root

Go Up

Name	GNN
Version	2.0
Description	Generalized Neural Network Bundle
License	SeeLICENSE.TXT
Copyright	Copyright (C) 2020 HPCC SystemsÂő
Authors	HPCCSystems
DependsOn	ML_Core
Platform	7.4.0

Table of Contents

Generalized Neural Network (GNN)

July 21, 2022

This bundle provides a generalized ECL interface to Keras over Tensorflow.

It provides Keras / Tensorflow operations parallelized over an HPCC cluster.

Tensorflow Models are created transparently on each HPCC node, and training, evaluation and predictions are done in a distributed fashion across the HPCC cluster.

GNN is designed to handle any type of Neural Network model that can be built using Keras. This includes Classical (Dense) Neural Networks as well as Convolutional and Recursive Networks (such as LSTM), or any combination of the above.

GNN currently supports both Tensorflow 1.x and Tensorflow 2.x versions. It also supports the use of GPUs in conjunction with Tensorflow, with certain restrictions in the supported topology. Specifically: - All servers in a cluster must have the same GPU configuration - The number of HPCC nodes must equal the number of GPUs.

One GPU will be allocated to each HPCC node. See GNNI module documentation for details.

The Module GNNI defines the ECL interface to Keras. It supports any Keras model (Functional or Sequential), and allows models with multiple inputs and outputs.

Input to GNNI is in the form of Tensor records. The built-in Tensor module defines these Tensors and provides functions for operating on them. Tensors provide an efficient N-dimensional representation for data into and out of GNNI.

0.1 INSTALLATION

Python3 and Tensorflow must be installed on each server running HPCC Systems Platform software. Tensorflow should be installed using su so that all users can see it, and must be installed using the same version of Python3 as is embedded in the HPCC Systems platform. The file Test/SetupTest.ecl can be used to test the environment. It will verify that Python3 and Tensorflow are correctly installed on each Thor node. To Install GNN, run:

```
ecl bundle install https://github.com/hpcc-systems/GNN.git
```

0.2 EXAMPLES

The files Test/ClassicTest.ecl and ClassificationTest.ecl show annotated examples of using GNN to create a simple Classical Neural Networks using the Keras Sequential model.

The file Test/FuncModelTest.ecl shows an example of building a classical regression / classification network with multiple inputs and outputs using the Keras Functional model.

The folder Test/HARTests contains tests that show how to create more sophisticated Convolutional and Recurrent networks.

0.3 OTHER DOCUMENTATION

Programmer Documentation is available at: HPCC Machine Learning Library A tutorial on installing and running GNN is available at: Generalized Neural Network Blog

GNNI

Go Up

IMPORTS

python3 | _versions.GNN.V2_0.GNN.Internal | _versions.GNN.V2_0.GNN.Types | _versions.GNN.V2_0.GNN.Internal.Types | _versions.GNN.V2_0.GNN.Internal.Keras | _versions.GNN.V2_0.GNN.Tensor | std.system.Thorlib | std.system.Log | _versions.ML_Core.V3_2_2.ML_Core.Types |

DESCRIPTIONS

GNNI GNNI

	GNNI

Generalized Neural Network Interface

Provides a generalized ECL interface to Keras over Tensorflow. It supports the Keras Functional API as well as the Sequential API.

The Functional style allows models to be defined as a Directed Acyclic Graph (DAG), allowing branching and merging among the layers. It is required for models that have multiple inputs or outputs. For an annotated example using the Functional model with multiple inputs and outputs, see Test/FuncModelTest.ecl

The Sequential stle is a bit simpler, but requires a strict sequence of layers, each one feeding into the next. For an annotated example using the Sequential model, see Test/ClassicTest.ecl

THEORY OF OPERATION

A Keras / TF model is built on each HPCC node and training data is distributed among the nodes. Distributed Synchronous Batch Gradient Descent is performed across nodes, synchronizing weights periodically based on the 'batchSize' parameter. Each function performs its work in a distributed manner, using the built-in parallelization of HPCC. Weights are synchronized across nodes after 'batchSize' records are processed on each node.

PROGRAM FLOW

The flow of a program using this interface is as follows:

- GetSession() – Initialized Keras / TF and returns a session token. This must be called before any other operations.
- Define the model, either as a Sequential model or a Functional model:
  - DefineModel(...) – Construct a Keras Sequential model by providing a list of Python statements, one to construct each layer of the neural network as well as an optional compile definition statement.
  - DefineFuncModel(...) – Construct a Keras Functional Model. This allows construction of complex models that cannot be expressed as a sequential set of layers. These include models with multiple inputs or outputs, or models that use divergence or convergence among different layers.
- CompileMod(...) – (Optional) Pass the Keras model compilation statement and perform it on the model. This is only required if the compile definition was not provided in DefineModel (above).
- Fit(...) – Trains the model across the nodes of the cluster, based on provided training data.
- EvaluateMod(...) – (Optional) Evaluate the model against a set of data (typically your validation or test data) and return the loss and any other metrics that were defined by your compileDef.
- Predict(...) – (Optional) Use the model to predict the output based on a provided set of input (X) data.
- GetWeights(...) – (Optional) Return the trained weights of all layers of the model.

USE OF TENSORS

GNNI uses Tensors (effectively N-dimensional array representations) to provide data and weights in and out of Keras / TF. See the included Tensor module for details. These Tensor datasets provide an efficient way to store, distribute, and process N-Dimensional data. The data is packed into 'slices', which can be either sparse or dense, for efficiency and scalability purposes.

Tensors can be used to convey record-oriented information such as training data as well as block oriented data like weights. Both can be N-dimensional. For record-oriented data, the first shape component is set to 0 (unspecified) indicating that it can hold an arbitrary set of records.

For models with multiple inputs or outputs, Tensor Lists are used (see Tensor.ecl for details), with one Tensor per input or output.

USE OF NumericField

GNNI also provides a set of interfaces which take in and emit data as 2-dimensional NumericField datasets (see ML_Core.Types.NumericField). This is purely for convenience for applications that don't require the N-Dimensional capabilities of the Tensor format. Internally, these functions translate the NumericField format into Tensors, and convert the output from Tensors to NumericField. These functions have the same names as the tensor functions, but with NF appended to the name (e.g. FitNF(...), PredictNF(...)). Weights are always returned as Tensors, so there is no NF version of GetWeights(...).

