance increase on average, and in 10% of the cases more than 20% performance increase can be achieved by using the optimal block. Both of these findings are promising.

The cudaOccupancyMaxPotentialBlockSize API was also tested to some extent, but proved insensitive to matrix sizes which does play a role in choice of blocks.

Current and future work includes testing our LS-CAT dataset using NLP-ML models.

Acknowledgment

The authors would like to thank the Department of Computer Science and its HPC-Lab at NTNU for their support which enabled this project. The authors and the SFI Center for Geophysical Forecasting also plan to build on our LS-CAT dataset.

References

[1] Puschel, M. and Moura, J.M.F. and Johnson, J.R. and Padua, D. and Veloso, M.M. and Singer, B.W. and Jianxin Xiong and Franchetti, F. and Gacic, A. and Voronenko, Y. and Chen, K. and Johnson, R.W. and Rizzolo, N., "SPIRAL: Code Generation for DSP Transforms," Proceedings of the IEEE, 232–275, February 2005.

[2] Hartono, Albert and Norris, Boyana and Sadayappan, P., Annotation-based empirical performance tuning using Orio, IEEE International Symposium on Parallel Distributed Processing, 2009, pp.1–11.

[3] Falch, Thomas L., and Anne C. Elster. "Machine Learning-Based Auto-Tuning for Enhanced Performance Portability of OpenCL Applications." Concurrency and Computation: Practice and Experience 29, no. 8 (2017): e4029. https://doi.org/10.1002/cpe.4029.

[4] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, Cristina Silvano "A Survey on Compiler Autotuning using Machine Learning," ACM Computing Surveys 2018.

[5] Zhang, W., Hao, M. & Snir, Marc "Predicting HPC parallel program performance based on LLVM compiler." Cluster Comput 20, 1179–1192 2017.

[6] Mishra, Alok and Malik, Abid M and Chapman, Barbara, "Using Machine Learning for OpenMP GPU Offloading in LLVM,", ACM SRC to be held at SC20, 2020.

[7] Cummins, Chris and Petoumenos, Pavlos and Wang, Zheng and Leather, Hugh, "End-to-End Deep Learning of Optimization Heuristics,", IEEE, September 2017, p219–232.

[8] Cummins, Chris and Fisches, Zacharias V. and Ben-Nun, Tal and Hoefler, Torsten and Leather, Hugh, "ProGraML: Graph-based Deep Learning for Program Optimization and Analysis,", March 2020.

[9] Ben-Nun, Tal and Jakobovits, Alice Shoshana and Hoefler, Torsten, "Neural Code Comprehension: A Learnable Representation of Code Semantics,", IEEE, November 2018, p219–232.

[10] Brauckmann, Alexander and Goens, Andrés and Ertel, Sebastian and Castrillon, Jeronimo, "Compiler-based graph representations for deep learning models of code,", ACM, February 2020, p201–211.

[11] Kuznetsov, Evgeny & Kondratyuk, Nikolay & Logunov, Mikhail & Nikolskiy, Vsevolod & Stegailov, Vladimir. "Performance and Portability of State-of-Art Molecular Dynamics Software on Modern GPUs." 2020.

[12] Ilya Grigorik "GH Archive.", Accessed January 31, 2021. https://www.gharchive.org/.

[13] Bill Fiser, Sebastian Jodłowski "BEST PRACTICES WHEN BENCHMARKING CUDA APPLICATIONS", accessed: 01.10.2020

[14] Lim, Robert V. and Norris, Boyana and Malony, Allen D. "Autotuning GPU Kernels via Static and Predictive Analysis", 2017

[15] Ying Hu, Shane A Story, "Tips to Measure the Performance of Matrix Multiplication Using Intel® MKL", accessed: 07.011.2020

