# HNet 2.0 User Manual

Latest update: 2019-01-19

Heejo You

# 1. Brief description of HNet 2.0

## A. Install and Run

HNet 2.0 was built and tested in Python version 3.6.4 and has not been tested to work with Python 2.x. In addition, HNet 2.0 supports GPU acceleration based on Tensorflow, so if you need GPU acceleration, please configure the environment by installing CUDA etc. beforehand.

Before starting the program, enter the following command in the terminal (command line) to complete the upgrade of modules.

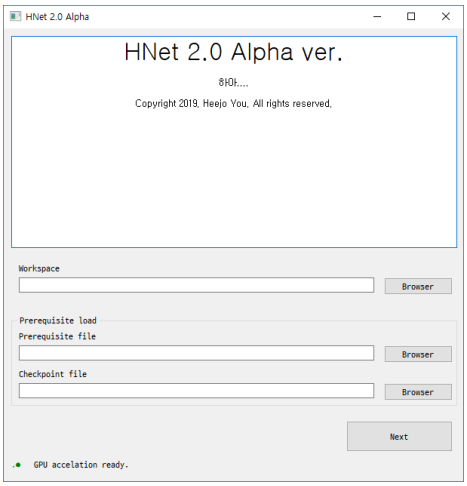
```
pip install -r requirements.txt
```

After completing the configuration for running HNet 2.0, execute the following command to tenimal to run HNet 2.0.

```
Python hnet2_gui.py
```

HNet 2.0 is a 5-step setup including the main, and the model is learned and tested.

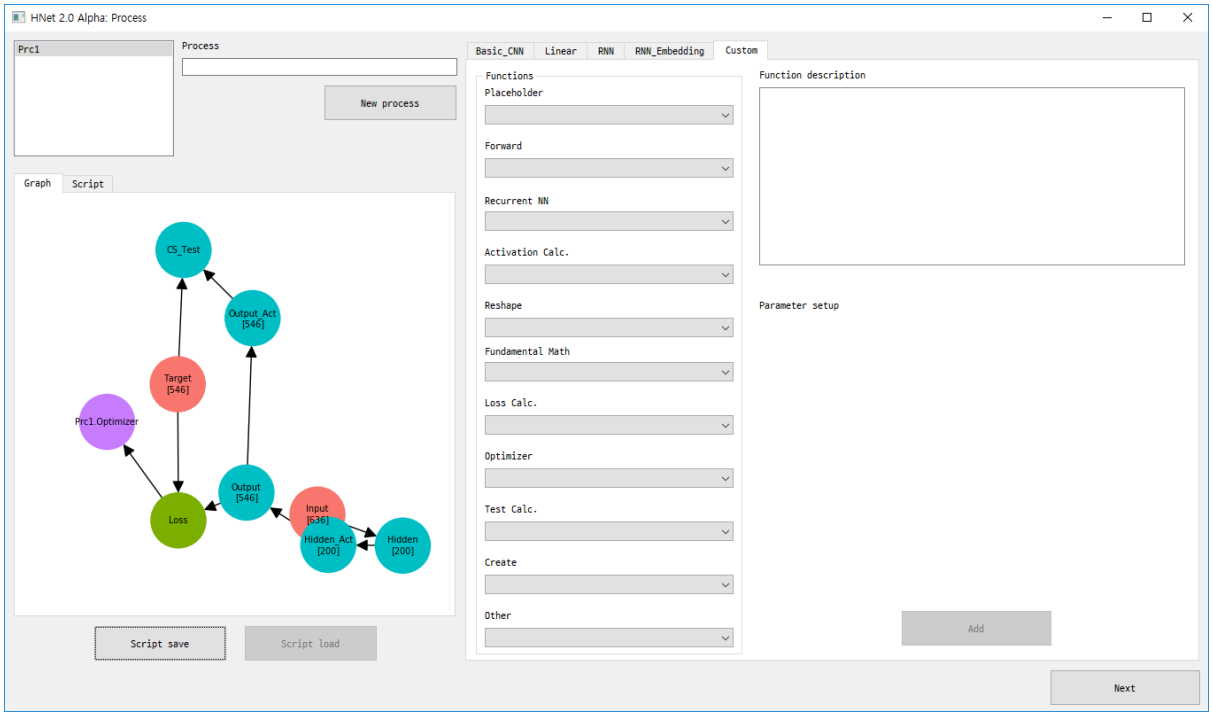
## B. Workspace and Load




Workspace is the folder (directory) where all the model's information is stored. In HNet 2.0, model structure information, learned weight information, and test information are recorded in workspace.

Prerequisite load is the function to load a previously configured model. The prerequisite file is a file of the model's set up information. You can load a file with '.pickle' and '.txt', which will be created in the workspace when you complete all five settings. If you import the model using the prerequisite file, skip all remaining steps and go straight to the train process. On the other hand, the checkpoint file is a file about the weight of the model that we learned before. In other words, the weights of model which is loaded by prerequisite file is changed to the state previously learned. If you only use the prerequisite file, you will start learning from scratch.

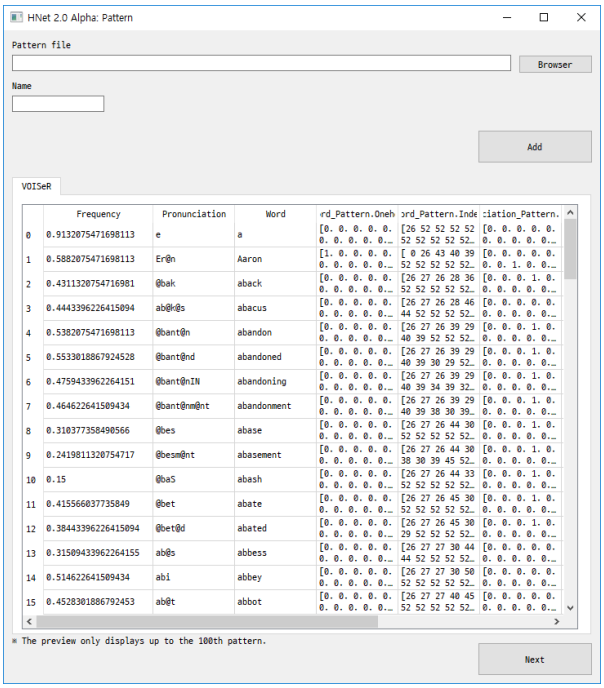
## C. Process setup




Process represents one activation flow in HNet 2.0. The user can use the shortcut tabs at the top right to load a process of a predetermined structure or freely configure a process using a custom tab. The basic elements of process are as follows. 1) A placeholder to insert input and target values (red of the graph). 2) A tensor representing the computation (blue green of the graph). 3) A loss that represents the degree how much model's result is wrong through a comparison the output tensor and the target placeholder (green of the graph). 4) An optimizer that modifies each weight to reduce future loss (purple of the graph).

