HJ-Net: A Connectionist Modeling Tool based Back-Propagation Algorithm

Supplementary Materials

# Using Tool

## Architecture Setup

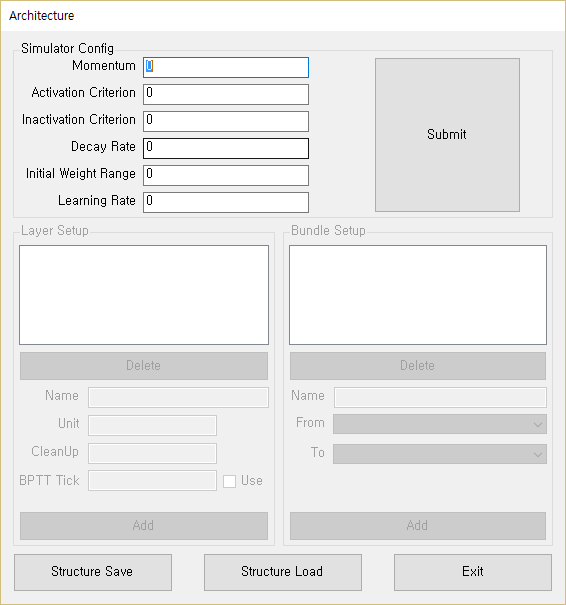


Figure 1. the architecture setup window.

Figure 1 is a window of architecture setup of program. In architecture setup, user set the initial variables and model structure. Initial variables decide the statuses of model. Therefore, when user set the architecture primarily, initial values should to be set before layer and connection setup.

### Initial values

Momentum is related the sigmoid function. In sigmoid function, momentum affect the slope of graph. The larger momentum makes steeper slope of function. Figure 2 shows the graph of the sigmoid function difference by momentum.

Figure 2. Sigmoid function difference by momentum

Activation criterion does not affect the model performance. However, this affect correctness data of result. Both of a unit’s target and activation are higher than this criterion, the unit is correct. On the other hand, unit’s target is higher but activation is lower than this criterion, the unit is incorrect.  
Inactivation criterion is similar to the activation criterion. Inactivation criterion does not also affect the model performance. However, this affect correctness data of result. Both of a unit’s target and activation are lower than this criterion, the unit is correct. On the other hand, unit’s target is lower but activation is higher than this criterion, the unit is incorrect.  
Decay rate is used at the weight change process in training. In weight change process, weights are moved to zero as shown in the decay rate. If a weight is positive number, the decay rate subtracts from weight. On the other hand, the weight is negative number, the decay rate adds from weight. And if the absolute number of weight is lower than decay rate, weight become zero.  
Initial weight range decide the initial status of connections’ weight. Every weight is assigned some randomize values between negative and positive initial weight range value. For example, when user set initial weight be 0.5, all weights have a value between -0.5 and +0.5.  
Learning rate means how many weights are changed at one training. If the learning rate is too big, model’s performance becomes fool. On the other hand, when the learning rate is too small, training speed becomes very slow. Users have to search a suitable learning rate.

One more important is that user can modify the initial variable after setup. When Users click the submit button, the initial variables are modified. All initial variable is modified, and all weights of model are re-initialized.

### Layer

Layer is a location which receive and send the activation. In layer section, user set how many and what layers make.  
Layer name does not affect any model performance in both of training and test. This is just index for user and program. Any name is OK, but it must not be overlapped to other layer names.  
Unit is the size of layer. Usual layer’s size is related the representation of model. On the other hand, the hidden layer’s size is related the computing power of model. Both of cases is user’s discretion. Only some constraints are that the maximum size of unit is 2,147,483,647 because programming variable, and too big unit size make problem about the memory.  
Cleanup is the size variable of a sub-layer which a layer have. Several connectionist modeling studies have reported the cleanup layer can contribute the performance of model (Reference). Moreover, some studies use the cleanup as a working memory (Reference). Although this can make the cleanup layer separately, this function is for user’s convenience. If user does not want to make the cleanup, the value can be set to 0.  
BPTT Tick is for the back-propagation through time(BPTT) algorithm. When user check the check-box of ‘USE’, layer become the layer for BPTT. BPTT Tick is the unfold size, so the flow of training and test is very different from general back-propagation(BP). Importantly, BPTT tick is start from 0 in this tool.

### Connection

A connection links two layers. In bundle section, user set what layer send activation to some layers. In this tool, connections are always one-way, not bi-direction.  
Like the layer name, bundle name is also just index for user and program. This does not affect any model performance, and it must not be overlapped to other bundle name.  
‘From’ and ‘To’ drop-box shows the layers which are made in layer section. The selected layer of “From” drop-box becomes the layer which activation sends. On the other hand, the selected layer of “To” drop-box becomes the layer which activation receives.  
One layer can send the activation to several layers, and can also receive the activation from several layers. However, if there is already same connection which both of ‘from’ and ‘to’ is same, it is impossible to make the connection.

### Save & Load

There are the functions of save and load. When user save the architecture, the save file storage the initial values, layer information, and connection information. Although the saved file is XML document, the saved file has “ACTXML” extension for interface convenience. If user want, user can open on text editors.  
When program load the ACTXML file, all current architecture information is swept. Although a caution message display, user have to be careful.

## Stimuli Pack Setup

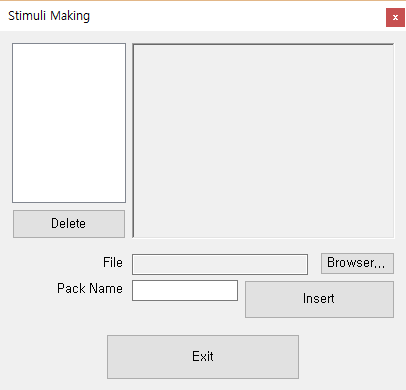


Figure 3. the stimuli pack setup window

In this tool, pattern information, which model use in training and test, is inserted from files.

### Stimuli File

Currently, this tool uses that the extension ‘txt’ as stimuli file. Although there can be multiple patterns in a file, this tool requires a strict form of file content to prevent a confusion of program.

### File content

Fundamentally, this tool demands several information at stimuli file: pattern names, stimulus names, stimulus probability, and patterns. Table 1 shows an example about file structure.

In first line, ‘In’ and ‘Out’ are the pattern names. Like other ‘name,’ these do not affect the performance, but it must not be overlapped. And, the ‘Name’, ‘Probability’, and ‘Time-Stamp’ are not used in this tool. These are just index.  
Other lines is the information about each stimulus. The first column is stimulus name. Stimulus name also affect to the performance, and this can be overlapped. However, after learning, when the tool output result data, these stimulus names use for result expression, so we recommend that the names also do not be overlapped.  
The second column is probability about the pattern learning. When model train the stimuli pack, a pattern is always not trained. Model will throw a dice, which cast from 0 to 1, and only if the cast value will be lower than this probability, model train the stimulus. Therefore, probability is higher, the model will train the stimulus more frequent. Of course, when probability is just 1 or 0, the stimulus is always or never trained, respectively.  
The third column is the time-stamp. When user does not use Simple Recurrent Network (SRN, Elman Network), this all values of this column is 0. This is just use in SRN algorithm. In SRN using, this time-stamp control the recurrent number of SRN because some user may want to change the recurrent number by stimulus.  
The forth and more columns are the patterns. Although all pattern values are 0 or 1 in example, it can be use all values which are between 0 and 1. We think that the patterns are used as input or output in training and test process, so the amount of number of a pattern (separated by space, not tap) has to be same to the unit size of the layer which user want to match. This tool does not check the amount correctness in this stage because model cannot inference what pattern and layer have to be matched.

