SpikeNN: A Quick Manual

Sina Tavakoli

1 Installation

Below comes an overview of files and folders structure of SpikeNN library.

```
SpikeNN
├── document
│   └── manual.pdf ..... a quick pdf manual
├── example ..... some examples of using SpikeNN
├── include ..... header files directory
├── src ..... source files directory
└── CMakeLists.txt ..... cmake file to build the library
```

Like any other library, to use spikeNN, its header files and library files should be linked to the project which is going to use it. To do this, the first step would be to build the binary file of SpikeNN. In orther to build the library, **boost::filesystem** and **boost::serialization** should be installed first. SpikeNN depends on these two libraries, so they should be linked to SpikeNN. To make the library to be cross-platform, we used CMake package which generates a project depending on what compiler and IDE a system has. Just creat an empty directory in the root directory of SpikeNN and from there, execute the following command in terminal (command prompt in windows).

```
1 cmake ..
```

This command will creat a project in the build directory according to the compiler and installed IDE. For example if one is using windows and Microsoft Visual Studio, cmake will make a visual studio solution file and links it to appropriate paths automatically (make file if using linux). Building this solution will result in creating the binary file of SpikeNN.
Next step to use SpikeNN is to link it to your own projects. For those who are not familiar with linking a library to a project, see `http://en.wikibooks.org/wiki/C%2B%2B_Programming/Compiler/Linker/Libraries` or `http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html`

2 Overview

As an overall look to the code,the important classes are introduced in this section and some fundamental task which is assigned to them are explained.

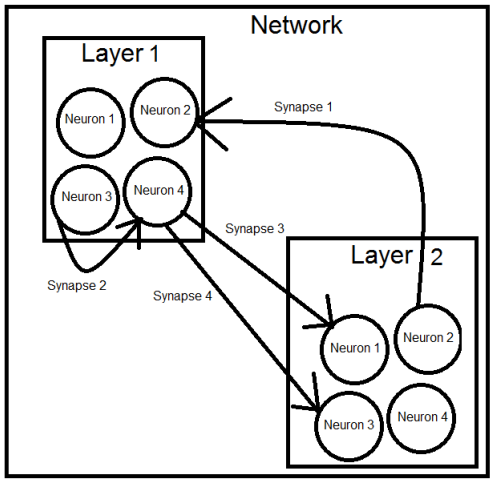

Figure 1: Overall look of a network.

2.1 Neuron

Neuron's behavior simulation is assigned to this class. According to a neuron model which is defined for this class, any object of this class will be called to update its components every time-step past. When model dictates a spike it will notify its post and pre-synapses of that spike through two lists of pointer-to-Synapse to access the pre-synapses and post-synapses.
"Neuron" is an abstract class, Which means that before using this class some functions should be overridden. These functions are "updatePotentional" which changes the potential every model's millisecond and "reset" which brings the neuron to its resting state. By redefining these two function one can put any arbitrary model of neuron to work in the constructed network. As an example, class "IzhikevichNeuron" is implemented to be the default completed neuron model which mimics the neuron's behavior with Izhikevich model (Izhikevich, 2003).

2.2 Synapse

This class is implemented for simulating the connections between neurons. So it has its own weight and delay for passing a spike. Another task of this class is to handling the operations of synaptic-time-dependent-plasticity (STDP) so it has access to its pre and post neurons using two pointers. It changes its own weight according to their spikes.

2.3 Layer

This class contains a list of neurons and synapses which their pre neurons are in this Layer so that it can call them when the model needs them to be updated. Beside that, it handles the input spikes from sources other than the network itself or currents or logs if user defines them. Another task of this class is to add neurons or make connections between them according to the users request. These task will be explained in detail in the next section.

2.4 DAHandler

For simulating the network's reinforcement using dopamine signaling (Izhikevich, 2007), this class is designed to handle the reward and punishment to the network. This class's objects will be called from "Layer" to check for the reward or punishment situations.

2.5 Network

This class is mainly a container which holds a list of layers and mainly passes the users order to the layers. The only exclusive task of this class is to hold time and calls layers for update every time-step.

3 Designing A Network

In this section a quick step-by-step guide to design an arbitrary network is presented. As an example it will construct the Network designed by Izhikevich, 2007 in his paper "Solving the Distal Reward Problem through Linkage of STDP and Dopamine Signaling".