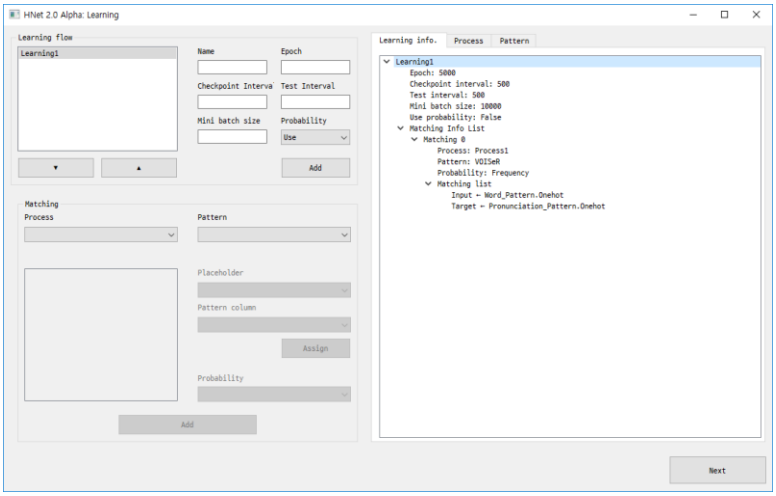
Custom tap allows you to use functions that are mainly used in TensorFlow or deep learning for process configuration. All function is divided into 11 types. If you select function, a brief description of the function is displayed in the upper right corner. The user can determine the detailed parameters of each function and add them to the process.

## D. Pattern setup



HNet 2.0 receives pickled Pandas dataframe as a pattern. If you set the pattern file and the name in the model of the pattern, you can see brief information of the pattern inputted at the bottom.

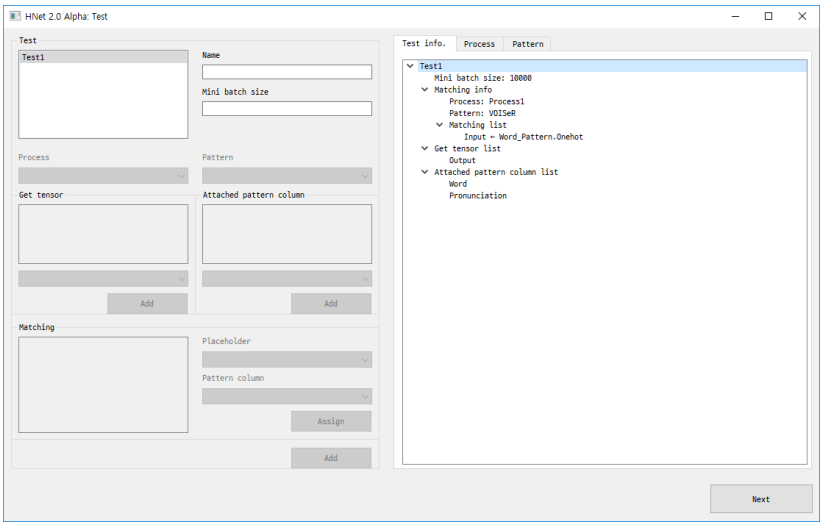
### E. Learning setup




The process is the specific way of learning, and the pattern is what you want model to learn. And a learning matches them and sets how much model learn them. In learning, you will determine how many epochs will be trained during learning, and how often you want to store checkpoint and test in the middle of learning. And, learning will match which pattern column entered in the process will be entered in the placeholder.

The entered learning information can be checked on the right panel, and the process and pattern can be checked again using the same tab.

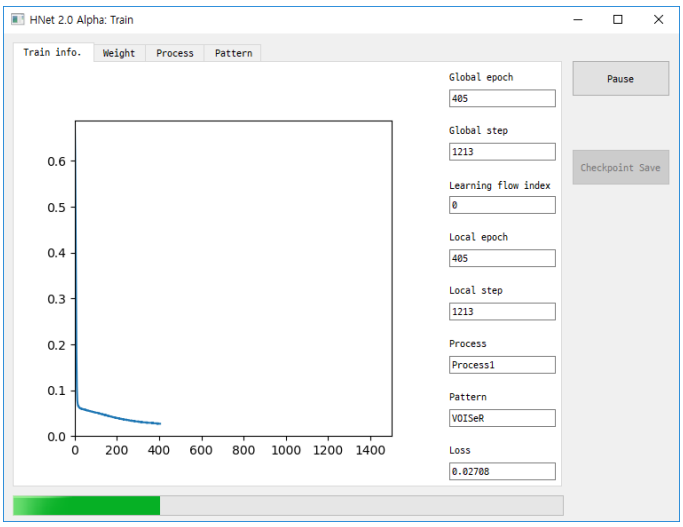
### F. Test setup




Test determines what information is extracted from the model during learning and at the end. If necessary, you can use the same or different process as learning, and you can extract most tensors in the process.

The inserted test information can be checked in the right panel like learning, and process and pattern can be checked again by using tabs in the same position.

G. Train




When the whole setting is finished, the train screen will be presented, and the learning will proceed. During learning, HNet 2.0 graphs current learning progress and simple loss variation. Thus, you can keep track of model's training situation. In addition, you can use the tabs at the top to see the model's current weights structure.

## 2. Details of each feature

### A. Process

#### i. Make new process

Enter the name of the process to be created in the upper left textbox and press the add button to create a blank process without tensor. You **CANNOT** delete an added process and tensor because of the setting of TensorFlow that works as a backend, so if you need to change the process, you must exit and restart the program.

