Neural Network on Compute Shader
Running and Training a Neural Network using GPGPU

Fredrik Åström
Bachelor Thesis in Computer Science
Blekinge Institute of Technology

2011-07-06

Abstract

In this thesis I look into how one can train and run an artificial neural network using Compute Shader and what kind of performance can be expected.

An artificial neural network is a computational model that is inspired by biological neural networks, e.g. a brain.

Finding what kind of performance can be expected was done by creating an implementation that uses Compute Shader and then compare it to the FANN library, i.e. a fast artificial neural network library written in C.

The conclusion is that you can improve performance by training an artificial neural network on the compute shader as long as you are using non-trivial datasets and neural network configurations.

Keywords: Artificial Neural Network, GPGPU, Compute Shader
Contents

1 Introduction 1
   1.1 Research Question .............................................. 1
   1.2 Hypothesis ....................................................... 2
   1.3 Method .......................................................... 2
   1.4 Target audience .................................................. 2

2 Neural Network Theory 2
   2.1 This Artificial Neural Networks ................................. 2
      2.1.1 Layered Feed forward Neural Network .................... 3
      2.1.2 iRprop− ....................................................... 3
      2.1.3 Sigmoid function ............................................ 4

3 The FANN Library 5

4 GPGPU 6
   4.1 Compute Shader in Shader Model 5.0 ............................. 6
      4.1.1 Synchronization ............................................. 6
      4.1.2 Buffers ...................................................... 6
      4.1.3 Shader Macro .............................................. 7

5 The implementation 7
   5.1 Data structures .................................................. 7
   5.2 The Run Shader ................................................... 8
   5.3 The Test Shader ................................................... 9
   5.4 The Learn Shader ................................................ 10

6 Benchmarks 10
   6.1 General Setup ................................................... 11
   6.2 Quality Benchmarks ............................................. 11
      6.2.1 Benchmark Setup .......................................... 11
      6.2.2 Benchmark Results ....................................... 12
   6.3 Performance Benchmarks ......................................... 15
      6.3.1 Connection Benchmark Results ........................... 16
      6.3.2 Dataset Benchmark Results ............................... 18

7 Discussion 19

8 Conclusion 20
9 Future Work

A Appendix

A.1 NNCommon.hlsl — Used by all shaders ............... 21
A.2 NNCommonLR.hlsl — Used by Test and Run shaders .... 21
A.3 NNRun.hlsl — The Run Shader code ................. 22
A.4 NNTest.hlsl — The Test Shader code ............... 23
A.5 NNLearn.hlsl — The Learn Shader code ............ 25
1 Introduction

Our brains are biological neural networks that can perform a multitude of task and their design has inspired a computational model known as artificial neural networks (ANN). An artificial neural network is just as the name suggests a network of connected neurons. To change what a network does all we has to do is change the connections. There are different kinds of neural networks and one of them is the feed forward ANN that consists of layers of neurons where each neuron can only be connected to neurons in the previous and the next layer [5].

An ANN can learn in multitudes of ways and one such way is the iRprop–training algorithm. iRprop– uses datasets of input and wanted output that is fed into the network and then adapts it so that the actual output hopefully comes closer to the wanted output [2].

The FANN library is a C implementation of an ANN originally created by Steffen Nissen. It can handle a number of different kinds of networks, a number of different training algorithms and the original design are documented in [5] and details about some of the newer functions can be found at [1].

A Graphics Processing Unit (GPU) is a processor that is very good at parallelization and also specialized at graphics processing. To make it easier for programs to be able to use different manufacturers’ GPU there have evolved different standards and different API’s. DirectX is one such API and at the time of writing, 2011-07-06, it is in version 11. To do General-purpose computing on Graphics Processing Units (GPGPU) with the DirectX 11 API they have something called a Compute Shader. A Shader is a program for the GPU and a Compute Shader is the only shader in the DirectX 11 API made for GPGPU [3].

1.1 Research Question

What difference is there between running a feed forward ANN using iRprop– on Compute Shader compared to using the Fast Artificial Neural Network (FANN) library from a performance viewpoint?
1.2 Hypothesis
Running a feed forward ANN on the GPU instead of on the CPU increases
the performance on larger networks and for bigger datasets.

