Test Specifications

This document details the tests that have been written, what component of the software they test, testing data used, and level of testing. It is intended to be a reference document for markers which is more readable than the actual test code whilst also providing some explanation for the rationale behind choices of test cases and methodology.

Notes on General Methodology

Tests are expressed as functions, wrapped in google test's TYPED_TEST macro. All tests are run for float and double types. To account for the different levels of precision, equality testing is done at the lower level of precision (i.e., float). In cases where randomness affects results (e.g., dropout layers), an appropriate error window is used for equality testing. Most instrumentation is done in the test functions – specifically, generating test data, instantiating required objects, and setting up internal state. Helper functions for populating matrices with ascending whole numbers and testing matrix equality are defined in ./test/test_helpers.hpp.

Utils Tests

The utils module contains numerous helper functions which are reused throughout the rest of the codebase, the bulk of which are matrix operations. Source code is located in ./nn-cpp/utils.hpp and ./nn-cpp/utils.cpp. Tests are located in ./test/utils_unit_tests.cpp and ./test/utils_integration_tests.cpp.

Unit Tests

The unit tests test individual functions defined in utils. To achieve maximal coverage and capture a wide spectrum of inputs, a combinatorial approach was used. For each function there is a small, medium, and big test case (which denotes the size of the input test data). Then, each case is run with both float and double data types (making use of google test's TYPED_TEST_SUITE macro). So for each function we have at minimum $3 \times 2 = 6$ test cases at runtime (because of timing constraints some test cases were omitted, but enough were written to demonstrate a thorough understanding of the approach). Test outputs are compared against numpy outputs; a trusted and widely used library.

Test Name	Description of Test Data	Functionality Tested
MatMulSmall	2x2 matrix multiplied by 2x3	Matrix-matrix multiplication defined in lines 5 to 43 of utils.cpp
MatMulMed	15x5 matrix multiplied by 20x20	As above
MatMulBig	50x20 matrix multiplied by 20x50	As above
ScalarMulSmall	2x2 matrix multiplied by 2	Matrix-scalar multiplication defined in lines 45 to 57 of utils.cpp
ScalarMulMed	15x5 matrix multiplied by 43	As above
ScalarMulBig	20x50 matrix multiplied by 1.5	As above
MatDivSmall	2x2 matrix divided by 2x2	Matrix-matrix division defined in lines 79 to 88 of utils.cpp

MatDivMed	15x5 matrix divided by 15x5	As above
MatDivBig	20x50 matrix divided by 20x50	As above
PowerSmall	2x2 matrix taken to the power of 2	Matrix exponentiation defined in lines 142 to 150 of utils.cpp
PowerMed	10x20 matrix taken to the power of 1.5	As above
PowerBig	50x50 matrix taken to the power of 3	As above
TransposeSmall	3x2 matrix	Matrix transposition defined in lines 194 to 202 of utils.cpp
TransposeMed	10x15 matrix	As above
TransposeBig	50x20 matrix	As above
MeanSmall	2x2 matrix	Taking the overall mean of all values in a matrix, defined in lines 204 to 212 of utils.cpp
MeanMed	10x10 matrix	As above
MeanBig	20x50 matrix	As above

In addition to these test cases, assertions are also placed throughout the body of utils.cpp to validate preconditions for input data (e.g., line 80 asserts that input matrix dimensions match before dividing them by each other).

Integration Tests

I also adopted a combinatorial approach for the utils integration tests. Here we create cases such that for each function, the output is used as the input for each other function in at least one test case (i.e., pairwise testing). This way we can test how different components interact with each other. We don't need to do integration tests with other components of the library because the library is designed in such a way that invocation of functions defined in utils are encapsulated entirely within abstractions (such as Layer or Model classes); in effect the unit tests for those abstractions double up as integration tests for the utils functions. Each test case is instantiated for float and double data types as in the unit tests. (As mentioned above, some cases have been missed out due to time constraints but enough have been written to demonstrate understanding of the approach).

Test Name	Description of Test Data	Functionality Tested
MulMulPow	2x2 matrix a 2x3 matrix b 3x2 matrix c	power((a * b) * c, 2.0)
MulDivMean	3x2 matrix a 2x2 matrix b 3x2 matrix c	mean((a * b) / c)
MulAddRoot	3x2 matrix a 2x2 matrix b 3x2 matrix c	root((a * b) + c)
MulSubExp	3x2 matrix a 2x2 matrix b 3x2 matrix c	exp((a * b) – c)

DivAddLog	2x2 matrix a 2x2 matrix b 2x2 matrix c	log((a / b) + c)
DivSubAbs	2x2 matrix a 2x2 matrix b 2x2 matrix c	abs((a / b) – c)
AddSubTranspose	3x2 matrix a 3x2 matrix b 3x2 matrix c	transpose((a / b) – c)

Evaluation

If I were to approach deriving these tests again, I would do a more comprehensive category partition. Currently, the expectation is that all tests should pass; a better approach would be to have some boundary and erroneous test data. Those cases are currently caught by assertions in the source code, but it would be nice to have explicit unit tests for them. Currently, expected results are hardcoded into the test code. A better approach would be, at runtime, to dynamically run the test cases in the trusted reference implementation (in this case numpy) and compare the results; this way we could dynamically generate input test data (i.e., random number generation), and the only constants between different test runs would be dimensions of the input matrices. This would increase the thoroughness of our tests and is essentially a form of fuzzing.

Layer Tests

The layer classes are the backbone of the neural network library. Source code is located in ./nn-cpp/Layer.hpp and ./nn-cpp/Layer.cpp. Tests are located in ./test/layer_unit_tests.cpp and ./test/layer_integration_tests.cpp.

