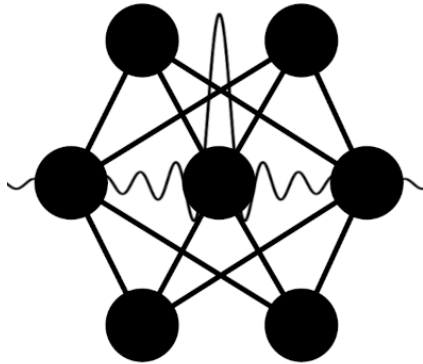


IdealINN

A simple, educational, neural network framework inspired to pytorch APIs.

Cesare Montresor

github.com/cesare-montresor/idealNN



University of Verona

Master's degree in Computer Science and Engineering

High performance software design in C++

Table of contents

Introduction	3
Abstract	3
Features & Limitations	3
Environment Setup	4
Third-party Libraries	4
Dependencies	4
Project Design	5
Design choices	5
Shared Pointers	5
Auto	5
Compile-time evaluations	5
Templates	6
Polymorphism	6
std::move	6
std::vector	6
Naming Conventions	6
Development process	7
Difficulties and Challenges	10
Backpropagation and Autograd	11
Weight initialization	12
Resources	12
Code Quality	12
Project Structure	13
Code organization	13
Results	16
Execution Output	16
test/classification_test.cpp	16
test/regression_test.cpp	17
Code Coverage	18
Cloc Analysys	18
Sanitizer	18
Valgrind	19
Roadmap	20
Milestone 1	20
Milestone 2	20
Milestone 3	20
Milestone 4	20
Milestone 5	20

Introduction

Abstract

The goal of the project is to develop a Neural Network library in C++ that mimics the APIs of one of the most renowned, used and pleasant to use neural network frameworks: pytorch.

The library is mainly intended for educational purposes, to demystify the complexities behind neural networks and neural networks frameworks and allows to rediscover typical challenges, common pitfalls and solutions.

Because of this, there has been an intentional attempt to try to keep a balance between providing “quality of life” features and trying to keep the main logic as plain as possible, in order to provide a library easy to use, but also easy to read and understand its inner workings.

On the contrary, pytorch focuses mostly on providing an easy to use library, but in order to do so, it leverages a lot of python features, like reflection and magic methods, which ultimately makes the codebase not so straight-forward to read and learn from.

The motivation behind the project's name, IdealINN, is an attempt to try to convey the naive exploration process done during development, on how to implement a NN library.

Ideal is intended in the same way as ‘ideal filters’ in the frequency domain. Ideal filters are how everyone approaching the field would initially design frequency filters, but with experience, it turns out that such naive design is far from being ideal.

Features & Limitations

At the current state, the project provides a basic set of features to train a small sized neural network for regression and classification tasks, using tabular data.

This choice has been made because, once the core is validated and working properly, it's relatively easy to extend the framework with additional functionalities due to its modular design.

The limitation about using only tabular data is due to the current state of the library used for Linear Algebra, Eigen3.

While Eigen3 is currently offering an implementation of multidimensional tensors, these features are provided as “unsupported” which explicitly states that the code is provided “as is” and no guarantees or support is provided.

<https://eigen.tuxfamily.org/dox/unsupported/index.html>

https://eigen.tuxfamily.org/dox/unsupported/eigen_tensors.html

Therefore the solution has been developed using standard *Eigen::MatrixXd* which is instead guaranteed to work properly. The project has been developed in a way that would make it relatively easy to switch from Matrix to Tensors when higher dimensional data will be needed and the core logic is fully validated.

```
using TensorData = Eigen::MatrixXd;
```

Environment Setup

Third-party Libraries

Eigen3	Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers and related algorithms. The project expects to find libeigen3 available on the system.
catch2	Is a unit testing framework for C++ and being a single header library, has been included directly in the project source code. Being the whole project a library, end-to-end tests are also intended to be used as usage examples.

Being a CPU-only project at the moment, in the roadmap we plan to also add support for OpenMP as a simple way to parallelize loops and tasks, almost without altering the main source code.