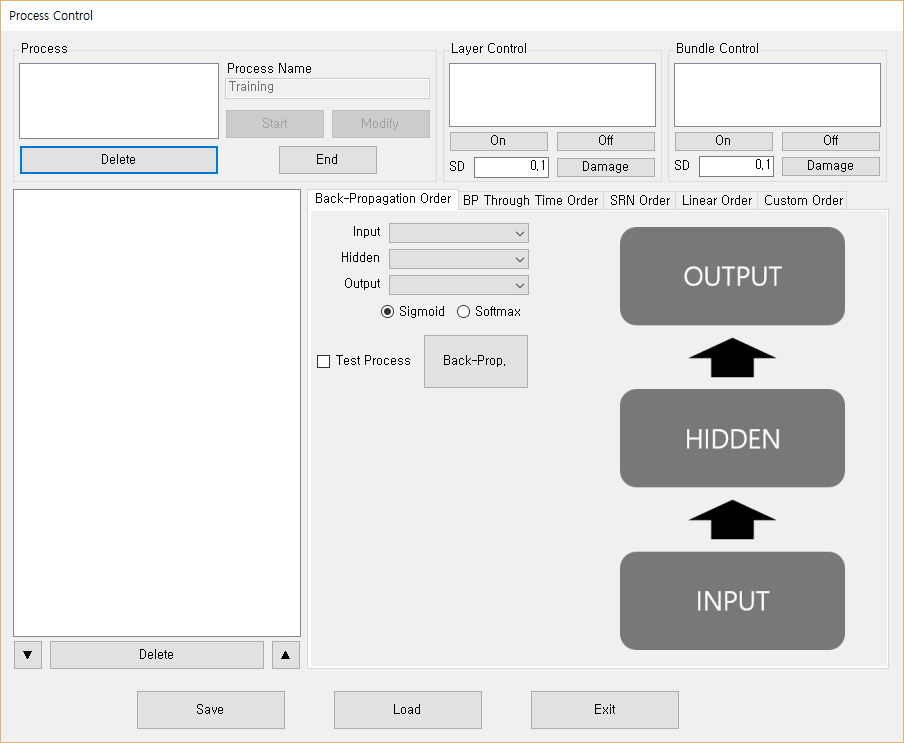
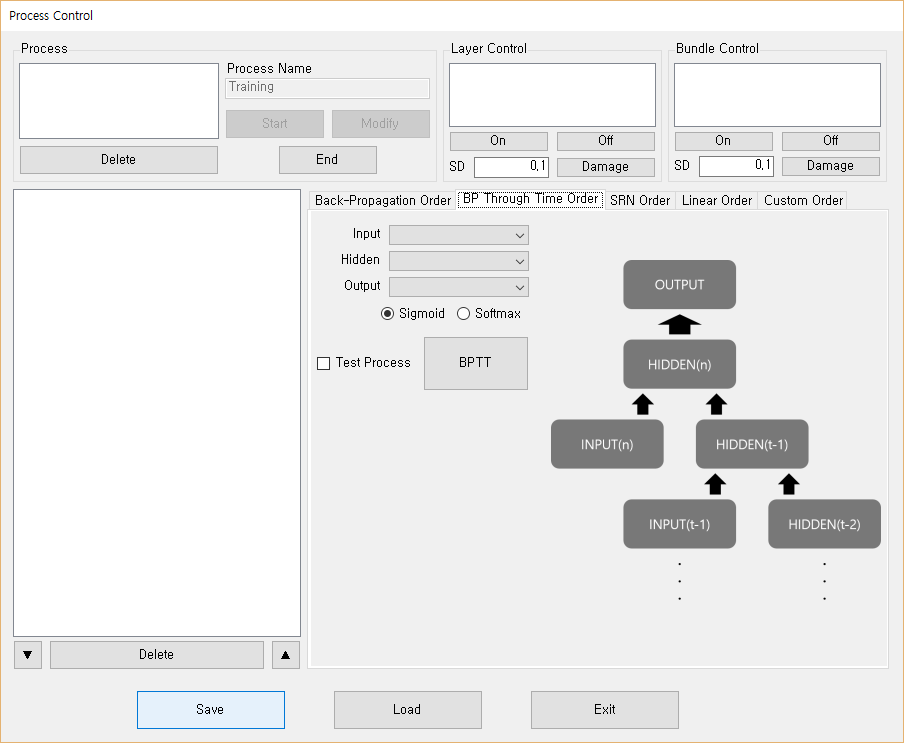
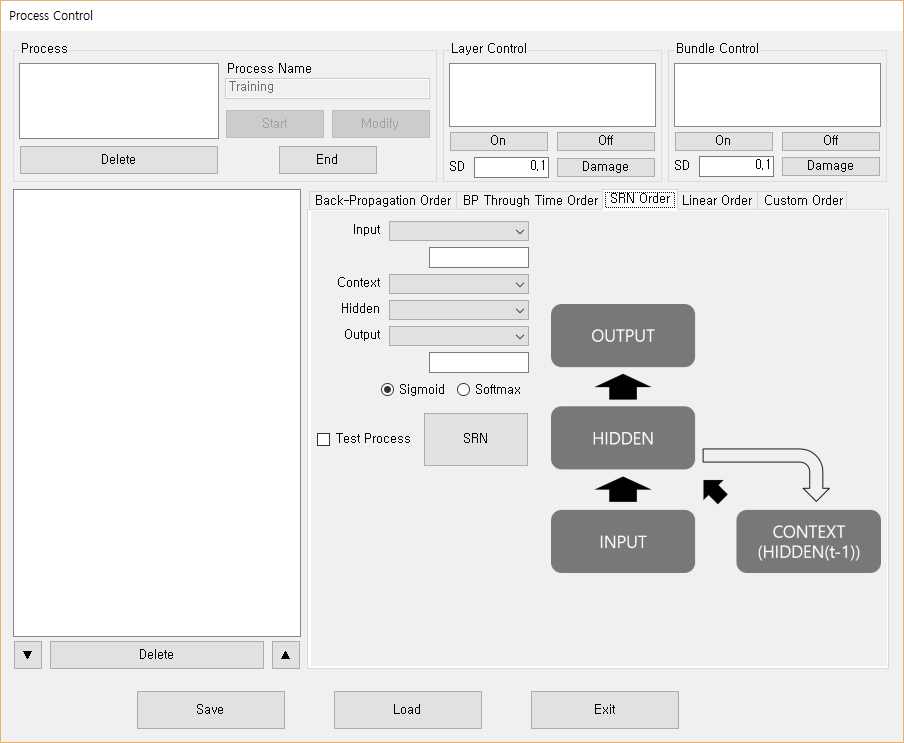
Table . an example of stimuli pack file

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pattern Name | Possibility | Time-Stamp | Output1 | Output2 | … |
| school | 0.765 | 10 | 1 1 0 1 0 1 1 … | 1 1 1 0 0 0 0 … | … |
| hierarchy | 0.043 | 21 | 0 0 0 1 1 0 0 … | 0 1 1 0 1 0 0 … | … |
| have | 0.914 | 33 | 1 0 0 0 0 0 0 … | 1 1 0 0 1 0 0 … | … |

### Pack Name

Pack name does not affect any model performance in both of training and test. This is just index for user and program. Any name is OK, but it must not be overlapped to other pack names.

## Process Setup

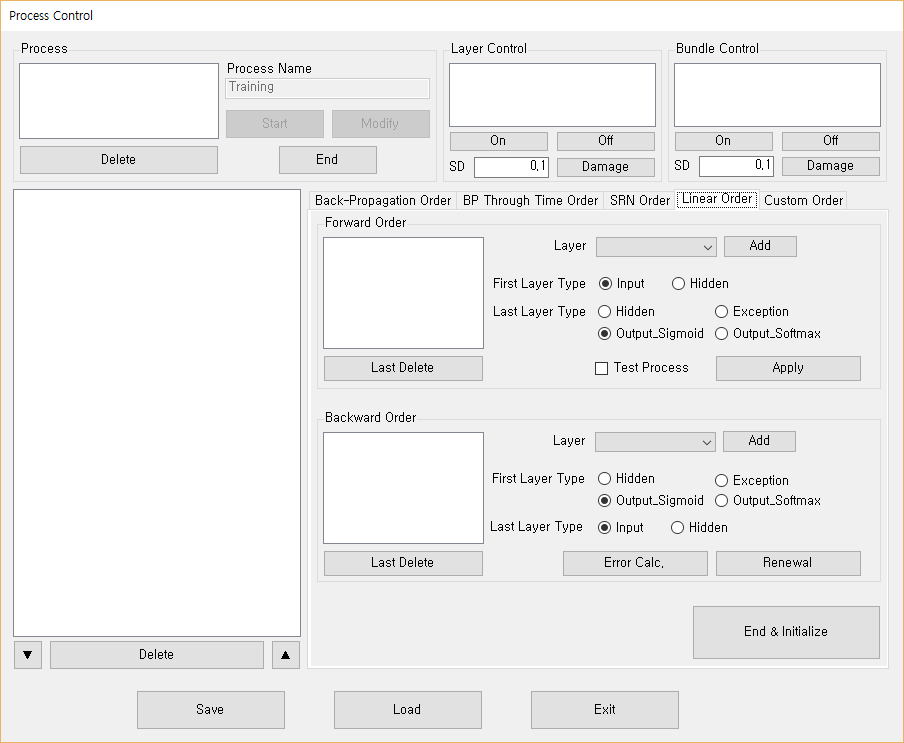
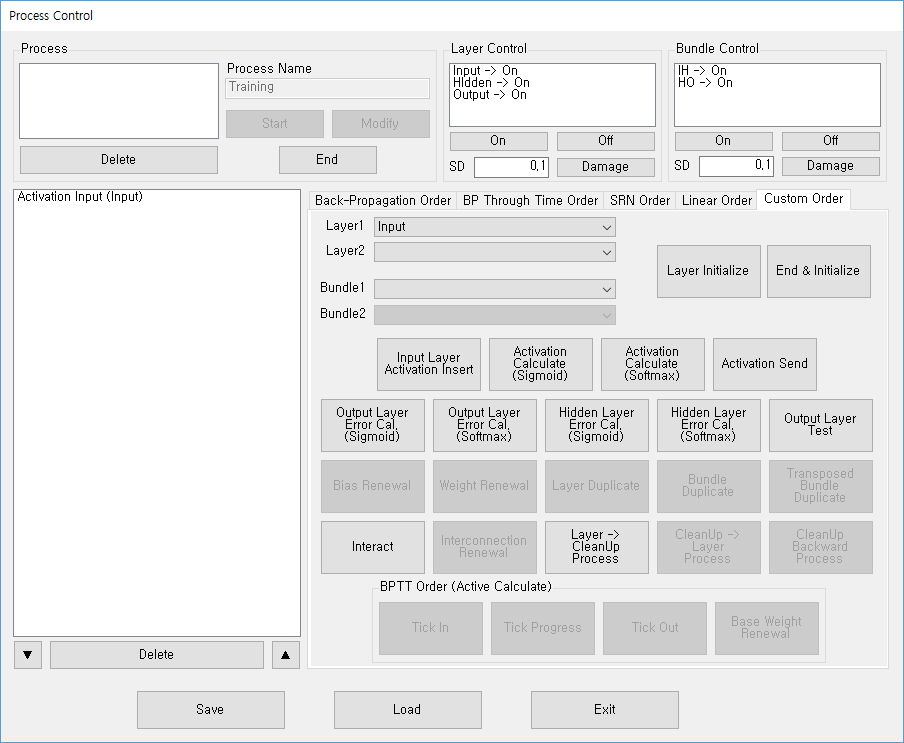
 

