Test Planning Document

Priority and Pre-requisites

We will consider the correctness and performance requirements

Correctness is the requirement that individual operators must produce the correct outputs according to their functionality specified in the requirements document (e.g., the ReLU operator must compute the function $\max(0, w_i)$ for a given matrix input w). This is a high priority requirement because it underpins the correctness of any models constructed as the composition of individual operators. The code is templated, so we will need to test with a variety of different numerical types. We should aim to detect correctness errors as early as possible, preferably before we start system-level tests of complex models. We can use the principle of partition to divide the problem into self-contained unit tests for each operator, integration tests to verify operators interact correctly with each other, and system-level tests which will train commonly-used models built using the operators and ensure they achieve an appropriate level of accuracy.

Performance is the requirement that the library runs fast enough to be competitive with other similar libraries. This requirement is less important than correctness, but is still high priority because it is the requirement that gets people using the software. Once again, we can use partition to decompose the testing into manageable tasks. We begin with performance tests for individual operators. These can be used during the development process to iteratively improve performance once correctness has been ensured. We will also require system-level tests for performance later on in the development process. This is because operators may be fast individually, but they could suffer when chained together, possibly due to suboptimal access patterns or cases of redundant calculations (see e.g., Pytorch's CrossEntropyLoss which runs softmax and categorical cross entropy in one step to save time https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html). These system-level tests will measure how long it takes to train commonly-used models and ensure it doesn't take too long. They can be combined with the system-level correctness tests so that we don't have to waste time retraining the same models in one testing cycle.

Scaffolding and Instrumentation

For the correctness requirement we will need scaffolding to pass test cases to the operators, and compare our output with the output from a trusted machine learning library (e.g., Pytorch or TensorFlow) to test correctness. For complex classes, like the Adam optimizer, lots of variables need to be setup so we will have to write scaffolding code to carry this out. In cases where a class has private attributes which we don't want to be accessible in the user-facing API, but we do want to be accessible to the test code, we will create test fixture classes which will inherit from the operator but will be declared as friend classes. In C++ being a friend of a parent class gives a class access to the private attributes of the base class. That way the test code will have access to private attributes for testing while the user code won't. The test fixture classes should only be used in test code, not in user code.

We will use the popular tool Gcov to calculate test coverage for our C++ code.

We will use the Google test framework as the library to support test implementation. Google test provides a collection of macros to assist test development. The software we are testing is templated, so we will need to test with different data types (float and double). Google test offers the TYPED_TEST_SUITE macro to automate the process of instantiating tests over a selection of user-specified data types.

For the performance requirement we will need scaffolding to measure the execution time of our code. We would like the granularity of such measurements to extend from individual operators all the way up to the whole model (std::chrono should suffice, if a finer grain is needed, can use profiling tools like Xcode Instruments). That way we can identify where the performance bottlenecks are, even in deep and complex models. At the unit level, input test data can be randomly generated. At the system level, input data will come from popular machine learning datasets such as MNIST.

We also intend to build a CI pipeline to run automated testing. This will be configured using Github actions to run regression tests whenever code is pushed to the codebase, or a pull request is merged. The CI will run on a MacOS server, and compile with clang.

Evaluation: The plan for scaffolding and instrumentation is adequate for the scope of the coursework, but there are some areas for improvement. Firstly, there are more opportunities for automation in the process. For example one could automate running the performance tests (i.e., the benchmarks) and even create a pipeline for automatically producing graphs and performing analysis on performance data. The automation process could be further improved by running regression tests on a greater range of operating systems and compilers – this would especially help us test the performance requirement better because we can use the results to convince ourselves that the code runs fast across many different hardware platforms, not just one. We could also use more targeted regression testing schemes – such as spec-based regression test selection or prioritized rotating selection – rather than running all of the tests every time a change is made to the code. In terms of instrumentation for the unit tests, Gcov only generates reliable coverage metrics when the input program is compiled without optimizations. However, end users will be using a version of the code compiled with aggressive compiler optimizations enabled. This shouldn't pose too much of a problem, but it is a discrepancy that one should bear in mind when evaluating test coverage.

Process and Risk

Assume we are following the Extreme Programming process. The above identified unit tests for correctness should be developed early on in the development process, preferably before development of the library itself actually begins. This is because the costs of faults grow exponentially as the timeline progresses. The system-level integration tests can take place towards the end of each iteration of the XP loop as, by that point, individual components will be nearing completion, and we can begin implementing and testing complex models. If we wanted to start system testing before all operators are complete, we could add scaffolding to fill in missing operators with code from other machine learning libraries, then gradually phase out the other libraries until we have a fully working system of our own. Each incremental release will need to go through acceptance testing to evaluate the quality of the

artifact. This is where we will run e.g., the benchmarking to verify whether the software fulfils the performance requirement, or run a user survey to measure how usable the API is for other developers.

Risk 1 – Depending on reference implementations for correctness: for the correctness tests, there is a risk that the outputs from other machine learning libraries that we test against are themselves incorrect. We can mitigate this by only using very trusted and reputable libraries, and possibly comparing against several different libraries (this follows the same principle as Buzz Aldrin wearing three watches; the more libraries we check against, the lower the probability of having incorrect test data).

Risk 2 – How representative is the data: randomly generated data may not cover real-world scenarios, and reliance on standardized datasets like MNIST may not be representative of more complex and varied use cases. However I don't think this risk is too relevant for us as the library developer; it is more the model designer's responsibility to tailor their model to the data and use case, the library simply provides the building blocks for them.

Risk 3 – overhead from test scaffolding/instrumentation code skewing performance results: we must ensure that scaffolding code is as lightweight as possible to ensure that it doesn't distort performance results. This may mean that we have to abandon the idea of adding operator-level granularity to our timings if it proves to add too much overhead.

Evaluation of the Quality of the Test Plan

The test plan presented in this document presents a compelling plan for testing the chosen requirements: correctness and performance. Detailed scaffolding and instrumentation strategies are presented to facilitate comprehensive testing. Some areas for improvement are presented below:

1. Present a vision for long-term maintenance: the test plan is sparse on details of how tests will evolve alongside changes to the codebase. It would be beneficial to implement guidelines for future test development to help sustain test quality as the project grows.
2. More diversity in test environments: the current CI pipeline is only planned to run on a MacOS server with clang, thus limiting the scope of performance and testing results to one hardware/compiler combination. Ideally, the pipeline would run on several different hardware platforms with several different compilers to enhance the robustness of results.