SEQUENCING OF OPERATIONS The Keras / Tensorflow operations take place under the hood from an ECL perspective. Therefore normal ECL data dependencies are not sufficient to ensure proper sequencing. For this reason, GNNI uses a series of tokens passed from one call to the next to ensure the correct order of command execution. For example:

- GetSession() returns a session-token which must be passed to DefineModel()
- Subsequent calls return a model-token which must be passed to the following call. Each call creates a new model token which becomes the input to the next call in sequence.
- It is critical that this token passing is chained, or calls may occur out of order. For example, Fit() could be called before DefineModel(), which would not produce good results.

MULTIPLE MODEL SUPPORT

GNNI supports multiple Keras models within the same work unit. Multiple models are created by using multiple calls to DefineModel() using the same sessionId. The returned modelIds are used in subsequent calls to discriminate between the models. See Test/MultiModel.ecl for an example.

WORKING WITH GPUs

GNN can be used with hardware accelleration (i.e. GPUs) provided certain configuration rules are followed:

- All servers have the same configuration in terms of number of GPUs per server
- The number of Thor nodes = GPUs Per Server * Number of Servers
- The number of GPUs per Server is passed to GetSession()

One GPU will be allocated to each Thor node.

PERFORMANCE CONSIDERATIONS

Performance of GNN is a complex topic, and it is very difficult to make recommendations as to the optimal configuration for training a given problem. Below are some observations regarding performance that may be useful in understanding the tradeoffs.

- Many factors influence the training performance, and these are interrelated in complex ways:
  - The complexity of the network design.
  - The amount of training data (number of records, and record size).
  - The complexity of the relationship(s) being modeled.
  - The Learning Rate specified in the Compile line for the selected optimizer.
  - The number of nodes in the HPCC cluster.
  - GNN training parameters (see Fit Below). Specifically: batchSize, learningRateReduction, batchSizeReduction, and localBatchSize.
  - The random starting point of the neural network weights.
- It important to separate the speed of running Epochs from the final loss achieved. Using large batchSize (i.e. the number of records to process on each node before synchronizing weights), will process epochs faster, but with less loss reduction on each epoch.
- Once the optimal training loss is determined (e.g. by running multiple epochs until consistent gain no longer occurs), it is recommended to use the "trainToLoss" parameter to abort training when a given loss level is reached.
- Training parameters usually need to be adjusted when running on different sized clusters. The number of records processed on each batch is nNodes * batchSize. For a larger cluster, it may be necessary to reduce the batchSize to avoid blurring of the training. An alternative is to use the batchSizeReduction parameter, which will cause the batchSize to automatically reduce as the training progresses. In some cases, reducing the learning rate or using the learningRateReduction parameter can also compensate for larger batch sizes.
- If the network doesn't converge (e.g. loss goes to infinity, or loss is unstable across epochs), it generally means that the learning rate is set too high, or the batchSize is too large.

Children

1. GetSession : Initialize Keras on all nodes and return a "session" token to be used on the next call to GNNI
2. DefineModel : Define a Keras / Tensorflow model using Keras sytax
3. DefineFuncModel : DefineFuncModel(. . .) – Construct a Keras Functional Model
4. ToJSON : Return a JSON representation of the Keras model
5. FromJSON : Create a Keras model from previously saved JSON
6. CompileMod : Compile a previously defined Keras model
7. GetWeights : Return the weights currently associated with the model
8. SetWeights : Set the weights of the model from a list of Tensors
9. GetLoss : Get the accumulated average loss for the latest epoch
10. Fit : Train the model using synchronous batch distributed gradient descent

11. EvaluateMod : Determine the loss and other metrics in order to evaluate the model
12. Predict : Predict the results using the trained model
13. Shutdown
14. FitNF : Fit a model with 2 dimensional input and output using NumericField matrices
15. EvaluateNF : Evaluate a model with 2 dimensional input and output using NumericField matrices
16. PredictNF : Predict the results for a model with 2 dimensional input and output using NumericField matrixes for input and output

---

GETSESSION GetSession

GNNI \

UNSIGNED4	**GetSession**
(UNSIGNED GPUsPerServer = 0)	

Initialize Keras on all nodes and return a "session" token to be used on the next call to GNNI.

This function must be called before any other use of GNNI.

PARAMETER **GPUsPerServer** ||| UNSIGNED8 — – The number of GPUs per Server. This is only used when working with GPUs. See "WORKING WITH GPUs section of module documentation. Defaults to 0 when GPUs not used.

RETURN **UNSIGNED4** — A session token (UNSIGNED4) to identify this session.

DEFINEMODEL DefineModel

GNNI \

UNSIGNED4	**DefineModel**
(UNSIGNED4 sess, SET OF STRING ldef, STRING cdef = ")	

Define a Keras / Tensorflow model using Keras sytax. Optionally also provide a "compile" line with the compilation parameters for the model.

If no compile line is provided (cdef), then the compile specification can be provided in a subsequent call to CompileMod (below).

The symbols "tf" (for tensorflow) and "layers" (for tf.keras.layers) are available for use within the definition strings. See GNN/Test/ClassicTest.ecl for an annotated example.

PARAMETER **sess** ||| UNSIGNED4 — The session token from a previous call to GetSesion().

PARAMETER **ldef** ||| SET (STRING) — A set of python strings as would be passed to Keras model.add(). Each string defines one layer of the model.

PARAMETER **cdef** ||| STRING — (optional) A python string as would be passed to Keras model.compile(...). This line should begin with "compile". Model is implicit here.

RETURN **UNSIGNED4** — A model token to be used in subsequent GNNI calls.

DEFINEFUNCMODEL DefineFuncModel

GNNI \

UNSIGNED4	**DefineFuncModel**
`(UNSIGNED sess, DATASET(FuncLayerDef) lDefs, SET OF STRING inputs, SET OF STRING outputs, STRING cdef = '')`	

DefineFuncModel(...) – Construct a Keras Functional Model. This allows construction of complex models that cannot be expressed as a sequential set of layers. These include models with multiple inputs or outputs, or models that use divergence or convergence among different layers.

Layers are connected together using the layerName and predecessor fields of the FuncLayerDef. The inputs of a layer are connected to the predecessor layers in the order specified by the set of names in the predecessor field.