Appendix C

Source Code

```
import collections
import datetime
import pickle
from utils.utils3 import *
from utils.data_loader3 import load_data
#from model.lstm import LSTM
from model.lstm import LSTM
from config.default import cfg
from model.multiattention2f import seq2seq
from model.transformers import Transformer
from sklearn.preprocessing import MinMaxScaler
import torch
import pathlib
import matplotlib.pyplot as plt
import time
import typing
from torch import nn
import os
import gc
import numpy as np
import collections
import matplotlib.pyplot as plt
from torch.autograd import Variable
import sys
from sklearn.metrics import precision_recall_curve
class Trainer:

    def __init__(self, cfg):
        """
            Initialize our trainer class.
        """

        self.cfg=cfg
        self.device = get_default_device()
        self.model = to_device(seq2seq(self.device),self.device)
        self.checkpoint_dir = pathlib.Path("outputs")

        self.loss_criterion = nn.BCEWithLogitsLoss()

        # Define our optimizer. SGD = Stochastich Gradient Descent
        #self.optimizer=torch.optim.SGD(self.model.parameters(), lr=cfg.TRAINER.LR,
            ↪  weight_decay=cfg.TRAINER.WEIGHT_DECAY,momentum=cfg.TRAINER.
```

```
            ↪ MOMENTUM)
        self.optimizer=torch.optim.Adam(self.model.parameters(), lr=cfg.TRAINER.LR,
            ↪  weight_decay=cfg.TRAINER.WEIGHT_DECAY)
        # Load our dataset


        self.dataloader_val = load_data(cfg,'test',self.device)
        self.load_last_model()
    def prec_rec_curve(self, name):
        self.model.eval()

        labels = []
        predictions = []
        with torch.no_grad():
            for (X_batch,lengths, aux_batch, Y_batch, Y2_batch) in self.
                ↪ dataloader_val:
                output_probs = self.model(X_batch.float(),lengths.float(),aux_batch
                    ↪ )
                labels.append(Y_batch.cpu().detach().numpy())
                predictions.append(torch.sigmoid(output_probs).cpu().detach().numpy
                    ↪ ())
        labels = np.stack(labels, axis=0 ).flatten()


        predictions = np.stack(predictions, axis=0 ).flatten()

        precision, recall, thresholds = precision_recall_curve(labels,predictions)

        plot_path = pathlib.Path("plots")
        plt.title("Precision␣recall")
        plt.plot(recall,precision)
        plt.savefig(plot_path.joinpath(f"{name}_plot.png"))


    def validate(self,tresh):
        self.model.eval()
        average_loss = 0
        accuracy = 0
        precisions = 0
        recalls = 0
        counter = 0
        first = True
        with torch.no_grad():
            for (X_batch,lengths, aux_batch, Y_batch, Y2_batch) in self.
                ↪ dataloader_val:
                output_probs = self.model(X_batch.float(),lengths.float(),aux_batch
                    ↪ )
                counter+=1

                #_, predicted = torch.max(output_probs, 1)
                ones=to_device(torch.ones(output_probs.shape),self.device)
                zeros=to_device(torch.zeros(output_probs.shape),self.device)
                treshed=torch.where(torch.sigmoid(output_probs) > tresh, ones,
                    ↪ zeros)
                predicted=treshed+to_device(2*torch.arange(len(ones)),self.device)
                # Compute Loss and Accuracy
                accuracy += Y2_batch.flatten()[predicted.long()].mean()
                if first:
                    print("Output␣probabilities")
                    print(torch.sigmoid(output_probs[:10]).cpu().detach().numpy())
```

```
                    print("Tresholded")
                    print(treshed[:10].cpu().detach().numpy())
                    print("Indiced")
                    print(predicted[:10].cpu().detach().numpy())
                    print("Targets")
                    print(Y2_batch[:10].cpu().detach().numpy())
                    print("Selected values")
                    print(Y2_batch.flatten()[predicted.long()][:10].cpu().detach().
                        ↪ numpy())
                    #print(output_probs.shape, predicted.shape, Y2_batch.shape)
                    first=False
        print(accuracy/counter)
    def load_last_model(self):
        state_dict = load_last_checkpoint(self.checkpoint_dir)
        if state_dict is None:
            print(
                f"Could not load best checkpoint. Did not find under: {self.
                    ↪ checkpoint_dir}")
            return
        self.model.load_state_dict(state_dict)

    def load_best_model(self):
        state_dict = load_best_checkpoint(self.checkpoint_dir,self.cfg.DATASETS.
            ↪ NAME)
        if state_dict is None:
            print(
                f"Could not load best checkpoint. Did not find under: {self.
                    ↪ checkpoint_dir}")
            return
        self.model.load_state_dict(state_dict)


if __name__ == '__main__':
    print(sys.argv[:1])
    trainer = Trainer(cfg)
    if sys.argv[1]=="rec":
        trainer.prec_rec_curve(sys.argv[2])
    else:
        trainer.validate(float(sys.argv[2]))
```

