Documentation C++ Deep Learning Library

Bock, Philipp

July 31, 2019

1 Structure of the Library

Having a look at the repository, on the top level, there are several folders holding data, documentation and example applications. These can be used to get a more clear understanding of how to use the Deep Learning library.
The library code itself is located within /libdl/library/ and contains the following four different components:

1. **ComputationalGraph:** This component implements the compute logic using a singly linked list of nodes. Where each of the nodes holds specific data or a mathematical operation. Using a controller class it is further capable of updating values of parameter nodes.
2. **NeuralNetwork:** This component contains first a Layer factory which creates Deep Learning layers using Nodes of the computational Graph. Second it holds a class *NeuralNetwork* which holds typical functions like *train()*, *predict()*, *testAccuracy()*, etc.
3. **Utils:** This component contains all sort of utilities as for example IO operations, common data types, data loaders, etc.
4. **Bindings:** This component contains all python bindings. These create the top-level functions, needed to construct a Neural Network, possible to call with python. To use them the libdl.so shared-library must be imported from the build/library/bindings folder.

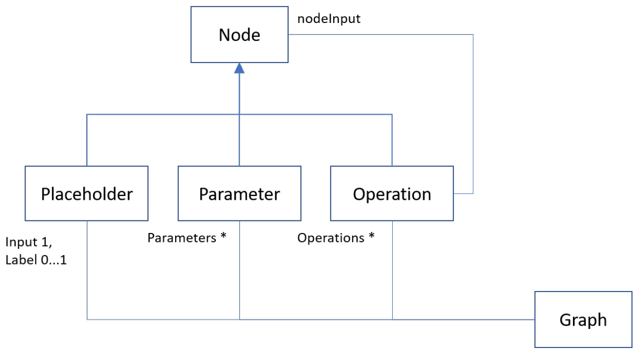
Note: In order for everything to build one has to mind the following dependency graph:

$$\textit{Utils} \rightarrow \textit{ComputationalGraph} \rightarrow \textit{NeuralNetwork} \rightarrow \textit{Bindings}$$

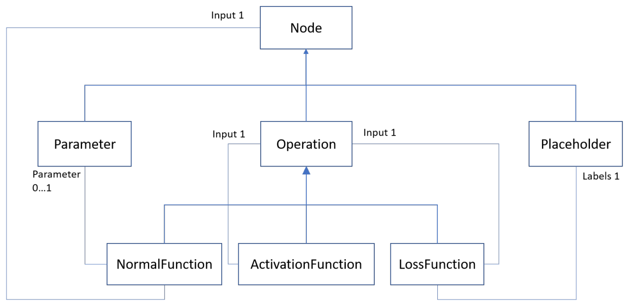
where the only non-dependent component is *Utils*.

The following two sections explain the **ComputationalGraph** and **NeuralNetwork** components more detailed.

2 Computational Graph



(a) General Structure



(b) Operations

Figure 1: Operations

The computational graph (1a) is stored a singly-linked List. However, there are three different main Node types:

1. **Placeholder** is used to input constants into the Graph. In terms of Neural Networks, this could be a sample or label. Therefore, this Node does not hold any input node.

2. **Parameter** is used to input variables into the Graph. In terms of Neural Networks, this could be a Weight or Bias. Therefore, it does not take any input node. But differently to the other nodes it has a method *updateParameter(HyperParameter)* which can be called to update its value. This is done according to the hyper-parameters of the Deep Learning logic.

3. **Operation** is the most important Node it provides an interface each Operation has to implement. This includes especially the *forwardPass()* and *backwardPass()* methods.

The other Nodes which are all derived from **Operation** are further separated as described in 1b. These sub-classes describe which input node possibilities a concrete operation should inherit depending on its type of operation in a Neural Network. E.g. a convolution (NormalFunction) does need in addition to the standard input a filter (represented by a Parameter), while a sigmoid (ActivationFunction) does never use any additional inputs.

2.1 Computation

When we want to create a new **Node** object, the class **OperationFactory** is used. When creating a Node with this factory it is assured that each created Node is added to a **Graph** object at the respective fields. This leads to having a list of *operations* and *parameters*.