#### ii. Shortcut tabs

Shortcut tabs are methods that configures a process that is configured in advance by inputting some variables needed for model configuration. When the process is created, the tabs at the top are enabled. Select the required process structure and input each parameter and click the 'Add' button to build the specified process.

#### iii. Custom tab

If you want to configure a process without using a shortcut process, you can use the custom tab. Custom tab is enabled when the process is created. You can build processes by adding the necessary placeholder, tensor, loss, and optimizer.

**Placeholder**

A placeholder is a special type of tensor, which is the area where user specified data is input. The user must decide on the size of the placeholder and add it later considering what data to input. For example, if the input pattern and the output pattern to be used for learning are vectors of size 200 and 100, respectively, you need to set size 200 and 100 as placeholders and add them to the process. Multiple placeholders can be added to a process.

To add a placeholder to the process, select it in the 'Placeholder' combobox, then set the detailed

parameters and press the add button at the bottom of the parameter area. Please refer to 4.A.i for details of each parameter.

**Tensor**

The exact definition of a tensor is the result of an operation being performed. If a placeholder or tensor that already exists is operated using a function, a new tensor is generated as the result. Thus, most tensor cannot be formed singly, and the first tensor is created by applying a function to the placeholder.

Although most functions take a single tensor and generate a single new tensor, this is not always the case. There are several functions which require multiple tensors for input such as 'concat', or which generate multiple tensors such as 'split'.

HNet 2.0 basically supports the functions listed in the 4.A, and you can add functions directly if you want. See 3.B for details on adding a function.

**Loss**

Loss is a value that represents the difference between two tensors. All learning in HNet 2.0 proceeds in the direction of reducing loss. The two tensors used to calculate the loss must have the same shape (or size). Also, in HNet 2.0, this value is not used as a tensor and cannot be used for functions other than the optimizer. HNet 2.0 basically supports four loss functions, see 4.A.vii for the use of each loss function.

**Optimizer**

In HNet 2.0, the optimizer is an operator that has led to reduce all losses added to the process so far. To learn the model, an optimizer is required for the process, and a process without an optimizer can only be used for testing. Also, one process can have only an optimizer, and no loss can be added after the optimizer is added. It is therefore not required, but we recommend adding an optimizer at the end of the process configuration. Please refer to 4.A.viii for details on the optimizer supported by HNet 2.0.

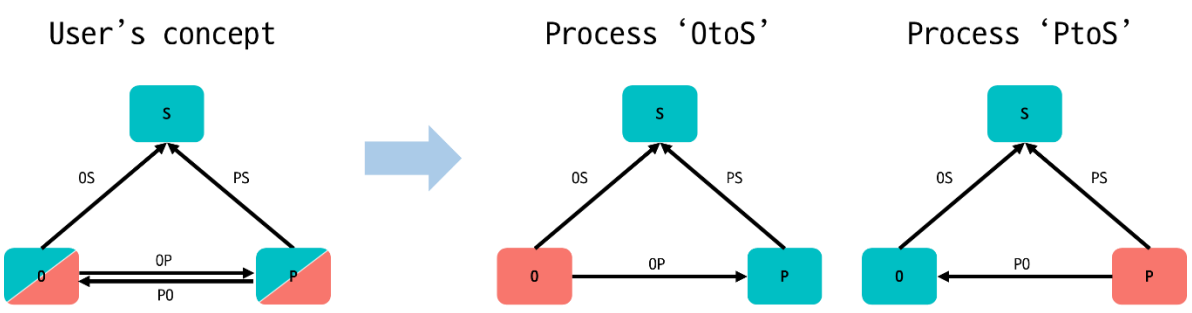
iv. Process graph

The process graph is located at the bottom left of the window. It shows how is connected each component of the process by a network, and the color of each node represents the category of the component. 1) placeholder: red. 2) tensor: blue. 3) loss: green. 4) optimizer: purple.

Placeholder and tensor nodes also have a number that indicates the size of the node, so you can refer to it when you add a tensor.

v. Multi processes and reuse

HNet 2.0 supports multi processes. Each process is basically independent of each other, and one training is also performed in a single process unit. However, depending on user conception, one model may need to learn several times with different processes. Following figure is a simplified example.



The concept model is a structure in which S is output. However, both of O or P can be input, and one is selected and the other becomes hidden. If process 'OtoS' and 'PtoS' are completely independent, you can organize one process and then learn the other. However, if you need to make the weights 'OS' and 'PS' of the two processes identical, you can set both processes and use reuse when entering the tensor. Some functions such as 'dense' or 'conv1d' have a reuse as a parameter. You can use reuse to associate multiple processes with each other, and to manage the processes as part of a larger model.

B. Pattern

HNet 2.0 uses pandas dataframe as a pattern. HNet 2.0 does not support the tools that make up the pattern itself because the pattern required by each user can be different. However, there are some rules for pattern composition. First, the column input to the placeholder must be a numpy array. Depending on the user's needs, information that is not used in model learning can also exist in columns. However, this information cannot be used to study HNet 2.0. Second, the pattern in the column must be the same shape of the same type. In HNet 2.0, one column is in the same placeholder, so an error occurs if the values in the column are not consistent. Third, the column to be used as the probability must be a scalar value between 0 and 1. Probability is discussed in detail in C. Fourth, columns used in one process must all be contained in one pattern. It is possible for multiple processes to use the same pattern, but a process is not allowed to use multiple patterns.

C. Learning

i. Make new learning

In the upper left corner, set up 6 parameters for learning and press the add button to add new learning. The meanings of the six parameters are as follows. 1) name: The identifier of the learning to be set. This value is used for your convenience and has no effect on learning. 2) epoch: Decide how much to learn. 1 epoch means that the entire input pattern has been learned once. 3) checkpoint interval: Determines how often the weight of the model is stored interim. 4) test interval: determines how often the specified test is performed. 5) mini batch size: Determines how many patterns will be learned once the learning is attempted. The larger the mini batch size, the faster the learning speed. However, using a mini batch size that is too large for your memory environment can cause out of memory problems. 6) probability: Decides whether to apply learning possibility of input pattern. When set to 'Use', in each epoch, each pattern is compared with the probability value of each item to judge whether the pattern is learned or not. For No use, all patterns are learned on every epoch.