Code listing C.1: Evaluation script for the binary classifier

```
import fasttext
import glob
import numpy as np
import re
def create_embeds():

    model = fasttext.load_model("../outputs/embedder.bin")

    path = "../data/raw/ir/"

    files = glob.glob(path+"*.ll")
    Lines = []
    for f in files:

        file1 = open(f, 'r')
        lines = []
        flag=False
        for l in (file1.readlines()):
```

```
            l=l.rstrip("\n")
            if l.find("}")!=-1:
                flag=False
            if flag:
                l=re.sub(r'\[.*?\]', lambda x: ''.join(x.group(0).split()), l)
                l=re.sub(r'\(.*?\)', "", l)
                #l=re.sub(r'\d', "", l)
                l=re.sub('\(', '␣',l)
                l=re.sub('\)', '␣',l)
                l=re.sub(',', '␣',l)
                l=re.sub('␣+', '␣',l)
                lines.append(l)
            if(l.find("define␣dso_local␣void")!=-1):
                flag=True

        Lines = "␣".join(lines)
        Lines = Lines.split("␣")
        to_save=np.array([model.get_word_vector(x) for x in Lines])
        print(f)
        name=f.split("/")[4].split(".")[0]
        np.save("../data/raw/embedded/"+name+".npy", to_save)


create_embeds()
```

Code listing C.2: Creation of the embeds

```
import fasttext
import glob
import re
def create_data():
    path = "../data/raw/ir/"

    files = glob.glob(path+"*.ll")
    Lines = []
    for f in files:
        file1 = open(f, 'r')
        lines = []
        flag=False
        for l in (file1.readlines()):
            l=l.rstrip("\n")
            if l.find("}")!=-1:
                flag=False
            if flag:
                l=re.sub(r'\[.*?\]', lambda x: ''.join(x.group(0).split()), l)
                l=re.sub(r'\(.*?\)', "", l)
                #l=re.sub(r'\d', "", l)
                l=re.sub('\(', '␣',l)
                l=re.sub('\)', '␣',l)
                l=re.sub(',', '␣',l)
                l=re.sub('␣+', '␣',l)
                lines.append(l)
            if(l.find("define␣dso_local␣void")!=-1):
                flag=True

        Lines.append(re.sub('␣+', '␣',"␣".join(lines)))

    with open('data.txt', 'w') as f:
        for item in Lines:
            f.write("%s\n" % item)
```

```
create_data()
model = fasttext.train_unsupervised('data.txt', model='skipgram', dim=40,epoch=10)
model.save_model("../outputs/embedder.bin")
```