1.3 Method
First create an ANN implementation on Compute Shader. Test the network
with regards to quality and to performance and then analyze the results.
The quality benchmarks will be utilizing datasets from the PROBEN1[6]
dataset collection.

1.4 Target audience
Anyone interested in programming for GPUs, the use or implementation of
artificial neural networks is the primary target audience. People interested
in optimizing and AI in general may also be interested in this thesis.

2 Neural Network Theory
An Artificial Neural Network is a collection of connected neurons. There
are many different ways to implement an artificial neuron. See Equation 1
for the general mathematical definition.

\[ y(x) = g \left( \sum_{i=0}^{n} w_i x_i \right) \]  

(1)

\( x \) is a neuron with \( n \) input connections \((x_0 \cdots x_n)\) and one output \( y(x) \)
where \((w_0 \cdots w_n)\) are weights that determines how the input should be
weighted. \( g \) is the activation function that determines the strength of the
output that can then be used as an input for other neurons. Depending on
the activation function used we get a different range of output values with
the most common ones using either the range of 0 to 1 or -1 to 1.

2.1 This Artificial Neural Networks
This section is going to focus upon the theories behind the implementation
used in the experiments. The implementation is going to use a layered feed
forward neural network with bias neurons and a Sigmoid activation function.
The learning algorithm is going to be the iRprop– algorithm.
2.1.1 Layered Feed forward Neural Network

A Layered Feed forward ANN is a network that is divided in different layers where each layer has a set of neurons that can only get input from the previous layer and give output to the next if it exists. There are three different kinds of layers, input, hidden and output layers, see Figure 1. Each layer except the output layer has an additional bias neuron that always outputs 1.

![Layered Feed forward Neural Network](image)

Figure 1: A three layer feed forward neural network with bias neurons

The input layer is the first layer and its neurons does not have any input connections instead their value is determined by another way. After the input layer come the hidden layers if there are any. The hidden neurons, i.e. the neurons in the hidden layer, have both input connections and output connections. The output layer is the last layer and its neurons do not have any output connections as its the last layer.

2.1.2 iRprop–

iRprop– is a first-order optimization algorithm for training a neural network. iRprop stands for improved Resilient backpropagation. There are mainly two different versions of iRprop, iRprop+ and iRprop–. The main difference between the two is that iRprop+ use weight-backtracking while iRprop– does not. See Figure 2 for the iRprop– algorithm.
for each \( w_{ij} \) do

\{
\begin{align*}
&\text{if } \frac{\partial E(t-1)}{\partial w_{ij}} \cdot \frac{\partial E^t}{\partial w_{ij}} > 0 \text{ then} \\
&\quad \Delta_{ij}^t := \min(\Delta_{ij}^{(t-1)} \cdot \eta^+, \Delta_{max}) \\
&\text{else if } \frac{\partial E(t-1)}{\partial w_{ij}} \cdot \frac{\partial E^t}{\partial w_{ij}} < 0 \text{ then} \\
&\quad \{ \\
&\quad \quad \Delta_{ij}^t := \max(\Delta_{ij}^{(t-1)} \cdot \eta^-, \Delta_{min}) \\
&\quad \quad \frac{\partial E^t}{\partial w_{ij}} := 0 \\
&\quad \} \\
&\quad w_{ij}^{(t+1)} := w_{ij}^t - \text{sign} \left( \frac{\partial E^t}{\partial w_{ij}} \right) \cdot \Delta_{ij}^t \\
\}
\end{align*}
\}

Figure 2: The iRprop– algorithm. Where \( 0 < \eta^- < 1 < \eta^+ \) and where \( \eta \) stands for learning rate. The partial derivative \( \frac{\partial E}{\partial w_{ij}} \) is the partial derivative for the activation function. \( t \) is for time or epoch. The function \( \text{sign}(x) \) returns +1 for positive values of \( x \), –1 for negative and otherwise 0.

A simplified explanation of how the iRprop– algorithm could be implemented follows but for a more elaborate and complete explanation of the algorithm see [2].

First before running the algorithm we need to have all the weights calculated for a specific set of input values and have a set of wanted output values for the input. The algorithm takes an error value derived from the difference in the output neuron values and the wanted output values and sends them backwards throughout the network. The error values are then used to change the weight values so to try and make the output neurons values more similar to the wanted values. In the end how much was changed is saved and used for the next iteration.