In the long run, it would make sense to integrate with cuBLAS and cuDNN to support more heavy computations, but extreme performance improvements are not the main goal of this project.

Dependencies

The project is compiled using standard C++17 and the whole development process has been done using standard Ubuntu 20.04 LTS, the code has been written using CLion.

In order to setup the environment, is needed to install the following packages

Required	<code>sudo apt install libeigen3-dev</code>
Test & Coverage	<code>sudo apt install valgrind</code> <code>sudo apt install gcovr</code>
Documentation	<code>sudo apt install doxygen</code> <code>sudo apt install texlive-font-utils</code> <code>sudo apt install graphviz</code>

Project Design

Design choices

Shared Pointers

In order to deliver a smooth coding experience for an hypothetical user that has decided to use IdealNN library without worrying too much about memory management and other C++ complexities, the project relies heavily on the use of shared pointers ([std::shared_ptr](#)), to further simplify the use of the library, all objects provides static methods to create objects already wrapped by shared pointers, those methods match one-to-one the available constructors for the object. [std::weak_ptr](#) are used to break circular ownership between Layer and Tensor.

This allows the user to focus on the design of the neural network architecture without worrying too much about memory management and potential leaks. Here an example extracted from [Tensor/Tensor.h](#)

```
struct Tensor {
    //make methods
    static TensorArrayRef MakeTensorArray();
    static TensorArrayRef MakeTensorArray(ArraySize size);
    static TensorArrayRef MakeTensorArray(TensorArray tensorArray);

    static TensorRef MakeTensor(ArraySize rows, ArraySize cols);
    static TensorRef MakeTensor(Tensor &tensor);
    static TensorRef MakeTensor(const TensorRef& tensor);
    static TensorRef MakeTensor(const Matrix &matrix);
    static TensorRef MakeTensor(const MatrixRef& matrix);

    //constructors
    Tensor(ArraySize rows, ArraySize cols);
    Tensor(Tensor const &tensor);
    explicit Tensor(const TensorRef& tensor);
    explicit Tensor(Matrix const &matrix);
    explicit Tensor(MatrixRef matrix);
}
```

Auto

The project heavily uses the *auto* keyword, however this feature is never used in function signatures, but only in the left side on assignments. The main reason being to keep the code flexible and greatly diminish the need of refactoring as the IdealNN library is an ongoing project, beside the purpose of this exam.

Compile-time evaluations

The project doesn't make use of compile-time evaluation such as constexpr except for the definitions of some constants. We also use [static_cast](#) to perform casting of unsigned long, used by the [std::vector](#) to the standard [IdealNN:ArraySize](#).

The major reason for not using compile-time evaluation features being that the typical performance challenges of neural networks usually come from numerical computation and therefore I think there is a little performance gain from using those features.

Templates

The project does use templates, but for the time being their use is quite limited, mostly to wrap types into internal `idealINN` types, to reduce future refactoring and a few utility functions. The use of templates might become more relevant in future releases when the codebase will grow further, but for the time being there is not a great advantage in doing so, considering that the use of templates makes the code harder to understand and debug.

Polymorphism

The project makes use of simple polymorphism to provide standard interfaces between classes for interoperability of the components. See project organization -> code structure for more details.

`std::move`

In the project we made use of `std::move` in order to turn local variables into lvalue in the attempt to reduce copying of values. This is especially important for `Tensor` and `TensorData`, as they represent the most memory consuming objects and are constantly passed between layers.

`std::vector`

In the project all arrays and lists of objects are implemented using `std::vector`. Dedicated methods are used to get the vector size without incurring into undefined behaviors.