Code listing C.3: The FastText embed trainer class

```
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import fasttext
from utils.utils import *
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size

        # Initialize GRU; the input_size and hidden_size params are both set to '
            ↪ hidden_size'
        #   because our input size is a word embedding with number of features ==
            ↪ hidden_size
        self.gru = nn.LSTM(embedding, hidden_size, n_layers,
                           dropout=(0 if n_layers == 1 else dropout), bidirectional=
                               ↪ True)

    def forward(self, input_seq, input_lengths, hidden=None):
        # type: (Tensor, Tensor, Optional[Tensor]) -> Tuple[Tensor, Tensor]
        # Convert word indexes to embeddings

        # Pack padded batch of sequences for RNN module
        packed = torch.nn.utils.rnn.pack_padded_sequence(input_seq, input_lengths,
            ↪ enforce_sorted=False)
        # Forward pass through GRU
        outputs, hidden = self.gru(packed, hidden)
        # Unpack padding
        outputs, _ = torch.nn.utils.rnn.pad_packed_sequence(outputs)
        # Sum bidirectional GRU outputs
        outputs = outputs[:, :, :self.hidden_size] + outputs[:, : ,self.hidden_size
            ↪ :]
        # Return output and final hidden state
        return outputs, hidden


class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0.1):
        super(DecoderRNN, self).__init__()

        # Keep for reference
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.dropout = dropout

        # Define layers
```

```
        self.gru = nn.LSTM(embedding, hidden_size, n_layers, dropout=(0 if n_layers
            ↪  == 1 else dropout))

    def forward(self, encoder_outputs, last_hidden,last_c):


        # Forward through unidirectional GRU

        rnn_output, hidden = self.gru(encoder_outputs, (last_hidden,last_c))


        return rnn_output, hidden



class seq2seq(nn.Module):
    def __init__(self,device):
        super(seq2seq, self).__init__()

        hidden_size=40
        embedding=40
        output_size=40

        self.attention = nn.MultiheadAttention(embedding,4)
        self.encoder1 = to_device(EncoderRNN(hidden_size,embedding,n_layers=1),
            ↪ device)
        self.decoder1 = to_device(DecoderRNN(hidden_size,embedding,n_layers=1),
            ↪ device)

        self._device = device
        self._decoder_n_layers = 1

        self.embed=nn.Embedding(7, 50)
        h_size=500
        self.linear1=nn.Linear(2*12000,h_size-50)
        self.seq=create_linear_layers(10,h_size)
        self.linear4=nn.Linear(h_size,25)
        self.linear5=nn.Linear(25,4)

        self.act = nn.ELU()
        self.bn1 = nn.BatchNorm1d(h_size-50)

        self.bn4 = nn.BatchNorm1d(25)

        self.out_act = nn.Softmax(dim=1)
        nn.init.xavier_uniform_(self.linear1.weight)
        self.seq.apply(weights_init)
        nn.init.xavier_uniform_(self.linear4.weight)
        nn.init.xavier_uniform_(self.linear5.weight)

    def forward(self, input_seq : torch.Tensor, input_length : torch.Tensor,aux):
        #input_seq sequence of tokens embedded, input_length length of sequences,
            ↪ aux, the matrix size vector
        input_length=input_length.cpu()
        length = int(np.max(input_length.numpy()))


        encoder_outputs1, encoder_hidden1 = self.encoder1(torch.transpose(input_seq
            ↪ , 0, 1), input_length)
```

```
        attn_output, _ = self.attention(encoder_outputs1, encoder_outputs1,
            ↪ encoder_outputs1)
        encoder_outputs1 = attn_output * encoder_outputs1
        e_h1, e_c1 = encoder_hidden1

        # Prepare encoder's final hidden layer to be first hidden input to the
            ↪ decoder
        decoder_hidden1 = e_h1[:self._decoder_n_layers]
        decoder_c1 = e_c1[:self._decoder_n_layers]

        decoder_output1, decoder_hidden1 = self.decoder1(encoder_outputs1,
            ↪ decoder_hidden1,decoder_c1)




        out = torch.transpose(encoder_outputs1,0,1)

        out = out.contiguous().view(128,-1)

        x = to_device(torch.zeros(128,2*12000),self._device)
        x[:,:length*40] = out

        out = self.linear1(x)
        out = self.act(out)
        out = self.bn1(out)

        aux = self.embed(aux)
        out = torch.cat((out,aux),1)

        out = self.seq(out)

        out = self.linear4(out)
        out = self.act(out)
        out = self.bn4(out)
        #aux = self.embed(aux)

        #out = torch.cat((out, aux),1)
        out = self.linear5(out)
        #out = self.out_act(out)

        return out

def create_linear_layers(n,size):
    layers = []
    for i in range(n):
        layers.append(nn.Linear(size,size))
        layers.append(nn.ELU())
        layers.append(nn.BatchNorm1d(size))
    return nn.Sequential(*layers)
def weights_init(m):
    if isinstance(m, nn.Linear):
        torch.nn.init.xavier_uniform_(m.weight)
```

Code listing C.4: The model class

```
import torch
import pandas as pd
import numpy as np
from utils.utils import to_device
```

```
def one_hot(a,a_max):
    a = a.astype("int")
    b = np.zeros((a.size, a_max+1))
    b[np.arange(a.size),a] = 1
    return b



def load_data(cfg,prefix,device):
    """
        Loads the labels from the judge test train sets transforms labels into
            ↪ numeric values creates a dataset and returns the dataset
    """
    df=None

    if(prefix=='train'):
        df = pd.read_csv('data/train/'+cfg.DATASETS.NAME+'_train.csv', low_memory =
            ↪  False) #[['path', 'matrix_id', 'time_14', 'time_15', 'diff_14','
            ↪ diff_15']]
    elif(prefix=='val'):
        df = pd.read_csv('data/val/'+cfg.DATASETS.NAME+'_val.csv', low_memory =
            ↪ False) #[['path', 'matrix_id','time_14', 'time_15', 'diff_14','
            ↪ diff_15']]
    elif(prefix=='test'):
        df = pd.read_csv('data/test/'+cfg.DATASETS.NAME+'_test.csv', low_memory =
            ↪ False) #[['path', 'matrix_id', 'time_14', 'time_15','diff_14','
            ↪ diff_15']]



    dataset = Dataset(df,cfg)
    data_set=torch.utils.data.DataLoader(dataset,batch_size=cfg.TRAINER.BATCH_SIZE,
        ↪  shuffle=True, num_workers=2,drop_last=True )
    data_set=DeviceDataLoader(data_set,device)
    return data_set


class Dataset(torch.utils.data.Dataset):
  'Characterizes␣a␣dataset␣for␣PyTorch'
  def __init__(self,df,cfg):
        'Initialization'
        self.df=df
        self.length=len(df)
        self.task=cfg.DATASETS.NAME


  def __len__(self):
        'Denotes␣the␣total␣number␣of␣samples'

        return self.length

  def __getitem__(self, index):
        'Generates␣one␣sample␣of␣data'
        # Select sample
        # Load data and get label
        path=self.df["path"].iloc[index]
        path="_".join(path.split("/")[1:3])
        aux = int(self.df["matrix_id"].iloc[index])
        Y=self.df.iloc[index].to_numpy()
```

```
        Y1=Y[6:10].astype(float) #2:8
        #print(Y1)
        #print(len(Y1))
        #Y1=Y1/Y1.max()
        #Y1 = np.exp(Y1 *5)
        #Y1 /= np.sum(Y1)
        Y2=Y[10].astype(float)

        X = np.zeros((600,40))
        t = np.load('data/raw/embedded/'+path+".npy")
        X[:len(t),:]=t
        return X,len(t),aux,Y1, Y2


class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)
    def __len__(self):
        """Number of batches"""
        return len(self.dl)
```