The inputs and outputs parameter specifies the names of the layers that form the input and output of the model.

This is similar to the Keras Functional API, except that the entire model is defined in one call rather than assembled piecemeal as in the Functional API. The same rules apply here as for the Keras Functional API, and there should be a simple translation of any program using the Functional API.

For models with multiple inputs, input is specified as a list of tensors (see Tensor.ecl).

For models with multiple outputs, output will be a list of tensors.

PARAMETER **sess** ||| UNSIGNED8 — The session token from a previous call to GetSesion().

PARAMETER **lDefs** ||| TABLE (FuncLayerDef) — A series of layer definitions using the Types.FuncLayerDef format.

PARAMETER **inputs** ||| SET (STRING) — A list of the names of the layers that represent the inputs to the model.

PARAMETER **outputs** ||| SET (STRING) — A list of the names of the layers that represent the outputs of the model.

PARAMETER **cdef** ||| STRING — (optional) A python string as would be passed to Keras model.compile(...). This line should begin with "compile". Model is implicit here.

RETURN **UNSIGNED4** —

SEE Types.FuncLayerDef

SEE Test.FuncModelTest.ecl

TOJSON ToJSON

GNNI \

STRING	**ToJSON**
(UNSIGNED4 mod)	

Return a JSON representation of the Keras model.

PARAMETER **mod** ||| UNSIGNED4 — The model token as previously returned from DefineModel(...) above.

RETURN **STRING** — A JSON string representing the model definition.

FROMJSON FromJSON

GNNI \

UNSIGNED4	**FromJSON**
(UNSIGNED4 sess, STRING json)	

Create a Keras model from previously saved JSON.

Note that this call defines the model, but does not restore the compile definition or the trained model weights. CompileMod(. . .) should be called after this to define the model compilation parameters.

PARAMETER **sess** ||| UNSIGNED4 — A session token previously returned from GetSession(..).

PARAMETER **json** ||| STRING — A JSON string defining the model as previously returned from ToJSON(. . .).

RETURN **UNSIGNED4** — A model token to be used in subsequent GNNI calls.

COMPILEMOD CompileMod

GNNI \

UNSIGNED4	**CompileMod**
(UNSIGNED model, STRING compileStr)	

Compile a previously defined Keras model.

This is an optional call that can be used if you omit the compileDef parameter during DefineModel(. . .) or if the model was created via FromJSON(. . .).

The compile string uses the same python syntax as using Keras' model.compile(. . .). Model is implied in this call, so the line should begin with "compile".

The symbol "tf" (for tensorflow) is available for use within the compile string.

Example:

- "'compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])"'

It is convenient to use the triple single quote(”’) syntax as it allows strings to cross line boundaries, and allows special characters such as single or double quotes without escaping.

There is no need to make this call if the compileDef was provided in the DefineModel(. . .) call.

The returned model token should be used in subsequent calls to GNNI.

PARAMETER **model** ||| UNSIGNED8 — A model token as returned from DefineModel(. . .) or FromJSON(. . .).

PARAMETER **compileStr** ||| STRING — A python formatted string defining the Keras ”compile” call and its parameters.

RETURN **UNSIGNED4** — A new model token that should be used in subsequent GNNI calls.

GETWEIGHTS GetWeights

GNNI \

DATASET(t_Tensor)	**GetWeights**
(UNSIGNED4 model)	

Return the weights currently associated with the model.

The weights are returned as a Tensor List containing the weights for each Keras layer as a separate Tensor.

The weights from a given layer can be extracted by simply filtering on the work-item (wi). The first layer will use wi = 1, and the Nth layer uses wi = N.

This call is typically made after training the model via Fit(. . .), but can also be called before Fit(. . .) to retrieve the initial weights.

PARAMETER **model** ||| UNSIGNED4 — The model token as returned from DefineModel(. . .), CompileMod(. . .), or Fit(. . .).

RETURN **TABLE (t_Tensor)** — A t_Tensor dataset representing the weights as a list of Tensors.

SEE Tensor.t_Tensor

SETWEIGHTS SetWeights

GNNI \

UNSIGNED4	**SetWeights**
(UNSIGNED4 model, DATASET(t_Tensor) weights)	

Set the weights of the model from a list of Tensors.

Typically, the weights to be set were originally obtained using GetWeights(...) above. They must be of the same number and shape as would be returned from GetWeights(...).

These will contain one Tensor for each defined Keras layer. The shape of each tensor is determined by the definition of the layer.

PARAMETER **model** ||| UNSIGNED4 — The model token from the previous step.

PARAMETER **weights** ||| TABLE (t_Tensor) — The Tensor List (DATASET(t_Tensor)) containing the desired weights.

RETURN **UNSIGNED4** — A new model token to be used in subsequent calls.

SEE Tenosr.t_Tensor

GETLOSS GetLoss

GNNI \

REAL	**GetLoss**
(UNSIGNED4 model)	

Get the accumulated average loss for the latest epoch.

This represents the average per sample loss.

PARAMETER **model** ||| UNSIGNED4 — The model token as returned from Fit(...).

RETURN **REAL8** — The average loss.

FIT Fit

GNNI \

UNSIGNED4	Fit

```
(UNSIGNED4 model, DATASET(t_Tensor) x, DATASET(t_Tensor) y, UNSIGNED4
batchSize = 512, UNSIGNED4 numEpochs = 1, REAL trainToLoss = 0, REAL
learningRateReduction = 1.0, REAL batchSizeReduction = 1.0, UNSIGNED4
localBatchSize = 32)
```

Train the model using synchronous batch distributed gradient descent.

The X tensor represents the independent training data and the Y tensor represents the dependent training data.

Both X and Y tensors should be record-oriented tensors, indicated by a first shape component of zero. These must also be distributed (not replicated) tensors. If the model specifies multiple inputs or outputs, then tensor lists should be supplied, using the work-item id (wi) to distinguish the order of the tensors in the tensor list (see Tensor.ecl).