3.1 Creating A Network Object

Creating a network a is simply done by creating a Network object with an arbitrary input which defines time-step. All the initializations and tunings will be done with the defined Network methods. So for example this line will be enough.

```
Network net(0.2);
```

3.2 Creating Layers

The below method creates an empty layer in a Network and return its ID for the later uses.

```
int Network::addLayer(bool shouldLearn=true,bool isContainer=false);
```

where variable "shouldLearn" determines that should the synapse weights in this layer be evolved in STDP or not, and "isContainer" sets if the layer is a

container or not (a container layer wont update its neurons over time).
For our example we only need one layer with default flags. So the below code will do the work.

```
int l = net.addLayer();
```

3.3 Creating Neurons

To add neurons to the layers one should specify the completed class of a neuron so that the network can generate new neurons from that specified kind. The following prototype shows how creating neurons can be done.

```
template <class NeuronTemp>
bool Network::addNeuron(int layerIndex,int neuronNum,
        ChannelType type=EXCITATORY, ParameterContainer* params=0);
```

where variable "layerIndex" is the ID of the layer that new neurons should be added to, "neuronNum" is the number of neurons that should be added, "type" is the channel type of those "to be added" neurons which is from the type of a pre defined enum "ChannelType" that can take either EXCITATORY or INHIBITORY. And at last "params" is the parameters that should be passed for initializing the neurons. This Variable is a "ParameterContainer" object which is a simple container of some float variables as a list.
For the example, as we need 800 excitatory and 200 inhibitory with discussed parameters in the Izhikevich's paper, the following code will construct the desired settings.

```
net.addNeuron<IzhikevichNeuron>(l,800,EXCITATORY,
    new IzhikevichParameters(0.02f, 0.2f, -65,8));
net.addNeuron<IzhikevichNeuron>(l,200,INHIBITORY,
    new IzhikevichParameters(0.1f, 0.2f, -65,2));
```

3.4 Setting Parameters

Setting models parameter devides to following two groups.

3.4.1 STDP Parameters

The following prototype handles the setting for STDP parameters.

```
void Netwok::setSTDPParameters(int layerIndex,float CMultiplier,
          float AP,float AN, float decayMultiplier = 1.0f,
          int STDPTimeStep = 100, float TaoP = 20,float TaoN = 20)
```

where "layerIndex" indicates to which layer these settings should be applied (if not passed, it will be applied to all layers). Weights and weight derivation will be multiplied by "decayMultiplier" and "CMultiplier" respectively every "STDPTimeStep" milliseconds. "AP" and "AN" are intercept of positive part and negative part of the STDP function respectively and finally "TaoP" and

"TaoN" are the time denominators of the STDP function.

3.4.2 Bounding Parameters

There are some boundaries over synapse weights and delays or input currents which model looks up to them. These variables can be set using the following method.

```
void Network::setBoundingParameters(int layerIndex,
                float exMaxWeight, float inMaxWeight,
                float exMinRandWeight, float exMaxRandWeight,
                float inMinRandWeight, float inMaxRandWeight,
                int minRandDelay, int maxRandDelay)
```

where "layerIndex" is the layer which these settings should be applied to and if it is not passed, it will be applied to all layers. "exMaxWeight" and "inMaxWeight" are the maximum weights that respectively an excitatory or inhibitory synapse can get through STDP, "exMinRandWeight", "inMinRandWeight", "exMaxRandWeight" and "inMaxRandWeight' specify a range in which initial random weights should be into (excitatory and inhibitory weights are separated). "minRandDelay" and "maxRandDelay" is for initial random delay generating of synapses.
For the example, this command will set the bounding parameters.

3.4.3 Current Parameters

Some variables control the random input current generating. The below method can be used to set these variables.

```
void Network::setCurrentParameters(int layerIndex,
      float MinInputCurrent,float MaxInputCurrent)
```

The example can be set using the following code.

```
net.setSTDPParameters(1, 0.99f, 0.1f, 0.15f, 1.0f, 100);
net.setCurrentParameters(1, -6.5, 6.5);
net.setBoundingParameters(1, 4, 4, 0, 4, 0, 4, 0, 20);
```

3.5 Making Connections

Connecting neurons can be done by a method called "makeConnection" which is overloaded to the followings, for different kinds of connecting patterns.

