antk Documentation

Release 0.3.0

Aaron Tuor

July 23, 2016

CONTENTS

Principal Authors Aaron Tuor , Brian Hutchinson

The Automated Neural-graph toolkit is a high level machine learning toolkit built on top of Google's Tensorflow to facilitate rapid prototyping of Neural Network models which may consist of multiple models chained together. This includes models which have multiple input and/or multiple output streams.

ANTk will be most useful to people who have gone through some of the basic tensorflow tutorials, have some machine learning background, and wish to take advantage of some of tensorflow's more advanced features. The code itself is consistent, well-formatted, well-documented, and abstracted only to a point necessary for code reuse, and complex model development. The toolkit code contains tensorflow usage developed and discovered over six months of machine learning research conducted in tensorflow, by Hutch Research based out of Western Washington University's Computer Science Department.

The kernel of the toolkit is comprised of 4 independent, but complementary modules:

loader: Implements a general purpose data loader for python non-sequential machine learning tasks. Contains functions for common data pre-processing tasks.

config: Facilitates the generation of complex tensorflow models, built from compositions of tensorflow functions.

node_ops: Contains functions taking a tensor or structured list of tensors and returning a tensor or structured list of tensors. The functions are commonly used compositions of tensorflow functions which operate on tensors.

generic_model: A general purpose model builder equipped with generic train, and predict functions which takes parameters for optimization strategy, mini-batch, etc...

Motivation:

Working at a high level of abstraction is important for the rapid development of machine learning models. Many successful state of the art models chain together or create an ensemble of several complex models. To facilitate the need for building models whose components are models we have developed a highly modularized set of utilities.

While this high level of abstraction is often attractive for development, when working with a highly abstracted machine learning toolkit it is often difficult to assess details of implementation and the underlying math behind a packaged model. To address this concern we have made the toolkit implementation and underlying math as transparent as possible. There are links to source code, and relevant scientific papers in the API and we have taken pains to illuminate the workings of complex code with well formatted mathematical equations. Also, we have taken care to allow easy access to tensor objects created by high level operations such as deep neural networks.

Design methodology:

ANTK was designed to be highly modular, and allow for a high level of abstraction with a great degree of transparency to the underlying implementation. We hope that this design can eliminate the reproduction of coding efforts without sacrificing important knowledge about implementation that may effect the overall performance of a model.

CHAPTER

ONE

DEPENDENCIES

Tensorflow, scipy, numpy, sklearn, graphviz.

Install tensorflow

Install graphviz

CHAPTER

TWO

INSTALLATION

A virtual environment is recommended for installation. Make sure that tensorflow is installed in your virtual environment and graphviz is installed on your system.

In a terminal:

```
(venv)$ mkdir antk
(venv)$ cd antk
(venv)$ git init
Initialized empty Git repository in /home/tuora/garbage/.git/
(venv)$ git remote add origin https://github.com/aarontuor/antk.git
(venv)$ git pull origin master
...
(venv)$ python setup.py develop
```

CHAPTER

THREE

DOCUMENTATION

3.1 API: ANT modules

3.1.1 loader

Implements a general purpose data loader for python non-sequential machine learning tasks. Several common data transformations are provided in this module, e.g., tfidf, whitening, etc.

Loader Tutorial

The loader module implements a general purpose data loader for python non-sequential machine learning tasks.

Supported Data Types

loader is designed to operate on numpy arrays, scipy sparse csr_matrices, and `HotIndex` objects.

HotIndex objects

In the discussion below we distinguish "one hot" meaning a matrix with exactly a single 1 per row and zeros elsewhere from "many hot", matrices with only ones and zeros. In order to address the pervasive need for one hot representations the loader module has some functions for creating one hot matrices (`toOnehot`), transforming one hots to indices (`toIndex`) and determining if a matrix is a one hot representation (`is_one_hot`).

Also there is a compact index representation of a one hot matrix, the `HotIndex` object which has a field to retain the row size of the one hot matrix, while representing the *on* columns by their indices alone.

Supported File Formats

.mat: Matlab files of matrices made with the matlab save command. Saved matrices to be read must be named **data**. As of now some Matlab implementations may load the files with the *load* function but the loaded matrices will have different values.

.sparsetxt Plain text files where lines correspond to an entry in a matrix where a line consists of values **i j k**, so a matrix A is constructed where $A_{ij} = k$. Tokens must be whitespace delimited.

.densetxt: Plain text files with a matrix represented in standard form. Tokens must be whitespace delimited.

.sparse: Like `.sparsetxt` files but written in binary (no delimiters) to save disk space and speed file i/o. Matrix dimensions are contained in the first bytes of the file.

.binary / .dense: Like `.densetxt` files but written in binary (no delimiters) to save disk space and speed file i/o. Matrix dimensions are contained in the first bytes of the file.

.index: A saved *HotIndex* object written in binary.

Import and export data

export_data : Scipy sparse matrices and numpy arrays may be saved to a supported file format with this function.

import_data: Scipy sparse matrices and numpy arrays may be loaded from a supported file format with this function.

```
>>> from antk.core import loader
>>> import numpy
>>> test = numpy.random.random((3,3))
>>> test
array([[ 0.65769658,  0.22230913,  0.41058879],
      [ 0.71498391,  0.47537034,  0.88214378],
      [ 0.37795028,  0.02388658,  0.41103339]])
>>> loader.export_data('test.mat', test)
>>> loader.import_data('test.mat')
array([[ 0.65769658,  0.22230913,  0.41058879],
      [ 0.71498391,  0.47537034,  0.88214378],
      [ 0.37795028,  0.02388658,  0.41103339]])
```

The `DataSet` object

DataSet objects are designed to make data manipulation easier for mini-batch gradient descent training. It is necessary to package your data in a *DataSet* object in order to create a *Model* object from antk's generic_model module. You can create a DataSet with a dictionary of numpy arrays, scipy sparse csr_matrices, and *HotIndex* objects.

```
>>> test2 = numpy.random.random((3,4))
>>> test3 = numpy.random.random((3,5))
>>> datadict = {'feature1': test, 'feature2': test2, 'feature3': test3}
>>> data = loader.DataSet(datadict)
>>> data
antk.core.DataSet object with fields:
    '_labels': {}
    '_num_examples': 3
    '_epochs_completed': 0
    '_index_in_epoch': 0
    '_mix_after_epoch': False
    '_features': {'feature2': array([[ 0.3053935 ,  0.19926099,  0.43178954,  0.21737312],
   [ 0.47352974,  0.33052605,  0.22874512,  0.59903599],
   [ 0.62532971,  0.70029533,  0.13582899,  0.39699691]]), 'feature3': array([[ 0.98901453,  0.4
   [ 0.46123761,  0.94292179,  0.13315178,  0.55212266,  0.09410787],
   [ 0.90358241,  0.88080438,  0.51443528,  0.69531831,  0.32700497]]), 'feature1': array([[ 0.5
   [ 0.95176126,  0.37265882,  0.72076518],
   [ 0.97364273,  0.79038134,  0.83085418]])}
```

There is a *DataSet.show* method that will display information about the DataSet.

```
>>> data.show()
features:
        feature2: (3, 4) <type 'numpy.ndarray'>
        feature3: (3, 5) <type 'numpy.ndarray'>
        feature1: (3, 3) <type 'numpy.ndarray'>
labels:
```

There is an optional argument for labels in case you wish to have features and labels in separate maps.

```
>>> label = numpy.random.random((3,10))
>>> data = loader.DataSet(datadict, labels={'label1': label})
>>> data.show()
features:
                feature2: (3, 4) <type 'numpy.ndarray'>
                feature3: (3, 5) <type 'numpy.ndarray'>
                feature1: (3, 3) <type 'numpy.ndarray'>
labels:
                label1: (3, 10) <type 'numpy.ndarray'>
```

Matrices in the DataSet can be accessed by their keys.

```
>>> data.features['feature1']
array([[ 0.65769658,  0.22230913,  0.41058879],
      [ 0.71498391,  0.47537034,  0.88214378],
      [ 0.37795028,  0.02388658,  0.41103339]])
>>> data.labels['label1']
        array([[ 0.95719927,  0.5568232 ,  0.18691618,  0.74473549,  0.13150579,
                         0.18189613,  0.00841565,  0.36285286,  0.52124701,  0.90096317],
                  [ 0.73361071,  0.0939201 ,  0.22622336,  0.47731619,  0.91260044,
                         0.98467187,  0.01978079,  0.93664054,  0.92857152,  0.25710894],
                  [ 0.024292  ,  0.92705842,  0.0086137 ,  0.33100848,  0.93829355,
                         0.04615762,  0.91809485,  0.79796301,  0.88414445,  0.72963613]])
```

If your data is structured so that your features and labels have rows corresponding to data points then you can use the *next_batch* function to grab data for a mini-batch iteration in stochastic gradient descent.

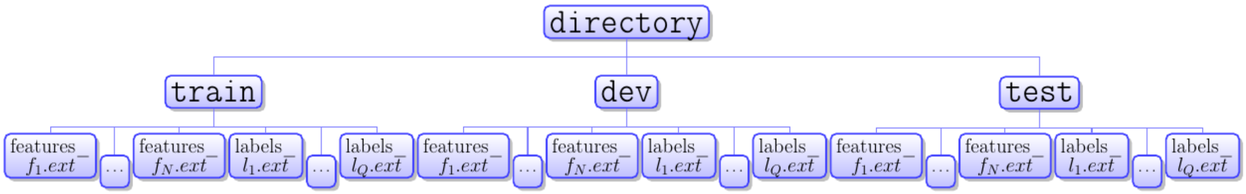
```
>>> minibatch = data.next_batch(2)
>>> minibatch.show()
features:
        feature2: (2, 3) <type 'numpy.ndarray'>
        feature3: (2, 3) <type 'numpy.ndarray'>
        feature1: (2, 3) <type 'numpy.ndarray'>
labels:
        label1: (2, 10) <type 'numpy.ndarray'>
```

You can ensure that the order of the data points is shuffled every epoch with the *mix_after_epoch* function, and see how many epochs the data has been trained with from the `epochs_completed` property.

```
>>> data.mix_after_epoch(True)
>>> data.next_batch(1)
<antk.core.loader.DataSet object at 0x7f5c48dc6b10>
>>> data.epochs_completed
1
>>> data.features['features1']
array([[ 0.71498391,  0.47537034,  0.88214378],
       [ 0.65769658,  0.22230913,  0.41058879],
       [ 0.37795028,  0.02388658,  0.41103339]])
```

read_data_sets: The loading function

read_data_sets will automatically load folders of data of the supported file formats into a *DataSets* object, which is just a record of DataSet objects with a *show()* method to display all the datasets at once. Below are some things to know before using the *read_data_sets* function.



Directory Structure *directory* at the top level can be named whatever. There are by default assumed to be three directories below *directory* named **train**, **dev**, and **test**. However one may choose to read data from any collection of directories using the *folders* argument. If the directories specified are not present *Bad_directory_structure_error* will be raised during loading. The top level directory may contain other files besides the listed directories. According to the diagram:

> *N* is the number of feature sets. Not to be confused with the number of elements in a feature vector for a particular feature set. *Q* is the number of label sets. Not to be confused with the number of elements in a label vector for a particular label set. The hash for a matrix in a *DataSet.features* attribute is whatever is between **features_** and the file extension (*.ext*) in the file name. The hash for a matrix in a *DataSet.labels* attribute is whatever is between **labels_** and the file extension (*.ext*) in the file name.

Note: Rows of feature and data matrices should correspond to individual data points as opposed to the transpose. There should be the same number of data points in each file of the **train** directory, and the same is true for the **dev** and **test** directories. The number of data points can of course vary between **dev**, **train**, and **test** directories. If you have data you want to load that doesn't correspond to the paradigm of matrices which have a number of data points columns there you may use the *read_data_sets* **folders** argument (a list of folder names) to include other directories besides **dev**, **train**, and **test**. In this case all and only the folders specified by the **folders** argument will be loaded into a *DataSets* object.

Examples Below we download, untar, and load a processed and supplemented Movielens 100k dataset, where data points are user/item pairs for observed movie ratings.

