Raul: on-device training library - API

AI enabling engineering team, Moscow

Noah`s Ark Lab

# Functionality

1.1 General

Training library main functionality:

* describe neural network topology
* train (fine-tune) neural network
* get trained weights for inference

1.2 Command syntax

All functions return API\_STATUS as a result code:

API\_STATUS <function\_name>(arguments…)

* 1. Operation statuses

|  |  |
| --- | --- |
| Error | Description |
| STATUS\_OK | No error |
| STATUS\_ERROR | General error |
| STATUS\_NOT\_IMPLEMENTED | Function not implemented |
| STATUS\_ERROR \_BAD\_SIZE | Incorrect size of argument or data array |
| STATUS\_ERROR \_BAD\_NAME | Wrong layer or layer parameter name |

Table 1.1 Summary of result codes

In case of error, it’s description can be queried with call to get\_last\_error**:**

const char\* get\_last\_error().

This function is thread-safe.

* 1. Data types

Graph\_Description\_t – describes the topology of neural network calculation graph

Graph\_Description\_Params\_t – represents parameters of one network layer

Graph\_t – represent neural network calculation graph itself

Optimizer\_t – represent weight optimization algorithm implementation

* 1. Create neural network topology

The workflow for defining neural network is the following:

1. Create Graph\_Description\_t:

Graph\_Description\_t\* graph\_descr;

create\_graph\_description(&graph\_descr);

1. Create layer description:

Graph\_Description\_Params\_t\* layer\_descr;

create\_graph\_description\_params(LAYER\_TYPE, LAYER\_NAME, &layer\_descr);

1. Set layer input and output tensors names

set\_graph\_description\_param\_inputs(layer\_descr, INPUTS);

set\_graph\_description\_param\_outputs(layer\_descr, OUTPUTS);

1. Set parameters for particular layer type. For example, dropout probability:

set\_graph\_description\_param\_dropout(layer\_descr, 0.5);

1. Add created layer to network. Graph\_Description\_Params\_t variable becomes inaccessible after adding to network.

add\_layer(graph\_descr, layer\_descr);

1. After adding all necessary layers, build the graph:

Graph\_t\* graph;

create\_graph(graph\_descr, &graph);

Steps 2-5 must be repeated for each layer in the network (input data layer, working layers, loss function). So there are shortcuts allowing to do steps 2-5 in a one function call.

|  |  |  |
| --- | --- | --- |
| Function | Params | Description |
| add\_activation\_layer |  | Sigmoid, Tanh, SoftMax, LogSoftMax, ReLU |
| add\_batchnorm\_layer |  |  |
| add\_convolution\_layer |  | 2D Convolution and 2D Depthwise Convolution |
| add\_data\_layer |  |  |
| add\_dropout\_layer |  |  |
| add\_elementwise\_layer |  |  |
| add\_fullyconnected\_layer |  |  |
| add\_globalaveragepooling\_layer |  |  |
| add\_loss\_layer |  | Cross-Entropy Loss or Negative Log-Likelihood Loss |
| add\_pooling2d\_layer |  | Max-Pooling and Average-Pooling layers |

* 1. Create optimizers for stochastic gradient descent

To train the network we need an optimizer. Currently, library supports following optimizers: Adadelta, Adagrad, Adam, Adamax, Momentum, Nesterov and naïve SGD.

Optimizer instance is created with one of the following creator functions:

|  |  |  |
| --- | --- | --- |
| Function | Params | Description |
| create\_adadelta\_optimizer |  |  |
| create\_adagrad\_optimizer |  |  |
| create\_adam\_optimizer |  |  |
| create\_adamax\_optimizer |  |  |
| create\_momentum\_optimizer |  |  |
| create\_nesterov\_optimizer |  |  |
| create\_sgd\_optimizer |  |  |

Typical usage of optimizer is the following:

Optimizer\_t\* optimizer;

float learning\_rate = 0.5;

create\_adadelta\_optimizer(&optimizer, learning\_rate);

… // use optimizer

delete\_optimizer(optimizer);

* 1. Dealing with weights

By default, all weights and biases are initialized to zero.

Initial weights and biases for layers are set the following way:

float\* weights;

float\* biases;

... // prepare weight and biases

set\_weights(graph, LAYER\_NAME, weights, SIZE);

set\_biases(graph, LAYER\_NAME, weights, SIZE);

After model training updated weights and biases can be queried using similar functions

get\_weights(graph, LAYER\_NAME, weights, SIZE);

get\_biases(graph, LAYER\_NAME, weights, SIZE);

* 1. Model training

Model training consists of two steps repeated for each data batch:

1. Provide training data to each input of the model and training labels to calculate loss

float\* input\_data;

... // load training data for batch

set\_input\_data(graph, INPUT\_NAME, data, SIZE);

1. Do a single iteration of training (forward, backward and model update) and obtain loss for particular batch:

float loss;

train\_single\_pass(graph, optimizer, &loss);

# Sample program

const float LEARNING\_RATE = 0.0001f;

const size\_t BATCH\_SIZE = 3;

const size\_t NUM\_CLASSES = 2;

const size\_t IMAGE\_WIDTH = 2;

const size\_t IMAGE\_HEIGHT = 1;

const size\_t IMAGE\_CHANNELS = 1;

const size\_t FC\_SIZE = 2;

Graph\_Description\_t\* desc = NULL;

ASSERT\_OK(create\_graph\_description(&desc));

{

const char\* tensors[] = { "data", "labels" };

add\_data\_layer(desc, "data", tensors, 2, IMAGE\_WIDTH, IMAGE\_HEIGHT, IMAGE\_CHANNELS, NUM\_CLASSES, BATCH\_SIZE);

}

add\_fullyconnected\_layer(desc, "fc", "data", "fc", FC\_SIZE);

add\_activation\_layer(desc, SOFTMAX\_ACTIVATION, "softmax", "fc", "softmax");

{

const char\* tensorsIn[] = { "softmax", "labels" };

add\_loss\_layer(desc, CROSS\_ENTROPY\_LOSS, "loss", tensorsIn, 2);

}

float input\_data[] = { 1, 0, 0, 1, 1, 1 };

float label\_data[] = { 1, 0, 0, 1, 1, 0 };

Graph\_t\* graph = NULL;

create\_graph(desc, &graph);

Optimizer\_t\* sgd\_optimizer = NULL;

create\_sgd\_optimizer(&sgd\_optimizer, LEARNING\_RATE);

auto totalBatchSize = IMAGE\_WIDTH \* IMAGE\_HEIGHT \* IMAGE\_CHANNELS \* BATCH\_SIZE;

set\_input\_data(graph, "data", input\_data, totalBatchSize);

set\_input\_data(graph, "labels", label\_data, NUM\_CLASSES \* BATCH\_SIZE);

for (size\_t epoch = 1; epoch < 10000; ++epoch)

{

float testLoss = 0.f;

auto timeStart = std::chrono::steady\_clock::now();

train\_single\_pass(graph, sgd\_optimizer, &testLoss);

printf("Epoch = %zu, Average loss = %f\n", epoch, testLoss);

}

delete\_optimizer(sgd\_optimizer);

delete\_graph(graph);