Code listing C.5: The data loader class

```
import torch
import matplotlib.pyplot as plt
import numpy as np
import pathlib
np.random.seed(0)
torch.manual_seed(0)

torch.backends.cudnn.deterministic = True
from sklearn.metrics import recall_score, precision_score

torch.cuda.is_available()

def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')

def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)



def plot_loss(Z: int,loss_dict: dict, label: str = None, fmt="-"):
    """
```

```
    Args:
        loss_dict: a dictionary where keys are the global step and values are the
            ↪ given loss / accuracy
        label: a string to use as label in plot legend
    """
    global_steps = list(loss_dict.keys())
    loss = list(loss_dict.values())
    plt.plot(global_steps, loss, fmt, label=label,zorder=Z)


def save_checkpoint(state_dict: dict,
                    filepath: pathlib.Path,
                    is_best: bool,
                    max_keep: int = 1,
                    dataset_name: str = "unnamed"):
    """
    Saves state_dict to filepath. Deletes old checkpoints as time passes.
    If is_best is toggled, saves a checkpoint to best.ckpt
    """
    filepath.parent.mkdir(exist_ok=True, parents=True)
    list_path = filepath.parent.joinpath("latest_checkpoint")
    torch.save(state_dict, filepath)
    if is_best:
        torch.save(state_dict, filepath.parent.joinpath(dataset_name+".ckpt"))
    previous_checkpoints = get_previous_checkpoints(filepath.parent)
    if filepath.name not in previous_checkpoints:
        previous_checkpoints = [filepath.name] + previous_checkpoints
    if len(previous_checkpoints) > max_keep:
        for ckpt in previous_checkpoints[max_keep:]:
            path = filepath.parent.joinpath(ckpt)
            if path.exists():
                path.unlink()
    previous_checkpoints = previous_checkpoints[:max_keep]
    with open(list_path, 'w') as fp:
        fp.write("\n".join(previous_checkpoints))


def get_previous_checkpoints(directory: pathlib.Path) -> list:
    assert directory.is_dir()
    list_path = directory.joinpath("latest_checkpoint")
    list_path.touch(exist_ok=True)
    with open(list_path) as fp:
        ckpt_list = fp.readlines()
    return [_.strip() for _ in ckpt_list]

def load_last_checkpoint(directory: pathlib.Path):
    loc = get_previous_checkpoints(directory)
    print(loc)
    filepath = directory.joinpath(loc[-1])
    if not filepath.is_file():
        return None
    return torch.load(directory.joinpath(loc[-1]))


def load_best_checkpoint(directory: pathlib.Path,dataset):
    filepath = directory.joinpath(dataset+".ckpt")
    if not filepath.is_file():
        return None
    return torch.load(directory.joinpath(dataset+".ckpt"))
def plot_loss(loss_dict: dict, label: str = None, fmt="-"):
```

```
    """
    Args:
        loss_dict: a dictionary where keys are the global step and values are the
            ↪ given loss / accuracy
        label: a string to use as label in plot legend
    """
    global_steps = list(loss_dict.keys())
    loss = list(loss_dict.values())
    plt.plot(global_steps, loss, fmt, label=label)



def compute_loss_and_accuracy(
        dataloader: torch.utils.data.DataLoader,
        model: torch.nn.Module,
        loss_criterion: torch.nn.modules.loss._Loss,
        device):
    """
    Computes the average precisions, recalls, loss and the accuracy over the whole
        ↪ dataset
    in dataloader.
    Args:
        dataloder: Validation/Test dataloader
        model: torch.nn.Module
        loss_criterion: The loss criterion, e.g: torch.nn.CrossEntropyLoss()
    Returns:
        [average_loss, accuracy, precision, recall]: both scalar.
    """
    average_loss = 0
    accuracy = 0
    precisions = 0
    recalls = 0
    counter = 0
    first = True
    with torch.no_grad():
        for (X_batch,lengths, aux_batch, Y1_batch, Y2_batch) in dataloader:
            output_probs = model(X_batch.float(),lengths.float(),aux_batch)
            counter+=1


            average_loss+=loss_criterion(output_probs,Y1_batch.float())
            #_, predicted = torch.max(output_probs, 1)


            if first:
                first=False
                #print(output_probs.shape)
                print(output_probs[0])
                print(Y1_batch[0])
            # Compute Loss and Accuracy
            accuracy += Y1_batch.flatten()[output_probs.argmax(dim=1)+output_probs.
                ↪ shape[1]*torch.arange(output_probs.shape[0])].mean()
    return average_loss/counter, accuracy/counter
```