3.5.1 Connecting Two Neurons

The first overloaded method is the following prototype.

```
void Network::makeConnection(int sourceLayerIndex,
      int sourceNeuronIndex, int destLayerIndex,
      int destNeuronIndex, float weight = -1, int delay = -1);
```

this method simply connects the neuron with ID “sourceNeuronIndex” from the layer with ID “sourceLayerIndex” to the neuron with ID “destNeuronIndex” from the layer with ID “destLayerIndex” and puts “weight” and “delay” as its initial weight and delay respectively. If these variable was set as -1, the network will choose a random value according to the set bounding parameters.

3.5.2 Connecting With Probability

The next overloaded method is the following.

```
void Network::makeConnection(int sourceLayerIndex,
      int destLayerIndex,float synapseProb,
      float excitatoryWeight=-1.0f,float inhibitoryWeight=-1.0f,
      int excitatoryDelay=-1.0f,int inhibitoryDelay=-1.0f);
```

This methods connects every neuron in layer with ID “sourceLayerIndex” to every neuron with ID “destLayerIndex” with probability “synapseProb”. And sets the inhibitory (excitatory) connection’s weight and delay as “inhibitoryWeight” and “inhibitoryDelay” (“excitatoryWeight” and “excitatoryDelay”). Again if these parameters passed -1, they will be choosen randomly.

3.5.3 Connecting Each Neuron to a Fixed Number of Neurons

Another overloaded method for making connections is the below prototype.

```
void Network::makeConnection(int sourceLayerIndex,int destLayerIndex,
     int neuronsNumToConnect,
     float excitatoryWeight = -1.0f, float inhibitoryWeight = -1.0f,
     int excitatoryDelay = -1.0f, int inhibitoryDelay = -1.0f);
```

This method connects every neuron from layer with ID “souceLayerIndex” to a fixed number of neurons (“neuronsNumToConnect”) from layer with ID “destLayerIndex” randomly, and puts the weights and delays with its following passed parameters like the previous overloaded function.

3.5.4 Connecting Using a Manual Pattern

The next overloaded function is designed to let user connect neurons manually with a passed pointer-to-function. Its prototype is following.

```
void Network::makeConnection(int sourceLayerIndex,int destLayerIndex,
            ConnectionInfo (*pattern)(int,int)=0);
```

where connects any neuron from layer ID “sourceLayerIndex” to any neuron from layerID “destLayerIndex” that the function which “pattern” is pointing to, decides to connect. This function gets two index variable as input which the first one is the source neuron ID from the source layer and the second one is the destination neuron ID from the destination layer and should return a “ConnectionInfo” type which containes the connection informations. This types constructor is as follows.

```
ConnectionInfo::ConnectionInfo(bool connectFlag,
          ChannelType type=EXCITATORY, float weight=-1, int delay=-1);
```

where "connectFlag" is a boolian variable which indicates that the connection should be made or not, "type" is the channel type of the connection if it should be made and "weight" and "delay" are its weight and delay.
When this form of "makeConnection" is called, the network will pass any two neurons from source and destination layer to the function and connects them according to the returned information.
For our example, since we should connect every two neurons with probability of 0.1, we can either use the below code or make it manually.

```
net.makeConnection(l,l,0.1,1.0f, 1.0f, CHOOSE_RANDOM, 1);
```

Note that CHOOSE_RANDOM is a variable globally defined as -1.
To make this connections manually we have to first write a pattern function which results the desired connections.

```
ConnectionInfo connectingPattern(int sourceID, int destID)
{
   if ((float)rand()/RAND_MAX < 0.1)
      return ConnectionInfo(true, DEPENDENT, 1.0f, 1);
   else
      return ConnectionInfo(false);
}
```

Note that with passing "DEPENDENT" as the channel type of the synapse, network will set the channel type of the neuron as the channel type of the pre-neuron of that synapse.
After defining the connectingPattern we should pass it to the network with the following code.

```
net.makeConnection(l,l,&connectingPattern);
```

3.6 Controling Input Spikes or Currents

By default, a layer dosn't have any input patterns, so user should define an input patter for layers that should have inputs from sources outside of the network. This adjusment can be done using the following method.

```
void Network::setInputPattern(int layerIndex, InputPatternMode mode,
      std::vector<InputInformation> (*pattern)(int) = 0);
```

This method sets an input pattern for layer with "layerIndex" ID. This pattern is determined with "mode" variable which is from the enum type "InputPatternMode". This type can get the followings.