BatchSize defines how many observations are processed on each node before weights are re-synchronized. There is an interaction between the number of nodes in the cluster, the batchSize, and the complexity of the model. A larger batch size will process epochs faster, but the loss reduction may be less per epoch. As the number of nodes is increased, a smaller batchSize may be required. The default batchSize of 512 is a good starting point, but may require tuning to increase performance or improve convergence (i.e. loss reduction). Final loss should be used to assess the fit, rather than number of epochs trained. For example, for a given neural network, a loss of .02 may be the optimal tradeoff between underfit and overfit. In that case the network should be trained to that level, adjusting number of epochs and batchSize to reach that level. Alternatively, the trainToLoss parameter can be used to automatically stop when a given level of loss is achieved. See the top-level module documentation for more insight on Performance Considerations.

PARAMETER **model** ||| UNSIGNED4 — The model token from the previous GNNI call.

PARAMETER **x** ||| TABLE (t_Tensor) — The independent training data tensor or tensor list.

PARAMETER **y** ||| TABLE (t_Tensor) — The dependent training data tensor or tensor list.

PARAMETER **batchSize** ||| UNSIGNED4 — The number of records to process on each node before re-synchronizing weights across nodes.

PARAMETER **numEpochs** ||| UNSIGNED4 — The number of times to iterate over the full training set.

PARAMETER **trainToLoss** ||| REAL8 — Causes training to exit before numEpochs is complete if the trainToLoss is met. Defaults to 0, meaning that all epochs will be run. When using trainToLoss, numEpochs should be set to a high value so that it does not exit before the training goal is met. Note that not all network / training data configurations can be trained to a given loss. The nature of the data and the network configuration limits the minimum achievable loss.

PARAMETER **learningRateReduction** ||| REAL8 — Controls how much the learning rate is reduced as epochs progress. For some networks, training can be improved by gradulally reducing the learning rate. The default value (1.0), maintains the original learning rate across all epochs. A value of .5 would cause the learning rate to be reduced to half the original rate by the final epoch.

PARAMETER **batchSizeReduction** ||| REAL8 — Controls how much the batchSize is reduced as epochs progress. For some networks, training can be improved by gradually reducing the batchSize. The default value (1.0), maintains the original batchSize across all epochs. A value of .25 would cause the learning rate to be reduced to one quarter the original rate by the final epoch.

PARAMETER **localBatchSize** ||| UNSIGNED4 — The batch size to use when calling Keras Fit() on each local machine. The default (32) is recommended for most uses.

RETURN **UNSIGNED4** — A new model token for use with subsequent GNNI calls.

EVALUATEMOD EvaluateMod

GNNI \

DATASET(Types.metrics)	**EvaluateMod**
(UNSIGNED4 model, DATASET(t_Tensor) x, DATASET(t_Tensor) y)	

Determine the loss and other metrics in order to evaluate the model.

Returns a set of metrics including loss and any other metrics that were defined in the compile definition for a set of provided test data.

Both X and Y tensors should be record-oriented tensors, indicated by a first shape component of zero. These must also be distributed (not replicated) tensors.

This is typically used after training the model, using a segregated set of test data, in order to determine the ”out of sample” performance (i.e. performance on data outside of the training set).

PARAMETER **model** ||| UNSIGNED4 — The model token from the previous GNNI call (e.g. Fit).

PARAMETER **x** ||| TABLE (t_Tensor) — The independent test data tensor or tensor list.

PARAMETER **y** ||| TABLE (t_Tensor) — The dependent test data tensor or tensor list.

RETURN **TABLE ({ UNSIGNED4 metricId , STRING metricName , REAL8 value })** — A dataset of metrics indicating the performance of the model.

SEE Types.metrics

PREDICT Predict

GNNI \

DATASET(t_Tensor)	**Predict**
(UNSIGNED4 model, DATASET(t_Tensor) x)	

Predict the results using the trained model.

The X tensor represents the independent (input) data for the neural network and the output is returned as a tensor or tensor list (for multiple output networks). Input and output will be Tensor Lists if there is more than one input or output tensor for the NN.

The X tensor should be a record-oriented tensor, indicated by a first shape component of zero. It must also be distributed (not replicated) tensor.

PARAMETER **model** ||| UNSIGNED4 — A model token as returned from the previous GNNI call (e.g. Fit).

PARAMETER **x** ||| TABLE (t_Tensor) — The independent (i.e. input) data tensor or tensor list.

RETURN **TABLE (t_Tensor)** — The output predicted by the model as a record-oriented tensor or tensor list.

SHUTDOWN Shutdown

GNNI \

UNSIGNED4	**Shutdown**
(UNSIGNED4 model)	

PARAMETER **model** ||| UNSIGNED4 — A model token as returned from a previous GNNI call.

RETURN **UNSIGNED4** — A new model token.

NODOC Shutdown Keras / Tensorflow and free up any allocated memory. This function is not required at this time but is here for future use.

FITNF FitNF

GNNI \

UNSIGNED4	**FitNF**

```
(UNSIGNED4 model, DATASET(NumericField) x, DATASET(NumericField) y,
UNSIGNED4 batchSize = 512, UNSIGNED4 numEpochs = 1, REAL trainToLoss = 0,
REAL learningRateReduction = 1.0, REAL batchSizeReduction = 1.0, UNSIGNED4
localBatchSize = 32)
```

Fit a model with 2 dimensional input and output using NumericField matrices.

This is a NumericField wrapper around the Fit function. See Fit (above) for details.

PARAMETER **model** ||| UNSIGNED4 — The model token from the previous GNNI call.

PARAMETER **x** ||| TABLE (NumericField) — The independent training data.

PARAMETER **y** ||| TABLE (NumericField) — The dependent training data.

PARAMETER **batchSize** ||| UNSIGNED4 — The number of records to process on each node before re-synchronizing weights across nodes.

PARAMETER **numEpochs** ||| UNSIGNED4 — The number of times to iterate over the full training set.

PARAMETER **trainToLoss** ||| REAL8 — Causes training to exit before numEpochs is complete if the trainToLoss is met. Defaults to 0, meaning that all epochs will be run. When using trainToLoss, numEpochs should be set to a high value so that it does not exit before the training goal is met. Note that not all network / training data configurations can be trained to a given loss. The nature of the data and the network configuration limits the minimum achievable loss.

PARAMETER **learningRateReduction** ||| REAL8 — Controls how much the learning rate is reduced as epochs progress. For some networks, training can be improved by gradulally reducing the learning rate. The default value (1.0), maintains the original learning rate across all epochs. A value of .5 would cause the learning rate to be reduced to half the original rate by the final epoch.