Naming Conventions

An effort has been made to maintain naming consistency all along the project

<code><type>Ref</code>	Indicates an object wrapped by a shared pointer (currently <code>std::shared_ptr</code>)
<code><type>WeakRef</code>	Indicates an object wrapped by a weak pointer (currently <code>std::weak_ptr</code>)
<code><type>Array</code>	Indicates an array of objects of a specific type (currently <code>std::vector</code>)
<code>Make<type></code>	Indicates a static method that construct an object and returns it wrapped by a shared pointer
<code><type>Layer</code>	Given that the library should allow the user to compose their neural network

<type>Activation <type>Loss <type>Optimizer <type>DataLoader	freely, the project provides basic interfaces for object types, all derived classes append the name of the used interface to their own class names. Examples: LinearLayer is derived from Layer MSELoss is derived from Loss SigmoidActivation is derived from Activation The interfaces class mostly contains pure virtual methods and properties, but they may also contain concrete implementations for common code.
---	--

NOTE: the notation <type> used above does not indicate the use of C++ templates, it's just used as a wildcard for this very document.

Development process

Given the nature of this project, which aims to replicate pytorch APIs, test driven development has been used as the main development pattern. The following pytorch example have been used to guide the development process, it can be also found as comment in the project inside the file [test/main_test.cpp](#)

```
import torch
from torch import transforms
from torch import torchvision
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

transform = transforms.Compose( [transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
)
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                         download=True, transform=transform)
trainloader = torch.utils.data.CSVDataLoader(trainset, batch_size=4,
                                              shuffle=True, num_workers=2)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
```



```

        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for epoch in range(2): # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward + backward + optimize
        outputs = net(inputs)
        print(outputs)
        break

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999: # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0
print('Finished Training')

```

Increasingly complex end-to-end tests have been written until it was possible to write a similar example using the IdealNN library. The following list, besides being a comprehensive test list, also reflects the order of implementation of features, during the development process. Within each test file, simpler tests can be found at the bottom of the file, and more complex ones at the top.

test/main_test.cpp	Act as wrapper for catch2 tests, containing only the CATCH_CONFIG_MAIN that represents the entry point of the test executable. As mentioned above, given that the file is mostly empty it also contains the pytorch example above, as a comment.
test/csv_loader.cpp	Test the CSV dataloader, which loads tabular data from standard CSV files.
test/forward_test.cpp	Test the forward pass, it doesn't use the logic necessary for the actually training (backward pass)
test/loss_test.cpp	Test the forward, loss and backward pass and the loss function for the forward pass.

test/backward_test.cpp	Test the forward, loss and backward pass, verify that the gradients for each layer's weights are generated.
test/optimizer_test.cpp	Test forward, backward, loss, backward pass and add the SDG optimizer, that would update the layer's weights.
test/regression_test.cpp	Test a complete end-to-end training for regression tasks, verifying the decrease in the loss function.
test/classification_test.cpp	Test a complete end-to-end training for classification tasks, verifying the decrease in the loss function.

For the time being all tests have been done using the well-known IRIS dataset (see more: https://en.wikipedia.org/wiki/Iris_flower_data_set).