3.6.1 ALL_RANDOM_CURRENT

If "mode" sets to be this value, then in each millisecond every neuron in the layer ID "layerIndex" will get a random current in the range that is specified in

current parameters.

3.6.2 ONE_RANDOM_CURRENT

This value randomly, gives one of the neurons a random current.

3.6.3 ALL_MAX_CURRENT

This value gives all of the neurons in the layer a max current which is set in current parameters.

3.6.4 ONE_MAX_CURRENT

This value randomly, gives one of the neurons a max current.

3.6.5 MANUAL_INPUT

If "mode" is set to be this value, the input currents or spikes handling will be given to user through the a pointer-to-function which is passed as "pattern". This function which "pattern" points to, should get an int value as the network time according to millisecond and return a list of informations about which neuron should get an input and how this input should be. The "InpuInformation" structure has the following constructor prototype.

```
InputInformation::InputInformation(int neuronIndex,
        CurrentMode inputMode, float manualCurrent=0);
```

Where "neuronIndex" is the ID of the neuron that this information is belong to and "inputMode" is from the enum type "CurrentMode" which can get one of the following values.

- **FORCE_FIRE:** This value forces the neuron to pass a spike.
- **RANDOM_CURRENT:** This value gives the neuron a randomly generated current in the range specified in current parameters.
- **MAX_CURRENT:** This value gives the neuron a maximum current set in current parameters.
- **MANUAL_CURRENT:** This value gives the neuron a current that is manually set by the variable "manualCurrent".

To advance our example, it is required that each neuron should get a random current between -6.5 to 6.5. Since we set the right parameters for minimum and maximum current in previous steps, we can use the following to implement this input behavior.

```
net.setInputPattern(I,ALL_RANDOM_CURRENT);
```

But if we want to do it manually, first we need to construct the input pattern function.

```
std::vector<InputInformation> inputPattern(int time)
{
    std::vector<InputInformation> re;
    for (int i=0;i<100;++i)
    {
        re.push_back(InputInformation(i,MANUAL_CURRENT,
              ((float)rand()/RAND_MAX-0.5)*13));
    }

    return re;
}
```

And then pass it to the network.

```
net.setInputPattern(1, MANUAL_INPUT, &inputPattern);
```

3.7 Add DAHandler

This class is implemented to control the reinforcement learning process of the network. It is an abstract class which should be completed according to needs of the user. Any object of this class which is assigned to a layer will be notified of every spike accured in that layer. If a layer is assigned to have a DAHandler, the synapses belonging to that layer will emerge in STDP according to the dopamine signals provided by the DAHandler object.
For our example, a complete DAHandler class has been implemented by default under the name of "IzhikevichDAHandler". One can use this class to handle the rewards of that example. Otherwise, according to the needs of user a new DAHandler should be implemented. So for the example that we have been following, the below commands will add the DA module to our network.

```
IzhikevichDAHandler* dah = new IzhikevichDAHandler();
dah->setSynapse(net.getSynapse(0));
net.addDAModule(1, dah);
```

3.8 Creating Log

To demonstrate the results of running a network, users can order to log some actions or key variables in a text file (".log" extention). These log files can be found in the root directory of the user's application. There are different kinds of things that can be logged which are explained in the foloowing sections.

3.8.1 Logging Activity

Any desired layer's spikes can be recorded using the following function.

```
void Network::logLayerActivity(int layer);
```

which records the spikes of the neurons belonging to the layer with index "layer", in some text files. Each text file represents the spikes during a minute of the

network, and every line of the file corresponds to a recorded spike which consists of two numbers seperated by space. The first number is the time of the spike (in terms of milliseconds) and the second number is ID of the neuron which fired that spike.

3.8.2 Logging a Neuron's Potential

Any neuron's potential can be recorded in a log file using the following method.

```
void Network::logPotential(int layer, bool (*pattern)(int) = 0);
```

where "layer" is the ID of the layer the desired neuron belongs to, and "pattern" is a pointer to a function which gets the ID of a neuron and returns a boolian value by which the user decides whether to log that neuron's potential or not. All the recorded data will be in a log file which every line correspondes to a time's potential. In every line there are two numbers which the first one is the time of record and the second one is the potential on that time.

