# Development and Implementation of an OpenCL Driven Neural Network for Use in Python

Heinrich Dinkel
1140339107
heinrich.dinkel@gmail.com

Daniel Laidig
J114030910
daniel@laidig.info

Christian WÃijrthner
c.wuerthner@me.com

## ABSTRACT

This paper describes the development and implementation process of a neural network which is driven by OpenCL using GPU acceleration. The neural network should be usable within Python using a wrapper for the shared object file of the C++ library.

## Categories and Subject Descriptors

H.4 [**Computer Science**]: Neural Networks

## General Terms

Theory

## Keywords

Neural Network, OpenCL, Python, ctypes, Stochastic Gradient Descent

## 1. BASICS

We use the following notation:

- Matrices are written in uppercase bold e.g. $\mathbf{X}$.
- Vectors are written in lowercase bold e..g $\mathbf{x}$.
- scalars are written in lowercase or uppercase. Lowercase indicates that it is a counting variable and uppercase that it is one of the limits in an finite set.
- For all functions , e.g. $f(x)$, where $x = \mathbf{X}$ , we apply the function element wise.
- Dot product is indicated by simple concatenation of two matrices/vectors e.g. $\mathbf{Xy}$.
- Element wise multiplication is indicated as $\mathbf{X} \otimes \mathbf{Z}$.

Neural networks are a statistical model for regression or classification. Even though naturally they do not produce any classification output, we can model the regressed output to be used as a classifier.

Overall we have two different modes: Training and evaluation. In the training phase we use the back propagation algorithm to train the network and adjust it so that it produces output, which is close to our target output.

Furthermore a neural network contains $L$ hidden layers, which are called hidden since their output is not directly observable.

The weights are in our simple case randomly initialized and then updated using the back propagation rule.

A basic neural network has therefore the following parameters:

- The weights from an neuron $k$ in layer $a$ to a neuron $j$ in layer $b$
- The input to the network, usually a vector $\mathbf{x}$ with $k$ values
- The target which should be estimated, usually a vector $\mathbf{t}$ with $q$ values.
- The learning rate $\eta$, which is the indicator how fast the network learns.

Moreover we can add some advanced techniques as the momentum to speed up the training the momentum

### 1.1 Gradient Descent

Gradient descent is a first-order optimization algorithm. To find a local minimum of a function using gradient descent, it takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point. If instead one takes steps proportional to the positive of the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent [2, 3].

Gradient descent is based on the observation that if the multivariable function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases fastest if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$, $-\nabla F(\mathbf{a})$ [4]. It follows that, if

$$\mathbf{b} = \mathbf{a} - \gamma \nabla F(\mathbf{a}) \tag{1}$$

for $\gamma$ small enough, then $F(\mathbf{a}) \geq F(\mathbf{b})$. With this observation in mind, one starts with a guess $\mathbf{x}_0$ for a local minimum of F, and considers the sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ such that

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n),\ n \geq 0. \tag{2}$$

Finally we have [1]:

$$F(\mathbf{x}_0) \geq F(\mathbf{x}_1) \geq F(\mathbf{x}_2) \geq \cdots, \tag{3}$$

## 1.2 Minibatch Stochastic Gradient Descent

Both statistical estimation and machine learning consider the problem of minimizing an objective function that has the form of a sum:

$$Q(w) = \sum_{i=1}^{n} Q_i(w) \tag{4}$$

where the parameter $w^*$ which minimizes Q(w) is to be estimated. Each summand function $Q_i$ is typically associated with the $i$-th observation in the data set (used for training).

When used to minimize the above function, a standard (or "batch") gradient descent method would perform the following iterations :

$$w := w - \eta \nabla Q(w) = w - \alpha \sum_{i=1}^{n} \nabla Q_i(w) \tag{5}$$

where $\eta$ is the learning rate.

But since in many cases, the update of one gradient would take to estimate the gradient over the whole dataset, the computation would take too much time. Often times we only need a rough estimate of the gradient, not a precise one.

What stochastic gradient descent does is to remove the sum and only uses one sample to estimate the gradient.

$$w := w - \eta \nabla Q(w) = w - \alpha \nabla Q(w) \tag{6}$$

In practice this turns out to be less effective than batch gradient descent, since we only get a very rough estimate of the gradient. Therefore the minibatch gradient descent technique is used. The difference here is that while batch gradient descent uses $n$ samples and stochastic gradient descent uses only 1, minibatch gradient uses $b$ training samples, where $b << n$. Typical sizes for $b$ are in the range if $b \in 2, \ldots, 100$.