Details of the entered learning can be seen in the panel on the right. And if the entered learning is wrong, you can delete it by double clicking the learning in the left listbox.

ii. Matching

All learning must be set to match. When you select one process and pattern to use in the learning, the matching panel at the bottom is enabled. Select the placeholder and the pattern column, press the assign button and the corresponding information is entered. The user must assign a specific column to all the placeholders. And if learning uses probability, set the probability column to use. Pressing the Add button assigns the matching set to learning.

iii. Learning flow and multi matching

**Learning flow**

HNet 2.0 supports learning flow. For example, let's say you want to build a model for late bilingual which

learn a second language at they become an adult. This requires a change in the environment. At the initial time, model only learns about the mother language. However, after a certain epoch, the model becomes a form that learns both mother language and second language information. Users can set up multiple learning, and by setting their order, he can configure the model when the environment changes. You can change the order of learning through the '▲' and '▼' buttons on the left learning panel.

**Multi matching**

Apart from learning flow, HNet 2.0 supports Multi matching. Let's reconsider the figure for the multi-process mentioned in the A. Both processes cannot learn either side first, and both learning processes must be done in one learning. To do this, you can set up multiple matching in a single learning. When multi matching is set, the learning sequence of matching is random in mini batch and learning of all mini batches becomes one epoch. For example, when there are mini batches batch 1-1, batch 1-2, and batch 1-3 for matching 1 and batch 2-1 and batch 2-2 for matching 2, these 5 batches are trained in random order, and epoch 1 increases when all five mini batches are learned.

D. Test

The test is performed every test interval set in learning and the result is recorded. Unlike learning, test becomes a test including matching. You can set up multiple tests, and if you have set up multiple tests, all tests will be done at every test interval.

i. Name and mini batch

In test, name does not affect the test result, but it is the identifier of the resulting file to be generated in the future. Mini batch size determines how many patterns are tested at one time. The larger the mini batch size, the smaller the test time, but using an excessively large mini batch size can cause out of memory problems. Mini batch size does not affect test results.

After entering the name and mini batch, select the process and pattern to be used in the test, and the matching panels at the bottom will be enabled.

ii. Get tensor

During the test, select tensors to extract the activation value. All tensors except loss and optimizer are selectable, and at least one tensor must be selected.

iii. Attached pattern column

Select the columns to be attached to the test result file. All columns in the pattern are selectable.

iv. Matching

All tests must be set to match. Select each placeholder and pattern column and press the assign button to enter the corresponding information. **The user must assign a specific column to all the placeholders needed to extract the tensor value selected in 'Get tensor'.** Thus, some placeholders do not need to be assigned values, and if there are unnecessary assignments, they are notified by a message and then excluded.

E. Train

Once all the settings are completed, the screen will jump to the train window. Training starts when the user presses the start button. The progress of learning can be checked in the progress bar at the bottom of the screen. If you need to pause in the middle, press the pause button (same as start button). Training is paused when the current epoch ends. You can save additional checkpoints at the time of pause.

i. Train information

The train information tab displays the learning status except the progress bar. The following information is displayed on the screen. 1) global epoch: shows how many epochs are learned for all learning flows. 2) global step: shows the number of learning steps for all learning flows. 3) learning flow index: Indicates the number of learning in progress in the entire learning flow. The starting value is 0. 4) local epoch: Shows how many epochs are learned for the current learning. 5) local step: shows how many times the learning has been done for the current learning. 6) process: Displays the process used by the current step. 7) pattern: Displays the pattern used by the current step. 8) loss: Displays the loss value derived from the current step. 9) graph: Displays the flow of change of loss sum output from the learning. If several steps in an epoch are learned, loss is the average of the losses calculated at every step in the epoch.

ii. Weight

The weight tab displays an image of the weights currently in use by the model being studied. In case of 3D (convolution 1d) or 4D (convolution 2d) weight like convolution filter, you can select desired filter by index slide. The width of the weight image represents the dimension of the inserted tensor, and the height represents the dimension of the tensor that is calculated.

F. Result file

At the middle of learning, and at the end of learning, HNet 2.0 saves the test result file. The save location is the 'Result' folder in the specified workspace folder. And the file name is following ({} is the corresponding variable):

'GE_{global_epcoh}.LI_{learning_flow_index}.LE_{local_epch}.pickle'

The file is in pickled dict format, and you can see inside through the python pickle library.

'global_Epoch', 'global_Step', 'learning_Index', 'local_Epoch' and 'local_Step' of each item in the Dict indicate information about the learning progress at the time the test was performed. In addition, the actual test results are contained in the subdict called 'Result'. 'Result' dict holds each test name as a key. Each of these items consists of a pandas dataframe in which the test results are recorded. The columns of the dataframe consist of 'get tensor' and 'attached pattern column' when setting the test. If there are overlapping names between 'get tensor' and 'attached pattern column' at test setup, the tags of '.Result' and '.Pattern' are added to column name respectively.

## 3. Advanced use

A. Prerequisite files and modify script

When you finish setting up test, there will be a model folder under the workspace folder, and two files 'Prerequisite.pickle' and 'Prerequisite.txt' will be created.

i. Prerequisite.pickle

'Prerequisite.pickle' has all configuration information except workspace. Users can later load this file to continue learning the same model, or load a checkpoint together to continue the previously discontinued learning. Python's pickle library allows for internal modifications, but is not recommended.

ii. Prerequisite.txt

'Prerequisite.txt' is the same configuration information file. However, unlike pickle, pattern information is not included. It is in txt file and can be opened through text editor. The internals are part of the

Python code that manipulates HNet 2.0, and you can modify the file to configure other types of models or change some parameters as needed. This file can also be loaded as '.pickle'. However, there are a few things to consider when modifying and using the file. 1) Set the pattern path. '#Pattern' comment following code is the part that receives pattern. Since the pattern file is not specified in the parameter, the user must manually add the pattern path to the 'pattern_Path =' parameter. Enter the path to the pattern you used to create the settings for the first time. 2) If you modify the process, you may not be able to use checkpoint. Modifications to the process may change the shape of some or all the weights used by the model, in which case checkpoints are not compatible. 3) Once loaded and the train screen is presented, check the process, pattern, learning, and test tabs before you start learning. If the structure and the pattern you use are not logically correct, in most cases an error occurs during loading and does not go away. However, in some cases you may need to verify it yourself.