2.1.3 Sigmoid function

The Sigmoid function is a common activation function in artificial neural networks. See Equation 2 for the formula and Figure 3 for a graph.

\[ g(x) = \left( \frac{1}{1 + e^{-2sx}} \right) \]  \( (2) \)
Figure 3: A graph of the sigmoid function when $s = 0.5$.

$s$ is in this case the steepness. The steepness is used to control how steep the function should be.

3 The FANN Library

The Fast Artificial Neural Network library is an open source neural network library written in C. It has support for many different network structures, different activation functions and different training algorithms. It has extensive documentation, a whole web page [1] with links to a wiki and much more, it also has a university report [5] that can also be found at the web page. The university report is looking at version 1.0 when the library is today, 2011-07-06, at version 2.1.

There are many different language APIs available, e.g. C++, Python and Lua.
4 GPGPU

GPGPU stands for General-purpose computing on Graphics Processing Units and to do any computing on DirectX 11 supported hardware we need to use shaders. A shader is a program that is executed on the GPU. There are different kinds of shaders, e.g. Compute Shader, Pixel Shader and Vertex Shader. Of the ones in the Direct3D 11 interface the Compute Shader is the only one focused upon GPGPU.

4.1 Compute Shader in Shader Model 5.0

DirectCompute have three concepts when dealing with parallelization, a thread, a threadgroup and a dispatch. A dispatch is a number of threadgroups, a threadgroup is a group of threads and a thread is one execution of the shader. The number of groups in a dispatch can be defined at runtime while the threadgroup size is defined at shader compile time.

The maximum number of threads per threadgroup is 1024 [4] in Shader Model 5.0.

4.1.1 Synchronization

The only synchronization possibility is between threads in the same threadgroup. Synchronization cannot be in a branch so if we want to loop trough things while synchronizing between each loop we need to be able to know in compile time how many times it will be called.

4.1.2 Buffers

A buffer can be bound to a register. A bound buffer can be used by shaders by defining a variable that uses that register. Binding the buffers and creating them is done purely on the CPU.

There are different kinds of buffers, e.g. append buffer and structured buffer. An append buffer is a buffer that we can only append values to. The actual size of the append buffer needs to be defined from the beginning from the CPU and does never actually get bigger.

A structured buffer can be either read only or read and write. What makes a structured buffer a structured buffer is that it can be of any type or a struct.
4.1.3 Shader Macro

Shader Macros are just like regular macros but for shader files. These can be set from the program at shader compile time. Just like regular macros they consist of a name and a definition and each time the compiler finds an identifier with the same name as the macro name then it replaces that with the macro definition.

Macro definitions can be used to set threadgroup size at compile time and to be used to make it possible to loop through a number of elements defined by the macro while using synchronization.

5 The implementation

My implementation is based upon the FANN library’s implementation in that it uses the same default values and the same algorithms and uses similar tricks, e.g. the partial derivative of the activation function clips the neuron values from 0.1 to 0.99 in both the FANN implementation and my implementation.

My implementation uses three different shaders, the Run shader, the Test shader and the Learn shader. To train a network the program first calls the Test shader to calculate the slopes and MSE for each dataset and then it calls the Learn shader to update the weights and add together all the different dataset’s MSE. The run shader is used when we want to calculate the output and be able to retrieve it from the GPU.

The activation function is a Sigmoid function described in section 2.1.3 with the steepness as 0.5 as default. To change the function one unfortunately need to change the function activation and activationDerived in a shader file named "NNCommon.hlsl".

In regards to the iRprop– algorithm the Test shader calculates all the $\frac{\partial E}{\partial w_{ij}}$ and then the Learn shader does the rest.

As according to the defaults values in the FANN library my implementation is using the values $\eta^- = 0.5$, $\eta^+ = 1.2$, $\Delta_{min} = 0$ and $\Delta_{max} = 50$. It also sets the previous steps as minimum 0.001 so to make sure the training does not stop thanks to rounding [5].

5.1 Data structures

Each network consists of a number of layers. Each layer consists of a number of neurons and each neuron is associated to a number of connections. In this implementation we have a layer buffer that holds a number of layers defined
in a shader macro and each layer consist of the layer size, the first neuron index and the first connection index for the first neuron.