Figure 4. the process setup window

Process is a plan of the training and test flow. In process setup, this tool provides some usual algorithms and customizing.

### Layer & Bundle Control

This function set the layer and bundle’s status. Currently, there are three statuses: on, off, and damaged. The basic status is ‘on.’ If user change the status to ‘off,’ the layer or bundle’s value become 0 in the process. For layer, the unit activation of the layer become 0 in all calculations which use the layer activation like activation send. On the other hand, for bundle, all weights of the connection become 0.  
When layer or bundle status is the damage status, the Gaussian noise is added to the unit activation or weight. User set the standard deviation of Gaussian noise to control the degree of the lesion.  
Importantly, this does not mean the real weights are changed. The real value is saved. These status changes only affect the process.

### Back-Propagation Order

The back-propagation order menu is a simple method for using the BP algorithm. When user set three layers, this tool makes the flow for BP algorithm.  
To use this menu, model’s architecture has to be constructed like right figure of program (the upper left of Figure 4). There are two connections that one is from input to hidden and another is from hidden to output.  
Below the output layer drop-box, the sigmoid and softmax radio-buttons are related with the activation calculation of output layer. This is only output layer, not hidden. Hidden layer’s activation always uses the sigmoid function.  
The test process checkbox decides that this process is for training or test. A note of caution is that there is no error calculation and weight renewal in test process.

### BP Through Time Order

The BPTT order menu is similar to the BP Order menu. However, to use this menu, user has to set the hidden layer as BPTT layer in architecture setup. As mentioned in architecture setup, BPTT layer’s tick start from 0 in this tool.

### SRN Order

SRN Order menu requires an additional layer from BP Order menu: context. Because the context layer is the duplication of previous hidden layer’s activation, hidden and context layers’ unit sizes have to be same.  
Unfortunately, the custom order of this tool does not provide the dynamic loop (using different time-stamp) currently. Therefore, the order list show only two orders. If we modify or add the order loop function, the display will also be modified.  
The textboxes below input and output drop-box are the place to write the pattern name. Because SRN algorithm uses similar patterns repeatedly, user has to set the pattern name’s regularity. For example, if user wants to use 15 recurrent process and pattern names set ‘input\_1’, ‘input\_2’, …., ‘input\_15’, the ‘input\_’ is regularity.

### Linear Order

In linear order menu, user can set the linear forward and backward processes. This menu is similar to the back-propagation order menu, but this can be used in more complex architectures like multiple hidden layers or separated input layers.  
In forward order, each layer calculates and sends the activation. The layers, which inserted in the list-box, have to sequentially connected like back-propagation order menu.  
On the other hand, in backward order, the error of layers are calculated and weights and biases are renewed. Importantly, the layers, which inserted in the list-box, have to reversely connected.  
The radio-buttons set the types of first and last layer. The inserted orders are different by these types, except the renewal. Because the orders, which renewal function uses, are not changed by the type of layer, the type setting does not affect.

### Custom Order

If user want to make a detail flow, the process can be customized. Table 2 shows how each order operates in the process flow.  
A method for using this custom order menu is ‘modifying.’ Except the SRN, this function can be combined with other algorithm. For example, user is designing the original BP model which has two input layers, he can modify the flow of original BP algorithm flow after using the BP Order menu.  
Of course, user also can create all process himself.

Table . order operation in the process flow

|  |  |
| --- | --- |
| Order Name | Function |
| Input Layer Activation Insert | Insert input pattern to selected layer as activation. |
| Activation Calculate (Sigmoid) | Calculate the inserted storage values from other layer to current layer’s activation by sigmoid function. |
| Activation Calculate (Softmax) | Calculate the inserted storage values from other layer to current layer’s activation by softmax function. However, we do not recommend to use this order in hidden layers. The using to hidden layer maybe decrease the performance of model. |
| Activation Send | Insert the multiplication value of activation and weight to connected other layers as storage (storage is not activation). |
| Output Layer Error Calc. (Sigmoid) | Calculate the layer’s error by comparison between target pattern and layer’s activation. This order is valid when activation was calculated by sigmoid function. |
| Output Layer Error Calc. (Softmax) | Calculate the layer’s error by comparison between target pattern and layer’s activation. This order is valid when activation was calculated by softmax function. |
| Hidden Layer Error Calc. (Sigmoid) | Calculate the layer’s error by connected layer’s error and weight. This order is valid when activation was calculated by sigmoid function. |
| Hidden Layer Error Calc. (Softmax) | Calculate the layer’s error by connected layer’s error and weight. This order is valid when activation was calculated by softmax function. |
| Output Layer Test | Calculate and write several results like mean squared error or activation average. Because of some result calculation, this order need a target pattern. |
| Bias Renewal | Renew selected layer’s bias. The error calculation has to precede this order. |
| Weight Renewal | Renew selected connection. The error calculation has to precede this order. |
| Layer Duplicate | Copy the first layer’s all information to the second layer. Two layers have to be same unit and cleanup size. |
| Bundle Duplicate | Copy the first connection’s weight to the second connection. Two connections’ send and receive layers have to be same unit size. |
| Transposed Bundle Duplicate | Copy the first connection’s weight to the second connection. The unit sizes of send and receive layers of first connection have to be same the unit sizes of receive and send layers of second connection. In short, the unit sizes have to be reversed. |
| Interact | Insert the multiplication each unit activation and weight to same layer’s other units. Activation is not calculated. |
| Interconnection Renewal | Renew the weights which for interact. The error calculation has to precede this order. Self-connection is always 0. |
| Layer -> Cleanup Process | Insert the multiplication value of activation and weight to connected cleanup as storage. And then, calculate the activation of cleanup. |
| Cleanup -> Layer  Process | Insert the multiplication cleanup’s activation and weight to layer as storage. But, this order does not calculate the activation of layer. |
| Cleanup Backward Process | Renew two connections of cleanup. The layer error calculation has to precede this order. |
| Tick In | Initialize BPTT layer. And then, calculate the inserted storage values from other layer to layer- 0’s activation. |
| Tick Progress | Copy the storage values from other layer to next layer. Next, add the multiplication previous layer’s activation and weight to next layer’s storage. And then, calculate the storage values to next layer’s activation. |
| Tick Out | Finish BPTT progress and use the activation of current tick’s layer. |
| Base Weight Renewal | Renew BPTT layer’s internal weight |
| Layer Initialize | Initialize selected layer. All storage, activation, and error become 0. |
| End & Initialize | There is no function. This is a marker to express end. However, all processes have to assign this order at the end. |

### Save & Load

Like the architecture setup, there are the functions of save and load. When user save the process, the save file storage all processes which include layer and bundle status and order. The saved file is XML document, and the saved file has “PRCXML” extension. Although we do not recommend because order list of process is complex, user can open and edit on text editors.  
When program load the PRCXML file, user should not use the other model’s file. Although we make independent files about each setting, PRCXML files depend on the architecture of tool. Currently, this tool does not verify that loaded processes are suitable with current architecture setting. User has to be careful.  
Also, like the architecture setup, when program load the PRCXML file, all current process information is swept. Although a caution message display, user has to be careful.