B. Function adding

HNet 2.0 basically includes functions that are mostly used by most users. However, there are some functions that are frequently used depending on the user, or they may be used to create customized functions directly. To do this, you can add functions that you can use directly on the Process screen as needed.

All the functions of the process screen are recorded in 'func_Dict' of 'HNet2/GUI/Process/Process_Func.py' in dict form. The user who wants to add a new function must set the function according to the rules of HNet 2.0. Below is the code of 'Layer_dense' set in HNet.

```
func_Dict["Layer_Dense"] = {
    "Func_Type": Func_Type.Forward,
    "Tensor_Type": Tensor_Type.Tensor,
    "Parameter": OrderedDict([
        ("name", Parameter_Type.Name),
        ("input_Tensor_Names", Parameter_Type.Tensor),
        ("tensor_Func", tf.layers.dense),
        ("units", Parameter_Type.Positive_int),
        ("use_bias", Parameter_Type.Bool),
        ("reuse_Tensor_Name", Parameter_Type.Process_and_Tensor_with_None)
        ]),
    "Required": ["name", "input_Tensor_Names", "tensor_Func", "units"],
    "Description": ""
    }
```

When you look at the code, you will see that a total of five pieces of information must be entered for each function. 1) Func_Type: This indicates which category the function is contained in. See the 'Func_Type' class in 'HNet2/GUI/GUI_Enum.py' for the available function categories. 2) Tensor_Type: This indicates which tensor the function generates. For the Tensor type, see the 'Tensor_Type' class in 'HNet2/HNet_Enum.py'. 3) Parameter: This sets which parameters are input when the function is actually processed. Since the required parameters depend on the tensor type and function, you need to define them clearly. Please refer to 'HNet2/Manager/Process.py' to check the parameters required for each type of Tensor. In addition, since different parameters are required for each parameter types such as 'name' is 'letter' or 'unit' is 'int', you need to set it as parameter type in advance. For parameter types, see the 'Parameter_Type' class in 'HNet2/GUI/GUI_Enum.py'. 4) Required: Set the parameter that must be input. Parameters set to Required are not entered in the process unless they are assigned. 5) Description: When the function is selected, a description of the function appears in the upper right part of the process window. This value does not affect the function of the function.

If you want to create and add your own functions in addition to the built-in functions included in

tensorflow, refer to the files in the folder 'HNet2/Compatible_Functions' as an example.

In addition, HNet 2.0 does not consider batch size changes. It is recommended that you do not use HNet 2.0 if the function covers batch size. This feature is intended for users who are familiar with python and tensorflow.

C. Shortcut add

The '.py' files in the 'HNet2/GUI/Process/Shortcut/' path are the files that make up the shortcut. You can add a shortcut by adding a new script to this path. Shortcut script requires three pieces of information. 1) description: This contains a short description of the shortcut. This value does not affect the execution of the shortcut. 2) parameter_Dict: This contains variables that determine what information to input as variables in the GUI environment. The keys in the dict can be used as variables in the a script using the same key. 3) Run_Shortcut: A python method that contains a script that handles the process input of HNet 2.0. You can get a script that can be used for Run_Shortcut in two ways. One is to use the script save function in the same tab as the graph when the user configures a process. If you press the Script save button, 'Process_Script.txt' is created in the model folder under the Workspace folder and you can use it. The other is to use the process part in 'Prerequisite.txt' which is created when all settings are finished. You can configure a shortcut by putting the script that you get from these two files into Run_Shortcut and changing the static value to a parameter which is a variable configured in parameter_Dict. Please refer to the files in the path 'HNet2/GUI/Process/Shortcut/' for specific examples.

# 4. Appendix

## A. Function description

In description, single letter 'B' is a batch symbol.

### i. Placeholder

Add a placeholder within the process. Plaecholder accepts three parameters: name, dtype, shape. Name and dtype parameters are required.

1) Name: An identifier within the process. This value does not affect learning.

2) Dtype: Determines the type of placeholder. You can enter one of three types: float32, int32, bool.

3) shape: Determines the shape of the placeholder. This parameter can contain a single number or a list of numbers separated by ','. Please make sure that the pattern matches the pattern you want to insert in the future. If no value is assigned, it is assumed that scalar will be entered. ※Unstructured type currently does not support user's setting.

### ii. Forward

Forward is a collection of functions where the calculation of the tensor takes place using the weight.

**Layer dense**

Layer dense adds a one-dimensional vector tensor of the specified size by matrix multiplying the input one-dimensional vector and a weight matrix. If the input tensor is n-dimensional, the last n-dimensional values are treated as a vector and the output is also an n-dimensional tensor with the last dimension specified (e.g. [B,3,5,7] -dense-> [B,3,5, 10]). The parameter list is as follows.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) units: Determines the size of the output vector. This value is required.

4) use_bias: Decides whether to add a bias.

5) reuse_Tensor_Name: Import weight from another layer dense. In this case, the weight is shared with another layer.

**Conv1d**

Apply filters to input 3d data and convolute to other 3d data.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Names: Determines from which tensor the calculation is to proceed. The shape of the input tensor must be 3D in the form of [B, step, channel]. This value is required.

3) filters: Decide how many filters to use for the operation. This value is the same as the channel number of the tensor to be output. This value is required.

4) kernel_size: Determines the size of the filter used in the operation. This value is required.

5) strides: Determines the step size of filter movement at the time of convolution.

6) padding: Determines whether the step size after convolution is equal to the entered tensor by padding.

7) use_bias: Decides whether to add a bias.

5) reuse_Tensor_Name: Import weight from another conv1d. In this case, the weight is shared with another layer.

**Conv2d**

Apply the filter to the input 4d [B, height, width, channel] data and convolute it to other 4d data.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Names: Determines from which tensor the calculation is to proceed. The shape of the input tensor must be 3D in the form of [B, height, width, channel]. This value is required.

3) filters: Decide how many filters to use for the operation. This value is the same as the channel number of the tensor to be output. This value is required.