PARAMETER **batchSizeReduction** ||| REAL8 — Controls how much the batchSize is reduced as epochs progress. For some networks, training can be improved by gradually reducing the batchSize. The default value (1.0), maintains the original batchSize across all epochs. A value of .25 would cause the learning rate to be reduced to one quarter the original rate by the final epoch.

PARAMETER **localBatchSize** ||| UNSIGNED4 — The batch size to use when calling Keras Fit() on each local machine. The default (32) is recommended for most uses.

RETURN **UNSIGNED4** — A new model token for use with subsequent GNNI calls.

SEE ML_Core.Types.NumericField

EVALUATENF EvaluateNF

GNNI \

DATASET(Types.metrics)	**EvaluateNF**
(UNSIGNED4 model, DATASET(NumericField) x, DATASET(NumericField) y)	

Evaluate a model with 2 dimensional input and output using NumericField matrices.

This is a NumericField wrapper around the EvaluateMod function. See EvaluateMod (above) for details.

PARAMETER **model** ||| UNSIGNED4 — The model token from the previous GNNI call.

PARAMETER **x** ||| TABLE (NumericField) — The independent test data.

PARAMETER **y** ||| TABLE (NumericField) — The dependent test data.

RETURN **TABLE ({ UNSIGNED4 metricId , STRING metricName , REAL8 value })** — A dataset of metrics indicating the performance of the model.

SEE Types.metrics

SEE ML_Core.Types.NumericField

PREDICTNF PredictNF

GNNI \

DATASET(NumericField)	**PredictNF**
(UNSIGNED4 model, DATASET(NumericField) x)	

Predict the results for a model with 2 dimensional input and output using NumericField matrixes for input and output.

This a a NumericField wrapper around the Predict function. See Predict (above) for details.

PARAMETER **model** ||| UNSIGNED4 — A model token as returned from the previous GNNI call (e.g. Fit).

PARAMETER **x** ||| TABLE (NumericField) — The independent (i.e. input) data NumericField matrix.

RETURN **TABLE ({ UNSIGNED2 wi , UNSIGNED8 id , UNSIGNED4 number , REAL8 value })** — The output predicted by the model as a NumericField matrix.

SEE ML_Core.Types.NumericField

Tensor

Go Up

IMPORTS

python3 | _versions.ML_Core.V3_2_2.ML_Core |
_versions.ML_Core.V3_2_2.ML_Core.Types | std.system.Thorlib | std.system.Log |

DESCRIPTIONS

TENSOR Tensor

	Tensor

ECL Tensor Module.

Overview:

Tensor datasets provide an efficient way to store, distribute, and process N-Dimensional data. Tensors represent an N dimensional array. They can represent data of from 0 dimensions (scalar), 1 dimension (vector), 2 dimensions (matrix), or up to a high number of dimensions.

Tensors are typed – the module currently only supports REAL4 type Tensors, but is set up to accomodate other data types in the future. The Tensor.R4 submodule is used to manage REAL4 type Tensors.

Two main record types are defined for use with Tensors:

- TensorData is used to define the content of a Tensor. This is a sparse data format – each record represents one cell of the Tensor.
- t_Tensor is used to define the Tensor's metadata such as it's N dimensional shape, its type, etc. It manages the tensor as a series of slices (i.e. partitions) with the data packed into the slices in either sparse or dense form, depending on the nature of the data.

A Tensor is created by calling the MakeTensor(...) function with the appropriate meta-data and a TensorData dataset.

Inversely, the data is read out of a Tensor using the GetData(...) function.

Tensor Shape:

A Tensor is defined with a shape. Shapes are given by a set of integers defining the length of each dimension of the Tensor. For example: shape [4, 3, 2] represents a 4 x 3 x 2 tensor. Record-oriented Tensors may have the first shape component unspecified. Zero is used to indicated that the index is unspecified. For example: a shape of [0, 5, 8, 4] specifies a Tensor with an unspecified number of rows, each with a 3 dimensional shape [5, 8, 4].

Distribution Modes:

Tensors have 2 distribution modes:

- Distributed – The slices are distributed across the nodes of the cluster.
- Replicated – All slices are present on all nodes (for local operations on each node).

Tensor Lists:

A t_Tensor dataset also allows for multiple tensors of different shapes to be stored in a single dataset. The work item (wi) field of the Tensor is used to distinguish between the different Tensors. A Tensor with multiple work items is considered an ordered list of Tensors.

Tensor Data Types:

At some point, we will support Tensors of different data types such as REAL4, REAL8, INTEGER4, INTEGER8, and STRING. This release, however, supports only REAL4 type tensors. The methods operating on these tensors are found in the R4 (i.e. REAL4) submodule. Future versions will add more submodules for different tensor types.

The dat module (e.g. Tensor.R4.dat) provides methods for packing and unpacking scalar, vector, and matrix data. These methods allow, for example, a Tensor of shape [2,3,3] to be built by packing two 3 x 3 matrices into a Tensor.

EXAMPLES:

```
// Scalar (0-D)
tensDatScalar := Tensor.R4.dat.fromScalar(3.14159); // 0D (Scalar) Tensor data
// Vector (1-D)
tensDatVector := Tensor.R4.dat.fromVector([.013, .015, -.312, 0, 1.0]); // 1D (Vector) Tensor
// Matrix (2-D)
tensDatMatrix := Tensor.R4.dat.fromMatrix(myNF); // 2D (Matrix) Tensor data
// N-D Tensor
```

```
tensDat := DATASET([\{[1,1,1,1], .01\},
                    \{[5,2,111,3], .02\}], Tensor.R4.TensDat); // 4D (nD) Tensor data
```

Children

1. R4 : REAL4 tensor type attributes

R4 R4

Tensor \

	R4

REAL4 tensor type attributes

Children

1. TensData : REAL4 Tensor Data Format
2. t_SparseDat : Record format for the sparseData child dataset within a Tensor
3. t_Tensor : Record format for a REAL4 valued Tensor slice
4. dat : Submodule for manipulating TensorData
5. Replicate : Replicate the Tensor Slices to all nodes of the cluster
6. MakeTensor : Make a Tensor from a set of TensorData and some meta-data
7. GetData : Extract the data from a tensor and return it in sparse TensData format
8. Reshape : Reshape a tensor to a new compatible shape
9. Add : Add two tensors
10. GetRecordCount : Get the number of records in a record-oriented Tensor
11. AlignTensors : Aligns a list of Tensors (seperated by wi) so that all of the tensors' corresponding records are stored on the same node

---

TENSDATA TensData

Tensor \ R4 \

	TensData

REAL4 Tensor Data Format.