Unit Tests

The library implements three types of layer: input, dropout, and dense. Each layer has the same API consisting of the methods compute, backward, and a constructor (setters are ignored for testing in the interests of time, getters are tested as part of the constructor unit tests). Layers also have hyperparameters (e.g., dropout rate or regularization coefficients) which affect their outputs. Outputs are tested against Pytorch outputs which were cross-verified with TensorFlow outputs, adding an extra guarantee of correctness. In practice, dense neural network layers are initialized with random weights, but for testing we need deterministic outputs, so we define a new constructor for the dense layer which allows us to deterministically set initial weight values.

Test Name	Description of Test Data	Functionality Tested
DenseLayerInit	Initializes a dense layer with 784 inputs and 128 outputs, 1 L1 weight regularization, 3 L1 bias regularization, 2 L2 weight regularization, and 4 L2 bias regularization	Verifies if initializer parameters are correctly assigned (and by proxy tests getter functions).

DenseLayerCompute	50x50 weight matrix of ascending whole numbers 1x50 bias vector (ascending whole numbers) 50x50 input data (ascending whole numbers) All other layer parameters are default	Tests correct computation of layer forward propagation.
DenseLayerBackpropNoReg	20x20 weight matrix of ascending whole numbers 1x20 bias vector (ascending whole numbers) 20x20 input data (ascending whole numbers) 20x20 output derivatives (ascending whole numbers) All other layer parameters are default	Tests correct computation of layer back propagation pass without regularization (i.e., regularization terms set to 0). Verifies that gradients of weights, biases, and inputs are correct.
DenseLayerBackpropL1Reg	As above but with L1 weight regularization set to 1 and L1 bias regularization set to 2	As above, but this time with L1 regularization applied
DenseLayerBackpropL1Reg	As above but with L2 weight regularization set to 3 and L2 bias regularization set to 4	As above, but this time with L2 regularization applied
DropoutLayerComputeTrain	Dropout rate set to 0.2 50x50 input matrix of ascending whole numbers Training layer mode	Tests dropout layer during training is zeroing an appropriate amount of neurons in accordance with the given dropout rate (because randomness is involved an error window of 0.05 is used)
DropoutLayerComputeEval	Dropout rate set to 0.2 50x50 input matrix of ascending whole numbers Eval layer mode	Tests dropout during inference which computes the identity function
DropoutLayerBackprop	50x50 input matrix of ascending whole numbers 50x50 output derivatives (ascending whole numbers) Training layer mode	Tests dropout back propagation during training. Same note on randomness applies from DropoutLayerComputeTrain
InputLayerCompute	50x50 input matrix of ascending whole numbers	Tests correct computation of input layer forward propagation
InputLayerBackprop	50x50 input matrix of ascending whole numbers	Tests correct computation of input layer backwards propagation

	50x50 output derivatives (ascending whole numbers)	

Integration Tests

Layer objects are designed to be coupled together in succession, separated by activation functions, and with a final loss function tacked onto the end; so our integration tests follow this pattern of usage. Test cases are derived such that each layer type interacts with every type of loss function and every type of activation function at least once (i.e., pairwise testing). There wasn't enough time to write integration tests for backpropagation, but the methodology would be the same. As with the unit tests, results are checked against Pytorch and TensorFlow and tests are evaluated with floats and doubles.

Test Name	Description of Test Data	Functionality Tested
DenseReluDenseCCE	3x3 dense layer ReLU activation layer 3x3 dense layer Categorical Cross Entropy loss	Computes forward propagation through each layer in sequence, then computes and validates accuracy (with a 0.01 error window to account for decimal rounding)
DenseSoftmaxDenseBCE	3x3 dense layer Softmax activation layer 3x3 dense layer Binary Cross Entropy loss	As above
DenseSigmoidDenseMSE	3x3 dense layer Sigmoid activation function 3x3 dense layer Mean Squared Error loss	As above
DenseLinearDenseMAE	3x3 dense layer Linear activation layer 3x3 dense layer Mean Absolute Error loss	As above

Evaluation

Together, the unit and integration tests achieve a good degree of coverage, however I think that the input data space isn't explored thoroughly enough with the current set of test data. For example, the unit tests only test each layer function with a single input dimension. Preferable each function would be tested on a range of input dimensions. There are other features of the input space which could be varied, such as layer hyperparameters, or ranges of input values (e.g., a test case with all negative numbers in the input, or all fractions).

System Tests

The system test trains a neural network model for classifying items of clothing from the MNIST fashion data set. Source code is located in ./test/system_tests.cpp The model trains for 15 epochs and is then tested. The model must achieve an accuracy greater than 0.8 on unseen test data to pass the test. This lower bound was chosen by training five identical models implemented in Pytorch for the same number of epochs, and then taking the rounded mean of their accuracies.

Architecture

1. 784x128 dense layer
2. Relu activation
3. 128x128 dense layer
4. Relu activation
5. 128x10 dense layer
6. Softmax activation

- Categorical Cross Entropy loss
- Adam optimizer (learning rate 0.001 and decay 0.001)

Evaluation

This architecture and 15 epochs were chosen because a more complex architecture or more epochs would take lots of computing power, storage, and time to train. If this test were to run as part of a periodic regression testing suite, then that would slow down the development process. The chosen architecture and number of epochs evaluates fast enough whilst also covering enough of the codebase to thoroughly test the overall system correctness of a model constructed using the library. If I had more time, I would like to add more models which incorporate some of the other components in the library, such as dropout layers or linear activation.