Code listing C.6: The utility class

```
import collections
import datetime
import pickle
from utils.utils2 import *
```

```
from utils.data_loader2 import load_data
#from model.lstm import LSTM
from model.lstm import LSTM
from model.reg_enc_dec_attn import seq2seq
from model.transformers import Transformer
from sklearn.preprocessing import MinMaxScaler
import torch
import pathlib
import matplotlib.pyplot as plt
import time
import typing
from torch import nn
import os
import gc
import numpy as np
import collections
import matplotlib.pyplot as plt
from torch.autograd import Variable


class Trainer:

    def __init__(self, cfg, name):
        """
            Initialize our trainer class.
        """
        self.batch_size = cfg.TRAINER.BATCH_SIZE
        self.epochs = cfg.TRAINER.EPOCH
        self.plot_name=name
        self.cfg=cfg
        self.device = get_default_device()
        self.model = to_device(seq2seq(self.device),self.device)


        self.loss_criterion = nn.MSELoss() #reduction="sum")
        self.extra_loss = nn.CrossEntropyLoss()
        # Define our optimizer. SGD = Stochastich Gradient Descent
        #self.optimizer=torch.optim.SGD(self.model.parameters(), lr=cfg.TRAINER.LR,
            ↪  weight_decay=cfg.TRAINER.WEIGHT_DECAY,momentum=cfg.TRAINER.
            ↪ MOMENTUM)
        self.optimizer=torch.optim.Adam(self.model.parameters(), lr=cfg.TRAINER.LR,
            ↪  weight_decay=cfg.TRAINER.WEIGHT_DECAY)
        # Load our dataset

        self.dataloader_train = None
        self.dataloader_test = None
        self.dataloader_val = None

        self.alpha = 0.5
        self.gamma = 0.01

        # Tests model for each five steps
        self.num_steps_per_test = 150
        self.num_steps_per_val = 500
        self.global_step = 0
        self.start_time = time.time()

        # Tracking variables
        self.TEST_LOSS = collections.OrderedDict()
        self.TRAIN_LOSS = collections.OrderedDict()
```

```
        #self.TRAIN_CL = collections.OrderedDict()
        self.TEST_ACC = collections.OrderedDict()
        self.VAL_LOSS = collections.OrderedDict()
        self.VAL_ACC = collections.OrderedDict()

        self.checkpoint_dir = pathlib.Path("outputs")
        if cfg.TRAINER.SHOULD_TRAIN:
            self.dataloader_train = load_data(cfg,'train',self.device)
            self.dataloader_test = load_data(cfg,'test',self.device)
            self.dataloader_val = load_data(cfg,'val',self.device)
            if cfg.TRAINER.LOAD_LAST==1:
                self.load_best_model()
            elif cfg.TRAINER.LOAD_LAST==2:
                self.load_last_model()
            self.train(cfg)
        else:
            self.load_best_model()


    def predict(self,df,cfg):
        print("not implemented")


    def validation_epoch(self):
        """
            Computes the loss/accuracy and precision/recall/f1
            Train, validation and test.
        """
        self.model.eval()




        val_loss, val_acc = compute_loss_and_accuracy(
            self.dataloader_test, self.model, self.loss_criterion,self.device
        )
        used_time = time.time() - self.start_time

        self.VAL_ACC[self.global_step] = val_acc
        self.VAL_LOSS[self.global_step] = val_loss.cpu().detach().numpy()
        print(
            f"Epoch: {self.epoch:>2}",
            f"Batches per seconds: {self.global_step / used_time:.2f}",
            f"Global step: {self.global_step:>6}",
            f"Validation Loss: {val_loss:.3f},",
            f"Validation Accuracy: {val_acc:.3f}",
            sep="\t")

        self.model.train()

    def test_epoch(self):
        """
            Computes the loss/accuracy and precision/recall/f1
            Train, validation and test.
        """
        self.model.eval()
        test_loss, test_acc = compute_loss_and_accuracy(
            self.dataloader_test, self.model, self.loss_criterion,self.device
        )
        used_time = time.time() - self.start_time
```

```
        self.TEST_ACC[self.global_step] = test_acc
        self.TEST_LOSS[self.global_step] = test_loss.cpu().detach().numpy()
        print(
            f"Epoch:␣{self.epoch:>2}",
            f"Batches␣per␣seconds:␣{self.global_step␣/␣used_time:.2f}",
            f"Global␣step:␣{self.global_step:>6}",
            f"Test␣Loss:␣{test_loss:.3f},",
            f"Test␣Accuracy:␣{test_acc:.3f}",
            sep="\t")

        self.model.train()

    def train(self,cfg):
        """
        Trains the model for [self.epochs] epochs.
        """