Basic usage:

```
>>> loader.maybe_download('ml100k.tar.gz', '.', 'http://sw.cs.wwu.edu/~tuora/aarontuor/ml100k.tar.gz
>>> loader.untar('ml100k.tar.gz')
>>> loader.read_data_sets('ml100k).show()
reading train...
reading dev...
reading test...
dev:
features:
        item: vec.shape: (10000,) dim: 1682 <class 'antk.core.loader.HotIndex'>
        user: vec.shape: (10000,) dim: 943 <class 'antk.core.loader.HotIndex'>
        words: (10000, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        time: (10000, 1) <type 'numpy.ndarray'>
labels:
        genre: (10000, 19) <type 'numpy.ndarray'>
        ratings: (10000, 1) <type 'numpy.ndarray'>
        genre_dist: (10000, 19) <type 'numpy.ndarray'>
```

```
test:
features:
        item: vec.shape: (10000,) dim: 1682 <class 'antk.core.loader.HotIndex'>
        user: vec.shape: (10000,) dim: 943 <class 'antk.core.loader.HotIndex'>
        words: (10000, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        time: (10000, 1) <type 'numpy.ndarray'>
labels:
        genre: (10000, 19) <type 'numpy.ndarray'>
        ratings: (10000, 1) <type 'numpy.ndarray'>
        genre_dist: (10000, 19) <type 'numpy.ndarray'>
train:
features:
item: vec.shape: (80000,) dim: 1682 <class 'antk.core.loader.HotIndex'>
        user: vec.shape: (80000,) dim: 943 <class 'antk.core.loader.HotIndex'>
        words: (80000, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        time: (80000, 1) <type 'numpy.ndarray'>
labels:
        genre: (80000, 19) <type 'numpy.ndarray'>
        ratings: (80000, 1) <type 'numpy.ndarray'>
        genre_dist: (80000, 19) <type 'numpy.ndarray'>
```

Other Folders:

```
>>> loader.read_data_sets('ml100k', folders=['user', 'item']).show()
reading user...
reading item...
item:
features:
        genres: (1682, 19) <type 'numpy.ndarray'>
        bin_doc_term: (1682, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        month: vec.shape: (1682,) dim: 12 <class 'antk.core.loader.HotIndex'>
        doc_term: (1682, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        tfidf_doc_term: (1682, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        year: (1682, 1) <type 'numpy.ndarray'>
labels:
user:
features:
        occ: vec.shape: (943,) dim: 21 <class 'antk.core.loader.HotIndex'>
        age: (943, 1) <type 'numpy.ndarray'>
        zip: vec.shape: (943,) dim: 1000 <class 'antk.core.loader.HotIndex'>
        sex: vec.shape: (943,) dim: 2 <class 'antk.core.loader.HotIndex'>
labels:
```

Selecting Files:

```
>>> loader.read_data_sets('ml100k', folders=['user', 'item'], hashlist=['zip', 'sex', 'year']).show()
reading user...
reading item...
item:
features:
        year: (1682, 1) <type 'numpy.ndarray'>
labels:
user:
features:
        zip: vec.shape: (943,) dim: 1000 <class 'antk.core.loader.HotIndex'>
        sex: vec.shape: (943,) dim: 2 <class 'antk.core.loader.HotIndex'>
labels:
```

Loading, Saving, and Testing

export_data

import_data

is_one_hot

read_data_sets

Classes

DataSet

DataSets

HotIndex

Data Transforms

center

l1normalize

l2normalize

pca_whiten

tfidf

toOnehot

toIndex

unit_variance

Exceptions

Bad_directory_structure_error

Mat_format_error

Sparse_format_error

Unsupported_format_error

Loading, Saving, and Testing

save

export_data

load

import_data

is_one_hot

read_data_sets

Classes

DataSet

DataSets

HotIndex

Data Transforms

center

l1normalize

l2normalize

pca_whiten

tfidf

toOnehot

toIndex

unit_variance

Exceptions

Bad_directory_structure_error

Mat_format_error

Sparse_format_error

Unsupported_format_error

API

exception `loader.`**`Bad_directory_structure_error`**

Raised when a data directory specified, does not contain a subfolder specified in the *folders* argument to *read_data_sets*.

class `loader.`**`DataSet`** (*features*, *labels=None*, *num_examples=None*, *mix=False*)

General data structure for mini-batch gradient descent training involving non-sequential data.

Parameters

- **features** – A dictionary of string label names to data matrices. Matrices may be *HotIndex*, scipy sparse csr_matrix, or numpy arrays.
- **labels** – A dictionary of string label names to data matrices. Matrices may be *HotIndex*, scipy sparse csr_matrix, or numpy arrays.
- **num_examples** – How many data points.
- **mix** – Whether or not to shuffle per epoch.

Returns

Attributes

Methods

features
A dictionary of feature matrices.

index_in_epoch
The number of datapoints that have been trained on in a particular epoch.

labels
A dictionary of label matrices

mix_after_epoch(*mix*)
Whether or not to shuffle after training for an epoch.

Parameters **mix** – True or False

next_batch(*batch_size*)

Return a sub DataSet of next batch-size examples.

If shuffling enabled: If *batch_size* is greater than the number of examples left in the epoch then a batch size DataSet wrapping back to beginning will be returned.

If shuffling turned off: If *batch_size* is greater than the number of examples left in the epoch, points will be shuffled and batch_size DataSet is returned starting from index 0.

Parameters **batch_size** – int

Returns A `DataSet` object with the next *batch_size* examples.

num_examples
Number of rows (data points) of the matrices in this `DataSet`.

show()
Pretty printing of all the data (dimensions, keys, type) in the `DataSet` object

showmore()
Print a sample of the first up to twenty rows of matrices in DataSet

class loader.**DataSets**(*datasets_map*, *mix=False*)
A record of DataSet objects with a display function.

Methods

show()
Pretty print data attributes.

showmore()
Pretty print data attributes, and data.

class loader.**HotIndex**(*matrix*, *dimension=None*)
Index vector representation of one hot matrix. Can hand constructor either a one hot matrix, or vector of indices and dimension.

Attributes

Methods

dim
The feature dimension of the one hot vector represented as indices.

hot()

Returns A one hot scipy sparse csr_matrix

shape
The shape of the one hot matrix encoded.

vec
The vector of hot indices.

class loader.**IndexVector**(*matrix*, *dimension=None*)

Attributes

Methods

exception loader.**Mat_format_error**
Raised if the .mat file being read does not contain a variable named *data*.

exception loader.**Sparse_format_error**
Raised when reading a plain text file with .sparsetxt extension and there are not three entries per line.

exception loader.**Unsupported_format_error**
Raised when a file is requested to be loaded or saved without one of the supported file extensions.

loader.**center**(*X*, *axis=None*)

Parameters X – A matrix to center about the mean(over columns axis=0, over rows axis=1, over all entries axis=None)

Returns A matrix with entries centered along the specified axis.

loader.**export_data**(*filename*, *data*)
Decides how to save data by file extension. Raises *Unsupported_format_error* if extension is not one of the supported extensions (mat, sparse, binary, dense, index). Data contained in .mat files should be saved in a matrix named *data*.

Parameters

- **filename** – A file of an accepted format representing a matrix.
- **data** – A numpy array, scipy sparse matrix, or *HotIndex* object.

loader.**import_data**(*filename*)
Decides how to load data into python matrices by file extension. Raises *Unsupported_format_error* if extension is not one of the supported extensions (mat, sparse, binary, dense, sparsetxt, densetxt, index).

Parameters filename – A file of an accepted format representing a matrix.

Returns A numpy matrix, scipy sparse csr_matrix, or any:*HotIndex*.

loader.**is_one_hot**(*A*)

Parameters A – A numpy array or scipy sparse matrix

Returns True if matrix is a sparse matrix of one hot vectors, False otherwise

Examples

```
>>> import numpy
>>> from antk.core import loader
>>> x = numpy.eye(3)
>>> loader.is_one_hot(x)
True
>>> x *= 5
>>> loader.is_one_hot(x)
False
>>> x = numpy.array([[1, 0, 0], [1, 0, 0], [1, 0, 0]])
>>> loader.is_one_hot(x)
True
>>> x[0,1] = 2
>>> loader.is_one_hot(x)
False
```

`loader.`**`l1normalize`**(*X*, *axis=1*)

axis=1 normalizes each row of X by norm of said row. $l1normalize(X)_{ij} = \frac{X_{ij}}{\sum_k |X_{ik}|}$

axis=0 normalizes each column of X by norm of said column. $l1normalize(X)_{ij} = \frac{X_{ij}}{\sum_k |X_{kj}|}$

axis=None normalizes entries of X by norm of X. $l1normalize(X)_{ij} = \frac{X_{ij}}{\sum_k \sum_p |X_{kp}|}$

Parameters

- **`X`** – A scipy sparse csr_matrix or numpy array.
- **`axis`** – The dimension to normalize over.

Returns A normalized matrix.

`loader.`**`l2normalize`**(*X*, *axis=1*)

axis=1 normalizes each row of X by norm of said row. $l2normalize(X)_{ij} = \frac{X_{ij}}{\sqrt{\sum_k X_{ik}^2}}$

axis=0 normalizes each column of X by norm of said column. $l2normalize(X)_{ij} = \frac{X_{ij}}{\sqrt{\sum_k X_{kj}^2}}$

axis=None normalizes entries of X by norm of X. $l2normalize(X)_{ij} = \frac{X_{ij}}{\sqrt{\sum_k \sum_p X_{kp}^2}}$

Parameters

- **`X`** – A scipy sparse csr_matrix or numpy array.
- **`axis`** – The dimension to normalize over.

Returns A normalized matrix.

`loader.`**`load`**(*filename*)

Calls *`import_data`*. Decides how to load data into python matrices by file extension. Raises *`Unsupported_format_error`* if extension is not one of the supported extensions (mat, sparse, binary, dense, sparsetxt, densetxt, index).

Parameters **`filename`** – A file of an accepted format representing a matrix.

Returns A numpy matrix, scipy sparse csr_matrix, or any:*HotIndex*.

`loader.`**`makedirs`**(*datadirectory*, *sub_directory_list=('train', 'dev', 'test')*)

Parameters

- **datadirectory** – Name of the directory you want to create containing the subdirectory folders. If the directory already exists it will be populated with the subdirectory folders.
- **sub_directory_list** – The list of subdirectories you want to create

Returns void

`loader.`**`maxnormalize`**(*X*, *axis=1*)

axis=1 normalizes each row of X by norm of said row. $maxnormalize(X)_{ij} = \frac{X_{ij}}{max(X_{i:})}$

axis=0 normalizes each column of X by norm of said column. $maxnormalize(X)_{ij} = \frac{X_{ij}}{max(X_{:j})}$

axis=None normalizes entries of X norm of X. $maxnormalize(X)_{ij} = \frac{X_{ij}}{max(X)}$

Parameters

- **X** – A scipy sparse csr_matrix or numpy array.
- **axis** – The dimension to normalize over.

Returns A normalized matrix.

`loader.`**`maybe_download`**(*filename*, *work_directory*, *source_url*)

Download the data from source url, unless it's already here. From https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/learn/python/learn/datasets/base.py

Parameters

- **filename** – string, name of the file in the directory.
- **work_directory** – string, path to working directory.
- **source_url** – url to download from if file doesn't exist.

Returns Path to resulting file.

`loader.`**`pca_whiten`**(*X*)

Returns matrix with PCA whitening transform applied. This transform assumes that data points are rows of matrix.

Parameters

- **X** – Numpy array, scipy sparse matrix
- **axis** – Axis to whiten over.

Returns

`loader.`**`read_data_sets`**(*directory*, *folders=('train', 'dev', 'test')*, *hashlist=()*, *mix=False*)

Parameters

- **directory** – Root directory containing data to load.
- **folders** – The subfolders of *directory* to read data from by default there are train, dev, and test folders. If you want others you have to make an explicit list.
- **hashlist** – If you provide a hashlist these files and only these files will be added to your `DataSet` objects. It you do not provide a hashlist then anything with the privileged prefixes **labels_** or **features_** will be loaded.

Returns A `DataSets` object.

`loader.`**`save`**(*filename*, *data*)

Calls :any'export_data'. Decides how to save data by file extension. Raises `Unsupported_format_error` if extension is not one of the supported extensions (mat, sparse, binary, dense, index). Data contained in .mat files should be saved in a matrix named *data*.

Parameters

- **filename** – A file of an accepted format representing a matrix.
- **data** – A numpy array, scipy sparse matrix, or *HotIndex* object.

loader.**tfidf**(*X*, *norm='l2row'*)

Parameters

- **X** – A document-term matrix.
- **norm** – Normalization strategy: 'l2row': normalizes the scores of rows by length of rows after basic tfidf (each document vector is a unit vector), 'count': normalizes the scores of rows by the the total word count of a document. 'max' normalizes the scores of rows by the maximum count for a single word in a document.

Returns Returns tfidf of document-term matrix X with optional normalization.

loader.**toIndex**(*A*)

Parameters **A** – A matrix of one hot row vectors.