Here an example of the resulting code, implemented using IdealNN (the example is not intended to replicate the above example in functionalities, just as coding style)

```
#include <Layer/LinearLayer.h>
#include <DataLoader/CSVDataLoader.h>
#include <Utils.h>
#include <Loss/CrossEntropyLoss.h>
#include <Optimizer/SDGOptimizer.h>
#include <Activation/SoftmaxActivation.h>
#include <Activation/RELUActivation.h>
#include <catch2/catch.hpp>
#include <iostream>

namespace IdealNN {
    TEST_CASE("Regression: train with SDG on multiple epochs") {

auto learning_rate = 0.0001;
auto batch_size = 5;
auto path = "/home/cesare/Projects/idealNN/extra/iris/IRIS.csv";
auto dl = CSVDataLoader::MakeCSVDataLoader(batch_size, path);

auto xs = Tensor::MakeTensorArray();
auto ys = Tensor::MakeTensorArray();

auto fc1 = LinearLayer::MakeLinearLayer(4, 10);
auto act1 = RELUActivation::MakeRELUActivation();
auto fc2 = LinearLayer::MakeLinearLayer(10, 1);

auto criterion = MSELoss::MakeMSELoss();

auto trainLayers = Utils::MakeLayerArray();
trainLayers->push_back(fc1);
trainLayers->push_back(fc2);
auto optimizer = SDGOptimizer::MakeSDGOptimizer(trainLayers, learning_rate);

auto epoch = 0;
auto epoch_max = 30;
```

```

auto num_batches = 0;

ScalarValue initialLoss=0;
ScalarValue finalLoss=0;
ScalarValue loss;
ScalarValue epochLoss=0;
dl->shuffle();

while(true) {
    auto batch = dl->getData();
    auto bs = Utils::getSize(batch);
    if(bs == 0){
        std::cout << "[EPOCH \t" << epoch << "]" << " --- " << "AVG Loss: ";
        std::cout << epochLoss / num_batches << " --- " << std::endl;
        if(epoch < epoch_max){
            if (epoch == 0){ initialLoss = (epochLoss / num_batches); }
            epochLoss = 0;
            num_batches = 0;
            ++epoch;
            dl->shuffle();
            continue;
        }else{
            finalLoss = (epochLoss / num_batches);
            break;
        }
    }
    CSVDataLoader::splitXY(batch, xs, ys, 0, 4, 4, 1);

    auto x1 = fcl->forwardBatch(xs);
    auto a1 = act1->forwardBatch(x1);
    auto ys_hat = fc2->forwardBatch(a1);

    loss = criterion->loss(ys_hat,ys);
    epochLoss += loss;
    ++num_batches;

    criterion->backward();
    optimizer->step();
    optimizer->zero_grad();
}

} //namespace TEST_CASE
} //namespace IdealNN

```

Difficulties and Challenges

One of the biggest challenges of this project is faced during this project is that even if everything seems to work correctly, code wise, the mathematics behind it full of potential pitfalls, which are very hard to debug because the project appears to work correctly, but then it turns out that the training simply doesn't occur and the loss function doesn't decrease.

This is made even worse by the fact that the neural network is composed of many pieces, and the

mathematics of each of them must be correct both for the forward and the backward pass and therefore at times have been challenging pinpointing the actual causes that lead to failure in training.

Backpropagation and Autograd

The most challenging part of this project has been the correct implementation of the backpropagation algorithm, which lies as the core of the training of general purpose neural networks.

The principle is simple, in theory all it takes is to compute the partial derivative of each mathematical operation done during the forward pass, compute the gradients, accumulate them and correct the weights. On top of this, it appears that for the same mathematical operation a different partial derivative might be used based on variables being or not-being dependent, which made it difficult to understand which derivation was most appropriate and both were technically correct.

However this becomes a non-trivial task when dealing with higher dimensional data, like matrices, and when you concatenate multiple operations, because even if it seems correct, a simple + or - sign can mess up the entire back propagation process.

On top of this, modern neural network frameworks rely on all sorts of normalization techniques to keep activation and gradients from exploding or vanishing. Those techniques, used in practice, are pretty much never taken into account by theoretical videos and articles explaining the mathematics.

Therefore, even with a correct implementation of the whole backpropagation algorithm, distributed across multiple classes, the training might still not happen or stop at some point because of the intrinsic numerical instability (NaN, inf, -inf) of some operation (ex: Softmax, CrossEntropy).

As a result I needed to get the correct mathematical implementation of several pieces, in order to be able to verify the correct implementation.

Also, the current implementation of the backpropagation algorithm included in IdealNN is not yet to its fullest potential as it allows only for sequential models, for example making it impossible to implement modern architectures like ResNet or Recurrent Neural Networks.

A better implementation of IdealNN backpropagation algorithm is currently being developed, but is not included in the current release.

The major difference being, at the moment it is possible only to specify a layer as the last operation for a tensor, the new version would support a general purpose Operation class that would be able to accept both the mathematical “operator” as well as a series of parameters, allowing for the construction of a proper DAG for backward propagation.

While this will provide a general purpose autograd capable of performing any partial derivative, it would also be very wasteful of computation resources. Because when one computing derivatives on paper, it is often possible to simplify the formulas making a lot of mathematical operations unnecessary.