Therefore in minibatch stochastic Gradient descent we randomly subset the training set and only take some samples out of it and estimate the gradient out of the seen sample set.

The important thing to do is to randomize the training set, since otherwise only a small portion of the training set will be seen and leads to wrong gradients.

## 1.3 The Forward Backward procedure in detail

To train the network we use the standard back-propagation algorithm, which essentially does a forward and a backward pass of the network in a row. Firstly we initialize the weights in a matrix.

$$\mathbf{W}_{j\times k} = \begin{bmatrix} w_{11} & w_{12} & \ldots & w_{1k} \\ w_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ w_{j1} & \ldots & \ldots & w_{jk} \end{bmatrix}, \mathbf{x}_{1\times k} = \begin{bmatrix} a_1 \\ \vdots \\ \vdots \\ a_k \end{bmatrix} \tag{7}$$

*Forward propagation.* is one of the two passes the network needs to do ( hence it's name ). In forward propagation the network calculates the predicted output of the network.

The output of the hidden layer $j$ of the network can be calculated as the weighted sum of the inputs

$$\mathbf{nnet}_j = \sum_{k=1}^{n} w_{kj} x_k = \mathbf{Wx} \tag{8}$$

Even though this method should already work, we add a bias to every node to increase the learning speed and void that the network stops learning.

$$\mathbf{nnet}_j = \sum_{k=1}^{n} w_{kj} x_k + b_k = \mathbf{Wx} + \mathbf{b} \tag{9}$$

r

Later we see that back-propagation needs some variables, which can be already precomputed during the forward step. These two variables are the output $\mathbf{o}_j$ of layer $j$ and the derivative of the output w.r.t to the weighted sum $\mathbf{nnet}_j$. We represent the derivatives in a matrix $\mathbf{D}$ and the output of the current layer

$$\mathbf{D}_j = \frac{\mathbf{o}_j}{\mathbf{nnet}_j} \tag{10}$$

In case of sigmoid activation function, we obtain:

$$\mathbf{D}_j = \varphi(\mathbf{nnet}_j)(1 - \varphi(\mathbf{nnet}_j)) \tag{11}$$

For the outputs, we simply feed forward the network and store the output $\mathbf{o}_j$ in out buffered lists.

$$\mathbf{o}_j = \varphi(\mathbf{Wx} + \mathbf{b}) \tag{12}$$

As soon as the feed forward is done, we produced an output of the network, which we can denote as $\mathbf{o}_L$. Now we need to decide how close our output is to the targetted output $\mathbf{t}$. For this we use a cost function in usual cases it is sufficient to use $MSE$ as such, which is defined as:

$$\mathbf{err} = \frac{1}{2}(\mathbf{o}_L - \mathbf{t})^2 \tag{13}$$

Here (13) we need to make sure that the last output layer has the same dimensions as the target output.

*Backpropagation.* starts when all layers are trained. The idea behind back propagation is that the hidden layers do not produce any output. So we cannot modify their weights directly, since we donâĂŹt know how large the error was. With back propagation we calculate the error at the output layer $L$ and then propagate this error to the hidden layers back. Therefore we update all $L - 1$ weight matrices and biases.

At first we calculate the differences between the target output and the estimated output.

$$\boldsymbol{\delta}_L = \mathbf{err} \otimes \mathbf{D}_L \tag{14}$$

$$\nabla \mathbf{W}_L = \mathbf{o}_L \boldsymbol{\delta}_L^T \tag{15}$$

We store both, the deltas and the nablas to later update the weights ( with the $\nabla$s) and the biases (using the $\delta$). From here on we then calculate the other layers backwards, we do:

$$\boldsymbol{\delta}_i = \mathbf{D}_i \otimes \left( \mathbf{W}_{i+1}^T \boldsymbol{\delta}_{i+1} \right) \tag{16}$$

$$\nabla \mathbf{W}_i = \mathbf{o}_i \boldsymbol{\delta}_i^T \tag{17}$$

Whereas we again store the deltas and the nablas to later update the weights.

To update the weights, we use the usual gradient descent update rule:

$$\mathbf{W}^* = \mathbf{W} - \alpha \nabla \mathbf{W}_i^T \tag{18}$$

$$\tag{19}$$

To update the biases, we use essentially the same update rule, but only consider the given $\boldsymbol{\delta}_i$s.

$$\mathbf{b}^* = \mathbf{b} - \alpha \boldsymbol{\delta}_i^T \tag{20}$$

Finally, if we would like to improve the converging speed we can apply momentum. Momentum adds extra ”velocity”towards the gradient curve, by using the last estimated value of the gradient $\mathbf{W}_{i-1}^*$ ($i$ denotes the current iteration) and applying on that the momentum ($\alpha$) as:

$$\mathbf{W}_{i+1} = \mathbf{W}_i - \eta \nabla \mathbf{W}_i + \alpha \mathbf{W}_i \tag{21}$$

The momentum is initialized with 0 and takes effect after one iteration of gradient descent.