Returns The hot indices.

Examples

```
>>> import numpy
>>> from antk.core import loader
>>> x = numpy.array([[1,0,0], [0,0,1], [1,0,0]])
>>> loader.toIndex(x)
array([0, 2, 0])
```

loader.**toOnehot**(*X*, *dim=None*)

Parameters

- **X** – Vector of indices or *HotIndex* object
- **dim** – Dimension of indexing

Returns A sparse csr_matrix of one hots.

Examples

```
>>> import numpy
>>> from antk.core import loader
>>> x = numpy.array([0, 1, 2, 3])
>>> loader.toOnehot(x)
<4x4 sparse matrix of type '<type 'numpy.float64'>'...
>>> loader.toOnehot(x).toarray()
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> x = loader.HotIndex(x, dimension=8)
>>> loader.toOnehot(x).toarray()
array([[ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.]])
```

loader.**unit_variance**(*X*, *axis=None*)

> **Parameters** **X** – A matrix to transfrom to have unit variance (over columns axis=0, over rows axis=1, over all entries axis=None)
>
> **Returns** A matrix with unit variance along the specified axis.

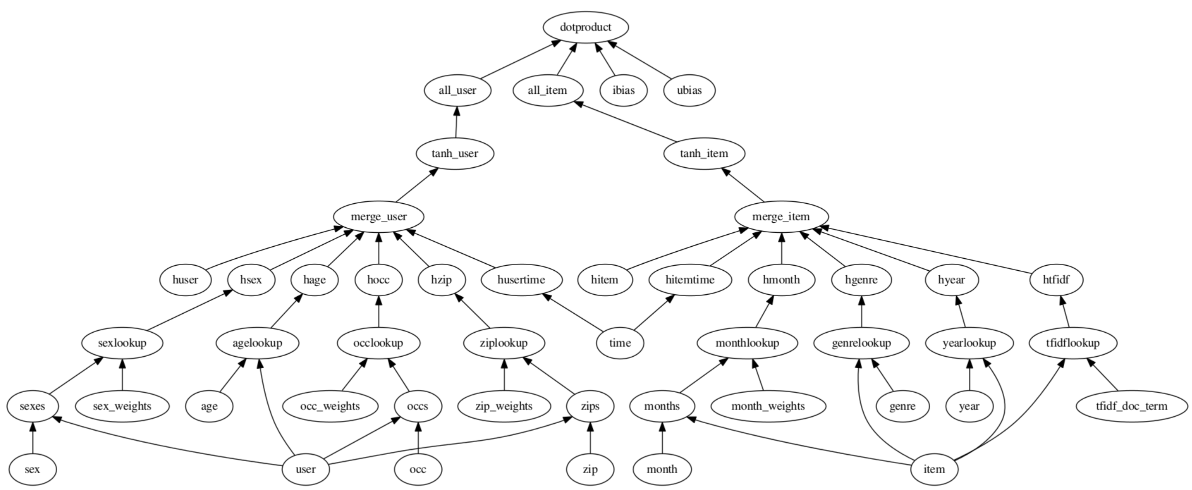
loader.**untar**(*fname*)

3.1.2 config

Facilitates the generation of complex tensorflow models, built from compositions of tensorflow functions.

Config Tutorial

The config module defines the *AntGraph* class. The basic idea is to represent any directed acyclic graph (DAG) of higher level tensorflow operations in a condensed and visually readable format. Here is a picture of a DAG of operations derived from it's representation in .config format:



Here are contents of the corresponding .config file:

```
dotproduct x_dot_y()
-all_user dnn([$kfactors,$kfactors,$kfactors], activation='tanh',bn=True,keep_prob=None)
--tanh_user tf.nn.tanh()
---merge_user concat($kfactors)
----huser lookup(dataname='user', initrange=$initrange, shape=[None, $kfactors])
----hage dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----agelookup embedding()
------age placeholder(tf.float32)
------user placeholder(tf.int32)
----hsex dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----sexlookup embedding()
------sex_weights weights('tnorm', tf.float32, [2, $kfactors])
------sexes embedding()
-------sex placeholder(tf.int32)
-------user placeholder(tf.int32)
----hocc dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----occlookup embedding()
```

```
------occ_weights weights('tnorm', tf.float32, [21, $kfactors])
------occs embedding()
-------occ placeholder(tf.int32)
-------user placeholder(tf.int32)
----hzip dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----ziplookup embedding()
------zip_weights weights('tnorm', tf.float32, [1000, $kfactors])
------zips embedding()
-------zip placeholder(tf.int32)
-------user placeholder(tf.int32)
----husertime dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----time placeholder(tf.float32)
-all_item dnn([$kfactors,$kfactors,$kfactors], activation='tanh',bn=True,keep_prob=None)
--tanh_item tf.nn.tanh()
---merge_item concat($kfactors)
----hitem lookup(dataname='item', initrange=$initrange, shape=[None, $kfactors])
----hgenre dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----genrelookup embedding()
------genre placeholder(tf.float32)
------item placeholder(tf.int32)
----hmonth dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----monthlookup embedding()
------month_weights weights('tnorm', tf.float32, [12, $kfactors])
------months embedding()
-------month placeholder(tf.int32)
-------item placeholder(tf.int32)
----hyear dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----yearlookup embedding()
------year placeholder(tf.float32)
------item placeholder(tf.int32)
----htfidf dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----tfidflookup embedding()
------tfidf_doc_term placeholder(tf.float32)
------item placeholder(tf.int32)
----hitemtime dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----time placeholder(tf.float32)
-ibias lookup(dataname='item', shape=[None, 1], initrange=$initrange)
-ubias lookup(dataname='user', shape=[None, 1], initrange=$initrange)
```

The lines in the .config file consist of a possibly empty graph marker, followed by a node name, followed by a node function call. We will discuss each of these in turn.

Terms

Node description: A line in a .config file

Graph marker: A character or sequence of characters that delimits graph dependencies. Specified by the graph marker p
for the constructor to AntGraph. By default '-'.

Node name: The first thing on a line in a .config file after a possibly empty sequence of graph markers and possible whitespace.

Node function: A function which takes as its first argument a tensor or structured list of tensors, returns
a tensor, or structured list of tensors, and has an optional name argument.

Node function call: The last item in a node description.

Graph Markers

In the .config file depicted above the graph marker is '-'. The graph markers in a .config file define the edges of the DAG. Lines in a .config file with no graph markers represent nodes with outorder = 0. These are the 'roots' of the DAG. The graph representation in .config format is similar to a textual tree or forest representation, however, multiple lines may refer to the same node. For each node description of a node, there is an edge from this node to the node described by the first line above of this node description that has one less graph marker.

Node Names

The next thing on a line following a possibly empty sequence of graph markers is the node name. Node names are used for unique variable scope of the tensors created by the node function call. The number of nodes in a graph

    is the number of unique

node names in the .config file.

Examples

The best way to get a feel for how to construct a DAG in this format is to try some things out. Since node function calls have no bearing on the high level structure of the computational graph let's simplify things and omit the node function calls for now. This won't be acceptable .config syntax but it will help us focus on the exploration of this form of graph representation.

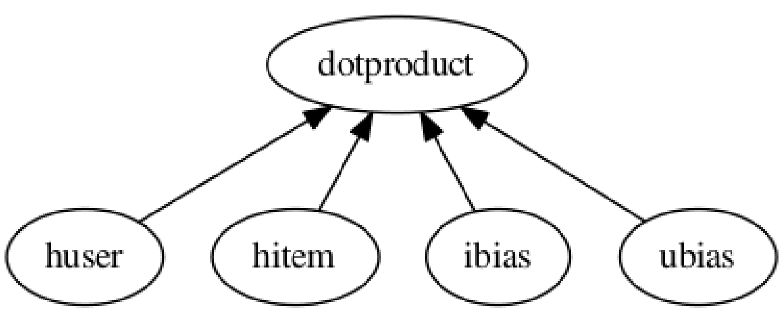
Here is a .config file minus the function calls (notice the optional whitespace before graph markers):

```
dotproduct
    -huser
    -hitem
    -ibias
    -ubias
```

Save this content in a file called test.config. Now in an interpreter:

```
>>>from antk.core import config
>>>config.testGraph('test.config')
```

This image should display:

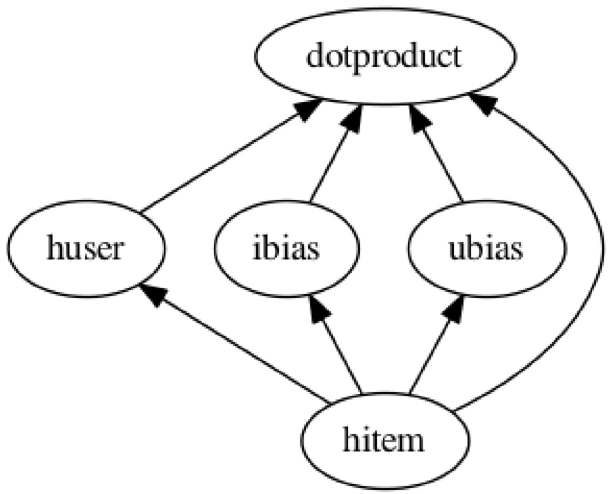


Now experiment with test.config to make some more graphs.

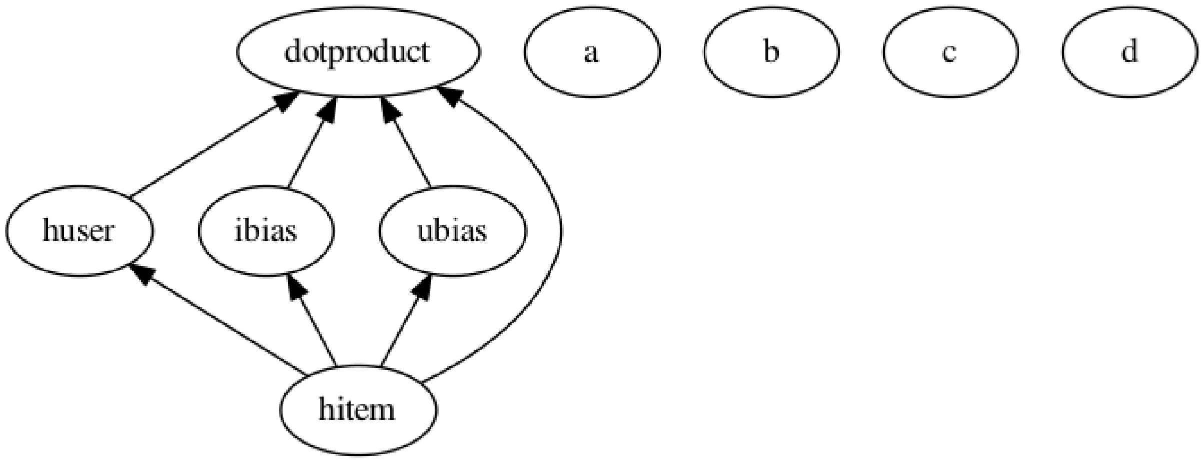
```
dotproduct
    -huser
        --hitem
    -ibias
        --hitem
```

```
6       -ubias
7           --hitem
8       -hitem
```

Note: Repeated Node Names Graph traversal proceeds in the fashion of a postorder tree traversal. When node names are repeated in a .config file, the output of this node is the output of the node description with this name which is first encountered in graph traversal. So, for the above example .config file and its corresponding picture below, the output of the hitem node would be the output of the node function call (omitted) on line 3. The order in which the nodes are evaluated for the config above is: **hitem**, **huser**, **ibias**, **ubias**, **dotproduct**.

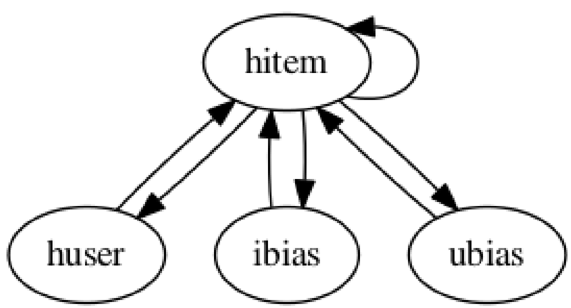



```
dotproduct
    -huser
        --hitem
    -ibias
        --hitem
    -ubias
        --hitem
    -hitem
a
b
c
d
```



Warning: **Cycles**: ANTk is designed to create directed acyclic graphs of operations from a config file, so cycles are not allowed. Below is an example of a config setup that describes a cycle. This config would cause an error, even if the node function calls were made with proper inputs.

```
hitem
    -huser
        --hitem
    -ibias
        --hitem
    -ubias
        --hitem
    -hitem
```



Node Functions

The first and only thing that comes after the name in a node description is a node function call. Node functions always take tensors or structured lists of tensors as input, return tensors or structured lists of tensors as output, and have an optional name argument. The syntax for a node function call in a .config is the same as calling the function in a python script, but omitting the first tensor input argument and the name argument. The tensor input is derived from the graph. A node's tensor input is a list of the output of it's 'child' nodes' (nodes with edges directed to this node) function calls. If a node has inorder = 1 then its input is a single tensor as opposed to a list of tensors of length 1.