The index values are used to access a value in different buffers. Most buffers needs to have a dataset offset so that all datasets can run at the same time and is meant to be used with one of the indexes, e.g. the neuron index and the connection index.

![Figure 4: Data structure for a three layer network when using one dataset. If we use more datasets then there will be a couple of offset for each threadgroup to be added to all indexes.](image)

So to access e.g. the neuron value for this dataset’s group in the first layer I take the neuron offset and add the first neuron index from the layer buffer first layer and then use that new index in the neuron value buffer.

### 5.2 The Run Shader

The Run shader is the shader to use when we need to run through the neural net and get the output. Each threadgroup is associated with one dataset. The threadgroup size is determined by the biggest layer size and set with a shader macro sent in at shader compile time. It also needs the layer’s size
as a shader macro so that it can use synchronization while looping through each layer.

The buffers available for the shader are the layer buffer, input buffer and the output buffer. The output buffer is later used to store the output neuron values. It has a groupshared float array that is the size of the threadgroup and is used to store the neuron values between each layer.

The first thing it does is to put all the input values for each threadgroup’s dataset into the groupshared array. Then it enters the layer loop that goes through each layer. Each neuron thread then calculates the value for the neuron associated with it and then waits for synchronization before it stores it into the groupshared array. The groupshared array only ever holds one layer’s neuron values as that is all that is needed to traverse the network. This is repeated until it has calculated the neuron values for the output layer and then stores those values inside the output buffer.

See Appendix A.3 for the actual shader code.

5.3 The Test Shader

The Test shader is used to run through a dataset, backpropogate the error values and then to calculate the slopes. The threadgroups and the layers size are set the same way as the Run shader does and because of the same reasons.

The buffers it uses are the layer buffer, input buffer, mse buffer, value buffer and slope buffer. This output buffer is not the same as the Run shaders output buffer as it holds the data that we want the output to become and is read only. The slope buffer holds the slopes for each neuron in each threadgroup, the slope values are to be added up in the Learn shader later, see Section 5.4, as it is hard to add together values from different groups effectively and thread safe. The value buffer is needed as the neuron values are needed when backpropogating the error value.

The first part of the Test shader is almost exactly as the Run shader with the only difference being that the Test shader saves the neuron values not only to the groupshared array but also to the values buffer.

After calculating the values for all the neurons it calculates the errors by comparing the output layers values to the output buffer values. At this stage it also calculates the MSE for that dataset by first moving all the output neurons MSE to groupshared memory and then one thread adds them all up and puts it in the mse buffer. After that it puts the error values to be backpropogated into the groupshared array. Then the error values needs to be backpropogated and for the slopes o be calculated. This implementation
does this by first using the error for that layer to calculate the slopes and then calculates the errors for the next layer. The slopes for each layer are placed into the slope buffer. This then repeats until it has reached the top layer.

See Appendix A.4 for the actual shader code.

5.4 The Learn Shader

The Learn shader runs after the Test shader and changes the connection weights in the network according to the slopes that have been calculated in the Test shader. The total amount of threads is determined by the amount of connections in the network. If the weights are 1024 or fewer then everything is done in one threadgroup but otherwise it is using 64 as the threadgroup size. The reason for this is to try lessen the amount of threads not doing anything but at the same time not have a large amount of threadgroups.

The buffers it uses are the weight buffer, slopes buffer, the previous slopes and step buffer, the MSE buffer and the MSE output buffer. The MSE output buffer is used to output the MSE for all the datasets to the CPU while the MSE buffer is the buffer holds each datasets own MSE.

The first thing the Learn shader does is to add up the MSE values in the MSE buffer and then one of the threads adds it to the MSE output buffer. It then adds up all slopes for that weight and load the previous slope and step data and uses that to change the weights. The previous slope and step buffer have the values associated with this weight is then set to the new slope and step values.

See Appendix A.5 for the actual shader code.

6 Benchmarks

To test my implementation against the FANN implementation I will run a couple of benchmarks. The benchmarks can be divided in two groups, quality benchmarks and performance benchmarks.

The Quality Benchmarks focuses upon how good the implementation is at training different networks. One shows the mean square error over time and the other shows it over epochs.