4) kernel_size: Determines the size of the filter used in the operation. This value is required.

5) strides: Determines the step size of filter movement at the time of convolution.

6) padding: Determines whether the step size after convolution is equal to the entered tensor by padding.

7) use_bias: Decides whether to add a bias.

5) reuse_Tensor_Name: Import weight from another conv2d. In this case, the weight is shared with another layer.

**Conv2d_Transpose**

Apply the filter to the input 4d [B, height, width, channel] data and transposed convolute it to other 4d data.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Names: Determines from which tensor the calculation is to proceed. The shape of the input tensor must be 3D in the form of [B, height, width, channel]. This value is required.

3) filters: Decide how many filters to use for the operation. This value is the same as the channel number of the tensor to be output. This value is required.

4) kernel_size: Determines the size of the filter used in the operation. This value is required.

5) strides: Determines the step size of filter movement at the time of convolution.

6) padding: Determines whether the step size after convolution is equal to the entered tensor by padding.

7) use_bias: Decides whether to add a bias.

5) reuse_Tensor_Name: Import weight from another conv2d transpose. In this case, the weight is shared with another layer.

iii. RNN

Recurrent neural network (RNN) is a collection of functions for which the calculation of tensors takes into account time information.

**RNN LSTM**

Perform RNN operation using LSTM cell.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. The shape of the input tensor must be 3D in the form [B, time, vector]. This value is required.

3) state_Reset: If True, the initial state at the beginning of the training is the last state of the previous learning. If False, initial state is used where all values are zero. Test does not apply the last state of previous learning (always 0 state).

4) num_units: Determines the size of the output vector. This value is required.

5) reuse_Tensor_Name: Import weight from another RNN LSTM. In this case, the weight is shared with another layer.

**RNN GRU**

Perform RNN operation using GRU cell.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. The shape of the input tensor must be 3D in the form [B, time, vector]. This value is required.

3) state_Reset: If True, the initial state at the beginning of the training is the last state of the previous learning. If False, initial state is used where all values are zero. Test does not apply the last state of previous learning (always 0 state).

4) num_units: Determines the size of the output vector. This value is required.

5) reuse_Tensor_Name: Import weight from another RNN GRU. In this case, the weight is shared with another layer.

**RNN Basic**

Perform RNN operation using Basic cell (back propagation through time).

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. The shape of the input tensor must be 3D in the form [B, time, vector]. This value is required.

3) state_Reset: If True, the initial state at the beginning of the training is the last state of the previous learning. If False, initial state is used where all values are zero. Test does not apply the last state of previous learning (always 0 state).

4) num_units: Determines the size of the output vector. This value is required.

5) reuse_Tensor_Name: Import weight from another RNN Basic. In this case, the weight is shared with another layer.

**RNN Feedback**

Performs RNN operation that receives both basic cell and projection as state. (Elman + Jordan network). This cell outputs both hidden and projection. The size of the output vector is num_hidden_units + num_proejction_units. The hidden vector is applied with tanh and the projection vector is non scaled logit. It is recommended to divide the output tensor by split.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. The shape of the input tensor must be 3D in the form [B, time, vector]. This value is required.

3) state_Reset: If True, the initial state at the beginning of the training is the last state of the previous learning. If False, initial state is used where all values are zero. Test does not apply the last state of previous learning (always 0 state).

4) num_hidden_units: Determines the size of hidden part of output vector. This value is required.

5) num_projection_units: Determines the size of projection part of output vector. This value is required.

6) projection_state_activation: Determines the activation function to apply to the projection within the RNN operation. This value is required. ※ This value does not apply to the projection vector.

7) reuse_Tensor_Name: Import weight from another RNN Feedback. In this case, the weight is shared with another layer.

### iv. Activation Calc.

A collection of functions that transform a tensor into a normalized activation value with a range of values.

**Sigmoid**

Apply the expression y = 1 / (1+exp(-x)) to the tensor to convert it to a value within the range of 0 and 1.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

**Softmax**

Apply y = exp(x)/sum(exp(logits)) to the tensor to convert the vector to a probability of the total sum of 1. Logits is the entire vector entered. If more than two dimensions, it applies to the last dimension.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

**Tanh**

Apply a hyperbolic tangent(tanh) to the tensor to convert it to a value between -1 and 1.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

**ReLU**

Apply y = max(x,0) to the tensor to convert all values below zero to zero.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

**Leaky_ReLU**

Apply y = max(x,0) + min(0,alpha*x) to the tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) alpha: Sets the argument alpha to be used for the function. If not set, 0.2 is assigned.

**Softplus**

Apply y = log(exp(x)+1) to the tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

v. Reshape

A collection of functions that transform the shape of a tensor without changing the value.

**Concat**

The input tensors are combined based on the specified dimension. If it is more than 3D, the dimensions of the remaining dimensions except for the specified dimension must be the same (e.g. [B, 5, 32], [B, 5, 17] -> [B, 5, 49]).

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensors the calculation is to proceed. This value is required.

3) axis: Specifies the dimension to join. This value is required.

**Split**

Divides input tensor by specified dimension. Tensors are created for the count of numbers assigned to num_or_size_splits.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) num_or_size_splits: Determines the size of each divided tensor. The sum of the numbers in the whole must be equal to the size of the dimension before the split. This value is required.

3) axis: Specifies the dimension to split. This value is required.

**Expand Dimds**

Inserts the dimension specified in the input tensor. (e.g., [B,3,5,7] -expand dim axis2-> [B,3,1,5,7])

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) axis: Specifies the dimension to be inserted. This value is required.

**Tile**

Creates a tensor that is repeated / copied the specified number of times for each dimension.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) multiples: Specifies the number of iterations for each dimension. The count of this value must match the number of dimensions of the tensor to be entered. This value is required.

**Reshape**

Overrides the shape of the input tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) shape: Specifies the shape to deform. The sum of the total values must be equal to the sum of the dimensions of the total dimensions of the input tensor. This value is required.

**Flatten**

Removes all dimensions of the input tensor and converts it to a batch of one-dimensional vector. (e.g. [B, 5, 7] -> [B, 35])

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

**Suqeeze**

Removes the specified dimensions from the input tensor. The size of specified dimension must be 1.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) axis: Specifies the dimensions to remove. Specified dimensions must be 1 in size. This value is required.

vi. Fundamental math

A collection of functions related to basic arithmetic operations.