        # Track initial loss/accuracy
        def should_test_model():
            return self.global_step % self.num_steps_per_test == 0
        def should_validate_model():
            return self.global_step % self.num_steps_per_val == 0

        for epoch in range(self.epochs):
            self.epoch = epoch
            # Perform a full pass through all the training samples
            for X_batch,lengths, aux_batch, Y1_batch,Y2_batch in self.
                ↪ dataloader_train:

                # Perform the forward pass
                #self.model.lstm1.init_hidden(cfg.TRAINER.BATCH_SIZE,self.device)

                #torch.cuda.empty_cache()


                predictions = self.model(X_batch.float(),lengths.float(),aux_batch)
                # Compute the cross entropy loss for the batch

                loss = self.loss_criterion(predictions, Y1_batch.float())
                los2 = self.extra_loss(predictions, Y2_batch.long())

                #loss, loss_shape, loss_temporal = dilate_loss(predictions,Y_batch.
                    ↪ float(),self.alpha, self.gamma, self.device)
                #loss = Variable(loss, requires_grad = True)

                #self.TRAIN_LOSS[self.global_step] = loss.cpu().detach().numpy()
                #self.TRAIN_CL[self.global_step] = los2.cpu().detach().numpy()
                #loss.backward()
                # Backpropagation
                self.optimizer.zero_grad()
                #los3=loss+2*los2
                los3=los2
                los3.backward()
                self.TRAIN_LOSS[self.global_step] = los3.cpu().detach().numpy()
                self.optimizer.step()
                self.global_step += 1
                 # Compute loss/accuracy for all three datasets.
                if(self.global_step%self.num_steps_per_test==0):
                    print(f"Train␣Loss:␣{loss:.3f},␣train␣CEL:␣{los2:.3f}")
                if should_test_model():
```

```
                    self.test_epoch()
                    self.save_model()
                if should_validate_model():
                    self.validation_epoch()

        create_plots(self, self.plot_name)


    def save_model(self):
        def is_best_model():
            """
                Returns True if current model has the lowest validation loss
            """
            validation_losses = list(self.TEST_LOSS.values())
            return validation_losses[-1] == min(validation_losses)

        state_dict = self.model.state_dict()
        filepath = self.checkpoint_dir.joinpath(f"{self.global_step}.ckpt")

        save_checkpoint(state_dict, filepath, is_best_model(),dataset_name=self.cfg
            ↪ .DATASETS.NAME)

    def load_last_model(self):
        state_dict = load_last_checkpoint(self.checkpoint_dir)
        if state_dict is None:
            print(
                f"Could not load best checkpoint. Did not find under: {self.
                    ↪ checkpoint_dir}")
            return
        self.model.load_state_dict(state_dict)

    def load_best_model(self):
        state_dict = load_best_checkpoint(self.checkpoint_dir,self.cfg.DATASETS.
            ↪ NAME)
        if state_dict is None:
            print(
                f"Could not load best checkpoint. Did not find under: {self.
                    ↪ checkpoint_dir}")
            return
        self.model.load_state_dict(state_dict)

    def display(self):
        self.model.eval()
        with torch.no_grad():
            for X_batch, Y_batch in self.dataloader_val:

                output_probs = self.model(X_batch.float())

                x = output_probs[0].cpu()
                y = Y_batch[0].cpu()


                return x, y


def create_plots(trainer: Trainer, name: str):

    plot_path = pathlib.Path("plots")
    plot_path.mkdir(exist_ok=True)
    # Save plots and show them
```

```
    plt.figure(figsize=(20, 8))
    plt.subplot(1, 2, 1)
    plt.title("Loss")
    plot_loss(trainer.TRAIN_LOSS, label="Training␣loss")
    plot_loss(trainer.TEST_LOSS, label="Test␣loss")
    plot_loss(trainer.VAL_LOSS, label="Val␣loss")
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.title("Accuracy")
    plt.title("Accuracy")
    plot_loss(trainer.VAL_ACC, label="Val␣accuracy")
    plot_loss(trainer.TEST_ACC, label="Test␣accuracy")
    plt.legend()
    f = open(plot_path.joinpath(f"{name}.pkl"),"wb")
    pickle.dump([trainer.TRAIN_LOSS,trainer.TEST_LOSS,trainer.VAL_LOSS,trainer.
        ↪ VAL_ACC,trainer.TEST_ACC],f)
    f.close()
    plt.legend()
    plt.savefig(plot_path.joinpath(f"{name}_plot.png"))
```