The Performance Benchmarks focuses upon how fast the implementations are. One set of benchmarks looks at the execution time per connections with different number of layers and neurons per layer. Another looks at the execution time per dataset.
6.1 General Setup

Both implementations are compiled with the same compiler and compile options. All Benchmarks are done on 4 different machines but with identical hardware. The processor is an Intel Core i7 920 with 4 cores, 8 threads and a clock speed of 2.66 GHz, the GPU is an NVIDIA GeForce GTX 480 and with 6 GB RAM.

The FANN library is running behind a C++ wrapper that is a part of the standard FANN library and can be found at [1].

In each graph MANN stands for My Artificial Neural Network, i.e. my implementation of an ANN on compute shader, and FANN stands for Fast Artificial Neural Network, i.e. the FANN library implementation.

6.2 Quality Benchmarks

To make sure that my implementation works as it should and to find out how similar its execution is to the FANN library it is needed to do some test between them. For this I am going to let both implementations train on the same dataset while saving the mean square error in between each epoch and save the execution time.

In the execution time I’m only counting the actual training of the network and not setting it up or setting the input values and the wanted output values.

To make sure the two implementations have an as similar as possible starting point they both use the same starting network weights and the same network structure. This is done by having my implementation initialize a network and then save it to two files, one for the FANN implementation and one for my implementation with both containing the same weight data. These files are then used during benchmarking.

6.2.1 Benchmark Setup

For this benchmark I will use the Proben1 collection of datasets\(^1\). I will only use three of the datasets as I am only really interested in making sure that my implementation works as it should. The network sizes that will be used are the same that are suggested in [6].

\(^1\)The FANN library includes the Proben1 datasets and it is those that are being used.
<table>
<thead>
<tr>
<th>Dataset name</th>
<th>Dataset size</th>
<th>Inputs</th>
<th>Hidden neurons</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>building</td>
<td>4208</td>
<td>14</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>mushroom</td>
<td>8124</td>
<td>125</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>soybean</td>
<td>683</td>
<td>82</td>
<td>16 + 8</td>
<td>19</td>
</tr>
</tbody>
</table>

Figure 5: The datasets used. 16 + 8 means that there are two hidden layers, the first with 16 neurons and the second with 8 neurons.

For testing I will run both implementations on the same configuration and starting weights for 1000 epochs and note the execution time between each epoch and the mean square error between them. In the case of my implementation the mean square error will be taken from the mse buffer afterwards but the execution times will still be gathered between each epoch. I will only do one training session for each problem as this is just for making sure that my implementation behaves similarly to the FANN library’s implementation. The results from these benchmarks should produce similar curves between the implementations but not exactly the same thanks to how floating point arithmetic works and its accuracy.

6.2.2 Benchmark Results

Figure 6: Graph showing the mean square error as a function of the time for the building problem.
Figure 7: Graph showing the mean square error as a function of the epoch for the building problem.

Figure 8: Graph showing the mean square error as a function of the time for the mushroom problem.
Figure 9: Graph showing the mean square error as a function of the epoch for the mushroom problem.

Figure 10: Graph showing the mean square error as a function of the time for the soybean problem.
6.3 Performance Benchmarks

To see how good performance the both implementations are in comparison to each other we need to do some benchmarks. The first benchmark, the connection benchmark, will measure execution time per connection on different sizes of networks. The second benchmark, the dataset benchmark, will measure the execution time per dataset over a different number of datasets and network sizes.

Both of them will train for 50 epochs on each data point so to get a more even result.

For the connection benchmark the different network configurations will be a 4 layered network and an 8 layered network. Each layer will begin with one neuron each and in the end have 1024 neurons each. Not every network size between these will be tested but there will rather be a step size between them. With a step size of 1 all neuron sizes are in the benchmark but with a step size of 2 then every other neuron size is investigated. That means that step size is the number of neurons added to each layer between each round of test.

Figure 11: Graph showing the mean square error as a function of the epoch for the soybean problem.
Figure 12: Step sizes for neurons depending for each number of layers. x/y means that the x value is for the FANN library and the y value is for the MANN library.

For the dataset benchmark the network configurations will be all 4 layered networks. Each layer will have 1, 32 and 128 neurons. The dataset sizes will go from 1 to 10000. To save time the network with 128 neurons in each layer only goes to 5000.