3.8.3 Logging a Synapse Weights

Any synapse's weight can be logged during model's time whith the following function.

```
void Network::logSynapseWeight(bool (*pattern)(int));
```

where pattern is a pointer to a function that decides if the weight of a synapse with the given ID, should be logged or not.

3.8.4 Logging Post/Pre Synapse Weights of a Neuron

Sometimes it is required to log all the pre or post synaptic weights of a neuron. Using the previous method this task may be hard sometimes. There is another method for making this situation easier by using the following methods

```
void Network::logPostSynapseWeights(int layer, int neuron);
void Network::logPreSynapseWeights(int layer, int neuron);
```

For our example, since we need the first synapse's weight to be logged, we have to cunstruct a function that responds to the desired ID.

```
bool firstSynapse(int ID)
{
    return ID == 0;
}
```

then we have to add this line to pass the function and make log as desired.

```
net.logSynapseWeight(&firstSynapse);
```

3.9 Running The Constructed Network

All setting are now done, so the last step would be order the network to run using the following method.

```
void Network::runNetwork(int maxTime);
```

where maxTime is the time according to millisecond which determines when network should finish its iterations.
For the example, as we need to run the model for 1 model's hour we need to write this code.

```
net.runNetwork(60*60*1000);
```

3.10 Save/Load Networks

In the last step, users should save their network's state if it is needed in the future. This can be done using the following methods.

```
void Network::saveNetwork(std::string path);
Network* Network::loadNetwork(std::string path);
```

where path is the location of the file which the Netwotk object should be saved into or be loaded from. For our example, if we want to save the result state of the network we constructed, we can add the following line.

```
net.saveNetwork("daspnet.sav")
```

where it will save the network in the file "daspnet.sav" which it is located in the root directory of the project. If in the future we had to load the saved file, we can use this command.

```
Network* loadedNet = Network::loadNetwork(daspnet . sav);
```

which creates a Network object according to the saved file and returns it's adress to be saved in loadedNet pointer.

4 Interfaces

As described before, the "Network" class can be used to construct any kinds of network, but for some specific networks it is often hard to handle the one-dimentional IDs which is given to any object in that network. In these situations it is recommanded to write a new class which behaves as an interface for "Network" with inheritation.
For example if we want to design a network for image processing works, we need layers to be two-dimentional just like a digital image is. For this matter class "VisualNetwork" is implemented to handle the networks related to image works.As an example we will construct a model to show how this class works. Consider the following code.

```
1  VisualNetwork visNet;
2  visNet.setBoundingParameters(3, 20, 0.5f, 2.5f, 10, 10, 1, 20);
3  visNet.setSTDPParameters(0.99f, 0.002f, 0.002f, 1.0f,
4            100, 20.0f, 30.0f);
5  int input = visNet.addLayer<IzhikevichNeuron>
6             (Point2D(100,100),INPUT_LAYER);
7  visNet.addReceptiveFieldSuperLayer<IzhikevichNeuron>
8            (input,4,Point2D(7,7),Point2D(3,3));
9  visNet.setInputImagesDirectory("inputPix");
10 visNet.logPreSynapseWeights(1,Point2D(10,10));
11 visNet.runNetwork(3600000);
```

In line 1, like before an object of this class is created.
In line 2 and 3, like before the parameters has been set.
In line 5, a layer is added to the network which contains 100x100 neurons (10000 neurons) the only difference here is that this layer is tow-dimentional and user can address to them using two coordination. This layer is also set as an input layer which means that the images which will be brought to network as inputs, enter the network through this layer.
In line 7, four new layers will be added to network which their neurons has a receptive field on the input layer, the settings of this receptive field connection are set so that each neuron from the new layer connects to a 7x7 neurons from input layer and these receptive fields will move 3 in first and second axis.
In line 9, a directory is set which all the input images are in that directory, network will read them and injects them as input one by one as time flows.
In line 10, neuron (10,10) from layer 1 is set to be logged.
And finally in line 11, the network is ordered to run for 1 model's hour.