## Learning

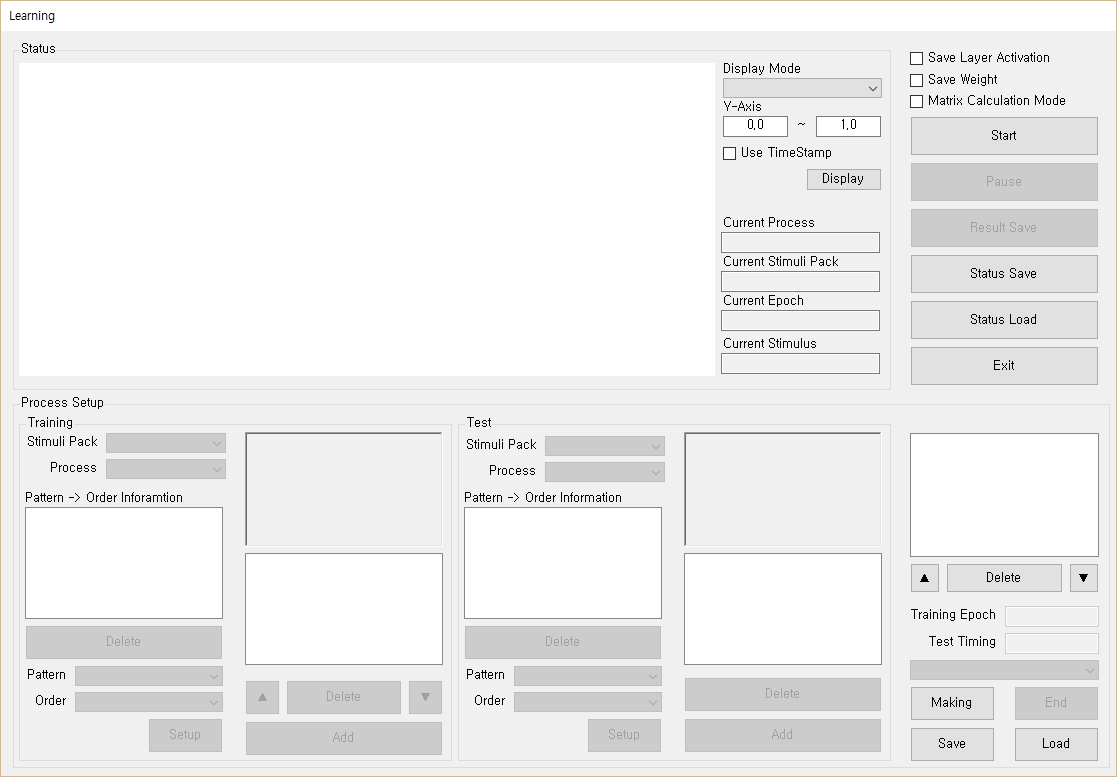


Figure 5. the learning window.

In learning, the model does learning and test. However, before learning, user matches the process and stimuli pack for training and test, and set several detail setting.

### Learning Setup

Learning setup is the setting about learning and test. Before this stage, user has to insert the model architecture, stimuli packs and processes. In this stage, user makes some pairs between the process and stimuli pack in both of training and test. Figure 6 shows examples about the structure of learning setup.

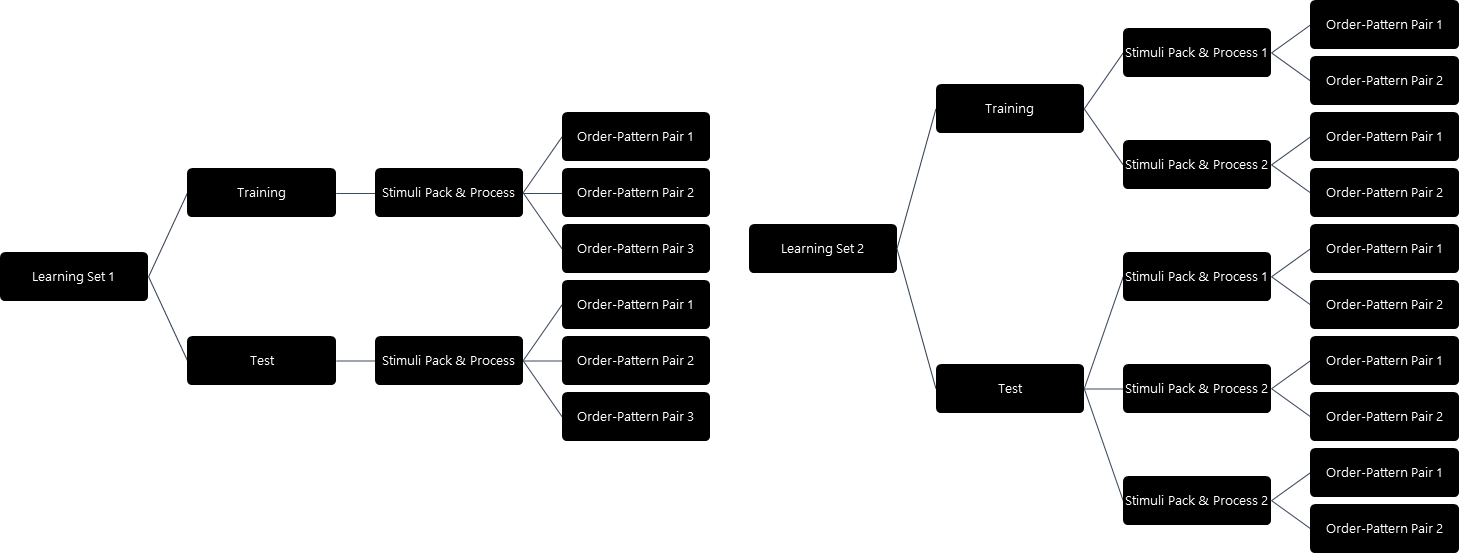
****

Figure 6. examples the structure learning setup

Like the Figure 6, learning setup can have multi learning sets. Each learning set mean a pair training and test information.  
In a learning set, a training information also can have multi pair of stimuli pack and process (learning set 2 of Figure 6). This means that several training flow can be used in a training. This function is to cope the situation which user premises that model uses several separate training flow in an epoch. In addition, this tool provides some randomize methods about this function. Table 3 shows how these methods operate.   
Importantly, as mentioned above, a stimuli pack has several patterns and there are some orders which need pattern information in a process. User also has to match between those patterns and order correctly. If there are some orders that pattern is not assigned, training information cannot be inserted.

On the other hand, same method is applied to test information except the randomize method. Because there is no renewal process in test, we think that there is no renewal order in test process, Therefore, test processes always use ‘sequential all’ method.  
In addition, each learning set have two variable: training epoch and test timing. Training epoch means how many this learning set train. The epoch is bigger, model trains the learning set more.  
Test timing means how often test process conduct. This tool runs the test process when the rest of current epoch divided by test timing is 0. The test result is saved at memory, and displayed in graph of tool. Important point is that small test timing make tool provides the detail transition of model, but at the same time that makes the model’s training be slower and memory use more.  
Matrix calculation size mean the how many stimuli trained in one time. Although some other connectionist model tools use the term ‘mini-batch’ about this mode, we use ‘the matrix calculation’ to avoid a confusion between this mode and other function. Usually, the matrix calculation size is bigger, the training speed is faster. However, it makes the training quality be lower.  
Importantly, increased matrix calculation size makes two problems. One is that this affects the randomize method. Because it is impossible that different processes are trained at a time, stimuli become a package. Therefore, shuffle is limited like Table 4.

Table . the difference among training randomize method.

|  |  |
| --- | --- |
| Name | Method |
| Random all | Both of the process order and training stimuli order are randomized. In other words, all processes’ all learning stimuli are randomized at each epoch.  Ex) Process1-Stimulus4 🡪 Process3-Stimulus2 🡪 Process2-Stimulus3 🡪 … |
| Random in Stimuli Pack | Training stimuli sequence is randomized. In all epochs, process is always sequential, but the learning stimuli of each process are randomized.  Ex) Process1-Stimulus4 🡪 Process1-Stimulus2 🡪 … 🡪Process2-Stimulus3 🡪 Process2-Stimulus16 🡪… |
| Sequential All | Both of the process order and training stimuli order are sequential.  Ex) Process1-Stimulus1 🡪 Process1-Stimulus2 🡪 … 🡪Process2-Stimulus24 Process2-Stimulus25🡪… |
| Sequential in Stimuli Pack | Only process sequence is randomized.  Ex) Process3-Stimulus1 🡪 Process3-Stimulus2 🡪 … 🡪Process1-Stimulus24 🡪 Process1-Stimulus25🡪… |

Table . The difference among training randomize method in matrix calculation.

|  |  |
| --- | --- |
| Name | Method |
| Random all | Ex) Process1-Stimulus4,2,3 🡪 Process3-Stimulus2,10,1 🡪 Process2-Stimulus3,4,7 🡪 … |
| Random in Stimuli Pack | Ex) Process1-Stimulus4,1,5 🡪 Process1-Stimulus2,10,9 🡪 … 🡪Process2-Stimulus3,4,6 🡪 Process2-Stimulus16,1,10 🡪… |
| Sequential All | Ex) Process1-Stimulus1,2,3 🡪 Process1-Stimulus4,5,6 🡪 … 🡪 Process2-Stimulus24,25,26 🡪 Process2-Stimulus27,28,29 🡪… |
| Sequential in Stimuli Pack | Ex) Process3-Stimulus1,2,3 🡪 Process3-Stimulus4,5,6 🡪 … 🡪Process1-Stimulus24,25,26 🡪 Process1-Stimulus27,28,29🡪… |

Second is that the matrix calculation restricts the SRN algorithm. Because context layer use the previous hidden layer’s activation, it makes a problem to change the matrix size. Therefore, the time stamp of all stimuli have to same in matrix calculation. If these problems affect the simulation results, user has to set the matrix calculation size to 1.