Not every dataset size is tested to save time. Just as with the connection benchmark there is a step size but this step size is for the dataset. All configurations have the step size of 10 up to 1000 and for the rest it is 100.

After a short amount of testing I noticed that the FANN benchmark results were very noisy while my implementation was not. Because of this all benchmarks for the FANN library is from running it 100 times and then taking the median execution time.

### 6.3.1 Connection Benchmark Results

![Connections graph](image)

Figure 13: Graph showing the execution time divided by connections as a function of the number of neurons in each layer.
Figure 14: Graph showing the execution time in my implementation per connection as a function of neurons per layer.

Figure 15: Graph showing the execution time in the FANN divided by connections as a function of the neurons in each layer.
6.3.2 Dataset Benchmark Results

![Graph showing the execution time per dataset as a function of number of datasets when the network is a 4 layered network with 128 neurons in each layer.](image1)

Figure 16: Graph showing the execution time per dataset as a function of number of datasets when the network is a 4 layered network with 128 neurons in each layer.

![Graph showing the execution time per dataset as a function of number of datasets when the network is a 4 layered network with 32 neurons in each layer.](image2)

Figure 17: Graph showing the execution time per dataset as a function of number of datasets when the network is a 4 layered network with 32 neurons in each layer.
Figure 18: Graph showing the execution time per dataset as a function of number of datasets when the network is a 4 layered network with 128 neurons in each layer.

7 Discussion

The benchmarks seem to point towards my implementation being good when using a lot of datasets, see Figure 16 to 18. This is probably because of dispatching one threadgroup per dataset will not increase overhead while being able to do more things at the same time in the graphics card.

Looking at Figure 13 it seems like what takes time is things that happens per connection and not per neuron when dealing with one dataset. We can also see that when using only one dataset we barely can get any kind of performance advantage from my implementation. The reason for this is probably because of overhead when dealing with the GPU.

In Figure 9 and 7 we see that the implementations did produce similar results as expected, see Section 6.2. In Figure 11 we can see a greater difference and it seems like the implementations come at rest at different minimums. The difference showed is probably not a problem but one could remove the doubt by doing some more tests with different starting weights.

To show further that my implementation works similarly to the FANN library we could also do additional checks by testing the same network with a couple of different starting weights to see if the average MSE is close.

One of the positive things about using the GPU while training a network
is that one can often still use the workstation for regular use with minimal change in training time.

I’m sure that my implementation can be improved to perform even better and to have support for more than 1024 neurons in a layer but at the moment it does seem to be able to do the job.

While running the dataset benchmarks my implementation was able to bluescreen my computer and crash the graphics card. This happened only when testing the network with 128 neurons in each network and only when changing large datasets, i.e. changing the input and output buffer over and over again. The first crash was when it was handling sets of 3000 and after that I could only do two tests at a time if I did not want it to shut down. Why this happened I’m not quite sure of but it might have something to do with the GPU having almost a 99% load while deleting big buffers and creating new ones very close to each other.

8 Conclusion

This thesis shows that by using compute shader we can cut down on training time for artificial neural networks when using big networks and many datasets. A good example of a problem that requires this is handwriting recognition.

9 Future Work

Since this was examining the performance of a single thread neural network library it would be interesting to see how well a multi-threaded neural network on the CPU would do compared to one on Compute Shader. Another interesting thing would be to see how well OpenCL or CUDA does and if there is anything in these that can improve performance even more.

My implementation could also be improved, both in regards to performance and in regards to functionality. Adding support for different activation functions, a neuron value range of -1 to 1 and many more things. Seeing if there is a way to implement a good reinforcement learning technique would also be interesting.

References

A Appendix

A.1 NNCommon.hlsl — Used by all shaders

```cpp
void sync()
{
    GroupMemoryBarrierWithGroupSync();
}

#define clipVal(x, lo, hi) (((x) < (lo)) ? (lo) :
                        (((x) > (hi)) ? (hi) : (x)))

inline float activation(float sum, float steepness)
{
    return 1.0f / (1.0f + exp(-2.0f * steepness * sum));
}

inline float activationDerived(float val, float steepness)
{
    float t = clipVal(val, 0.1, 0.99);
    return 2 * steepness * t * (1 - t);
}
```