Any node functions defined in node_ops may be used in a graph, as well as any tensorflow functions which satisfy the definition of a node function. For tensorflow node function calls 'tensorflow' is abbreviated to 'tf'. User defined node functions may be used in the graph when specified by the optional arguments *function_map*, and *imports*, to the `AntGraph` constructor.

The node name is used for the optional name argument of the node function.

The AntGraph object

To use a .config file to build a tensorflow computational graph you call the `AntGraph` constructor with the path to the .config file as the first argument, and some other optional arguments. We'll make the multinomial logistic regression model from tensorflow's basic MNIST tutorial, and then extend this model to a deep neural network in order to demonstrate how to use a .config file in your tensorflow code.

Create a file called antk_mnist.py and start off by importing the modules and data we need.

```
1 import tensorflow as tf
2 from antk.core import config
3 from tensorflow.examples.tutorials.mnist import input_data
4
5 mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

We'll need a config file called logreg.config with the content below:

```
pred mult_log_reg(numclasses=10)
-pixels placeholder(tf.float32)
```

Notice that we didn't specify any dimensions for the placeholder *pixels*. We need to hand a dictionary with keys corresponding to placeholders with unspecified dimensions, and values of the data that will later get fed to this placeholder during graph execution. This way the constructor will infer the shape of the placeholder. This practice can help eliminate a common source of errors in constructing a tensorflow graph. To instantiate the graph from this config file we add to antk_mnist.py:

```
6 with tf.name_scope('antgraph'):
7     antgraph = config.AntGraph('logreg.config', data={'pixels': mnist.test.images})
8 x = antgraph.placeholderdict['pixels']
9 y = antgraph.tensor_out
```

There are three accessible fields of a `AntGraph` object which contain tensors created during graph construction from a .config file:

- `tensordict`: a python dictionary of non-placeholder tensors.
- `placeholderdict`: a python dictionary of placeholder tensors.
- `tensor_out`: The output of the nodes of the graph with outorder 0 (no graph markers).

Note that we could replace line 9 above with the following:

```
9 y = antgraph.tensordict['pred']
```

We can now complete the simple MNIST model verbatim from the tensorflow tutorial:

```
10 y_ = tf.placeholder(tf.float32, [None, 10])
11
12 cross_entropy = -tf.reduce_sum(y_*tf.log(y))
13
14 train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
15
16 correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
17
18 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
19
20 # tensorboard stuff
21 accuracy_summary = tf.scalar_summary('Accuracy', accuracy)
```

```
22 session = tf.Session()
23 summary_writer = tf.train.SummaryWriter('log/logistic_regression', session.graph.as_graph_def())
24 session.run(tf.initialize_all_variables())
25
26 for i in range(1000):
27     batch_xs, batch_ys = mnist.train.next_batch(100)
28     session.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
29
30     acc, summary_str = session.run([accuracy, accuracy_summary], feed_dict={x: mnist.test.images,
31                                             y_: mnist.test.labels})
32     summary_writer.add_summary(summary_str, i)
33     print('epoch: %f acc: %f' % (float(i*100.0)/float(mnist.train.images.shape[0]), acc))
```

If we let antk_mnist.py take a command line argument for a .config file we can use antk_mnist.py with any number of .config files expressing arbitrarily complex architectures. This will allow us to quickly search for a better model. Let's use the argparse module to get this command line argument by adding the following lines to antk_mnist.py.

```
import argparse

parser = argparse.ArgumentParser(description="Model for training arbitrary MNIST digit recognition a
parser.add_argument("config", type=str,
                    help="The config file for building the ant architecture.")
args = parser.parse_args()
```

Now we change the former line 7 to:

```
antgraph = AntGraph(args.config, data={'pixels': mnist.dev.images})
```

We could try a neural network with nnet_mnist.config:

```
pred mult_log_reg(numclasses=10)
-network dnn([100,50,10], activation='tanh')
--pixels placeholder(tf.float32)
```

This should get us to about .94 accuracy. We might want to parameterize the number of hidden nodes per hidden layer or the activation function. For this we can use some more command line arguments, and the config file variable marker '$'.

First we change nnet_mnist.config as follows:

```
pred mult_log_reg(numclasses=10)
-network dnn([$h1, $h2, $h3], activation=$act)
--pixels placeholder(tf.float32)
```

Next we need some more command line arguments for antk_mnist.py. So we need to add these lines:

```
parser.add_argument("-h1", type=int,
                    help="Number of hidden nodes in layer 1.")
parser.add_argument("-h2", type=int,
                    help="Number of hidden nodes in layer 2.")
parser.add_argument("-h3", type=int,
                    help="Number of hidden nodes in layer 3.")
parser.add_argument("-act", type=int,
                    help="Type of activation function.")
```

Finally we need to bind the variables in the .config file in our call to the `AntGraph` constructor using the optional *variable_bindings* argument.

```
with tf.name_scope('antgraph'):
    antgraph = AntGraph(args.config, data={'pixels': mnist.dev.images},
```

```
                            variable_bindings={'h1': args.h1,
                                               'h2': args.h2,
                                               'h3': args.h3,
                                               'act': args.act})
```

For something really deep we might try a highway network with high_mnist.config:

```
pred mult_log_reg(numclasses=10)
-network3 dnn([50, 20])
--network2 highway_dnn([50]*20, activation='tanh', bn=True)
---network dnn([100, 50])
----pixels placeholder(tf.float32)
```

This may take 5 or 10 minutes to train but should get around .96 accuracy.

These higher level abstractions are nice for automating the creation of weight and bias Variables, and the Tensors involved a deep neural network architecture. However, one may need direct access to tensors created within a complex operation such as *highway_dnn*, to for instance analyze the training of a model. There is access to these tensors via a standard tensorflow function and some collections associated with each node defined in the .config file. To demonstrate accessing the tensors created by the *highway_dnn* operation in high_mnist.config, at the end of antk_mnist.py we can add:

```
weights = tf.get_collection('network')
bias = tf.get_collection('network_bias')
other = tf.get_collection('network')

for i, wght in enumerate(weights):
    print('weight %d: name=%s tensor=%s' % (i, wght.name, wght))
for i, b in enumerate(bias):
    print('bias %d: name=%s tensor=%s' % (i, b.name, b))
for i, tensor in enumerate(other):
    print('other %d: name=%s tensor=%s' % (i, tensor.name, tensor))
```

And post training we get the following output modulo two memory addresses:

```
weight 0: name=antgraph/network/layer0/add:0 tensor=Tensor("antgraph/network/layer0/add:0", shape=(?,
weight 1: name=antgraph/network/layer1/add:0 tensor=Tensor("antgraph/network/layer1/add:0", shape=(?,
bias 0: name=network/layer0/network/Bias:0 tensor=<tensorflow.python.ops.variables.Variable object at
bias 1: name=network/layer1/network/Bias:0 tensor=<tensorflow.python.ops.variables.Variable object at
other 0: name=antgraph/network/layer0/add:0 tensor=Tensor("antgraph/network/layer0/add:0", shape=(?,
other 1: name=antgraph/network/layer1/add:0 tensor=Tensor("antgraph/network/layer1/add:0", shape=(?,
```

class `config.`**`AntGraph`**(*config, tensordict={}, placeholderdict={}, data=None, function_map={}, imports={}, marker='-', variable_bindings=None, graph_name='no_name', graph_dest='antpics/', develop=False*)

Object to store graph information from graph built with config file.

Parameters

- **`config`** – A plain text config file
- **`tensordict`** – A dictionary of premade tensors represented in the config by key
- **`placeholderdict`** – A dictionary of premade placeholder tensors represented in the config by key
- **`data`** – A dictionary of data matrices with keys corresponding to placeholder names in graph.
- **`function_map`** – A dictionary of function_handle:node_op pairs to use in building the graph

- **imports** – A dictionary of module_name:path_to_module key value pairs for custom node_ops modules.
- **marker** – The marker for representing graph structure
- **variable_bindings** – A dictionary with entries of the form *variable_name:value* for variable replacement in config file.
- **graph_name** – The name of the graph. Will be used to name the graph pdf file.
- **graph_dest** – The folder to write the graph pdf and graph dot string to.
- **develop** – True|False. Whether to print tensor info, while constructing the tensorflow graph.

Attributes

Methods

display_graph(*pdfviewer='okular'*)
Display the pdf image of graph from config file to screen.

get_array(*collection_name*, *index*, *session*, *graph*)

placeholderdict
A dictionary of tensors which are placeholders in the graph. The key should correspond to the key of the corresponding data in a data dictionary.

tensor_out
Tensor or list of tensors returned from last node of graph.

tensordict
A dictionary of tensors which are nodes in the graph.

exception config.**GraphMarkerError**
Raised when leading character of a line (other than first) in a graph config file is not the specified level marker.

exception config.**MissingDataError**
Raised when data needed to determine shapes is not found in the *DataSet*.

exception config.**MissingTensorError**
Raised when a tensor is described by name only in the graph and it is not in a dictionary.

exception config.**ProcessLookupError**
Raised when lookup receives a dataname argument without a corresponding value in it's *DataSet* and there is not already a Placeholder with that name.

exception config.**RandomNodeFunctionError**
Raised when something strange happened with a node function call.

exception config.**UndefinedVariableError**
Raised when a a variable in config is not a key in variable_bindings map handed to graph_setup.

exception config.**UnsupportedNodeError**
Raised when a config file calls a function that is not defined, i.e., has not been imported, or is not in the node_ops base file.

config.**ph_rep**(*ph*)
Convenience function for representing a tensorflow placeholder.

Parameters **ph** – A tensorflow placeholder.

Returns A string representing the placeholder.

`config.`**`testGraph`**(*config*, *marker='-'*, *graph_dest='antpics/'*, *graph_name='test_graph'*)

Parameters

- **`config`** – A graph specification in .config format.
- **`marker`** – A character or string of characters to delimit graph edges.
- **`graph_dest`** – Where to save the graphviz pdf and associated dot file.
- **`graph_name`** – A name for the graph (without extension)

3.1.3 node_ops

The node_ops module consists of a collection of mid to high level functions which take a tensor or structured list of tensors, perform a sequence of tensorflow operations, and return a tensor or structured list of tensors. All node_ops functions conform to the following specifications.

- All tensor input (if it has tensor input) is received by the function's first argument, which may be a single tensor, a list of tensors, or a structured list of tensors, e.g., a list of lists of tensors.
- The return is a tensor, list of tensors or structured list of tensors.
- The final argument is an optional *name* argument for variable_scope.

Use Cases

node_ops functions may be used in a tensorflow script wherever you might use an equivalent sequence of tensorflow ops during the graph building portion of a script.

node_ops functions may be called in a .config file following the .config file syntax which is explained in Config Tutorial.

Making Custom ops For use With *config* module

The *`AntGraph`* constructor in the *config* module will add tensor operations to the tensorflow graph which are specified in a config file and fit the node_ops spec but not defined in the *node_ops* module. This leaves the user free to define new node_ops for use with the config module, and to use many pre-existing tensorflow and third party defined ops with the config module as well.

The *`AntGraph`* constructor has two arguments *function_map* and *imports* which may be used to incorporate custom node_ops.

- **function_map** is a hashmap of function_handle:function, key value pairs
- **imports** is a hashmap of module_name:path_to_module pairs for importing an entire module of custom node_ops.

Accessing Tensors Created in a node_ops Function

Tensors which are created by a node_ops function but not returned to the caller are kept track of in an intuitive fashion by calls to **tf.add_to_collection**. Tensors can be accessed later by calling **tf.get_collection** by the following convention:

For a node_ops function which was handed the argument **name='some_name'**:

- The **nth weight tensor** created may be accessed as

```
tf.get_collection('some_name_weights')[n]
```

- The **nth bias tensor** created may be accessed as

```
tf.get_collection('some_name_bias')[n]
```

- The **nth preactivation tensor** created may be accessed as

```
tf.get_collection('some_name_preactivation')[n]
```

- The **nth activation tensor** created may be accessed as