In order to mitigate computation waste I plan to move all the mathematical operations outside the layers/activations into dedicated classes, together with their simplified derivative.

These dedicated classes will become the actual “operator” fed into the Operation, allowing both for flexibility and performance. Here an example of use of the old system:

```
output->operation = weak from this(); //LinearLayer, it's a ptr to the layer itself
output->operation = Operation::MakeOperation(Operator::Linear, weights, bias );
```

Weight initialization

When dealing with mathematics, the devil is in the details, for example, it turned out to be of extreme importance the way tensor weights are initialized and this also depends greatly on the type of non-linearities (activation functions) used in the composition of the model. For example, on this topic alone, weight initialization, there have been several academic papers over the years describing several ways to do so. A module which has been temporarily removed from this release is the Initialization class that wraps all the complexities of weight initialization, for the time being, a parameterized version of `kaiming_uniform` is used. However, to fully resolve the issue and provide an easy and solid solution, it is necessary to modify some of the core logic of `LinearLayer` to include internally also the activation function, in this way the weights can be initialized accordingly.

Resources

Here a non-comprehensive list of the resources used during the development of this project to better understand the mathematical theory required and how to organize a neural network library.

- <https://www.youtube.com/watch?v=44tFKZhPyPQ>
- <https://www.youtube.com/watch?v=i94OvYb6noo> <3
- https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
- <https://www.youtube.com/watch?v=tleHLnjs5U8>
- <https://www.youtube.com/watch?v=09c7bkxpv9I>
- <https://www.youtube.com/watch?v=MswxJw-8PvE>
- <https://forums.fast.ai/t/gradients-for-softmax-are-tiny-solved/18970/11>
- <http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/>
- <https://iamtrask.github.io/2015/07/12/basic-python-network/>
- <https://pytorch.org/docs/stable/nn.init.html>

Code Quality

The code has been developed using CLion which includes static analyzer CLangTidy the used `.clang-tidy` has been included in the source code. The project has been compiled and tested using **-Wall -Wextra -Wpedantic -Werror** and by default the whole development was done with undefined behaviors sanitizer **-fsanitize=undefined -fno-omit-frame-pointer** turned on.

The undefined behaviors can be turned on/off with CMakeList parameters.

Project Structure

The project source code is organized as follows

src/	Contains all the source code that constitutes IdealNN lib.
test/	Contains all the test
doc/	Contains the final documentation and the code coverage (populated by 'make install')
bin/	Contains the statically compiled libidealnn.a (populated by 'make install')
extra/	Contains the datasets used for training
third_party/	Contains third party libraries.
build/	Contains the cmake and the build files.

Code organization

Common.h	Contains all the basic types used in the project, eventually wrapping native types in order to provide a simple mechanism for future refactoring and improvements. It also contains all the necessary forward declarations needed to avoid circular dependencies.
Utils.h Utils.cpp	Contains utility methods used across all the project, as well as method utility methods that don't find a better placement with the current project structure. For example, the type ScalarValue, defined in common.h, is used all across the project, but really is a simple wrapper for double and therefore doesn't really have a class/struct definition. In this case Utils offers utility methods to create pointers to vectors of ScalarValues.
Tensor/ Tensor.h Tensor.cpp	Represents a high level Tensor class, which internally contains the actual data, but it also allows for storing the accumulated gradients, tensor operations and all other information necessary for training a neural network.
Layer/ Layer.h Layer.cpp	General interface for Layer objects, most methods are defined as pure-virtual except for forwardBatch, for which class provides a concrete implementation as turned out to be appropriate for most Layers. The forwardBatch method also caches inputs and outputs of the layer as they are always needed during the backward pass.
Layer/ LinearLayer.h	Concrete implementation of Layer Implements a standard so called Linear, Dense or Fully Connected Layer. The basic operation is a linear