A.2 NNCommonLR.hlsl — Used by Test and Run shaders

```cpp
struct Layer
{
    uint size;
    uint fNeuron;
    uint fConnection;
};
```
RWStructuredBuffer<
float>
weight : register(u0);
groupshared float neurons [NEURONS];

inline float getVal(uint pos, uint lSize)
{
  float val = 0;
  for (uint n = 0; n < lSize; ++n, ++pos)
  {
    val += weight[pos] ∗ neurons[n];
  }
  val += weight[pos];
  return val;
}

A.3 NNRun.hlsl — The Run Shader code

#include "NNCommon.hlsl"
#include "NNCommonTR.hlsl"

cbuffer NeuronsData : register(b0)
{
  uint totNeurons, // replace with tot neurons
  uint totConnections, // replace with tot connection
  float steepness;
};

RWStructuredBuffer<
float>
output : register(u1);

StructuredBuffer<Layer>
layers : register(t10);
StructuredBuffer<
float>
input : register(t11);

[numthreads( NEURONS, 1, 1 )]
void main(uint tID : SV_GroupIndex, uint3 gID : SV_GroupID)
{
  uint lSize = layers[0].size;
  const uint NEURONS_OFFSET = gID.x ∗ totNeurons;
  const uint CONNECTION_OFFSET = gID.x ∗ totConnections;
  const uint INPUT_OFFSET = gID.x ∗ lSize;

  uint pos = tID ∗ (lSize + 1);
  if (tID < lSize)
  {
    neurons[tID] = input[INPUT_OFFSET + tID];
  }
  float val = 0;
  [unroll]
  for (int l = 1; l < LAYERS; ++l)
  {
    sync();
    lSize = layers[l - 1].size;
    pos = tID ∗ (lSize + 1) + layers[l].fConnection;
    val = getVal(pos, lSize);
    sync();
  }
neurons[tID] = activation(val, steepness);
}

lSize = layers[LAYERS - 1].size;
if (tID < lSize)
{
    output[lSize * gID.x + tID] = neurons[tID];
}

A.4 NNTest.hlsl — The Test Shader code

#include "NNCommon.hlsl"
#include "NNCommonTR.hlsl"

cbuffer NeuronsData : register(b0)
{
    uint totNeurons;
    uint totConnections;
    float steepness;
};

RWStructuredBuffer<float> slope : register(u1);
RWStructuredBuffer<float> value : register(u2);
RWStructuredBuffer<float> mseBuffer : register(u4);

StructuredBuffer<Layer> layers : register(t10);
StructuredBuffer<float> input : register(t11);
StructuredBuffer<float> output : register(t12);

groupshared float MSE;
groupshared uint mMSE;

void computeMSE(uint tID, uint outOffset)
{
    uint oNeurons = layers[LAYERS - 1].size;
    float diff = 0;
    float mseToAdd = 0;
    if (tID < oNeurons)
    {
        diff = output[outOffset + tID] - neurons[tID];
        mseToAdd = diff * diff;
        if (diff < -0.999999f)
            diff = -17.0f;
        else if (diff > 0.999999f)
            diff = 17.0f;
        else
            diff = log((1 + diff)/(1 - diff));
        diff = diff * activationDerived(neurons[tID], steepness);
        InterlockedAdd(mmse, 1);
    }

    neurons[tID] = mseToAdd;
    sync();
    if (tID == 0)
    {

float mseAdded = 0;
for (uint n = 0; n < oNeurons; ++n)
    mseAdded += neurons[n];
MSE = mseAdded / nrMSE;
}

sync();
neurons[tID] = diff;
}

void changeSlope(uint pos, float error, uint valuePos, uint limit)
{
    float change = 0;
    for (uint temp = 0, n = valuePos; temp < limit; ++temp, ++n, ++pos)
    {
        slope[pos] += error * value[n];
    }
    slope[pos] += error; // Bias
}

float changeError(uint pos, uint stepsize, uint vPos, uint limit)
{
    float error = 0;
    for (uint n = 0; n < limit; ++n, pos += stepsize)
    {
        error += weight[pos] * neurons[n]; // Neurons is now error
    }
    return error * activationDerived(value[vPos], steepness);
}