**Add_n**

Add tensors to input. All inserted tensors must have the same shape.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensors the calculation is to proceed. This value is required.

**Add**

Add two tensors to receive. Both tensors must have the same shape.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensors the calculation is to proceed. This value is required.

**Subtract**

Subtract the second tensor from the first tensor. Both tensors must have the same shape.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensors the calculation is to proceed. This value is required.

**Multiply**

The two input tensors are elementwise-multiply. Both tensors must have the same shape.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensors the calculation is to proceed. This value is required.

**Divide**

Elementwise-divide the second tensor in the first tensor. Both tensors must have the same shape.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensors the calculation is to proceed. This value is required.

**Mean**

Calculate the mean value of the input tensor with respect to the specified axis.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) axis: Sets the axis on which to obtain the average.

4) keepdims: After determining the average, decide whether to keep the dimension of the tensor. If it is False, it becomes 'number of axis input - 1' dimension.

**Sum**

Calculate the sum based on the specified axis of the input tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) axis: Sets the axis on which to obtain the sum.

4) keepdims: After determining the sum, decide whether to keep the dimension of the tensor. If it is False, it becomes 'number of axis input - 1' dimension.

**Max**

Calculate the max value based on the specified axis of the input tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) axis: Sets the axis on which to obtain the max value.

4) keepdims: After determining the max value, decide whether to keep the dimension of the tensor. If it is False, it becomes 'number of axis input - 1' dimension.

**Min**

Calculate the min value based on the specified axis of the input tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) axis: Sets the axis on which to obtain the min value.

4) keepdims: After determining the min value, decide whether to keep the dimension of the tensor. If it is False, it becomes 'number of axis input - 1' dimension.

vii. Loss calc.

A collection of functions used to generate the losses used in learning through comparison of two tensors.

**Absolute_Difference_Loss**

The absolute value of the difference between label and prediction (L1 loss). Label and prediction must have the same shape.

1) Name: An identifier within the process. This value does not affect learning.

2) label_Tensor_Name: Observation value that aims at learning. This value is required.

3) prediction_Tensor_Name: The predicted value calculated by the model. This value is required.

4) weights: Assign some weight to loss. 1 if not set.

**Mean_Squared_Error_Loss**

The sum of the squares of the label and prediction squares (L2 loss). Label and prediction must have the same shape.

1) Name: An identifier within the process. This value does not affect learning.

2) label_Tensor_Name: Observation value that aims at learning. This value is required.

3) prediction_Tensor_Name: The predicted value calculated by the model. This value is required.

4) weights: Assign some weight to loss. 1 if not set.

**Sigmoid_Loss**

Calculate sigmoid cross entropy loss of label and logit. Label and logit must have the same shape. ※Logit is the unscaled logit before applying sigmoid.

1) Name: An identifier within the process. This value does not affect learning.

2) label_Tensor_Name: Observation value that aims at learning. This value is required.

3) logit_Tensor_Name: The unscaled logit produced by the model. This value is required.

4) weights: Assign some weight to loss. 1 if not set.

**Softmax_Loss**

Calculates Softmax cross entropy loss of label and logit. Label and logit must have the same shape. ※Logit is an unscaled logit before applying softmax.

1) Name: An identifier within the process. This value does not affect learning.

2) label_Tensor_Name: Observation value that aims at learning. This value is required.

3) logit_Tensor_Name: The unscaled logit produced by the model. This value is required.

4) weights: Assign some weight to loss. 1 if not set.

**Sparse_Softmax_Loss**

Calculates softmax cross entropy loss of Label and logit. Label should be one dimension less than logit, and it must be an int32 dtype containing one of the indexes for the last dimension of logit (e.g. if logit shape = [B, 32, 50], label shape = [B, 32] and each value is from 0 to 49). ※Logit is an unscaled logit before applying softmax.

1) Name: An identifier within the process. This value does not affect learning.

2) label_Tensor_Name: Observation value that aims at learning. This value is required.

3) logit_Tensor_Name: The unscaled logit produced by the model. This value is required.

4) weights: Assign some weight to loss. 1 if not set.

viii. Optimizer

The optimizer is an operator that lets process add all the losses in the process and reduce the allocated loss to learn the model. The optimizer has a fixed name and only one can be created per process.

**ADAM_Optimizer**

Create an optimizer using the ADAM algorithm.

1) initial_learning_rate: Determines the initial learning rate. If not set, 0.001 is assigned.

2) beta1: Set up beta1. If not set, 0.9 is assigned.

3) beta2: Set up beta2. If not set, 0.999 is assigned.

4) epsilon: Set up epsilon. If not set, 1e-8 is assigned.

5) decay_method: Sets the decay method of the learning rate. If not set, no decay.

6) decay_steps: One of the decay criteria when decay_method is exponential. When learning reaches each decay_step, learning rate decays by decay_rate than before. If not set, it is 0.5. If not exponential, this value is ignored.

7) decay_rate: One of the decay criteria when decay_method is exponential. When learning reaches each decay_step, learning rate decays by decay_rate than before. If not set, it is 0.5. If not exponential, this value is ignored.

8) warmup_steps: The decay criterion when decay_method is Noam. The learning rate gradually increases until the learning reaches warmup_steps and becomes initial_learning_rate at warmup_steps. After that, it gradually decays. If not Noam, this value is ignored.

9) applied_variable: A list of weights for which updates are in progress during training. The excluded weights are fixed in the process without updating.

**GD_Optimizer**

Create an optimizer using the gradient descent algorithm.

1) initial_learning_rate: Determines the initial learning rate. If not set, 0.001 is assigned.

2) decay_method: Sets the decay method of the learning rate. If not set, no decay.

3) decay_steps: One of the decay criteria when decay_method is exponential. When learning reaches each decay_step, learning rate decays by decay_rate than before. If not set, it is 0.5. If not exponential,

this value is ignored.

4) decay_rate: One of the decay criteria when decay_method is exponential. When learning reaches each decay_step, learning rate decays by decay_rate than before. If not set, it is 0.5. If not exponential, this value is ignored.