```
tf.get_collection('some_name_activations')[n]
```

- The **nth post dropout** tensor created may be accessed as

```
tf.get_collection('some_name_dropouts')[n]
```

- The **nth post batch normalization tensor** created may be accessed as

```
tf.get_collection('some_name_bn')[n]
```

- The **nth tensor created not listed above** may be accessed as

```
tf.get_collection('some_name')[n],
```

- The **nth hidden layer size skip transform** (for `residual_dnn`):

```
tf.get_collection('some_name_skiptransform')[n]
```

- The **nth skip connection** (for `residual_dnn`):

```
tf.get_collection('some_name_skipconnection')[n]
```

- The **nth transform layer** (for `highway_dnn`):

```
tf.get_collection('some_name_transform')[n]
```

Weights

Here is a simple wrapper for common initializations of tensorflow **'Variables'_**. There is a option for l2 regularization which is automatically added to the objective function when using the generic_model module.

`weights`

Placeholders

Here is a simple wrapper for a tensorflow placeholder constructor that when used in conjunction with the config module, infers the correct dimensions of the placeholder from a string hashed set of numpy matrices.

`placeholder`

Neural Networks

> **Warning:** The output of a neural network node_ops function is the output after activation of the last hidden layer. For regression an additional call to `linear` must be made and for classification and additional call to `mult_log_reg` must be made.

Initialization

Neural network weights are initialized with the following scheme where the range is dependent on the second dimension of the input layer:

```
if activation == 'relu':
   irange= initrange*numpy.sqrt(2.0/float(tensor_in.get_shape().as_list()[1]))
else:
   irange = initrange*(1.0/numpy.sqrt(float(tensor_in.get_shape().as_list()[1])))
```

initrange above is defaulted to 1. The user has the choice of several distributions,

- 'norm', 'tnorm': *irange* scales distribution with mean zero and standard deviation 1.
- 'uniform': *irange* scales uniform distribution with range [-1, 1].
- 'constant': *irange* equals the initial scalar entries of the matrix.

Dropout

Dropout with the specified *keep_prob* is performed post activation.

Batch Normalization

If requested batch normalization is performed after dropout.

Networks

`dnn`

`residual_dnn`

`highway_dnn`

`convolutional_net`

Loss Functions and Evaluation Metrics

`se`

`mse`

`rmse`

`mae`

`cross_entropy`

`other_cross_entropy`

perplexity

detection

recall

precision

accuracy

fscore

Custom Activations

ident

tanhlecun

mult_log_reg

Matrix Operations

concat

x_dot_y

cosine

linear

embedding

lookup

khatri_rao

Tensor Operations

Some tensor operations from Kolda and Bader's *Tensor Decompositions and Applications* are provided here. For now these operations only work on up to order 3 tensors.

nmode_tensor_tomatrix

nmode_tensor_multiply

binary_tensor_combine

ternary_tensor_combine

Batch Normalization

batch_normalize

Dropout

Dropout is automatically 'turned' off during evaluation when used in conjuction with the generic_model module.

dropout

API

exception `node_ops.`**`MissingShapeError`**
Raised when *`placeholder`* can not infer shape.

`node_ops.`**`accuracy`**`(`**args*, ***kwargs*`)`

`node_ops.`**`batch_normalize`**`(`**args*, ***kwargs*`)`

Batch Normalization: Adapted from tensorflow nn.py and skflow batch_norm_ops.py . Batch Normalization Accelerating Deep Network Training by Reducing Internal Covariate Shift

Parameters

- **`tensor_in`** – input Tensor
- **`epsilon`** – A float number to avoid being divided by 0.
- **`name`** – For variable_scope

Returns Tensor with variance bounded by a unit and mean of zero according to the batch.

`node_ops.`**`binary_tensor_combine`**`(`**args*, ***kwargs*`)`
For performing tensor multiplications with batches of data points against an order 3 weight tensor.

Parameters

- **`tensors`** – A list of two matrices each with first dim batch-size
- **`output_dim`** – The dimension of the third mode of the weight tensor
- **`initrange`** – For initializing weight tensor
- **`name`** – For variable scope

Returns A matrix with shape batch_size X output_dim

`node_ops.`**`binary_tensor_combine2`**`(`**args*, ***kwargs*`)`

`node_ops.`**`concat`**`(`**args*, ***kwargs*`)`
Matrix multiplies each tensor in *tensors* by its own weight matrix and adds together the results.

Parameters

- **`tensors`** – A list of tensors.
- **`output_dim`** – Dimension of output
- **`name`** – An optional identifier for unique variable_scope.

Returns A tensor with shape [None, output_dim]

`node_ops.`**`cosine`**`(`**args*, ***kwargs*`)`
Takes the cosine of vectors in corresponding rows of the two matrix tensors in operands.

Parameters

- **`operands`** – A list of two tensors to take cosine of.
- **`name`** – An optional name for unique variable scope.

Returns A tensor with dimensions (operands[0].shape[0], 1)

Raises ValueError when operands do not have matching shapes.

`node_ops.`**`cross_entropy`**`(`**args*, ***kwargs*`)`

`node_ops.`**`detection`**`(`**args*, ***kwargs*`)` node_ops.**dropout** (**args*, ***kwargs*)

Adds dropout node. Adapted from skflow dropout_ops.py . Dropout A Simple Way to Prevent Neural Networks from Overfitting

Parameters

- **tensor_in** – Input tensor.
- **prob** – The percent of weights to keep.
- **name** – A name for the tensor.

Returns Tensor of the same shape of *tensor_in*.

node_ops.**embedding** (**args*, ***kwargs*)
A wrapper for tensorflow's embedding_lookup

Parameters

- **tensors** – A list of two tensors , matrix, indices
- **name** – Unique name for variable scope

Returns A matrix tensor where the i-th row = matrix[indices[i]]

node_ops.**fan_scale** (*initrange*, *activation*, *tensor_in*)

node_ops.**fscore** (**args*, ***kwargs*)

node_ops.**ident** (*tensor_in*, *name='ident'*)
Identity function for grouping tensors in graph, during config parsing.

Parameters **tensor_in** – A Tensor or list of tensors

Returns tensor_in

node_ops.**khatri_rao** (**args*, ***kwargs*)
From David Palzer

Parameters

- **tensors** –
- **name** –

Returns

node_ops.**linear** (**args*, ***kwargs*)
Linear map: $\sum_i(args[i] * W_i)$, where W_i is a variable.

Parameters

- **args** – a 2D Tensor
- **output_size** – int, second dimension of W[i].
- **bias** – boolean, whether to add a bias term or not.
- **bias_start** – starting value to initialize the bias; 0 by default.
- **distribution** – Distribution for lookup weight initialization
- **initrange** – Initrange for weight distribution.
- **l2** – Floating point number determining degree of of l2 regularization for these weights in gradient descent update.

- **name** – VariableScope for the created subgraph; defaults to "Linear".

Returns A 2D Tensor with shape [batch x output_size] equal to $\sum_i (args[i] * W_i)$, where W_i are newly created matrices.

Raises ValueError: if some of the arguments has unspecified or wrong shape.

`node_ops.`**`lookup`**`(`**args*, ***kwargs*`)`

A wrapper for tensorflow's embedding_lookup which infers the shape of the weight matrix and placeholder value from the parameter *data*.

Parameters

- **dataname** – Used exclusively by config.py
- **data** – A `HotIndex` object
- **indices** – A Placeholder. If indices is none the dimensions will be inferred from *data*
- **distribution** – Distribution for lookup weight initialization
- **initrange** – Initrange for weight distribution.
- **l2** – Floating point number determining degree of of l2 regularization for these weights in gradient descent update.
- **shape** – The dimensions of the output tensor, typically [None, output-size]
- **makeplace** – A boolean to tell whether or not a placeholder has been created for this data (Used by config.py)
- **name** – A name for unique variable scope.

Returns tf.nn.embedding_lookup(wghts, indices), wghts, indices

`node_ops.`**`mae`**`(`**args*, ***kwargs*`)`

Mean Absolute Error

`node_ops.`**`mse`**`(`**args*, ***kwargs*`)`

Mean Squared Error.

`node_ops.`**`mult_log_reg`**`(`**args*, ***kwargs*`)`

Performs mulitnomial logistic regression forward pass. Weights and bias initialized to zeros.

Parameters

- **tensor_in** – A tensor or placeholder
- **numclasses** – For classificatio
- **data** – For shape inference.
- **dtype** – For `weights` initialization.
- **initrange** – For `weights` initialization.
- **seed** – For `weights` initialization.
- **l2** – For `weights` initialization.
- **name** – For variable_scope

Returns A tensor shape=(tensor_in.shape[0], numclasses)

`node_ops.`**`nmode_tensor_multiply`**`(`**args*, ***kwargs*`)`

Nth mode tensor multiplication (for order three tensor) from Kolda and Bader Tensor Decompositions and Applications Works for vectors (matrix with a 1 dimension or matrices)

Parameters

- **tensors** – A list of tensors the first is an order three tensor the second and order 2
- **mode** – The mode to perform multiplication against.
- **leave_flattened** – Whether or not to reshape tensor back to order 3
- **keep_dims** – Whether or not to remove 1 dimensions
- **name** – For variable scope

Returns Either an order 3 or order 2 tensor

node_ops.**nmode_tensor_tomatrix**(**args*, ***kwargs*)

Nmode tensor unfolding (for order three tensor) from Kolda and Bader Tensor Decompositions and Applications

Parameters

- **tensor** – Order 3 tensor to unfold
- **mode** – Mode to unfold (0,1,2, columns, rows, or fibers)
- **name** – For variable scoping

Returns A matrix (order 2 tensor) with shape dim(mode) X $\Pi_{othermodes}$ dim(othermodes)

node_ops.**other_cross_entropy**(**args*, ***kwargs*)

Logistic Loss

node_ops.**perplexity**(**args*, ***kwargs*)

node_ops.**placeholder**(**args*, ***kwargs*)

Wrapper to create tensorflow Placeholder which infers dimensions given data.

Parameters

- **dtype** – Tensorflow dtype to initiliaze a Placeholder.
- **shape** – Dimensions of Placeholder
- **data** – Data to infer dimensions of Placeholder from.
- **name** – Unique name for variable scope.

Returns A Tensorflow Placeholder.

node_ops.**precision**(**args*, ***kwargs*)

Percentage of classes detected which are correct.

Parameters

- **targets** – A one hot encoding of class labels (num_points X numclasses)
- **predictions** – A real valued matrix with indices ranging between zero and 1 (num_points X numclasses)
- **threshold** – The detection threshold (between zero and 1)
- **detects** – In case detection is precomputed for efficiency when evaluating both precision and recall

Returns A scalar value

node_ops.**recall**(**args*, ***kwargs*)

Percentage of actual classes predicted

Parameters

- **targets** – A one hot encoding of class labels (num_points X numclasses)
- **predictions** – A real valued matrix with indices ranging between zero and 1 (num_points X numclasses)
- **threshold** – The detection threshold (between zero and 1)
- **detects** – In case detection is precomputed for efficiency when evaluating both precision and recall

Returns A scalar value

node_ops.**rmse**(**args*, ***kwargs*)
Root Mean Squared Error

node_ops.**se**(**args*, ***kwargs*)
Squared Error.

node_ops.**ternary_tensor_combine**(**args*, ***kwargs*)
For performing tensor multiplications with batches of data points against an order 3 weight tensor.

Parameters

- **tensors** –
- **output_dim** –
- **initrange** –
- **name** –

Returns

node_ops.**weights**(**args*, ***kwargs*)
Wrapper parameterizing common constructions of tf.Variables.

Parameters

- **distribution** – A string identifying distribution ‘tnorm’ for truncated normal, ‘rnorm’ for random normal, ‘constant’ for constant, ‘uniform’ for uniform.
- **shape** – Shape of weight tensor.
- **dtype** – dtype for weights
- **initrange** – Scales standard normal and trunctated normal, value of constant dist., and range of uniform dist. [-initrange, initrange].
- **seed** – For reproducible results.
- **l2** – Floating point number determining degree of of l2 regularization for these weights in gradient descent update.
- **name** – For variable scope.

Returns A tf.Variable.

node_ops.**x_dot_y**(**args*, ***kwargs*)
Takes the inner product for rows of operands[1], and operands[2], and adds optional bias, operands[3], operands[4]. If either operands[1] or operands[2] or both is a list of tensors then a list of the pairwise dot products (with bias when len(operands) > 2) of the lists is returned.

Parameters

- **operands** – A list of 2, 3, or 4 tensors (the first two tensors may be replaced by lists of tensors in which case the return value will a list of the dot products for all members of the cross product of the two lists.).

- **name** – An optional identifier for unique variable_scope.

Returns A tensor or list of tensors with dimension (operands[1].shape[0], 1).

Raises Value error when operands is not a list of at least two tensors.

3.1.4 generic_model

A general purpose model builder equipped with generic train, and predict functions which takes parameters for optimization strategy, mini-batch, etc...

class generic_model.**Model**(*objective, placeholderdict, maxbadcount=20, momentum=None, mb=1000, verbose=True, epochs=50, learnrate=0.003, save=False, opt='grad', decay=[1, 1.0], evaluate=None, predictions=None, logdir='log/', random_seed=None, model_name='generic', clip_gradients=0.0, make_histograms=False, best_model_path='/tmp/model.ckpt', save_tensors={}, tensorboard=False, train_evaluate=None*)

Generic model builder for training and predictions.

Parameters

- **objective** – Loss function
- **placeholderdict** – A dictionary of placeholders
- **maxbadcount** – For early stopping
- **momentum** – The momentum for tf.MomentumOptimizer
- **mb** – The mini-batch size
- **verbose** – Whether to print dev error, and save_tensor evals
- **epochs** – maximum number of epochs to train for.
- **learnrate** – learnrate for gradient descent
- **save** – Save best model to *best_model_path*.
- **opt** – Optimization strategy. May be 'adam', 'ada', 'grad', 'momentum'
- **decay** – Parameter for decaying learn rate.
- **evaluate** – Evaluation metric
- **predictions** – Predictions selected from feed forward pass.
- **logdir** – Where to put the tensorboard data.
- **random_seed** – Random seed for TensorFlow initializers.
- **model_name** – Name for model
- **clip_gradients** – The limit on gradient size. If 0.0 no clipping is performed.
- **make_histograms** – Whether or not to make histograms for model weights and activations
- **best_model_path** – File to save best model to during training.
- **save_tensors** – A hashmap of str:Tensor mappings. Tensors are evaluated during training. Evaluations of these tensors on best model are accessible via property *evaluated_tensors*.
- **tensorboard** – Whether to make tensorboard histograms of weights and activations, and graphs of dev_error.

Returns *Model*

Attributes

Methods

average_secs_per_epoch
The average number of seconds to complete an epoch.

best_completed_epochs
Number of epochs completed during at point of best dev eval during training (fractional)

best_dev_error
The best dev error reached during training.

completed_epochs
Number of epochs completed during training (fractional)

eval (*tensor_in*, *data*, *supplement=None*)
Evaluation of model.

Parameters **data** – *DataSet* to evaluate on.

Returns Result of evaluating on data for `self.evaluate`

evaluated_tensors
A dictionary of evaluations on best model for tensors and keys specified by *save_tensors* argument to constructor.

placeholderdict
Dictionary of model placeholders

plot_train_dev_eval (*figure_file='testfig.pdf'*)

predict (*data*, *supplement=None*)

Parameters **data** – *DataSet* to make predictions from.

Returns A set of predictions from feed forward defined by `self.predictions`

train (*train*, *dev=None*, *supplement=None*, *eval_schedule='epoch'*, *train_dev_eval_factor=3*)

Parameters **data** – *DataSet* to train on.

Returns A trained *Model*

`generic_model.`**get_feed_list** (*batch*, *placeholderdict*, *supplement=None*, *dropouts=None*, *dropout_flag='train'*)

Parameters

- **batch** – A dataset object.
- **placeholderdict** – A dictionary where the keys match keys in batch, and the values are placeholder tensors
- **supplement** – A dictionary of numpy input matrices with keys corresponding to placeholders in placeholderdict, where the row size of the matrices do not correspond to the number of datapoints. For use with input data intended for embedding_lookup.
- **dropouts** – Dropout tensors in graph.
- **dropout_flag** – Whether to use Dropout probabilities for feed forward.

Returns A feed dictionary with keys of placeholder tensors and values of numpy matrices, paired by key

generic_model.**parse_summary_val**(*summary_str*)
Helper function to parse numeric value from tf.scalar_summary

Parameters **summary_str** – Return value from running session on tf.scalar_summary

Returns A dictionary containing the numeric values.

3.1.5 Models

The models below are available in ANTk. If the model takes a config file then a sample config is provided.

Skipgram

class skipgram.**SkipGramVecs**(*textfile*, *vocabulary_size=12735*, *batch_size=128*, *embedding_size=128*, *skip_window=1*, *num_skips=2*, *valid_size=16*, *valid_window=100*, *num_sampled=64*, *num_steps=100000*, *verbose=False*)
Trains a skip gram model from Distributed Representations of Words and Phrases and their Compositionality

Parameters

- **textfile** – Plain text file or zip file with plain text files.
- **vocabulary_size** – How many words to use from text
- **batch_size** – mini-batch size
- **embedding_size** – Dimension of the embedding vector.
- **skip_window** – How many words to consider left and right.
- **num_skips** – How many times to reuse an input to generate a label.
- **valid_size** – Random set of words to evaluate similarity on.
- **valid_window** – Only pick dev samples in the head of the distribution.
- **num_sampled** – Number of negative examples to sample.
- **num_steps** – How many mini-batch steps to take
- **verbose** – Whether to calculate and print similarities for a sample of words

Methods

plot_embeddings(*filename='tsne.png'*, *num_terms=500*)
Plot tsne reduction of learned word embeddings in 2-space.

Parameters

- **filename** – File to save plot to.
- **num_terms** – How many words to plot.

skipgram.**build_dataset**(*words*, *vocabulary_size*)

Parameters

- **words** – A list of word tokens from a text file
- **vocabulary_size** – How many word tokens to keep.

Returns data (text transformed into list of word ids 'UNK'=0), count (list of pairs (word:word_count) indexed by word id), dictionary (word:id hashmap), reverse_dictionary (id:word hashmap)

`skipgram.`**`generate_batch`**(*data*, *batch_size*, *num_skips*, *skip_window*)

Parameters

- **`data`** – list of word ids corresponding to text
- **`batch_size`** – Size of batch to retrieve
- **`num_skips`** – How many times to reuse an input to generate a label.
- **`skip_window`** – How many words to consider left and right.

Returns

`skipgram.`**`plot_tsne`**(*embeddings*, *labels*, *filename='tsne.png'*, *num_terms=500*)

Makes tsne plot to visualize word embeddings. Need sklearn, matplotlib for this to work.

Parameters

- **`filename`** – Location to save labeled tsne plots
- **`num_terms`** – Num of words to plot

`skipgram.`**`read_data`**(*filename*)

Parameters **`filename`** – A zip file to open and read from

Returns A list of the space delimited tokens from the textfile.

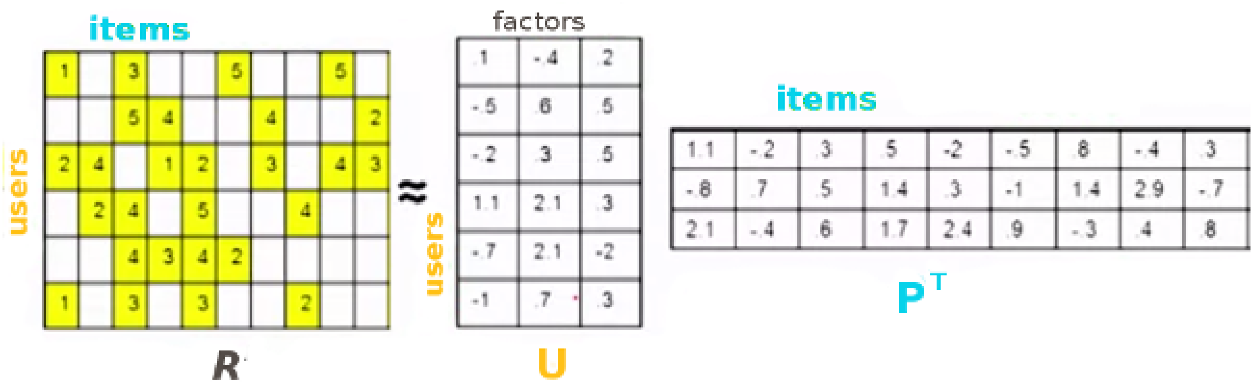
Matrix Factorization

`mfmodel.`**`mf`**(*data*, *configfile*, *lamb=0.001*, *kfactors=20*, *learnrate=0.01*, *verbose=True*, *epochs=1000*, *maxbadcount=20*, *mb=500*, *initrange=1*, *eval_rate=500*, *random_seed=None*, *develop=False*, *train_dev_eval_factor=3*)

Sample Config

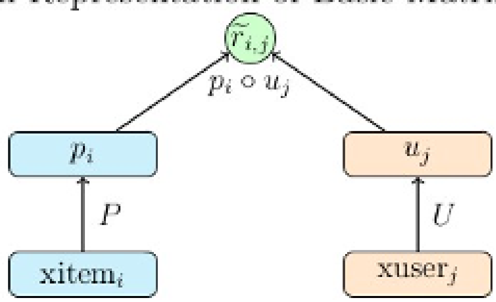
```
dotproduct x_dot_y()
    -huser lookup(dataname='user', initrange=0.001, shape=[None, 20])
    -hitem lookup(dataname='item', initrange=0.001, shape=[None, 20])
    -ibias lookup(dataname='item', initrange=0.001, shape=[None, 1])
    -ubias lookup(dataname='user', initrange=0.001, shape=[None, 1])
```

Low Rank Matrix Factorization is a popular machine learning technique used to produce recommendations given a set of ratings a user has given an item. The known ratings are collected in a user-item utility matrix and the missing entries are predicted by optimizing a low rank factorization of the utility matrix given the known entries. The basic idea behind matrix factorization models is that the information encoded for items in the columns of the utility matrix, and for users in the rows of the utility matrix is not exactly independent. We optimize the objective function $\sum_{(u,i)}(R_{ui} - P_i^T U_u)^2$ over the observed ratings for user *u* and item *i* using gradient descent.



We can express the same optimization in the form of a computational graph that will play nicely with tensorflow:

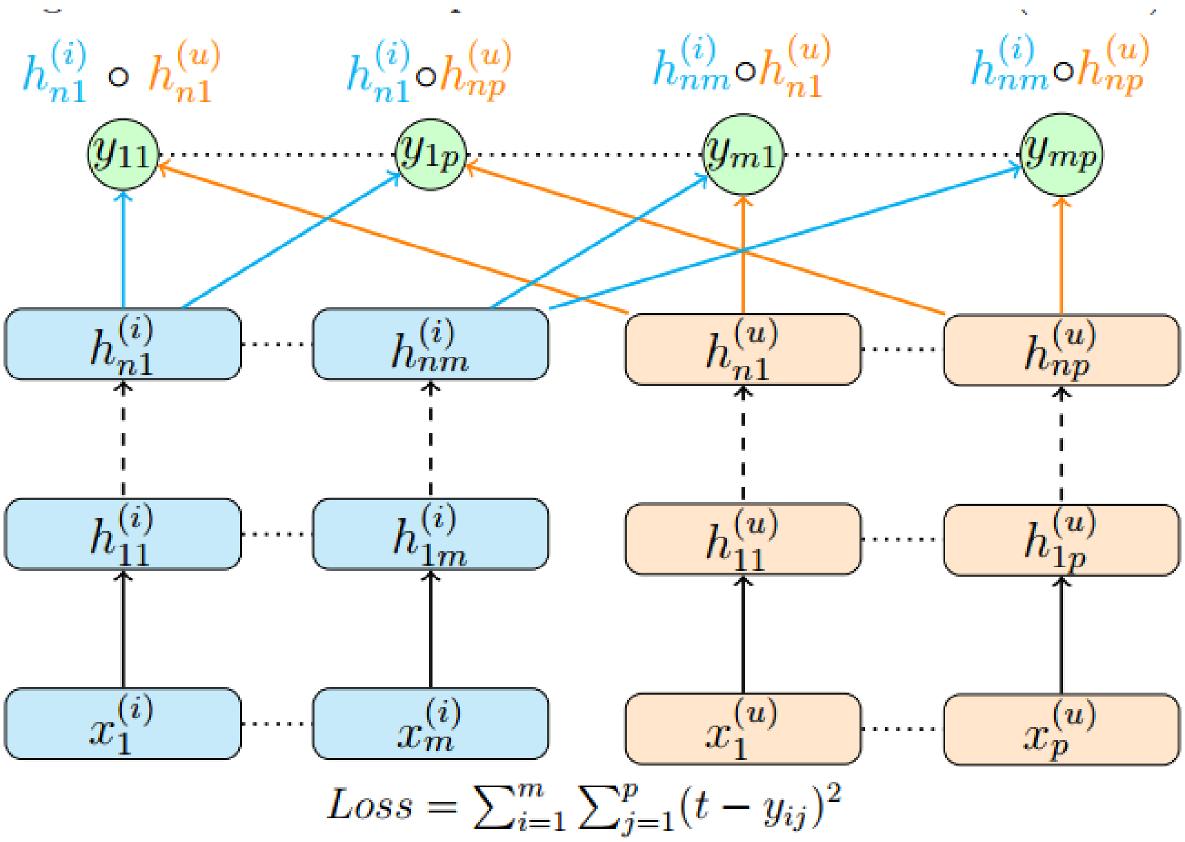
Figure 1: Graph Representation of Basic Matrix Factorization



Here $xitem_i$, and $xuser_j$ are some representation of the indices for the user and item vectors in the utility matrix. These could be one hot vectors, which can then be matrix multiplied by the *P* and *U* matrices to select the corresponding user and item vectors. In practice it is much faster to let $xitem_i$, and $xuser_j$ be vectors of indices which can be used by tensorflow's **gather** or **embedding_lookup** functions to select the corresponding vector from the *P* and *U* matrices.

DSSM (Deep Structured Semantic Model) Variant

`dssm_model.`**`dssm`**(*data, configfile, layers=[10, 10, 10], bn=True, keep_prob=0.95, act='tanhlecun', initrange=1, kfactors=10, lamb=0.1, mb=500, learnrate=0.0001, verbose=True, maxbadcount=10, epochs=100, model_name='dssm', random_seed=500, eval_rate=500*)



Sample Config