## 2. IMPLEMENTATION DETAILS

OpenCL offers multiple implementation types. It is natively written in C, but has also a C++ wrapper onboard. We used in our project the C++ wrapper since it is more naturally to use that in a C++ project. First one needs to include the necessary header into any class.

```
#include <CL/cl.hpp>
```

**Listing 1: OpenCL C++ header**

*The interface to OpenCL.* was written in C++11 and makes heavily use of the current variadic args feature. For this small scale task, the interface is definitely too complex, but it can be used for any other OpenCL task.

```
__kernel void mult(const int wSrc, __global const float* A,__global const float* B,__global float* output)
{
   const int idx = get_global_id(0);
   const int idy = get_global_id(1);

   output[idy*wSrc+idx] = A[idy*wSrc+idx] * B[idy*wSrc+idx];
}
```

**Listing 2: Example Kernel function**

OpenCL uses externally defined functions ( coined as kernels ) to compile the code and run it, during the runtime. Therefore one needs to write a kernel for its needs. An example kernel can be seen at 2. In our case we defined various kernels, for dot products and other matrix operations.

Since OpenCL is a multi device and platform GPU/CPU interface, one needs to first figure out which kind of platform (e.g. NVIDIA,AMD,Intel) the current machine is using and then decide which accelerator will be used 2.

```
void exampleplatform(const char * programpath){
std::vector<cl::Platform> all_platforms;
cl::Platform::get(&all_platforms);
//Assume having only one platform
cl::Platform defaultplatform = all_platforms[0];
std::vector<cl::Device> all_gpu_devices;
//CL_DEVICE_TYPE_GPU can also be CL_DEVICE_TYPE_CPU for CPU
defaultplatform.getDevices(CL_DEVICE_TYPE_GPU, &all_gpu_devices);
//Assume having only one device / take the first
cl::Device device = all_gpu_devices.front();
//Init the current context with the device
cl::Context(device);
//We wrote a helper function here to get the string data out of the kernel file
const char* content = util::file_contents(programpath);
}
```

**Listing 3: Get Platforms and devices**

Moreover since OpenCL is compiled during runtime, one needs to extract the code from the kernel file (e.g. kernel.cl) and give this content to the OpenCL.

As soon as the context is initialized we can run our kernels on the device. The problem here is that the kernel is defined with it's own parameters and types, but we dont know these during the compile time. To have a universal interface we used the variadicargs feature to allow the programmer a very straight forward way to use any kernel function.

```
void runKernel(const char *kernelname){
cl::Program::Sources sources;
//contents is the already read out content of the kernels file (e.g. kernels.cl)
sources.push_back(std::make_pair(contents,strlen(contents)+1));
//Init the program with the context and the source
cl::Program program(context,sources);
//Build the program on the device
program.build({device});
cl::CommandQueue queue(context,device);
//The operator allows us to set args to the kernel
cl::Kernel kernel_operator(program,kernelname);
//kernel_operator allows us to send arguments to the kernel by calling
//kernel_operator.setArgs(ARGNUMBER,ARGUMENT);

//Init space on the device using cl::Buffers
//Send Arguments to the device ....
// quene.enqueueWriteBuffer() .....
//Wait until the arguments did arrive on the device
queue.finish();

//Execute!
cl::Event event;
queue.enqueueNDRangeKernel(kernel_operator,cl::NullRange,cl::NDRange(10),cl::NullRange,NULL,&event);
//Wait until the execution has finished
event.wait();
// Read out the results by calling
// quene.enqueueReadBuffer()
}
```

**Listing 4: Kernel usage**

The usual Kernel execution can be seen in 2. Even though it is recommended to use OpenCL in this fashion, we cannot cope with different arguments for the kernel. Therefore we would need to init a device and context every time ( or at least as a singleton ) and then rewrite the argument passing for every parameter independently. This would take a lot of time, if the GPU is used extensively.