5) warmup_steps: The decay criterion when decay_method is Noam. The learning rate gradually increases until the learning reaches warmup_steps and becomes initial_learning_rate at warmup_steps. After that, it gradually decays. If not Noam, this value is ignored.

6) applied_variable: A list of weights for which updates are in progress during training. The excluded weights are fixed in the process without updating.

### ix. Test Calc.

It is a collection of functions that calculate loss values that are mainly used for analysis. Unlike a loss calc. category, it can be assigned to a get tensor during a test and is calculated for each pattern.

**Mean_Squared_error**

Generates a mean squared error tensor of two input tensors.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determine from which tensors to proceed the calculation. This value is required.

**Cross_Entropy**

Creates a cross entropy tensor of the input two tensors. The first tensor is assigned to label and the second tensor is assigned to prediction.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determine from which tensors to proceed the calculation. This value is required.

**Cosine_Similarity**

Creates a cosine similarity tensor of two input tensors.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determine from which tensors to proceed the calculation. This value is required.

**Euclidean_Distance**

Creates a Euclidean distance tensor of two input tensors.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determine from which tensors to proceed the calculation. This value is required.

**Multi_Test**

It generates the mean squared error, cross entropy, cosine similarity, and Euclidean distance tensor of each input tensor. For a cross entropy calculation, the first tensor is assigned to label and the second tensor is assigned to prediction.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determine from which tensors to proceed the calculation. This value is required.

**Arg_Max**

Creates a tensor that displays the index with the highest value based on the last dimension of the input tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

**Semantic_Stress**

Generates the semantic stress of input tensor. Please refer to the following article on the specific definition of semantic stress:

Plaut, D. C. (1997). Structure and function in the lexical system: Insights from distributed models of word reading and lexical decision. *Language and cognitive processes, 12*(5-6), 765-806.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

x. Create

A collection of create functions that automatically generate tensors based on a given shape and function, without inserting any other tensors.

**Ones**

Generates a tensor with all values equal to 1.

1) Name: An identifier within the process. This value does not affect learning.

2) shape: Sets the shape of the tensor to be created. This value is required.

**Zeors**

Generates a tensor with all values equal to 0.

1) Name: An identifier within the process. This value does not affect learning.

2) shape: Sets the shape of the tensor to be created. This value is required.

**Random_Normal**

Generate tensor according to normal distribution.

1) Name: An identifier within the process. This value does not affect learning.

2) shape: Sets the shape of the tensor to be created. This value is required.

3) mean: Sets the mean of the normal distribution. If not set, 0 is assigned.

4) stddev: Sets the standard deviation of the normal distribution. If not set, 1 is assigned.

**Random_Uniform**

Generate tensor according to uniform distribution.

1) Name: An identifier within the process. This value does not affect learning.

2) shape: Sets the shape of the tensor to be created. This value is required.

3) minval: Sets the minimum value of the tensor to be created. This value is required.

4) maxval: Sets the maximum value of the tensor to be generated. This value is required.

xi. Other

A collection of other functions not included in the above 10 categories.

**Clip**

It clips the value outside the given range of input tensor. Values lower than min are fixed at min and values higher than max are fixed at max.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) clip_value_min: Determines the min value. This value is required.

4) clip_value_max: Determines the max value. This value is required.

**Dropout**

Make each cell of input tensor zero at the specified probability rate. At the same time, the tensor is multiplied by 1 / rate. In Test, this function is not applied (output is same to input tensor)

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) rate: Sets the probability that dropout will be applied. If not set, 0.5 is assigned.

**Indexing**

Extracts only the specified index of the specified axis. In case of axis 2 and index 3 in the tensor of Shape [B,3,5,7], a tensor of shape [B,3,7] of [:,:, 3,:] of the input tensor is generated.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) axis: Decide on which axis to extract. This value is required.

3) index: Decide which index of the set axis to extract. This value is required.

**Embedding**

Generates an n+1 dimensional float tensor for an n dimensional int tensor that receives input. Within a single model, the same int value will always be the same float vector.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. Must be an int tensor. This value is required.

3) id_size: Sets the value range of the int tensor to be input. This value is required.

4) embedding_size: Sets the size of the float vector to be created. This value is required.

5) reuse_Tensor_Name: Import embedding variable from another embedding. In this case, both embeddings produce the same float vector for the same int value.

**Noise_Normal**

Add Gaussian noise to the input tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) mean: Sets the mean of the Gaussian noise. If not set, 0 is assigned.

4) stddev: Sets the standard deviation of the Gaussian noise. If not set, 1 is assigned.

**Noise_Uniform**

Add uniform noise to the input tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) minval: Sets the minimum value of noise. This value is required.

4) maxval: Sets the maximum value of noise. This value is required.

**Batch_Normalization**

Generate a tensor that performs batch normalization on the last dimension of the input tensor. For more details on batch normalization see the document on 'tf.layers.batch_normalization' by tensorflow and the following paper:

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv* preprint arXiv:1502.03167.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Name: Determines from which tensor the calculation is to proceed. This value is required.

3) momentum: Sets the momentum. If not set, 0.99 is assigned.

4) epsilon: Set epsilon. If not set, 0.001 is assigned.

5) reuse_Tensor_Name: Import weight from another batch normalization. In this case, the weight is shared with another layer.

**Max_Pooling1d**

Max pooling is performed on the input 3d tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Names: Determines from which tensor the calculation is to proceed. The shape of the input tensor must be 3D in the form of [B, step, channel]. This value is required.

3) pool_size: Determines the size of the pooling window. This value is required.

5) strides: Determines the size of the movement of the pooling window. This value is required.

6) padding: After pooling through padding, determine the shape of the tensor is equal to the entered tensor.

**Max_Pooling2d**

Max pooling is performed on the input 4d tensor.

1) Name: An identifier within the process. This value does not affect learning.

2) input_Tensor_Names: Determines from which tensor the calculation is to proceed. The shape of the input tensor must be 4D in the form of [B, height, width, channel]. This value is required.

3) pool_size: Determines the size of the pooling window. This value is required.

5) strides: Determines the size of the movement of the pooling window. This value is required.

6) padding: After pooling through padding, determine the shape of the tensor is equal to the entered tensor.