```
dotproduct x_dot_y()
-user_vecs ident()
--huser lookup(dataname='user', initrange=$initrange, shape=[None, $kfactors])
--hage dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=.8)
---agelookup embedding()
----age placeholder(tf.float32)
----user placeholder(tf.int32)
--hsex dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
---sexlookup embedding()
----sex_weights weights('tnorm', tf.float32, [2, $kfactors])
----sexes embedding()
-----sex placeholder(tf.int32)
-----user placeholder(tf.int32)
--hocc dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
---occlookup embedding()
----occ_weights weights('tnorm', tf.float32, [21, $kfactors])
----occs embedding()
-----occ placeholder(tf.int32)
-----user placeholder(tf.int32)
--hzip dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
---ziplookup embedding()
----zip_weights weights('tnorm', tf.float32, [1000, $kfactors])
----zips embedding()
-----zip placeholder(tf.int32)
```

```
-----user placeholder(tf.int32)
--husertime dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
---time placeholder(tf.float32)
-item_vecs ident()
--hitem lookup(dataname='item', initrange=$initrange, shape=[None, $kfactors])
--hgenre dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
---genrelookup embedding()
----genres placeholder(tf.float32)
----item placeholder(tf.int32)
--hmonth dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
---monthlookup embedding()
----month_weights weights('tnorm', tf.float32, [12, $kfactors])
----months embedding()
-----month placeholder(tf.int32)
-----item placeholder(tf.int32)
--hyear dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
---yearlookup embedding()
----year placeholder(tf.float32)
----item placeholder(tf.int32)
--htfidf dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
---tfidflookup embedding()
----tfidf_doc_term placeholder(tf.float32)
----item placeholder(tf.int32)
--hitemtime dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
---time placeholder(tf.float32)
-ibias lookup(dataname='item', shape=[None, 1], initrange=$initr
```

Weighted DSSM variant

`dsaddmodel.`**`dsadd`**(*data*, *configfile*, *initrange=0.1*, *kfactors=20*, *lamb=0.01*, *mb=500*, *learnrate=0.003*, *verbose=True*, *maxbadcount=10*, *epochs=100*, *model_name='dssm'*, *random_seed=500*, *eval_rate=500*)

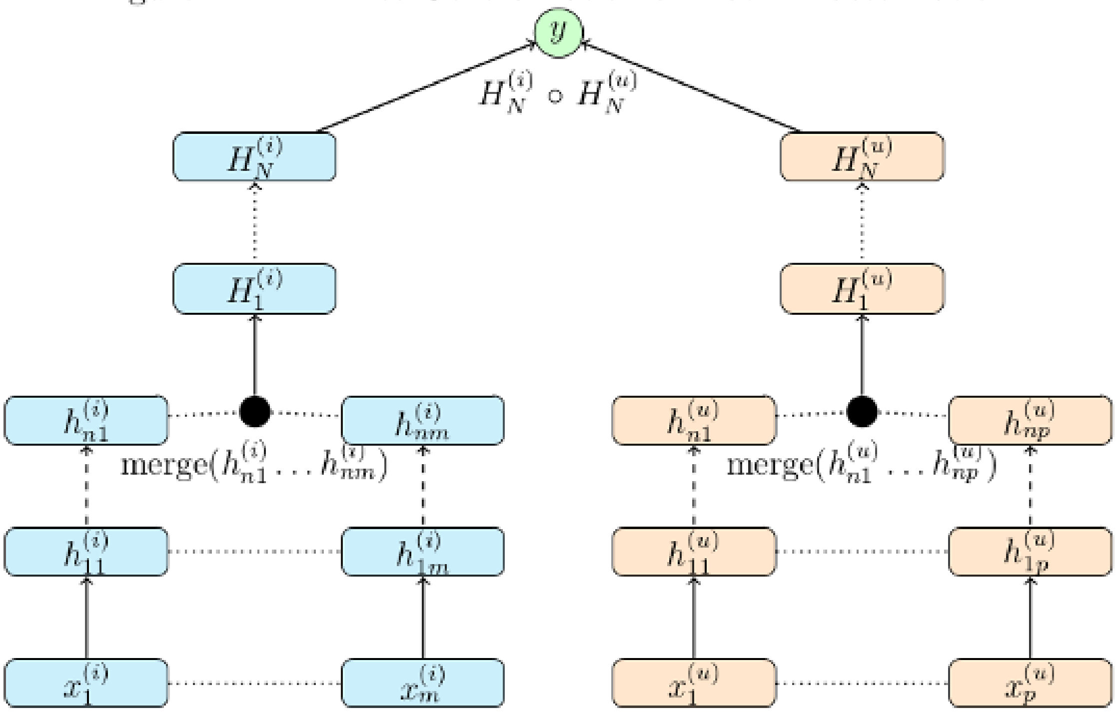
This model is the same architecture as the variant of DSSM above but with a different loss:

$$Loss = (t - \sum_{i=1}^{m} \sum_{j=1}^{p} w_{ij} y_{ij})^2$$

Binary Tree of Deep Neural Networks for Multiple Inputs

`tree_model.`**`tree`**(*data*, *configfile*, *lamb=0.001*, *kfactors=20*, *learnrate=0.0001*, *verbose=True*, *maxbadcount=20*, *mb=500*, *initrange=1e-05*, *epochs=10*, *random_seed=None*, *eval_rate=500*, *keep_prob=0.95*, *act='tanh'*)

Figure 2: DNN Tree Generalization of Matrix Factorization



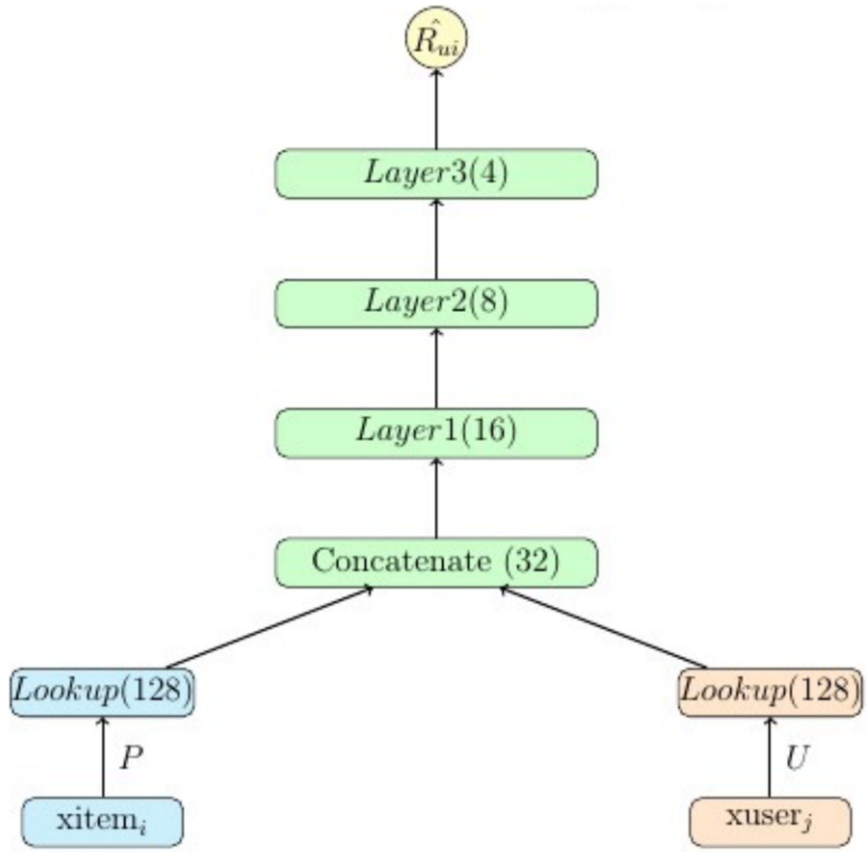
Sample Config

```
dotproduct x_dot_y()
-all_user dnn([$kfactors,$kfactors,$kfactors], activation='tanh',bn=True,keep_prob=None)
--tanh_user tf.nn.tanh()
---merge_user concat($kfactors)
----huser lookup(dataname='user', initrange=$initrange, shape=[None, $kfactors])
----hage dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----agelookup embedding()
------age placeholder(tf.float32)
------user placeholder(tf.int32)
----hsex dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----sexlookup embedding()
------sex_weights weights('tnorm', tf.float32, [2, $kfactors])
------sexes embedding()
-------sex placeholder(tf.int32)
-------user placeholder(tf.int32)
----hocc dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----occlookup embedding()
------occ_weights weights('tnorm', tf.float32, [21, $kfactors])
------occs embedding()
-------occ placeholder(tf.int32)
-------user placeholder(tf.int32)
----hzip dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----ziplookup embedding()
------zip_weights weights('tnorm', tf.float32, [1000, $kfactors])
------zips embedding()
-------zip placeholder(tf.int32)
-------user placeholder(tf.int32)
----husertime dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
```

```
-----time placeholder(tf.float32)
-all_item dnn([$kfactors,$kfactors,$kfactors], activation='tanh',bn=True,keep_prob=None)
--tanh_item tf.nn.tanh()
---merge_item concat($kfactors)
----hitem lookup(dataname='item', initrange=$initrange, shape=[None, $kfactors])
----hgenre dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----genrelookup embedding()
------genres placeholder(tf.float32)
------item placeholder(tf.int32)
----hmonth dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----monthlookup embedding()
------month_weights weights('tnorm', tf.float32, [12, $kfactors])
------months embedding()
-------month placeholder(tf.int32)
-------item placeholder(tf.int32)
----hyear dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----yearlookup embedding()
------year placeholder(tf.float32)
------item placeholder(tf.int32)
----htfidf dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----tfidflookup embedding()
------tfidf_doc_term placeholder(tf.float32)
------item placeholder(tf.int32)
----hitemtime dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=None)
-----time placeholder(tf.float32)
-ibias lookup(dataname='item', shape=[None, 1], initrange=$initrange)
-ubias lookup(dataname='user', shape=[None, 1], initrange=$initrange)
```

A Deep Neural Network with Concatenated Input Streams

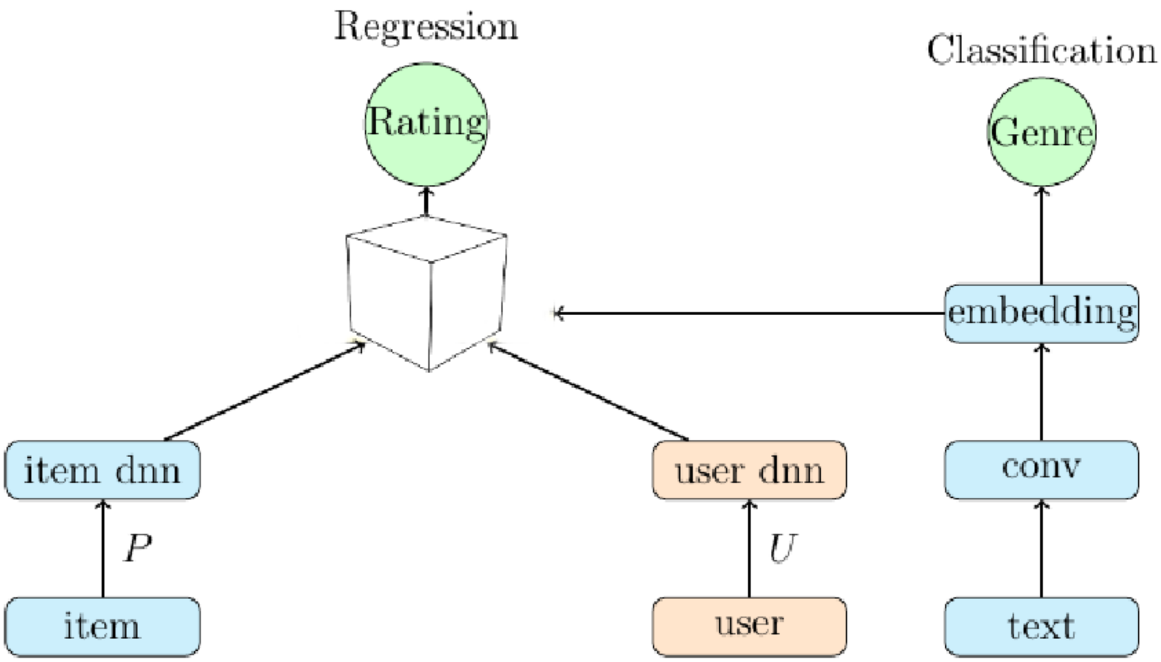
`dnn_concat_model.`**`dnn_concat`**(*data, configfile, layers=[16, 8], activation='tanhlecun', initrange=0.001, bn=True, keep_prob=0.95, concat_size=24, uembed=32, iembed=32, learnrate=1e-05, verbose=True, epochs=10, maxbadcount=20, mb=2000, eval_rate=500*)



Sample Config