LinearLayer.cpp	transformation done via matrix multiplication. It contains weights that can be subject to training.
Activation/ Activation.h Activation.cpp	General interface for Activation objects, extends Layer. The main goal of Activation classes is to provide a simple way to introduce the necessary non-linearities between linear layers. The major difference between Layer and Activation is lack of training parameters. Activation adds a concrete implementation of the parameter() method, which always returns an empty vector. For this purpose the method is marked as final.
Activation/ SigmoidActivation.h SigmoidActivation.cpp	Concrete implementation of the Activation interface to provide the most classic of the Activation functions, the Sigmoid function for the forward and the backward pass.
Activation/ TanhActivation.h TanhActivation.cpp	Concrete implementation of the Activation interface to provide another classic, the Tanh function, for the forward and the backward pass.
Activation/ RELUActivation.h RELUActivation.cpp	Concrete implementation of the Activation interface that implements what today is considered a default choice for non-linearities, the RELU function, for the forward and the backward pass.
Activation/ SoftmaxActivation.h SoftmaxActivation.cpp	Concrete implementation of the Activation interface. Unlike other Activations which are intended to be general purpose, softmax is generally used in combination with CrossEntropy loss for classification tasks. The softmax function is inherently numerically unstable and can easily produce 0, inf, -inf. To minimize this problem is typically advised to normalize the activations before feeding them to the softmax, this is typically done using a BatchNorm layer, which is not yet implemented within IdealNN. Instead the softmax has been slightly modified to partially normalize the data internally.
Loss/ Loss.h	General interface for Loss objects, all methods are defined as pure-virtual. It also defines properties for storing data necessary for the backward pass.
Loss/ MSELoss.h MSELoss.cpp	Concrete implementation of the Loss class, it provides the Mean Squared Error Loss function, typically used for regression tasks.
Loss/ CrossEntropyLoss.h CrossEntropyLoss.cpp	Concrete implementation of the Loss class, it provides the Cross Entropy Loss function, typically used for multi-class classification tasks. As it internally uses logarithms, it requires that all inputs are positive, non-zero numbers, otherwise it produces NaN as results. In order to fulfill this requirement, the Cross Entropy loss is often used in combination with Softmax Activation.
Optimizer/ Optimizer.h	General purpose interface for any Optimizer object, all methods are pure-virtual and it also defines some properties generally used by Optimizers.

Optimizer/ SDGOptimizer.h SDGOptimizer.cpp	Concrete implementation of the Optimizer class, it provides a vanilla implementation of Stochastic Gradient Descent algorithm.
DataLoader/ DataLoader.h	General purpose interface for any DataLoader object, all methods are pure-virtual.
DataLoader/ CSVDataLoader.h CSVDataLoader.cpp	Concrete implementation of the DataLoader class, it provides a general purpose CSV data loader, as output it provides a read to use array of Tensor, already organized in mini-batches, which is what is typically used during the training process.