Note: This is sparse format, and any cells not supplied are assumed to be zero.

FIELD **indexes** ||| SET (UNSIGNED4) — – the N-dimensional index of this tensor cell

FIELD **value** ||| REAL4 — – the numeric value of this tensor cell.

T_SPARSEDAT t_SparseDat

Tensor \ R4 \

	t_SparseDat

Record format for the sparseData child dataset within a Tensor

FIELD **offset** ||| UNSIGNED4 — The offset within the tensor slice.

FIELD **value** ||| REAL4 — The value at the given offset within the tensor slice.

T_TENSOR t_Tensor

Tensor \ R4 \

	t_Tensor

Record format for a REAL4 valued Tensor slice.

Tensors are stored as a Dataset of Tensor slices. Each slice contains Tensor metadata (e.g. shape, dataType), as well as the tensor data elements within the slice. Slices can be densely packed or sparsely packed depending on the density of the source data.

FIELD **nodeId** ||| UNSIGNED4 — The node number on which this slice currently resides.

FIELD **wi** ||| UNSIGNED4 — The work-item allows a list of tensors to be stored within a single dataset. Wi of 1 indicates the first tensor in the list, 2 for the second, etc.

FIELD **sliceId** ||| UNSIGNED4 — The id of this tensor slice. Each tensor is represented as 1 or more slices. Each tensor in a tensor list can have the same sliceIds.

FIELD **shape** ||| SET (UNSIGNED4) — The shape of the tensor (e.g. [10, 20, 5]).

FIELD **dataType** ||| UNSIGNED4 — The data type for each cell of the tensor.

FIELD **maxSliceSize** ||| UNSIGNED4 — The size of a full slice for this tensor.

FIELD **sliceSie** ||| — The size of this slice. Slices 1 - (N-1) will full slices, while slice N may have less than the maxSliceSize data.

FIELD **denseDat** ||| — A packed block of REAL4 values representing the linearized data within this slice.

FIELD **sparseDat** ||| — A child dataset for storeing sparse data as a set of local offset and value pairs. Note: Only denseData or sparseData are used for any slice. The other will be empty.

FIELD **slicesize** ||| UNSIGNED4 — No Doc

FIELD **densedata** ||| SET (REAL4) — No Doc

FIELD **sparsedata** ||| TABLE (t_SparseDat) — No Doc

DAT dat

Tensor \ R4 \

	dat

Submodule for manipulating TensorData.

Children

1. fromScalar : Create tensor data from a scalar
2. fromVector : Create tensor data from a vector

3. fromMatrix : Create tensor data from a NumericField matrix
4. toScalar : Extract a scalar from a position within the Tensor data
5. toVector : Extract a vector of values from a TensData dataset
6. toMatrix : Extract a matrix of values from a TensData dataset

---

FROMSCALAR fromScalar

Tensor \ R4 \ dat \

DATASET(TensData)	**fromScalar**
(REAL4 value, t_Indexes atIndx = [])	

Create tensor data from a scalar.

The scalar will be placed at the "atIndex" in the tensor.

Example: tdat := t_Tensor.R4.dat.fromScalar('3.14159', [1,3,1]); // The cell will be placed at index [1, 3, 1]

PARAMETER **value** ||| REAL4 — The value of the tensor cell at atIndex.

PARAMETER **atindx** ||| SET (UNSIGNED4) — No Doc

RETURN **TABLE (TensData)** — A TensData dataset with one record.

PARM atIndx The index of the cell being defined.

FROMVECTOR fromVector

Tensor \ R4 \ dat \

DATASET(TensData)	**fromVector**
(SET OF REAL4 vec, t_Indexes atIndx = [])	

Create tensor data from a vector.

The elements of the array will be placed under "atIndx". The first element will be at [atIndx, 1], and the Nth will be at [atIndx, N].

Example: tdat := t_Tensor.R4.dat.fromVector([.1, .2, -.1, -.2], [1, 3]); // The first element (.1) will be at index [1, 3, 1].

PARAMETER **vec** ||| SET (REAL4) — A set of numbers representing the value of the vector.

PARAMETER **atIndx** ||| SET (UNSIGNED4) — The index under which to place the vector.

RETURN **TABLE (TensData)** — A TensData dataset with length the same as the vector.

FROMMATRIX fromMatrix

Tensor \ R4 \ dat \

DATASET(TensData)	**fromMatrix**
(DATASET(NumericField) mat, t_Indexes atIndx = [])	

Create tensor data from a NumericField matrix.

The elements of the matrix will be placed at: [atIndx, id, number], where id and number are the row and column indexes for each matrix cell.

Note: The work-item (wi) field of the NF matrix is ignored, so multiple work-items should not be used in the input matrix.

Example: tdat := t_Tensor.R4.dat.fromMatrix(myNumericFieldDS, [3,5,2]); // The first element of the matrix will be at: [3,5,2,1,1].

PARAMETER **mat** ||| TABLE (NumericField) — A ML_Core.NumericField dataset representing the matrix to be added.

PARAMETER **atIndx** ||| SET (UNSIGNED4) — The index under which to place this matrix in the tensor data.

RETURN **TABLE ({ SET (UNSIGNED4) indexes , REAL4 value })** — A TensorData dataset with length the same as the NumericField data passed in.

SEE ML_Core.Types.NumericField

TOSCALAR toScalar

Tensor \ R4 \ dat \

REAL4	**toScalar**
`(DATASET(TensData) tens, t_Indexes fromIndx = [])`	

Extract a scalar from a position within the Tensor data.

Note: If the tensor shape has 5 indexes, then fromIndex should be 5 long, as the scalar is extracted from the actual tensor cell.

Example: REAL4 val := toScalar(myt_TensorDat, [1,3]); // Extract a cell from position [1,3] of a 2-D tensor.

PARAMETER **tens** ||| TABLE (TensData) — A TensData dataset from which to extract.

