Deep Learning from Scratch in C++

The DamkeNet Deep Learning Library Documentation

Introduction

The DamkeNet is a Deep Learning library that specializes in sign language recognition and was developed by Damir Bojadžić for the purposes of the “Deep Learning from Scratch in C++” practical course held in SS19 at the Technical University of Munich. The library can also be expanded and used for other DL projects. This document covers the most important aspects of this project including classes and functions and a guide on how to use them.

Layer Class

The layer is the building block of the neural network. On its own, the task of the layer is to receive data, transform the data and output the data. In order to do that efficiently, the library defines an abstract class Layer for Eigen::VectorXd and Eigen::MatrixXd input types. The interface of the class with each function is explained below.

predict() - The purpose of this function is to simply pass the data through the layer. It is marked as “const”, and therefore does not change the internal attributes of the class.

feedForward() - Essentially, the feedForward function does the same operations as predict(), but it also stores intermediate values which are used later (if needed) during the backward pass. For example, in the MaxPooling layer, there is no difference between predict() and feedForward().

feedBackward() - The purpose of this function is to compute the backward pass using the intermediate values that get stored in the forward pass and also to compute and internally store the gradient of the input with regard to the attributes. Please note that the feedBackward() function just computes and stores the derivative and does not actually modify the arguments (e.g. weights).

All the functions listed above work with both Eigen::VectorXd and std::vector<Eigen::MatrixXd>. The reason for representing images as vectors of matrices is that an image can have multiple channels (RGB or for example if the convolutional layer had multiple kernels it can output multiple images for each kernel). Therefore, this decision was made in order to make the layers as compatible with each other as possible.

However, due to fully connected layers taking in only eigen vectors as inputs and convolutional layers operating strictly on matrices, a very nasty problem emerges when trying to move from convolutional layers to fully connected layers. This problem can be fixed by implementing a new fully connected layer class, but due to time limitations that has not been done, and an explicit transformation is required (which will be explained in the Net Class section)

In order to actually make the layer learn, the class specifies an interface to work with the derivatives computed by the backward pass.

accumulateDerivative() - This function can be best described as a commit function. This is best illustrated through an example. Let’s imagine that the backward pass has been computed and the variable temp_derivative, which stores the derivative, has been created. After calling accumulateDerivative(), the temp_derivative has been committed and transferred to real_derivative. However, the internal attributes (e.g. weights) have still not changed. This implementation may be counter intuitive, but it is heavily used when computing mini-batches and gradients since it just accumulates the gradient, and does not immediately update the weights. That is the job of the derive() function.

derive() - The derive() function simply takes the value calculated by accumulateDerivative() and updates the internal parameters with it. An important thing to note is that derive takes into consideration the learning rate, regularization and momentum when calculating the derivative, and can therefore provide an adaptive learning step if needed.

resetDerivative() - Clears all derivatives that may be stuck in the layer. Should be called after every batch.

save()/load() - The save() and load() functionality allows the user to save the current state of the layer to a .txt file. The interesting thing is that the user can then access the .txt files and modify the parameters (should such a need arise) and/or visualize them. These functions accept an fstream object to the config.txt where the layer outputs its structure, and a path variable where the layer writes its parameters. The user is free to choose in what format he/she wants to save the state of the network, since the loading is also user defined. When writing to the config file, the save function has to output a char which distinguishes the type of layer. Note that this char also needs to be hardcoded in the save function of the Net class, so that it knows which type of layer to instantiate.

Currently convolutional, maxpooling and fully connected layers are inherited from the Layer class. Each one of these can be seen in the layer.h and layer.cpp files, alongside comments describing how each layer is created and how it should be handled.

Net Class

The Net class provides an interface for working with layers. It stores the layers inside an std::vector<Layer>, and provides wrapper functions which call the functions from the Layer classes.

If the user wants to add a layer, simply calling the adequate “add” function with proper parameters should do the job. If, however, the user wants to specify his own layer type, he would have to create an “add” function for that layer, which essentially just allocates a layer and pushes it into the vector.