Results

Execution Output

test/classification_test.cpp

Classification: train with SDG on multiple epochs

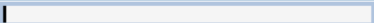



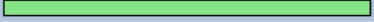






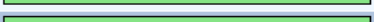





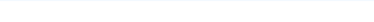
```
[EPOCH  0] --- AVG Loss: 1.15455 ---  
[EPOCH  1] --- AVG Loss: 1.15508 ---  
[EPOCH  2] --- AVG Loss: 1.15612 ---  
[EPOCH  3] --- AVG Loss: 1.1561 ---  
[EPOCH  4] --- AVG Loss: 1.15551 ---  
[EPOCH  5] --- AVG Loss: 1.15392 ---  
[EPOCH  6] --- AVG Loss: 1.15254 ---  
[EPOCH  7] --- AVG Loss: 1.15083 ---  
[EPOCH  8] --- AVG Loss: 1.1475 ---  
[EPOCH  9] --- AVG Loss: 1.14404 ---  
[EPOCH 10] --- AVG Loss: 1.13969 ---  
[EPOCH 11] --- AVG Loss: 1.13524 ---  
[EPOCH 12] --- AVG Loss: 1.12932 ---  
[EPOCH 13] --- AVG Loss: 1.12318 ---  
[EPOCH 14] --- AVG Loss: 1.11674 ---  
[EPOCH 15] --- AVG Loss: 1.11008 ---  
[EPOCH 16] --- AVG Loss: 1.10265 ---  
[EPOCH 17] --- AVG Loss: 1.09508 ---  
[EPOCH 18] --- AVG Loss: 1.0873 ---  
[EPOCH 19] --- AVG Loss: 1.07962 ---  
[EPOCH 20] --- AVG Loss: 1.072 ---  
[EPOCH 21] --- AVG Loss: 1.06459 ---  
[EPOCH 22] --- AVG Loss: 1.05716 ---  
[EPOCH 23] --- AVG Loss: 1.0504 ---  
[EPOCH 24] --- AVG Loss: 1.04386 ---  
[EPOCH 25] --- AVG Loss: 1.03755 ---  
[EPOCH 26] --- AVG Loss: 1.03163 ---  
[EPOCH 27] --- AVG Loss: 1.0265 ---  
[EPOCH 28] --- AVG Loss: 1.0216 ---  
[EPOCH 29] --- AVG Loss: 1.01666 ---  
[EPOCH 30] --- AVG Loss: 1.01252 ---
```

test/regression_test.cpp

Regression: train with SDG on multiple epochs

```
[EPOCH 0] --- AVG Loss: 9.43876 ---  
[EPOCH 1] --- AVG Loss: 6.04214 ---  
[EPOCH 2] --- AVG Loss: 4.60063 ---  
[EPOCH 3] --- AVG Loss: 3.78199 ---  
[EPOCH 4] --- AVG Loss: 3.24688 ---  
[EPOCH 5] --- AVG Loss: 2.86504 ---  
[EPOCH 6] --- AVG Loss: 2.57631 ---  
[EPOCH 7] --- AVG Loss: 2.34977 ---  
[EPOCH 8] --- AVG Loss: 2.16628 ---  
[EPOCH 9] --- AVG Loss: 2.01386 ---  
[EPOCH 10] --- AVG Loss: 1.88497 ---  
[EPOCH 11] --- AVG Loss: 1.77424 ---  
[EPOCH 12] --- AVG Loss: 1.67793 ---  
[EPOCH 13] --- AVG Loss: 1.59313 ---  
[EPOCH 14] --- AVG Loss: 1.51784 ---  
[EPOCH 15] --- AVG Loss: 1.45056 ---  
[EPOCH 16] --- AVG Loss: 1.39007 ---  
[EPOCH 17] --- AVG Loss: 1.33522 ---  
[EPOCH 18] --- AVG Loss: 1.28543 ---  
[EPOCH 19] --- AVG Loss: 1.23994 ---  
[EPOCH 20] --- AVG Loss: 1.19827 ---  
[EPOCH 21] --- AVG Loss: 1.15979 ---  
[EPOCH 22] --- AVG Loss: 1.12428 ---  
[EPOCH 23] --- AVG Loss: 1.09143 ---  
[EPOCH 24] --- AVG Loss: 1.0609 ---  
[EPOCH 25] --- AVG Loss: 1.03246 ---  
[EPOCH 26] --- AVG Loss: 1.00596 ---  
[EPOCH 27] --- AVG Loss: 0.981232 ---  
[EPOCH 28] --- AVG Loss: 0.958017 ---  
[EPOCH 29] --- AVG Loss: 0.936252 ---  
[EPOCH 30] --- AVG Loss: 0.915724 ---
```