### Learning Setup Save & Load

Like other menus, there are the save and load functions about the learning setup. The saved file is XML document, and the saved file has “LSXML” extension. User can open and edit the save file on text editors.  
Like the the PRCXML file, when program load the LSXML file, user should not use the other model’s file. LSXML files depend on the all other information. Currently, this tool does not verify that loaded processes are suitable with current architecture setting. User has to be careful.  
Also, like the other setup, when program load the LSXML file, all current learning setup information is swept. Although a caution message display, user has to be careful.

### Result Save

When model conduct their test processes, several test data is saved in memory. When result save function is run, these accumulated data is written to the several files: raw data, squared error, cross entropy, semantic stress, correctness, target activate unit average, target inactivate unit average, and optionally layer activation and weight. We will talk what data is written in the result file section.  
The button for this function is only activated when model training status is pause or end. In addition, when model finishes the training, this function is automatically run once.

### Save Layer Activation & Weight

Save Layer and save weight functions are the additional function. Although this tool basically provides the activation information of output layer, we think that some users need the information about other layers and connections. However, the information requires very big memory resources.  
Save layer is the function which save the activation information of selected layer at the timing of each stimulus test. In this function, user can select all types of layers like normal, cleanup, or hide layer of BPTT.  
On the other hand, save weight is the function which save the weights information of select connection at the start timing of test, not the timing of each stimulus. As mentioned in learning setup section, we think that there is no renewal order in test process. This means that weights are not changed in test processes. Therefore, this tool saves the weights information at only start timing of test. Like the save layer, user can select the various types of weights like normal connection, layer to cleanup, cleanup to layer, or hide connection of BPTT.

### Status Save & Load

This function is for continuation of model training or test. Like the result save, the button for weight save function is only activated when model training status is pause or end. When this tool saves the weight, model’s all current weights are saved as a file. This file has ‘WDATA’ extension.  
Weight load function can be used only before starting training. When tool loads a WDATA file, model architecture has to coincide the architecture of saved WDATA file. All weights of connections are changed, and previous weight statuses (maybe initial status) are swept.  
One of important things is that WDATA file does not have the transition of weights. This is only a status, so WDATA file cannot instruct how the status was constructed.

### Graph Display

As mentioned above, this tool saves the test result to the memory. In many cases, user want to know whether this model’s training is correct or not. To propose these information, this tool display several types of graphs.  
The types is related the result data. As mentioned, squared error, cross entropy, semantic stress, correctness, target activation unit average, and target inactivation unit average data is saved at test. Graph shows the ‘average’ of one of those data.  
In addition, there are two types of graph display methods whether using time-stamp or not. Figure 7 shows the difference. When user does not use time-stamp, the x-axis of graph is epochs sum of all learning sets (left of Figure 7). On the other hand, when user use the time-stamp, the x-axis is the time-stamp. In this case, each epoch becomes a line (right of Figure 7).

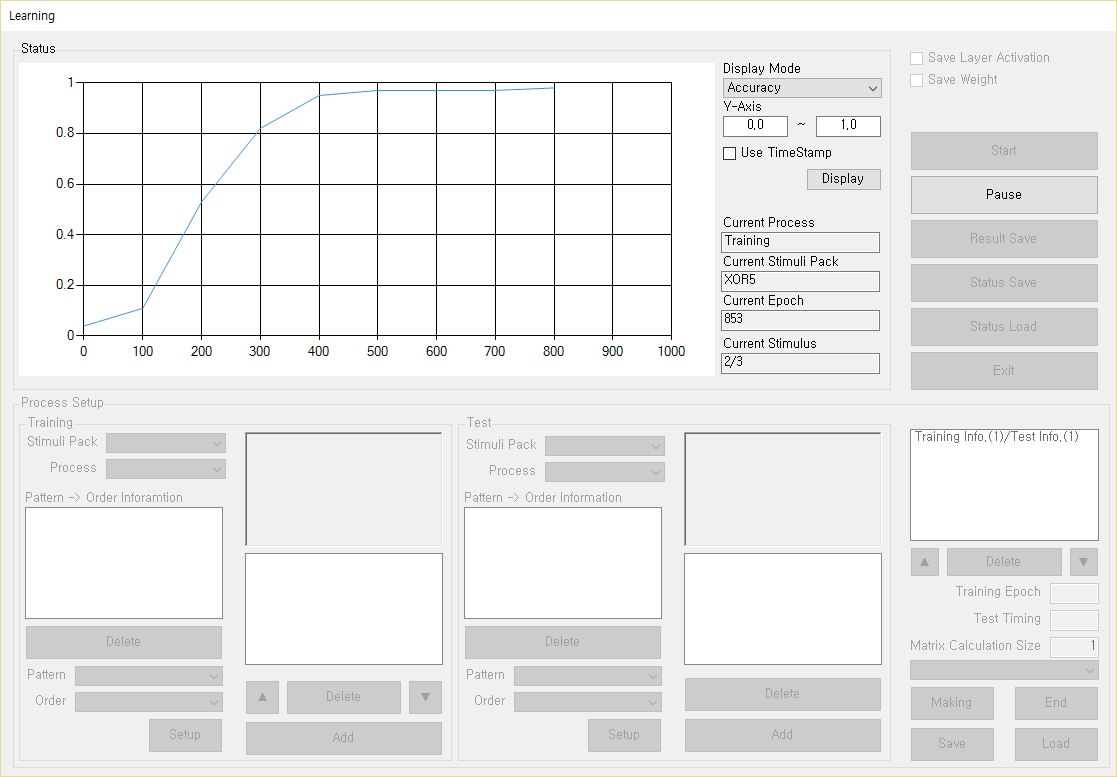
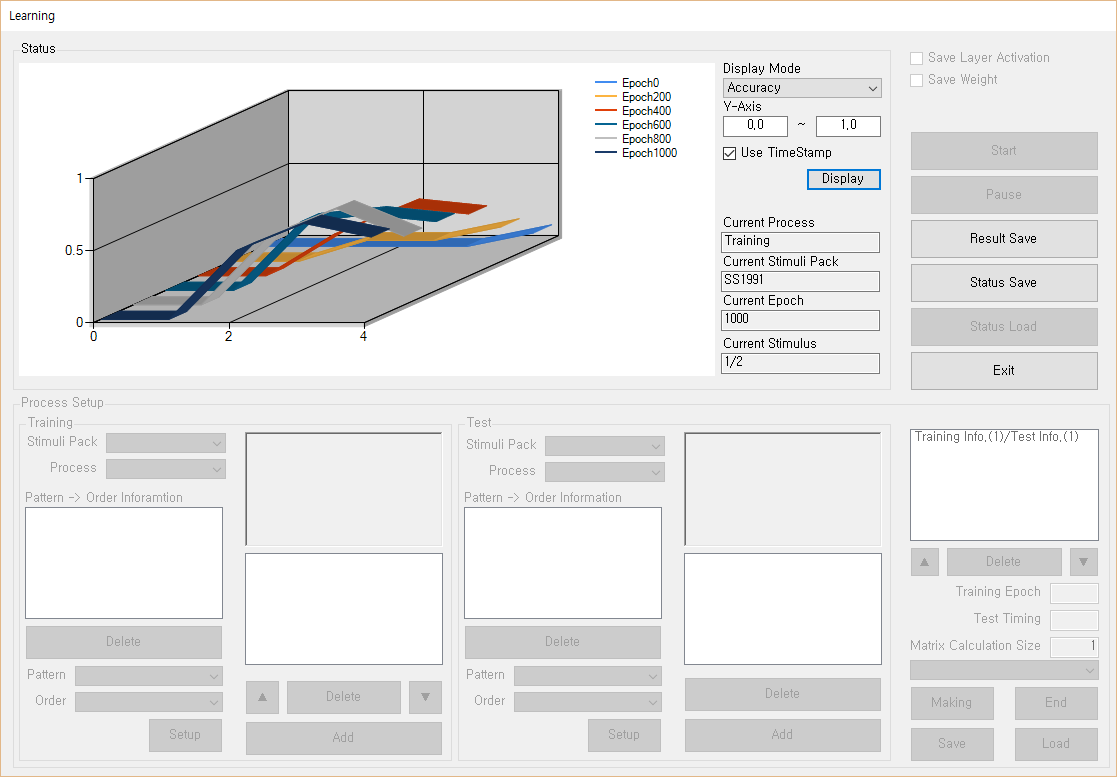
 

Figure 7. the graph display in training.