Our solution to that problem is as follows:

```
class OpenCL{

//Constructor ..... etc
// Hook for the iteration
template<std::size_t P=0,typename... Tp>
   typename std::enable_if<P == sizeof...(Tp), void>::type addkernelargs(std::tuple<Tp ...>&& t,cl::Kernel &k,cl::CommandQueue &,std::vector<cl::Buffer> &outputbuffers) const{
   // Do nothing
   }

//  Start of the iteration
template<std::size_t P = 0, typename... Tp>
   typename std::enable_if< P < sizeof...(Tp), void>::type addkernelargs(std::tuple<Tp...> && t,cl::Kernel &kernel,cl::CommandQueue &,std::vector<cl::Buffer> &outputbuffers) const{
      // Type
       typedef typename std::tuple_element<P, std::tuple<Tp...> >::type type;

       // Add the value of the current item from std::get<P> to the args in kernel
       // This function decides which type the kernel arg is
       addkernelarg(P, std::get<P>(t), kernel,queue,outputbuffers);

       // Recurse to get the remaining args
       addkernelargs<P + 1, Tp...>(std::forward<std::tuple<Tp...>>(t), kernel,queue,outputbuffers);
   }

// Adding Std::vector as type to the kernel args list
   template<typename T>
   void addkernelarg(std::size_t i, std::vector<T> const & arg, cl::Kernel & kernel,cl::CommandQueue &) const{
      cl::Buffer buffer(this->context,CL_MEM_READ_WRITE,arg.size()*sizeof(T));
      queue.enqueueWriteBuffer(buffer,CL_FALSE,0,sizeof(T)*arg.size(),&(arg[0]));
      kernel.setArg(i,buffer);

   }


// Adding any array into the kernel args
   template<typename T,std::size_t N>
   void addkernelarg(std::size_t i, T const (& arg)[N], cl::Kernel & kernel,cl::CommandQueue &) const{
      cl::Buffer buffer(this->context,CL_MEM_READ_WRITE,N*sizeof(T));
      queue.enqueueWriteBuffer(buffer,CL_FALSE,0,sizeof(T)*N,&arg);
      kernel.setArg(i,buffer);
   }

   // Adding any constant to the kernel
   template<typename T>
   void addkernelarg(std::size_t i, T const & arg, cl::Kernel & kernel,cl::CommandQueue &) const{
        cl::Buffer buffer(this->context,CL_MEM_READ_WRITE,arg.size()*sizeof(T));
        queue.enqueueWriteBuffer(buffer,CL_FALSE,0,sizeof(T)*arg.size(),&(arg[0]));
        kernel.setArg(i,buffer);
    }

}
```

**Listing 5: Add arguments to kernel dynamic way**

This recipe can be used to do the same actions for the reading buffers out after the transfer has finished. Therefore we can create a highly dynamic wrapper class for OpenCL.

## 2.1 Transparently handling OpenCL within a Matrix class

To be able to use high level C++ code to implement the feedforward and backpropagation algorithm, we introduced a CL_Matrix class, which provides common matrix operation like multiplication, addition and so on. Most matrix operations are implemented as OpenCL kernels on the GPU. To avoid unnecessary transfers between the RAM and the GPU memory, the matrix class implements a state handling. Two methods for this are shown below, `syncToRam()` and `moveToRam()` work in a similar way.

```
template<typename T>
inline void CL_Matrix<T>::syncToGpu() const {
    if (this->state == OnlyRam) {
        this->gpu_buf = this->_cl.putDataOnGPU(this->mat);
        this->state = Synced;
    }

}

template<typename T>
inline void CL_Matrix<T>::moveToGpu() {
    this->syncToGpu();
    this->state = OnlyGpu;

}
```

**Listing 6: State handling to allow the Matrix data to be both used in RAM and in the GPU memory**

This way, we can both perform OpenCL operations and access the data from the normal C++ code, for example to get the result. Data transfers are avoided if possible, so for example multiple consecutive OpenCL operations can be performed without transfering data.

## 2.2 Use a C++ Class in Python

To use a C++ class in a Python program, Python's ctypes library can be used to invoke functions on a shared object file. The class will be wrapped into a Python class which uses the ctypes library to invoke functions on the C++ object. As the ctypes library is not able to handle C++ objects directly, a wrapper function must be written in C for every function of the C++ class. These plain C functions can then be invoked by ctypes.