Code Coverage

File	Lines			Branches	
Activation/Activation.cpp		0.0%	0 / 2	-%	0 / 0
Activation/RELUActivation.cpp		100.0%	15 / 15	50.0%	20 / 40
Activation/SigmoidActivation.cpp		100.0%	16 / 16	50.0%	24 / 48
Activation/SoftmaxActivation.cpp		100.0%	18 / 18	50.0%	28 / 56
Activation/TanhActivation.cpp		100.0%	16 / 16	50.0%	20 / 40
DataLoader/CSVDataLoader.cpp		100.0%	49 / 49	60.3%	41 / 68
DataLoader/DataLoader.h		100.0%	1 / 1	-%	0 / 0
Layer/Layer.cpp		100.0%	10 / 10	62.5%	5 / 8
Layer/Layer.h		100.0%	1 / 1	-%	0 / 0
Layer/LinearLayer.cpp		100.0%	35 / 35	51.7%	31 / 60
Loss/CrossEntropyLoss.cpp		100.0%	24 / 24	54.3%	25 / 46
Loss/Loss.h		100.0%	1 / 1	-%	0 / 0
Loss/MSELoss.cpp		100.0%	23 / 23	55.9%	19 / 34
Optimizer/Optimizer.h		100.0%	1 / 1	-%	0 / 0
Optimizer/SDGOptimizer.cpp		100.0%	26 / 26	64.3%	18 / 28
Tensor/Tensor.cpp		63.4%	26 / 41	40.6%	13 / 32
Utils.cpp		65.0%	13 / 20	66.7%	8 / 12
Utils.h		100.0%	8 / 8	50.0%	2 / 4

Cloc Analysys

```
cesare@gazelle:~/Projects/idealNN$ cloc src/
  30 text files.
  30 unique files.
   0 files ignored.

github.com/AlDanial/cloc v 1.90  T=0.02 s (1890.6 files/s, 91818.9 lines/s)
-----
Language                     files      blank      comment      code
-----
C++                           13         145         103         393
C/C++ Header                  17         206         306         304
-----
SUM:                           30         351         409         697
-----
```

Sanitizer

```
cesare@gazelle:~/Projects/idealNN/build/debug$ make test
Running tests...
Test project /home/cesare/Projects/idealNN/build/debug
  Start 1: test
1/1 Test #1: test ..... Passed    0.21 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =  0.21 sec
```

Valgrind

```
cesare@gazelle:~/Projects/idealNN/build/debug$ valgrind --leak-check=full --show-leak-  
kinds=all --track-fds=yes --track-origins=yes ./idealln_test  
==51231== Memcheck, a memory error detector  
==51231== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.  
==51231== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info  
==51231== Command: ./idealln_test  
==51231==
```

```
=====  
All tests passed (14 assertions in 12 test cases)  
=====  
==51231==  
==51231== FILE DESCRIPTORS: 3 open (3 std) at exit.  
==51231==  
==51231== HEAP SUMMARY:  
==51231==     in use at exit: 0 bytes in 0 blocks  
==51231==   total heap usage: 96,208 allocs, 96,208 frees, 5,774,141 bytes allocated  
==51231==  
==51231== All heap blocks were freed -- no leaks are possible  
==51231==  
==51231== For lists of detected and suppressed errors, rerun with: -s  
==51231== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)  
cesare@gazelle:~/Projects/idealNN/build/debug$
```

Roadmap

Milestone 1

- Add LeakyRELUActivation to resolve the problem of dead neurons..
- Add BatchNormLayer to improve numerical stability.
- Add DropoutLayer to allow for an easy way to reduce overfitting.
- Add TensorInitialization class for flexible weight initialization.

With BatchNormLayer and DropoutLayer it should be possible to build much deeper models, also it would improve overall numerical stability.

Milestone 2

- Implement fully fledged Autograd backpropagation algorithm as described above
Adding Operation and Operator classes.
Moving the math logic outside the Layer/Activation/Loss function, together with their derivatives.
- Add operator overloading for Tensors integrated with Autograd.

With proper Autograd and operator overloading it will be possible to build non-sequential architectures like ResNet, RNN and transformers.

Milestone 3

- Add support for OpenMP parallelize everything but the backward pass

Larger models should be now much slower to train, OpenMP could provide a cheap, simple and clean solution to improve the performance.

Milestone 4

- Replace Eigen::MatrixXd with Eigen::Tensor
- Add extensive unit testing to ensure equal results in the mathematical operation.

Milestone 5

- Add ImageDataLoader with multi-threading loader
- Add Conv2DLayer, MaxPoolActivation.
- Look into cuDNN and cuBLAS to increase performance image processing.