The predict functions are supposed to predict the output of the neural network with regard to the input. Inputs can be eigen vectors or matrices and outputs can be just the position of the highest label or the whole prediction vector. More on this function can be found in net.h and net.cpp.

train() - The training function is the most complex and most important function of the whole network. Two versions of this function exist, one for eigen vectors and one for eigen matrices, but essentially both do the same thing. The function accepts at minimum an std::vector of inputs and an std::vector of targets (in the form of Eigen::Vectors) which are then used to train the network. Additionally the user can specify the number of epochs, the batch size, learning rate, the regularization strength, momentum strength (in case that adaptive gradient descent should be used) and early stopping. After the function is called, the network does the rest.

It should be noted that these are not all the parameters which can be tweaked with regard to the train() function. Inside the function, the user can specify how soon/late the early stopping should be triggered, and the user can also specify how often the function should log the loss. The submodule spdlog was used for logging, but was later replaced by a simple std::cout command for simplicity.

Batch learning was never implemented in its true form (passing multiple images at once) but was rather implemented by passing individual images and simply not updating the gradient step but accumulating the gradient. It would, however, be interesting to see both versions implemented and measure their performance. One of the advantages of this approach is minimal memory usage (which is crucial due to std::vector usage). On the other hand the eigen library and its fast matrix multiplication algorithms are not used, thus significantly reducing speed.

One of the worst consequences of having the fully connected layers accept only vectors was that eventually you would have to manually check and convert the input matrix into an eigen vector and vice versa. For this, the functions vec2mat() and mat2vec() were implemented, but the hideous hardcoded control statement using std::dynamic_pointer_cast had to be used. This can be avoided by reworking fully connected layers to use matrices.

It is also worthwhile going over the loss function of the network. The loss function can be defined by inheriting the Layer class and simply stacking a loss layer class at the end of the network. Currently the loss is hardcoded in the train() function to simply feed backwards the value of
2 * (output - target). The expansion of the existing model to use custom loss functions, if needed, can be done with ease.

mnist.h and signlang.h

The mnist.h and signlang.h files provide a variety of functions to load the MNIST handwritten digits dataset and the sign language dataset as well as their respective labels. The data can be loaded either as eigen matrices or vectors. It should be noted that these functions provide a high level of reusability in other projects and for other datasets should such a need arise. The mnist.h functions can be used to load from binary files while the signlang.h functions load from .csv files where the data is separated by a comma.

There are no unit tests that cover these files for a multitude of reasons. The most important reason is that in order to generate a “ground truth” the data would need to be loaded and again stored inside a file or hardcoded inside the test. And the problem is that the ground truth data would, in fact, have to be generated by the functions that we are trying to test. Since it doesn’t make sense to test the validity of a function by using that same exact output as the ground truth, no unit tests were written. However, this does not mean that the functions were not tested at all. After the loading was done, random images were selected from the loaded data and manually compared with the contents of the .csv file. Since the random images matched the content of the .csv file exactly, it is assumed that the functions load the data without errors.

If the user should want to visualize the images that are loaded or check their validity, the exportimg() function has been implemented which accepts an eigen matrix and a path where the image is exported in .txt form (or other forms). This .txt can later be visualized (for the purposes of testing, MATLAB was used for this task). Some of these images can be found in the folder txtimages.

Main.cpp (Demo Example)

Two examples (main functions) are provided inside the main.cpp file as demonstration of the functionality of the net.

The first one presents the state of the neural network before and after training by visualizing 2000 test labels and the respective output of the network. Before training, a clear bias towards one label can be seen as the neural network usually just predicts one letter, yielding an accuracy of 1/24. After the training a clear increase in predictive capabilities can be noticed with the neural network predicting with an ~80% accuracy. It should be noted that some sign language gestures look very similar (e.g. the letters A, M, N, S) and that the network mostly fails in differentiating between them.

The second demo example simply demonstrates the training process of the neural network and the gradual decrease of the loss function. The example is just set to 50 epoch which isn’t enough for achieving the above mentioned accuracy, but is simply used for purposes of demonstration. This demo example should, however, yield a test set accuracy of around 70% within 15 minutes of training. This result was measured on the 1st of August. Should any major discrepancies arise, please notify the author.

Testing

All the tests can be found in the tests.cpp file and provide coverage of the most important functions of the neural network. Additionally every function validates its inputs, which has also been covered by the tests. The user has the freedom to experiment with the network and play around with the different functionalities. Should there be any unwanted side effects or malfunctions please notify the author immediately.