A simple example of such a C++ class is seen in the following figure:

```
class WrapperExample {
    std::string* name;

public:
    WrapperExample(std::string name) {
        this->name = new std::string(name);
    }
    ~WrapperExample() {
        delete this->name;
    }
    void sayHello() {
        std::cout << "Hello " << *(this->name) << "!" << std::endl;
    }
};

extern "C" {
    WrapperExample* WrapperExample_new(char* name) {
        return new WrapperExample(name);
    }
    void WrapperExample_sayHello(WrapperExample* e) {
        e->sayHello();
    }
    void WrapperExample_delete(WrapperExample* e) {
        delete e;
    }
}
```

**Listing 7: Example C++ class for wrapping a C++ object in a Python object (Example.cpp)**

To compile the given C++ code into a shared object file which can be used by Python's ctypes library, following compilation statements can be used:

```
g++ -std=c++11 -c -fPIC Example.cpp -o example.o
g++ -std=c++11 -shared -Wl,-soname,example.so -o example.so  example.o
```

**Listing 8: Compilation of the C++ class into a shared object file**

The Python wrapper will call the functions defined in the `extern` section. For every function in the `extern` section, the Python class will have one function to wrap it. The function `WrapperExample_new()` will be called in the constructor of the Python class and `WrapperExample_delete()` in its destructor. All functions besides the `WrapperExample_new()` function receive a pointer to an instance of the C++ class `WrapperExample` on which the function should be invoked. These objects are created in `WrapperExample_new()` and the pointers to them are stored by Python.

The corresponding Python class to wrap the C++ class shown above looks like this:

```python
from ctypes import cdll

lib = cdll.LoadLibrary('example.so')

class ExampleWrapper:
    def __init__(self, name):
        self.obj = lib.WrapperExample_new(name)

    def sayHello(delf):
        lib.WrapperExample_sayHello(self.obj)

    def __del__(self):
        lib.WrapperExample_delete(self.obj)
```

**Listing 9: Python class wrapping the C++ class shown above (Example.py)**

The Python class `ExampleWrapper` can now be used as any other Python class.

## 2.3 Using the Interface

The Python class wrapping our `NeuralNetwork` class can be found in `NeuralNetwork.py`. To create a new `NeuralNetwork` the following constructor can be used:

```python
nn = NeuralNetwork(layerCount=3,
                   layerSize=np.array([784, 1000, 10]),
                   actFunctions=np.array([1, 1]),)
```

**Listing 10: Constructor to create a neural network**

In this case a neural network with 3 layers is created. The list `layerSize` contains the size of every layer. The neural network created in this example has an input layer with 784 nodes, one hidden layer with 1000 nodes and a output layer with 10 nodes. The activation functions are defined by the list `actFunctions`. The value 0 will set the activation function TAN_H and the value 1 SIGMOID. As the input layer has no activation function, the size of the list `actFunctions` is only 2. The sizes of the input and output layers can later be queried by the functions `getInputSize()` and `getOutputSize()`.

To train the neural network, the functions `trainBatch()` and `trainStochastic()` can be used. Both functions expect two matrices as parameters. The first matrix defines the input values which will be given to the neural network and the second parameter defines the expected output values. One row in one in the matrices is one train case for the neural network. The functions return a matrix with the errors. `trainStochastic()` uses stochastic gradient descent. The learning rate and momentum can be defined by the corresponding parameters `learningRate` and `momentum` and the number of epochs can be defined. An example usage can be seen in the following listing:

```python
errors = nn.train(trainImages[:, 0:20], trainOutput[:, 0:20], learningRate=0.2, momentum=0.1, numEpochs=10000)
print errors

errors = nn.trainsgd(trainImages[:, 0:20], trainOutput[:, 0:20], learningRate=0.2, momentum=0.1, numEpochs=10000)
print errors
```

**Listing 11: Usage of trainsgd() and train()**

The function `test()` can be used to test a input value with the neural network. In difference to `train()` only one matrix is expected as parameter. The values from the matrix will be given to the neural network. One column of the matrix is one test case. The function will return a matrix with the value of the output nodes for every test case.

```python
result = nn.test(testImages[:, 0:20])
print result
```

**Listing 12: Usage of test()**

The current state of the neural network including the weight biases and the configuration can be dumped into a file using the `save()` function. The path to the file must be supplied as a parameter. To restore the neural network later, following constructor can be used:

```python
nn.save(”path/to/savefile.bin”)

nn2 = NeuralNetwork(saveFile=”path/to/savefile.bin”);
```

**Listing 13: Constructor to load a dump file**

## 3. REFERENCES

[1] A. Cauchy. *Méthode générale pour la résolution des systèmes d'équations simultanées*. 1847.

[2] K. C. Kiwiel. Convergence and efficiency of subgradient methods for quasiconvex minimization. *Mathematical Programming*, 90(1):1–25, Mar. 2001.

[3] N. Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145 – 151, 1999.

[4] Y.-x. Yuan. Step-sizes for the gradient method. *AMS/IP Studies in Advanced Mathematics*, 42(2), 1999.