## Batch Mode

Batch mode is for the reservation of multiple simulations. This mode uses the save files of the all setup: ACTXML, PRCXML, LSXML. And, the inserting method of stimuli data is same to the stimuli making menu. User can use two functions: repeat and variable setup.

### Repeat function

It is very simple. Tool inserts same simulation repetitively. If you do not want to repeat, it is OK to insert 1 as the repeat value.

### Variable setup

We think that most users want to diversify some variables which have been mentioned as the major factors of modeling performance like learning rate. In variable setup, user can set some gradation of the variables. When variable setup function use, some variables of loaded original simulation is reassigned and diversified.  
Now, this function supports 5 variables: layer unit, layer damage, connection damage, learning rate, and initial weight range. We think that these variables have been controlled in many previous studies, and will be controlled in future studies. Also, it can be added some other variables.  
Besides the variable, user additionally has to decide the several factors. One is the range of variable. Start point and end point is the range of gradation. Therefore, the amount of inserted batch simulation is increased when the range is wide. Another is the step. Step decides the detail of the difference. When the step is small, the amount of inserted simulation batch is increased. For example, when user set start point, end point, and step of layer unit variable are 100, 200, and 10, respectively, 11 batch simulation will be inserted. On the other hand, when start point, end point, and step of layer unit variable are set 100, 200, and 50, respectively, 3 batch simulation will be inserted.  
In addition, one of the important thing is that the multiple variable can be assigned. All diversified variables of the original simulation are simultaneously. Therefore, when there are the variable setups, which make 10 batch and 3 batch simulations be inserted, the 30 batch will be inserted.

After inserting all training information, tool will sequentially conduct the all simulations. In batch model, there is no the result or status save functions in pause. Instead of, both of functions are executed automatically after training.

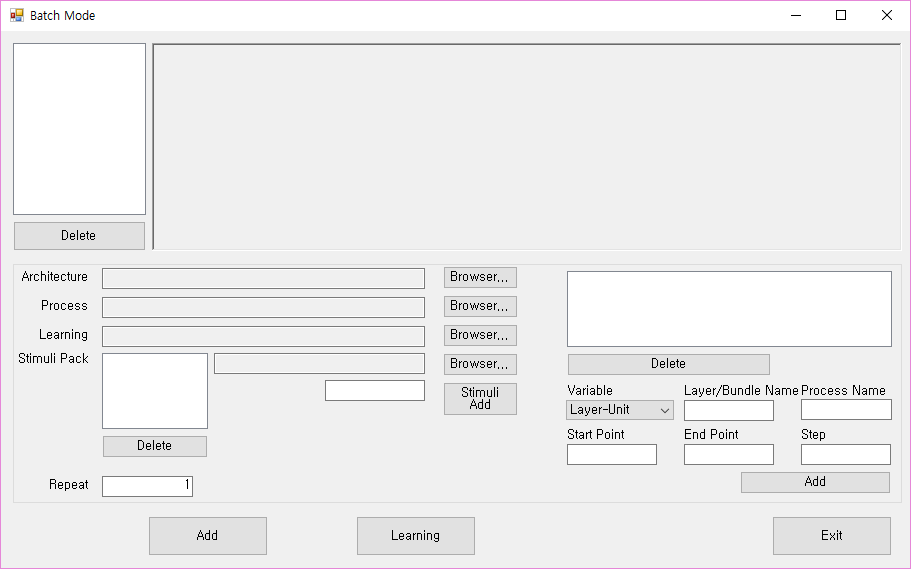
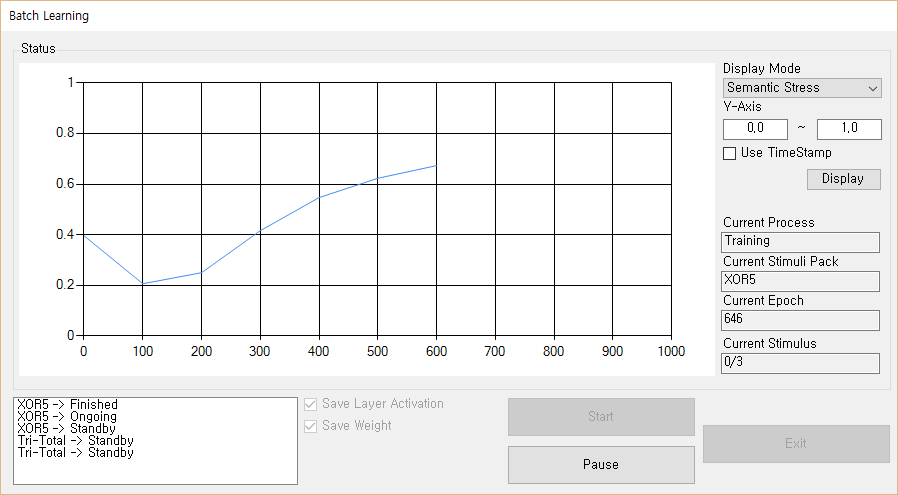
 

Figure 8. the batch mode windows.

## Result Files

As mentioned in result save section, model’s test results are saved as several files in a directory which named from saving time. All files have the ‘txt’ extension, so user can open these files on the text editor like ‘notepad’. However, we recommend to copy the data to a spreadsheet program like ‘Microsoft excel’ or ‘Apache OpenOffice Calc.’ for ease check and analysis.  
Most files, which is in the directory, have the information about the transitions of calculated value of each stimulus, except information, raw data, layer, and weight files (layer and weight files are optional). Table 5 shows the meaning of these files’ content.

Table . the types and contents of result files.

|  |  |
| --- | --- |
| File name | Content |
| SimulationResult-Accuracy.txt | Each stimulus’s correctness of is written. The correctness calculation method was mentioned in architecture setup section. The used criteria are set in architecture setup. If output layer’s activation is correct, the cell is 1, and the incorrect stimulus’s cell is 0. |
| SimulationResult-ActivateUnit.txt | The average of all units that target activation is higher than activate criterion is written. The unit’s correctness is not considered. |
| SimulationResult-CrossEntropy.txt | The cross entropy value of each stimulus is written. The cross entropy is calculated by below equation. |
| SimulationResult-InactivateUnit.txt | The average of all units that target activation is lower than inactivate criterion is written. The unit’s correctness is not considered. |
| SimulationResult-SemanticStress.txt | The semantic stress value of each stimulus is written. The semantic stress is calculated by below equation. |
| SimulationResult-SquaredError.txt | The squared error value of each stimulus is written. The squared error is calculated by below equation. |

Information data file simply shows the basic variables about the simulation. This model’s save directory name is just save time, so there is no information about saved simulation. When batch mode use, it is possible that user cannot know what result is user want. However, this information data is very simple, so we recommend to keep the other save files like architecture or stimuli.

Raw data file shows the all result information. In particular, this file shows each unit status of output layer, including activation and correctness. Basically, above files are summaries about this raw data. When user want to analyze some detail results of model, we recommend using this raw data.  
In addition, if user used save layer function or save weight function, the file ‘ActivationData.txt’ or ‘WeightData.txt’ is added in the directory, respectively. These files show the selected layer’s activation or weight information.

# Process Algorithm

This tool makes a learning flow through the assembling several orders, and these orders are the parts some basic algorithms like BP. In this section, we will mention the structure of base algorithms and orders through pseudo-code. We think that the understanding the algorithm used in this tool is not indispensable. However, we anticipate that this information can be used for improvement of this tool through catching some errors that we cannot notice.

## The Base Pseudo-Code

Table 6, Table 7, and Table 8 are the pseudo-codes about BP, SRN, and BPTT, respectively. This pseudo-codes were not directly used in this tool, but most orders refer to these pseudo-codes. The variables, which have ‘Pattern’, ‘Sum’, ‘Activation’, ‘Error’, or ‘Bias’ suffix, are vectors, and the variables, which have ‘Weight’ suffix, are matrices.

Table 6. the pseudo-code of basic back-propagation algorithm.