PARAMETER **fromIndx** ||| SET (UNSIGNED4) — The index from which to extract the cell value.

RETURN **REAL4** — The extracted value as a REAL4.

TOVECTOR toVector

Tensor \ R4 \ dat \

DATASET(NumericField)	**toVector**
`(DATASET(TensData) tens, t_Indexes fromIndx = [])`	

Extract a vector of values from a TensData dataset.

If the tensor shape has N terms, then the fromIndx should contain N-1 terms. It will return the cells: [fromIndx, 1] through [fromIndx, M], where M is the last shape term.

The data is returned as a NumericField matrix with a single row (i.e. id = 1). This is used rather than a SET to allow for sparse data. Only non-zero cells are returned. The number field indicates the position within the vector.

Example: DATASET(NumericField) vec := toVector(myt_TensorDat, [5,2]); // Extract a vector from [5,2] in the 3-D tensor data.

PARAMETER **tens** ||| TABLE (TensData) — The TensorData dataset from which to extract the vector.

PARAMETER **fromIndex** ||| — the index from which to extract.

PARAMETER **fromindx** ||| SET (UNSIGNED4) — No Doc

RETURN **TABLE ({ UNSIGNED2 wi , UNSIGNED8 id , UNSIGNED4 number , REAL8 value })** — A vector as a single row of a NumericField matrix.

SEE ML_Core.Types.NumericField

TOMATRIX toMatrix

Tensor \ R4 \ dat \

DATASET(NumericField)	**toMatrix**
(DATASET(TensData) tens, t_Indexes fromIndx = [])	

Extract a matrix of values from a TensData dataset.

If the tensor shape has N terms, then the fromIndx should contain N-2 terms. It will return the cells: [fromIndx, 1, 1] through [fromIndx, K, M], where K is the second to last shape term and M isthe last shape term.

Example: myNF := toNumericField(myt_TensorDat, [3,11]); // Extract a matrix from a 4-D tensor data dataset.

PARAMETER **tens** ||| TABLE (TensData) — The TensorData dataset from which to extract.

PARAMETER **fromIndx** ||| SET (UNSIGNED4) — The index from wich to extract the matrix.

RETURN **TABLE ({ UNSIGNED2 wi , UNSIGNED8 id , UNSIGNED4 number , REAL8 value })** — A matrix in NumericField format.

SEE ML_Core.Types.NumericField

REPLICATE Replicate

Tensor \ R4 \

DATASET(t_Tensor)	**Replicate**
(DATASET(t_Tensor) tens)	

Replicate the Tensor Slices to all nodes of the cluster.

This is used to provide a copy of the Tensor on each node of the cluster.

PARAMETER **tens** ||| TABLE (t_Tensor) — A t_Tensor dataset to be replicated.

RETURN **TABLE ({ UNSIGNED4 nodeId , UNSIGNED4 wi , UNSIGNED4 sliceId , SET (UNSIGNED4) shape , UNSIGNED4 dataType , UNSIGNED4 maxSliceSize , UNSIGNED4 sliceSize , SET (REAL4) denseData , TABLE (t_SparseDat) sparseData })** — A replicated t_Tensor dataset. If the original dataset contained N slices, the new dataset will contain N x nNodes slices.

MAKETENSOR MakeTensor

Tensor \ R4 \

DATASET(t_Tensor)	**MakeTensor**
(t_Indexes shape, DATASET(TensData) contents = DATASET([], TensData), BOOLEAN replicated = FALSE, UNSIGNED4 wi = 1, UNSIGNED4 forceMaxSliceSize = 0)	

Make a Tensor from a set of TensorData and some meta-data.

Tensors may be replicated (e.g. copied locally to each node), or distributed (slices spread across nodes).

PARAMETER **shape** ||| SET (UNSIGNED4) — The desired shape of the Tensor (e.g. [10, 5, 2]).

PARAMETER **contents** ||| TABLE (TensData) — Dataset of TensData representing the contents of the Tensor. If omitted, the tensor will be empty (i.e. all zeros).

PARAMETER **replicated** ||| BOOLEAN — True if this tensor is to be replicated to all nodes. Default = False (i.e. distributed).

PARAMETER **wi** ||| UNSIGNED4 — Work-item. This field allows multiple Tensors to be stored in the same dataset. Default = 1. This field should always be 1 for a single Tensor dataset. For a Tensor list, wi should always go from 1 to nTensors.

PARAMETER **forceMaxSliceSize** ||| UNSIGNED4 — If non-zero, it will override the default sizing of slices. Needed internally, but should always use the default (0) for external uses.

RETURN **TABLE ({ UNSIGNED4 nodeId , UNSIGNED4 wi , UNSIGNED4 sliceId , SET (UNSIGNED4) shape , UNSIGNED4 dataType , UNSIGNED4 maxSliceSize , UNSIGNED4 sliceSize , SET (REAL4) denseData , TABLE (t_SparseDat) sparseData })** — A dataset of t_Tensor representing the Tensor object.

GETDATA GetData

Tensor \ R4 \

DATASET(TensData)	**GetData**
(DATASET(t_Tensor) tens)	

Extract the data from a tensor and return it in sparse TensData format.

This is essentially the inverse of the MakeTensor(. . .) method.

PARAMETER **tens** ||| TABLE (t_Tensor) — The t_Tensor dataset from which to extract the data

RETURN **TABLE (TensData)** — TensData dataset of non-zero tensor data (sparse form).

RESHAPE Reshape

Tensor \ R4 \

	Reshape
(DATASET(t_Tensor) tens, t_Indexes newShape)	

Reshape a tensor to a new compatible shape.

Returns a new tensor with the desired shape.

If the shapes were not compatible, an empty tensor is returned.

PARAMETER **tens** ||| TABLE (t_Tensor) — The tensor to be reshaped.

PARAMETER **newShape** ||| SET (UNSIGNED4) — The desired new shape.

RETURN **TABLE ({ UNSIGNED4 nodeId , UNSIGNED4 wi , UNSIGNED4 sliceId , SET (UNSIGNED4) shape , UNSIGNED4 dataType , UNSIGNED4 maxSliceSize , UNSIGNED4 sliceSize , SET (REAL4) denseData , TABLE (t_SparseDat) sparseData })** — A new tensor with the desired shape, if the shapes were compatible. Otherwise, an empty tensor.