Code listing C.7: The trainer class

```

from trainer.trainer2 import Trainer
from config.default import cfg
import sys
import time
#raw2data(cfg)

#data_to_one_hots(cfg)
if __name__ == '__main__':
    print(sys.argv[:1])
    cfg.TRAINER.LOAD_LAST = 0
    for i in range(int(sys.argv[2])):
        trainer = Trainer(cfg,sys.argv[1]+"_"+str(i))
        cfg.TRAINER.LOAD_LAST = 2
        if i > int(sys.argv[2])/8 and cfg.TRAINER.LR>0.000005:
            cfg.TRAINER.LR -= 0.0000002
        del trainer
        time.sleep(10)
```

Code listing C.8: The Regression task

```
from yacs.config import CfgNode as CN

cfg = CN()
cfg.MODEL = CN()
# -----------------------------------------------------------------------------
# Embed
# -----------------------------------------------------------------------------
cfg.MODEL.EMBED = CN()
cfg.MODEL.EMBED.TRESHOLD = 1 #unused variable
cfg.MODEL.EMBED.OUTDIM = 3 #Outdim is the size of timepart, placepart, buspart,
    ↪ capacity, direction has default of one


# -----------------------------------------------------------------------------
# Regression forecaster lstm
# -----------------------------------------------------------------------------
```

```
cfg.MODEL.LSTM = CN()
cfg.MODEL.LSTM.TRESHOLD = 1 #unused variable

# ---------------------------------------------------------------------------
# Dataset
# ---------------------------------------------------------------------------
cfg.DATASETS = CN()
cfg.DATASETS.NAME = 'anomaly2'
cfg.DATASETS.REBUILD=False #rebuilds dataset
cfg.DATASETS.TRAIN=150 #amount of days added to the training set
cfg.DATASETS.TEST= 21 #amount of days added to the training set
cfg.DATASETS.VAL=  14 #amount of days added to the training set
cfg.DATASETS.LONG_MEM=  1000 #amount of trips remembered
cfg.DATASETS.SHORT_MIS=  300 #amount of trips forgotten
cfg.DATASETS.SEQ_LEN=  28 #Longest trip


# ---------------------------------------------------------------------------
# Trainer
# ---------------------------------------------------------------------------
cfg.TRAINER = CN()
cfg.TRAINER.GAMMA = 0.1
cfg.TRAINER.BATCH_SIZE = 128
cfg.TRAINER.LR = 0.00001  #0.0001
cfg.TRAINER.MOMENTUM = 0.9

cfg.TRAINER.EPOCH = 10
cfg.TRAINER.WEIGHT_DECAY = 5e-4
cfg.TRAINER.SHOULD_TRAIN = True #retrains the model
cfg.TRAINER.LOAD_LAST = 0 #loads the last checkpoint to continue training
```

Code listing C.9: Configuration file

Larrs Bjertnes

Applying NLP-Based ML Techniques to our LS-CAT Dataset

NTNU
Kunnskap for en bedre verden