In order to compute the forward pass, one can now iterate over the list of **Operation** objects and call for each *forwardPass()*. When the method is invoked, the current Operation will take the forward pass value of its input Node, perform its mathematical operation and then sets its own forward pass value.
Note: The forward pass value of placeholder and parameter is already set at construction

The backward pass works similar only that it is called while traversing the reversed list of **Operation** objects. It calculates the gradients w.r.t the inputs and passes those gradients to the respective input Node. In order to calculate the gradients, it uses the gradients set by its output node. However, as the last node has no output node, the *Graph* object sets the previous gradients for this node to one.

Further, as a last step after the backward pass, the **Graph** class can call *updateParameters(HyperParameters)* on all **Parameter** objects updating them with the calculated gradients.

2.2 Data Representation

In order to store the forward pass and gradient values, the library uses dynamic Eigen matrices with data type 'float' (Eigen::MatrixXf). In order to keep this consistent throughout the library a typedef $usingMatrix = Eigen :: MatrixXf$; is used.
Further, each operation handles the matrices as batches where one sample corresponds to one row. If a sample contains multiple channels, each is stored in complete form consecutively e.g.
$sampleOne = batch.row(0) = [channel\, 0]...[channel\, n]$

3 Neural Network

The core of this component is the **AbstractLayer** class. It gives the user an interface that allows one to easily create a Neural Network layer. Within a layer, multiple **Operation** objects can be created at once to implement Deep Learning logic. The other important part of this component is the *NeuralNetwork* class that allows us to handle our Deep Learning network.

3.1 Creating Layers

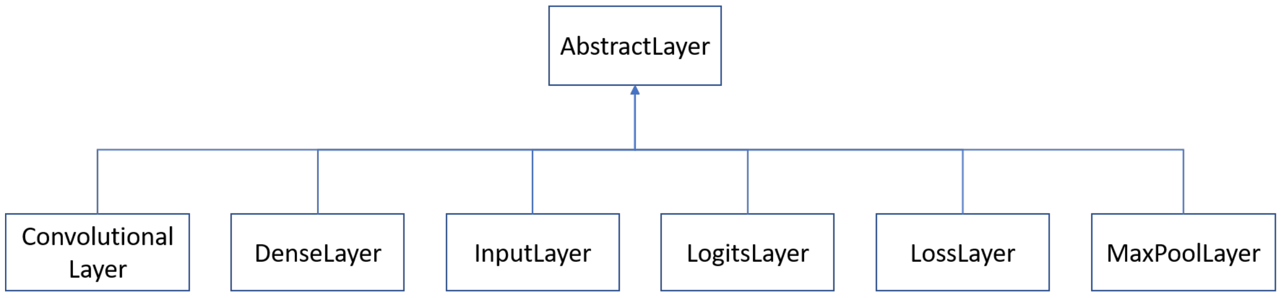


Figure 2: Layers Hierarchy

As shown in 2 each layer object inherits from **AbstractLayer** which defines the behavior each layer needs to have. This is that each layer (apart from **InputLayer**) must take a **Graph** object and an **AbstractLayer** object. The former is in order to be capable of adding the operations to a Computational Graph. The latter allows us to connect the nodes of different layers. Each **Abstract Layer** contains a member *outputNode* that stores the last node of the computation sequence within the layer. Therefore, when creating the first node within a layer we simply take the *outputNode* of the input layer as *inputNode*.

3.2 Executing a Neural Network

In order to finally create a Neural Network, one uses the *Graph* object as well as the input and output layer. Now, this class contains the logic to run training or prediction according to a **HyperParameter** object.

3.3 Features of the library

1. Optimization Algorithms: SGD, momentum, Adam
2. Layers: Convolutional, MaxPool, Dense, Logits (Softmax), LossLayer
3. Activation Functions: Sigmoid, ReLu
4. Initialization Types: Xavier for ReLu
5. Loss Types: Cross-Entropy, MSE
6. Use of Python bindings
7. Writing and Reading of Networks