[numthreads( NEURONS, 1, 1 )]
void main(uint tID : SV_GroupIndex, uint3 gID : SV_GroupID)
{
    const uint NEURONS_OFFSET = gID.x * totNeurons;
    const uint CONNECTION_OFFSET = gID.x * totConnections;
    const uint INPUT_OFFSET = gID.x * layers[0].size;
    const uint OUTPUT_OFFSET = gID.x * layers[LAYERS - 1].size;

    if (tID == 0)
        nrMSE = 0;

    uint lSize = layers[0].size;
    uint pos = tID * (lSize + 1);
    if (tID < lSize)
    {
        neurons[tID] = input[INPUT_OFFSET + tID];
        value[tID + NEURONS_OFFSET] = neurons[tID];
    }
    float val = 0;
lSize = layers[0].size;
[unroll]
for (int l = 1; l < LAYERS; ++l)
{
    sync();
    pos = tID * (lSize + 1) + layers[1].fConnection;
val = getVal(pos, lSize);
sync();
neurons[tID] = activation(val, steepness);

lSize = layers[1].size;
if (tID < lSize)
    value[tID + layers[1].fNeuron + NEURONS_OFFSET] = neurons[tID];
sync();
computeMSE(tID, OUTPUT_OFFSET);
if (tID == 0)
    mseBuffer[glD.x] = MSE;

// Back propagate and calc slopes
float error = neurons[tID];
uint nCurrOffset;

[unroll]
for (int l = LAYERS − 1; l > 0; −−l)
{
    lSize = layers[l − 1].size;
    nCurrOffset = NEURONS_OFFSET + layers[l − 1].fNeuron;
    sync();
    if (tID < layers[l].size)
        changeSlope(CONNECTION_OFFSET + layers[l].fConnection + tID
            * (lSize + 1), error, nCurrOffset, lSize);
    error = changeError(layers[l].fConnection + tID, lSize + 1,
        nCurrOffset + tID, layers[l].size);
    sync();
    neurons[tID] = error;
}

A.5  NNLearn.hlsl — The Learn Shader code

#include "NNCommon.hlsl"

// Needs SET_SIZE and THREADS macros to be defined
// THREADS defines the amount of THREADS per group
// SET_SIZE defines the size of the sets

cbuffer LearnData : register(b1)
{
    uint nrSets;
    uint fillerAt; // The thread where it should stop as its a filler
    uint fillerGroup; // The Thrdgroup with filler threads in it
    float goalMse;
};

// Remember offsets for device shared memory
RWSStructuredBuffer<float> weight : register(u0);
RWSStructuredBuffer<float> slopes : register(u1);
RWSStructuredBuffer<float2> slopeStep : register(u3);
RWStructuredBuffer<float> mseBuffer : register(u4);
AppendStructuredBuffer<float> mse : register(u5);

static const float increaseFactor = 1.2;
static const float decreaseFactor = 0.5;
static const float deltaMin = 0;
static const float deltaMax = 50;

[numthreads( THREADS, 1, 1 )]
void main( uint tID : SV_GroupIndex, uint3 gID : SV_GroupID )
{
  if (gID.x == fillerGroup && fillerAt <= tID)
    return;

  // add upp the total mse and append it
  float newMse = 0;
  for ( uint n = 0; n < nrSets; ++n )
  {
    newMse += mseBuffer[n];
  }
  newMse = newMse / nrSets;
  if (tID == 0 && gID.x == 0)
    mse.Append(newMse);
  uint pos = tID + gID.x * THREADS;

  float sstep = max(slopeStep[pos].y, 0.001); // So that it doesn’t stop
  float slope = 0;
  for ( uint i = 0; i < nrSets; ++i )
  {
    slope += slopes[pos + i * SET_SIZE];
    slopes[pos + i * SET_SIZE] = 0;
  }
  if (newMse <= goalMse)
    return;

  float same = slope * slopeStep[pos].x;
  if (same >= 0)
    sstep = min(sstep * increaseFactor, deltaMax);
  else
  {
    sstep = max(sstep * decreaseFactor, deltaMin);
    slope = 0;
  }
  if (slope < 0)
  {
    weight[pos] -= sstep;
    if (weight[pos] < -1500)
      weight[pos] = -1500;
  }
  else
  {
    weight[pos] += sstep;
    if (weight[pos] > 1500)
      weight[pos] = 1500;
  }
slopeStep[pos] = float2(slope, sstep);