ADD Add

Tensor \ R4 \

DATASET(t_Tensor)	Add
(DATASET(t_Tensor) t1, DATASET(t_Tensor) t2)	

Add two tensors.

This performs cell-wise addition of the contents of the two input tensors and returns a new tensor representing the sum of the two tensors.

Both tensors must be of the same shape.

This function can also add two tensor lists. Each tensor of list 1 must be of the same shape as the corresponding tensor in list 2. The lists must also be of the same length.

PARAMETER **t1** ||| TABLE (t_Tensor) — The first tensor or tensor list.

PARAMETER **t2** ||| TABLE (t_Tensor) — The second tensor or tensor list.

RETURN **TABLE ({ UNSIGNED4 nodeId , UNSIGNED4 wi , UNSIGNED4 sliceId , SET (UNSIGNED4) shape , UNSIGNED4 dataType , UNSIGNED4 maxSliceSize , UNSIGNED4 sliceSize , SET (REAL4) denseData , TABLE (t_SparseDat) sparseData })** — A new Tensor (DATASET(t_Tensor)) representing t1 + t2.

GETRECORDCOUNT GetRecordCount

Tensor \ R4 \

UNSIGNED	**GetRecordCount**
(DATASET(t_Tensor) tens)	

Get the number of records in a record-oriented Tensor.

PARAMETER **tens** ||| TABLE (t_Tensor) — The input Tensor.

RETURN **UNSIGNED8** — The number of records in the distributed tensor.

ALIGNTENSORS AlignTensors

Tensor \ R4 \

DATASET(t_Tensor)	**AlignTensors**
(DATASET(t_Tensor) tensList)	

Aligns a list of Tensors (seperated by wi) so that all of the tensors' corresponding records are stored on the same node. This prevents different sized records from being distributed differently among the nodes.

In most cases, the inputs and outputs to a neural network during training, and the inputs during prediction should be aligned so that various aspects of the same observation are presented together.

PARAMETER **tens** ||| — A Tensor List with at least two tensors identified by sequential work item ids from 1-N.

PARAMETER **tenslist** ||| TABLE (t_Tensor) — No Doc

RETURN **TABLE (t_Tensor)** — A new Tensor List with the same number of tensors as the input list, with all of the tensors being aligned.

Types

Go Up

DESCRIPTIONS

TYPES Types

	Types

Type definitions for use with the GNNI Interface.

Children

1. metrics : Return structure for call to EvaluateMod
2. FuncLayerDef : Record to use for defining complex (i.e

METRICS metrics

Types \

	metrics

Return structure for call to EvaluateMod.

Contains a series of metrics and their values.

FIELD **metricId** ||| UNSIGNED4 — A sequential id to maintain the metrics' order.

FIELD **metricName** ||| STRING — The Keras name identifying the metric.

FIELD **value** ||| REAL8 — The value of the metric.

FUNCLAYERDEF FuncLayerDef

Types \

	FuncLayerDef

Record to use for defining complex (i.e. Non-Sequential, Functional) models using the DefineFuncModel() GNNI method.

FIELD **layername** ||| STRING — No Doc

FIELD **layerdef** ||| STRING — No Doc

FIELD **predecessors** ||| SET (STRING) — No Doc

Utils

Go Up

IMPORTS

_versions.GNN.V2_0.GNN.Tensor |

DESCRIPTIONS

UTILS Utils

	Utils

Utility module for GNN. Contains various utility functions for use with GNN.

Children

1. ToOneHot : Convert Tensor Data to OneHot Encoding
2. FromOneHot : Convert One Hot encoded 2-D tensor data to class label format
3. Probabilities2Class : Convert a set of class probabilities to a class label
   Class probabilities are typically returned from a ”softmax” activation function

---

TOONEHOT ToOneHot

Utils \

DATASET(TensDat)	**ToOneHot**
(DATASET(TensDat) classDat, UNSIGNED numClasses)	

Convert Tensor Data to OneHot Encoding.

Input is a 1-D tensor data set with the value of each observation being the class.

Returns a 2-D TensDat dataset with numClasses being the cardinality of the 2nd dimension. The value will be 1 for the cell with second dimension corresponding to the class. All others will be zero. Since TensDat is a sparse format, all zero cells will be skipped.

Note that Classes are 0-based. Class 0 will be at final index = 1. Class 5 will be at final index = 6.

PARAMETER **classDat** ||| TABLE (TensDat) — A 1-D tensor with the index being the observation number, and the value ((0-(numClasses-1)) corresponds to the class label.

PARAMETER **numClasses** ||| UNSIGNED8 — The number of possible values for the class variable.

RETURN **TABLE ({ SET (UNSIGNED4) indexes , REAL4 value })** — A 2-D set of TensData one hot encoded.

SEE Tensor.R4.TensData

FROMONEHOT FromOneHot

Utils \

DATASET(TensDat)	**FromOneHot**
(DATASET(TensDat) ohTens)	

Convert One Hot encoded 2-D tensor data to class label format.

Input is a 2-D One Hot encoded TensDat dataset, as produced by ToOneHot above.

Output is a 1-D set of class labels corresponding to the highest value of the One Hot encoded fields for each observation.

Note that returned classes are zero based.

PARAMETER **ohTens** ||| TABLE (TensDat) — A one hot encoded 2-D tensor data set.

RETURN **TABLE ({ SET (UNSIGNED4) indexes , REAL4 value })** — A 1-D dataset of TensData with the value of each observation being the class label.

SEE Tensor.R4.TensData

PROBABILITIES2CLASS Probabilities2Class

Utils \

DATASET(TensDat)	**Probabilities2Class**
(DATASET(TensDat) td)	

Convert a set of class probabilities to a class label

Class probabilities are typically returned from a "softmax" activation function. This returns the class label associated with the maximum probability label.

Note that this function simply calls FromOneHot, which implements this functionality. Both names are included because it is sometimes more intuitive to think of the operation in different ways.

PARAMETER **td** ||| TABLE (TensDat) — A 2-D tensor data set with a probability for each class.

RETURN **TABLE ({ SET (UNSIGNED4) indexes , REAL4 value })** — A 1-D dataset of TensData with a class label for each observation.

SEE FromeOneHot

SEE Tensor.R4.TensData