For (epoch\_Index = 1:epoch)

{

Randomize pattern order

For (pattern\_Index = 1:pattern)

{

Input\_Pattern Insert

Hidden\_Sum = Input\_Pattern × Input-to-Hidden Weight;

Hidden\_Actiation = Sigmoid\_Function(Hidden\_Sum + Hidden\_Bias);

Output\_Sum = Hidden\_Activation × Hidden-to-Output\_Weight;

Output\_Activation = Sigmoid\_Function(Output\_Sum + Output\_Bias);

Output\_Error = (Target\_Pattern – Output\_Activation) \* Sigmoid\_Function’(Output\_Sum + Output\_Bias);

Hidden\_Error = Output\_Error × Hidden-to-Output\_WeightT \*  
 Sigmoid\_Function’(Hidden\_Sum + Hidden\_Bias);

Hidden-to-Output\_Weight = Hidden-to-Output\_Weight +  
Learning\_Rate × (Hidden\_ActivationT × Output\_Error);

Input-to-Hidden\_Weight = Input-to-Hidden\_Weight +  
 Learning\_Rate × (Input\_PatternT × Hidden\_Error);

Output\_Bias = Output\_Bias + Learning\_Rate × Output\_Error;

Hidden\_Bias = Hidden\_Bias + Learning\_Rate × Hidden\_Error;

}

}

※Each operator ‘+’, ‘×’, ‘\*’ is matrix add, matrix multiply, and matrix elementwise multiply, respectively.

※Superscript ‘T’ means the transposed matrix or vector.

Table 7. the pseudo-code of simple recurrent network (Elman network) algorithm.

For (epoch\_Index = 1:epoch)

{

Randomize pattern order

For (pattern\_Index = 1:pattern)

{

Context\_Activation = Zero\_Vector;

For (tick\_Index = 1:tick)

{

Input\_Patterntick Insert

Hidden\_Sum = Input\_Pattern × Input-to-Hidden Weight +  
 Context\_Activation × Context-to-Hidden Weight matrix;

Hidden\_Actiation = Sigmoid\_Function(Hidden\_Sum + Hidden\_Bias);

Output\_Sum = Hidden\_Activation × Hidden-to-Output\_Weight;

Output\_Activation = Sigmoid\_Function(Output\_Sum + Output\_Bias);

Output\_Error = (Target\_Pattern – Output\_Activation) \* Sigmoid\_Function’(Output\_Sum + Output\_Bias);

Hidden\_Error = Output\_Error × Hidden-to-Output\_WeightT \*  
 Sigmoid\_Function’(Hidden\_Sum + Hidden\_Bias);

Hidden-to-Output\_Weight = Hidden-to-Output\_Weight +  
Learning\_Rate × (Hidden\_ActivationT × Output\_Error);

Input-to-Hidden\_Weight = Input-to-Hidden\_Weight +  
 Learning\_Rate × (Input\_PatternT × Hidden\_Error);

Context-to-Hidden\_Weight = Context-to-Hidden\_Weight +  
 Learning\_Rate × (Context\_ActivationT × Hidden\_Error);

Output\_Bias = Output\_Bias + Learning\_Rate × Output\_Error;

Hidden\_Bias = Hidden\_Bias + Learning\_Rate × Hidden\_Error;

Context\_Activation = Hidden\_Actiation;

}

}

}

※Each operator ‘+’, ‘×’, ‘\*’ is matrix add, matrix multiply, and matrix elementwise multiply, respectively.

Table 8. the pseudo-code of back-propagation through time algorithm.

For (epoch\_Index = 1:epoch)

{

Randomize pattern order

For (pattern\_Index = 1:pattern)

{

For (tick\_Index = 0:tick)

{

Input\_Patterntick\_Index Insert

Hiddentick\_Index\_Sum = Input\_Patterntick\_Index × Hidden-to-Hidden Weight;

Hiddentick\_Index\_Actiation = Sigmoid\_Function(Hiddentick\_Index\_Sum + Hidden\_Bias);

}

Output\_Sum = Hiddentick\_Activation × Hidden-to-Output\_Weight;

Output\_Activation = Sigmoid\_Function(Output\_Sum + Output\_Bias);

Output\_Error = (Target\_Pattern – Output\_Activation) \* Sigmoid\_Function’(Output\_Sum + Output\_Bias);

Hiddentick\_Error = Hidden-to-Output\_Weight × Output\_Error \*  
 Sigmoid\_Function’(Hiddentick\_Sum + Hidden\_Bias);

For (tick\_Index = tick-1:0)

{

Hiddentick\_Index\_Error = Hidden(tick\_Index+1)\_Error × Hidden-to-Hidden\_WeightT \*  
 Sigmoid\_Function’(Hiddentick\_Index\_Sum + Hidden\_Bias);

}

Hidden-to-Output\_Weight = Hidden-to-Output\_Weight +  
Learning\_Rate × (Hiddentick\_ActivationT × Output\_Error);

Hidden\_Bias\_Changetick = Learning\_Rate × Hiddentick\_Error;

For (tick\_Index = tick-1:0)

{

Hidden-to-Hidden\_ Change\_ Weighttick\_Index = Learning\_Rate ×  
 (Hiddentick\_Index\_ActivationT × Hidden(tick\_Index+1)\_Error);

Hidden\_Bias\_Changetick\_Index = Learning\_Rate × Hiddentick\_Index\_Error;

}

Hidden-to-Hidden\_Weight = Hidden-to-Hidden\_Weight +  
Average(Hidden-to-Hidden\_Weight\_Change);

Input-to-Hidden\_Weight = Input-to-Hidden\_Weight +  
 Learning\_Rate × (Input\_PatternT × Hidden0\_Error);

Output\_Bias = Output\_Bias + Learning\_Rate × Output\_Error;

Hidden\_Bias = Hidden\_Bias + Average(Hidden\_Bias\_Change)

}

}

※Each operator ‘+’, ‘×’, ‘\*’ is matrix add, matrix multiply, and matrix elementwise multiply, respectively.

## Orders about Layer

In this tool, unlike the Base Pseudo-Code, the variables, which have ‘Pattern’, ‘Sum’, ‘Activation’, ‘Error’, or ‘Weight’ suffix, are matrices. the variables, which have ‘Bias’ suffix, are vectors. This is because this tool use the matrix calculation basically.   
In addition, to add the bias vectors to layer matrix, the matrices, which all rows are the bias vector, are made and used. We will display this as a function expression ‘BiasMatrix\_Function().’

### Input layer activation

Replace the activation vector of selected layer to input pattern vector.

Layer\_Activation = Input\_Pattern;

### Activation Calculate (Sigmoid)

Calculate the activation vector of selected layer using sigmoid function.

Layer\_Bias\_Matrix = BiasMatrix\_Function(Layer\_Bias);

Layer\_Activation = Sigmoid\_Function(Layer\_Sum + Layer\_Bias\_Matrix);

### Activation Calculate (Softmax)

Calculate the activation vector of selected layer using softmax function.

Layer\_Bias\_Matrix = BiasMatrix\_Function(Layer\_Bias);

Layer\_Activation = Softmax\_Function(Layer\_Sum + Layer\_Bias\_Matrix);

### Activation Send

Multiply the selected layer’s activation vector and send-connection’s weight. After that, add the vectors to each send connection’s receive layer’s sum vector. In this order, selected layer sends itself’s activation to all send-connected layer, not selectional.

For (connectin\_Index = 1:n) (n = 1,2 … send-connection’s count)

{

Layer\_Sumconnection\_Index = Layer\_Sumconnection\_Index +  
 Selected\_Layer\_Activation × Connection\_Weightconnection\_Index;

}

### Output Layer Error Calc. (Sigmoid)

Calculate the error vector of selected layer, which activation is calculated by sigmoid function, through the comparison between activation and target. Sigmoid\_Function’(prime) is the differential of sigmoid function.

Layer\_Bias\_Matrix = BiasMatrix\_Function(Layer\_Bias);

Layer\_Error = (Target\_Pattern – Layer\_Activation) \*  
 Sigmoid\_Function’(Layer\_Sum + Layer\_Bias\_Matrix);

### Output Layer Error Calc. (Softmax)

Calculate the error vector of selected layer, which activation is calculated by softmax function, through the comparison between activation and target pattern.

Layer\_Error = Target\_Pattern – Layer\_Activation

### Hidden Layer Error Calc. (Sigmoid)

Calculate the error vector of selected layer, which activation is calculated by sigmoid function. Because hidden layer does not have the target pattern, error is calculated by using the error of layers which are connected by send-connections. Like Output Layer Error Calc., Sigmoid\_Function’(prime) is the differential of sigmoid function.

For (connectin\_Index = 1:n) (n = 1,2 … send-connection’s count)

{

Layer\_Bias\_Matrix = BiasMatrix\_Function(Layer\_Bias);

Layer\_Error = Layer\_Error + (Layer\_Errorconnection\_Index × Connection\_Weightconnection\_IndexT \*  
 Sigmoid\_Function’(Layer\_Sum + Layer\_Bias\_Matrix));

}

### Hidden Layer Error Calc. (Softmax)

Calculate the error vector of selected layer, which activation is calculated by softmax function. Because hidden layer does not have the target pattern, error is calculated by using the error of layers which are connected by send-connections.

For (connectin\_Index = 1:n) (n = 1,2 … send-connection’s count)

{

Layer\_Error = Layer\_Error + Layer\_Errorconnection\_Index × Connection\_Weightconnection\_IndexT

}

### Output Layer Test

Calculate the several result values, which can instruct model situation and save to the memory. Semantic stress does not use the target pattern.

Result\_Squared\_Error(current\_epoch,stimulus) = Squared\_Error\_Function(Target\_Pattern, Layer\_Activation);

Result\_Cross\_Entropy(current\_epoch,stimulus) = Cross\_Entropy\_Function(Target\_Pattern, Layer\_Activation);

Result\_Semantic\_Stress(current\_epoch,stimulus) = Semantic\_Stress\_Function(Layer\_Activation);

Result\_Correctness(current\_epoch,stimulus) = Correctness\_Function(Target\_Pattern, Layer\_Activation);

Result\_Target\_Activate\_Avg(current\_epoch,stimulus) =  
 Target\_Activate\_Avg\_Function (Target\_Pattern, Layer\_Activation);

Result\_Target\_Inactivate\_Avg(current\_epoch,stimulus) =  
 Target\_Inactivate\_Avg\_Function (Target\_Pattern, Layer\_Activation);

### Bias Renewal

Renew the bias of layer using the error of layer. Because bias is a vector, a column sum function is used about Layer\_Error. This function calculates the sums of each column, and makes a vector. After that, bias is decayed.

Layer\_Bias = Layer\_Bias + Learning\_Rate × Column\_Sum\_Function(Layer\_Error);

Decay\_Function(Layer\_Bias)

### Layer Duplicate

Copy the first layer’s activation, sum, error, bias, and interconnection weight to the second layer.

Layer2\_Sum = Layer1\_Sum;

Layer2\_Activation = Layer1\_Activation;

Layer2\_Error = Layer1\_Error;

Layer2\_Bias = Layer1\_Bias;

Layer2\_Interconnection\_Weight = Layer1\_Interconnection\_Weight;

### Interact

Multiply the selected layer’s activation vector and interconnection weight. After that, add the vectors to selected layer’s sum vector. This order does not include the calculation of activation.

Layer\_Sum = Layer\_Sum + Layer\_Activation × Layer\_Interconnection\_Weight;

### Interconnection Renewal

Renew the interconnection of layer. The self-connection cells of interconnection weight like (1,1), (2,2) become 0. After that, interconnection is decayed.

Layer\_Interconnection\_Weight = Layer\_Interconnection\_Weight +  
Learning\_Rate × (Layer\_ActivationT × Layer\_Error);

For (rowIndex=1:n) (n = 1,2, …, Layer’s unit count)

{

Layer\_Interconnection\_Weightn,n = 0;

}

Decay\_Function(Layer\_Interconnection\_Weight)

### Layer -> CleanUp Process

Multiply the selected layer’s activation vector and the weight which is from layer to cleanup layer. After that, add the vectors to selected layer’s sum vector, and calculate the activation of cleanup layer.

Layer\_Cleanup\_Bias\_Matrix = BiasMatrix\_Function(Layer\_Cleanup\_Bias);

Layer\_Cleanup\_Sum = Layer\_Activation × Layer\_to\_Cleanup\_Weight;

Layer\_Cleanup\_Activation = Sigmoid\_Function(Layer\_Cleanup\_Sum + Layer\_Cleanup\_Bias\_Matrix);

### CleanUp -> Layer Process

Multiply the selected layer’s cleanup layer activation vector and the weight which is from cleanup to layer. After that, add the vectors to selected layer’s sum vector. This order does not include the calculation of activation.

Layer\_Sum = Layer\_Sum + Layer\_Cleanup\_Activation × Cleanup\_to\_Layer\_Weight;

### CleanUp Backward Process

Like bias renewal, a column sum function is used about Layer\_Cleanup\_Error at the calculation ‘Layer\_Cleanup\_Bias.’

Layer\_Cleanup\_Error = Layer\_Error × Cleanup\_to\_Layer\_WeightT

\* Sigmoid\_Function’(Layer\_Cleanup\_Sum + Layer\_Cleanup\_Bias);

Layer\_Cleanup\_Bias = Layer\_Cleanup\_Bias +  
 Learning\_Rate × Column\_Sum\_Function(Layer\_Cleanup\_Error);

Cleanup\_to\_Layer\_Weight = Cleanup\_to\_Layer\_Weight +  
Learning\_Rate × (Layer\_Cleanup\_ActivationT × Layer\_Error);

Layer\_to\_Cleanup\_Weight = Layer\_to\_ Cleanup\_Weight +  
Learning\_Rate × (Layer\_ActivationT × Layer\_Cleanup\_Error);

Decay\_Function(Layer\_Cleanup\_Bias);

Decay\_Function(Cleanup\_to\_Layer\_Weight);

Decay\_Function(Layer\_to\_Cleanup\_Weight);

## Orders about BPTT Orders

Some orders are only for BPTT Layer. BPTT layer has several variables, which are to control tick. In addition, in using BPTT Layer, some layer orders are also changed because BPTT layer has different structure.

### Tick In

Copy the layer sum vector to hide layer0, and calculate the activation of the hide layer0. After that, initialize layer sum to insert next activations. Current\_Tick value becomes a marker for the status of BPTT Layer.

Hide\_Layer\_Bias = BiasMatrix\_Function(Hide\_Layer\_Bias);

Hide\_Layer0\_Sum = Layer\_Sum;

Hide\_Layer0\_Activation = Sigmoid\_Function(Hide\_Layer0 \_Sum + Hide\_Layer\_Bias);

Layer\_Sum Initialize;

Current\_Tick = 0;

### Tick Progress

This is similar to ‘Tick In’ order, except the calculation hide layer sum. The layer of previous tick also sends the activation. In all ticks, Hide\_Weight is same (unfold).

Hide\_Layer\_Bias = BiasMatrix\_Function(Hide\_Layer\_Bias);

Hide\_LayerCurrent\_Tick+1\_Sum = Layer\_Sum + Hide\_Layer\_ActivationCurrent\_Tick × Hide\_Weight;

Hide\_Layer Current\_Tick+1\_Activation = Sigmoid\_Function(Hide\_Layer Current\_Tick+1\_Sum + Hide\_Layer\_Bias);

Layer\_Sum Initialize;

Current\_Tick = Current\_Tick + 1;

### Tick Output

Copy the calculated activation of final hide layer to the layer for external process.

Layer\_Activation = Hide\_LayerCurrent\_Tick\_Activation;

### Base Weight Renewal

Calculate the error of each hide layer. Although Hide\_Weight is same in all ticks, delta values, which is for renewal hide weight and hide layer bias, are different because the error of each hide layer is different.  
Layer\_Error is calculated twice. First calculation is same to the usual hidden layer error. This is used for the calculation of error of hide layers. Second calculation uses the hide layers’ error. This is used for the calculation of error of other layers connected.

Hidden Layer Error Calc. (Sigmoid);

Hide\_LayerCurrent\_Tick\_Error = Layer\_Error;

For (tickIndex=n:0) (n = Current tick - 1, …, 1, 0)

{

Hide\_LayertickIndex\_Error = Hide\_LayertickIndex+1\_Error × Hide\_WeightT;

}

For (tickIndex=n:0) (n = Current tick, …, 1, 0)

{

Hide\_LayertickIndex\_Bias\_Delta = Learning\_Rate × Column\_Sum\_Function(Hide\_LayertickIndex\_Error);

}

For (tickIndex=n:0) (n = Current tick - 1, …, 1, 0)

{

Hide\_WeighttickIndex\_Delta = Learning\_Rate ×  
 (Hide\_LayertickIndex\_ActivationT × Hide\_LayertickIndex+1\_Error);

}

Hide\_Weight = Hide\_Weight + Average(Hide\_Weight\_Delta);

Hide\_Layer\_Bias = Hide\_Layer\_Bias + Average(Hide\_Layer\_Bias\_Delta);

Decay\_Function(Hide\_Weight)

Decay\_Function(Hide\_Layer\_Bias)

Layer\_Error = Average(Hide\_Layer\_Error)

## Orders about Bundle

### Weight Renewal

Renew the weight of connection using the error of receive layer and activation of send layer. After that, the connection is decayed.

Connection\_Weight = Connection\_Weight +  
 Learning\_Rate × (Send\_Layer\_ActivationT × Receive\_Layer\_Error);

Decay\_Function(Connection\_Weight);

### Bundle Duplicate

Copy the first connection’s weight to the second connection.

Connection2\_Weight = Connection1\_Weight;

### Transposed Bundle Duplicate

Copy the transposed weight of the first connection’s weight to the second connection.

Connection2\_Weight = Connection1\_WeightT;

## Other Orders

### End & Initialize

This is no function. However, tool cognizes the end of process by this order. Therefore, this order has to be located in the last of all processes.