```
out linear(1, True)
-h1 dnn([16, 8], activation='tanhlecun', bn=True, keep_prob=.95)
--x concat(24)
---huser lookup(dataname='user', initrange=.001, shape=[None, $embed])
---hitem lookup(dataname='item', initrange=.001, shape=[None, $embed])
```

Multiplicative Interaction between Text, User, and Item



3.2 Tutorials

3.2.1 Node Ops Tutorial

Contains functions taking a tensor or structured list of tensors and returning a tensor or structured list of tensors. The functions are commonly used compositions of tensorflow functions which operate on tensors.

Weights and Placeholders

weights

placeholder

Loss Functions and Evaluation Metrics

se

mse

rmse

mae

cross_entropy

other_cross_entropy

perplexity

detection

recall

precision

accuracy

fscore

Custom Activations

ident

tanhlecun

mult_log_reg

Matrix Operations

concat

x_dot_y

cosine

linear

embedding

lookup

khatri_rao

Tensor Operations

nmode_tensor_tomatrix

nmode_tensor_multiply

binary_tensor_combine

ternary_tensor_combine

Tricks for Training

batch_normalize

dropout

Neural Networks

dnn

residual_dnn

highway_dnn

convolutional_net

Making an op

3.2.2 Generic Model Tutorial

The generic_model module abstracts away from many common training scenarios for a reusable model training interface.

Here is sample code in straight tensorflow for the simply Mnist tutorial.

```
1  import tensorflow as tf
2  from tensorflow.examples.tutorials.mnist import input_data
3  import os
4  os.environ["CUDA_VISIBLE_DEVICES"] = ''
5  mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
6
7  x = tf.placeholder(tf.float32, [None, 784])
8  W = tf.Variable(tf.zeros([784, 10]))
9  b = tf.Variable(tf.zeros([10]))
10 y = tf.nn.softmax(tf.matmul(x, W) + b)
11 y_ = tf.placeholder(tf.float32, [None, 10])
12 cross_entropy = -tf.reduce_sum(y_*tf.log(y))
13 correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
14 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
15
16 train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
17 accuracy_summary = tf.scalar_summary('Accuracy', accuracy)
18 session = tf.Session()
19 summary_writer = tf.train.SummaryWriter('log/logistic_regression', session.graph.as_graph_def())
20 session.run(tf.initialize_all_variables())
21
22 for i in range(1000):
23   batch_xs, batch_ys = mnist.train.next_batch(100)
24   session.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
25   acc, accuracy_summary_str = session.run([accuracy, accuracy_summary], feed_dict={x: mnist.test.imag
26                                                                                     y_: mnist.test.labels})
27   summary_writer.add_summary(accuracy_summary_str, i)
28   print('Accuracy: %f' % acc)
```

In the case of this simple Mnist example lines 1-14 process data and define the computational graph, whereas lines 16-28 involve choices about how to train the model, and actions to take during training. An ANTK `Model` object parameterizes these choices for a wide variety of use cases to allow for reusable code to train a model. To achieve the same result as our simple Mnist example we can replace lines 17-29 above as follows:

```
import tensorflow as tf
from antk.core import generic_model
from tensorflow.examples.tutorials.mnist import input_data
from antk.core import loader
import os
import sys
os.environ["CUDA_VISIBLE_DEVICES"] = ''
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.nn.softmax(tf.matmul(x, W) + b)
y_ = tf.placeholder("float", [None, 10])
cross_entropy = -tf.reduce_sum(y_*tf.log(y))
```

```
predictions = tf.argmax(y, 1)
correct_prediction = tf.equal(predictions, tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

trainset = loader.DataSet({'images': mnist.train.images}, {'labels': mnist.train.labels})
print(type(mnist.train.labels[0,0]))
devset = loader.DataSet({'images': mnist.test.images},{'labels': mnist.test.labels})
pholders = {'images': x, 'labels': y_}
model = generic_model.Model(cross_entropy, pholders,
                            mb=100,
                            maxbadcount=500,
                            learnrate=0.001,
                            verbose=True,
                            epochs=100,
                            evaluate=1 - accuracy,
                            model_name='simple_mnist',
                            tensorboard=False)

dev = loader.DataSet({'images': mnist.test.images, 'labels': mnist.test.labels})
dev.show()
train = loader.DataSet({'images': mnist.train.images, 'labels': mnist.train.labels})
train.show()
model.train(train, dev=dev, eval_schedule=100)
```

Notice that we had to change the evaluation function to take advantage of early stopping so that when the model does better the evaluation function is less. So we evaluate on 1 - accuracy = error. Using generic_model now allows us to easily test out different training scenarios by changing some of the default settings.

We can go through all the options and see what is available. Replace your call to the `Model` constructor with the following call that makes all default parameters explicit.

```
model = generic_model.Model(cross_entropy, pholders,
                            maxbadcount=20,
                            momentum=None,
                            mb=1000,
                            verbose=True,
                            epochs=50,
                            learnrate=0.01,
                            save=False,
                            opt='grad',
                            decay=[1, 1.0],
                            evaluate=1-accuracy,
                            predictions=predictions,
                            logdir='log/simple_mnist',
                            random_seed=None,
                            model_name='simple_mnist',
                            clip_gradients=0.0,
                            make_histograms=False,
                            best_model_path='/tmp/model.ckpt',
                            save_tensors={},
                            tensorboard=False):
```

Suppose we want to save our best set of weights, and bias for this logistic regression model, and make a tensorboard histogram plot of how the weights change over time. Also, we want to be able to make predictions with our trained model as well.

We just need to set a few arguments in the call to the `Model` constructor:

```
save_tensors=[W, b]
make_histograms=True
```

You can view the graph with histograms with the usual tensorboard call from the terminal.

```
$ tensorboard --logdir log/simple_mnist
```

Also, to be able to make predictions with our trained model we need to set the predictions argument in the call to the constructor as below:

```
predictions=tf.argmax(y,1)
```

Now we can get predictions from the trained model using:

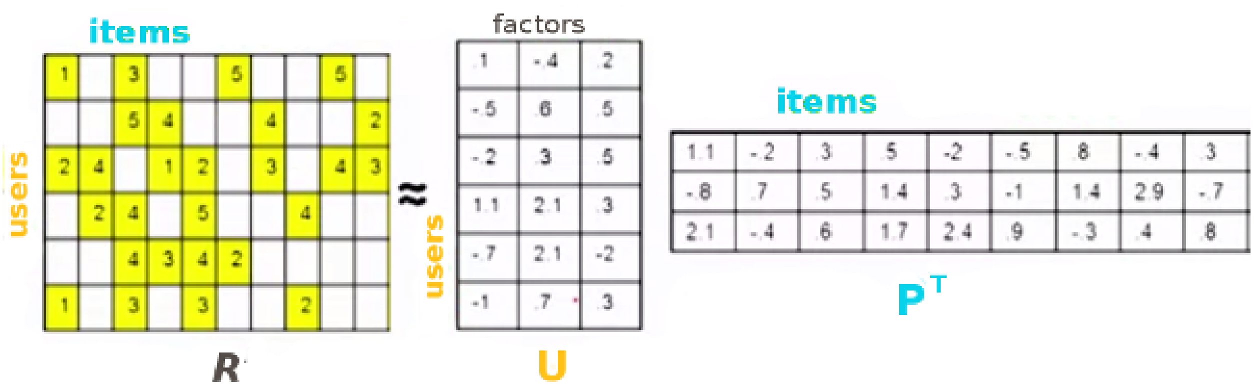
```
dev_classes = model.predict(devset)
```

3.2.3 All in One Tutorial via Matrix Factorization

Part 1 starts off with a somewhat gentle introduction to the toolkit by implementing basic matrix factorization ratings prediction on the MovieLens 100k dataset. Read the directions carefully and be prepared use your copy and pasting skills. Part 2 explores developing a more complex model using deep neural nets to incorporated user and item meta data into the model. Carefully reading parts 1 and 2 will pay off when you engage in the task of building a new model.

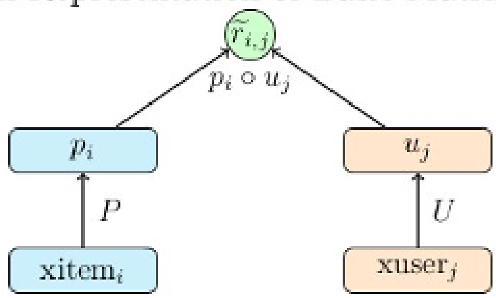
Part 1: Matrix Factorization Model

Low Rank Matrix Factorization is a popular machine learning technique used to produce recommendations given a set of ratings a user has given an item. The known ratings are collected in a user-item utility matrix and the missing entries are predicted by optimizing a low rank factorization of the utility matrix given the known entries. The basic idea behind matrix factorization models is that the information encoded for items in the columns of the utility matrix, and for users in the rows of the utility matrix is not exactly independent. We optimize the objective function $\sum_{(u,i)}(R_{ui} - P_i^T U_u)^2$ over the observed ratings for user *u* and item *i* using gradient descent.



We can express the same optimization in the form of a computational graph that will play nicely with tensorflow:

Figure 1: Graph Representation of Basic Matrix Factorization



Here $xitem_i$, and $xuser_j$ are some representation of the indices for the user and item vectors in the utility matrix. These could be one hot vectors, which can then be matrix multiplied by the *P* and *U* matrices to select the corresponding user and item vectors. In practice it is much faster to let $xitem_i$, and $xuser_j$ be vectors of indices which can be used by tensorflow's **gather** or **embedding_lookup** functions to select the corresponding vector from the *P* and *U* matrices.

This simple model isn't difficult to code directly in tensorflow, but it's simplicity allows a demonstration of the functionality of the toolkit without having to tackle a more complex model.

We have some processed MovieLens 100k data prepared for this tutorial located at http://sw.cs.wwu.edu/~tuora/aarontuor/ml100k.tar.gz . The original MovieLens 100k dataset is located at http://grouplens.org/datasets/movielens/ .

To start let's import the modules we need, retrieve our prepared data, and use the loader module's *read_data_sets* function to load our data:

```
import tensorflow as tf
from antk.core import config
from antk.core import generic_model
from antk.core import loader


loader.maybe_download('ml100k.tar.gz', '.',
                      'http://sw.cs.wwu.edu/~tuora/aarontuor/ml100k.tar.gz')
loader.untar('ml100k.tar.gz')
data = loader.read_data_sets('ml100k', folders=['dev', 'train'],
                              hashlist=['item', 'user', 'ratings'])
```

There is a lot more data in the ml100k folder than we need for demonstrating a basic MF model so we use the **hashlist** and **folders** arguments to select only the data files we want. We can view the dimensions types, and dictionary keys of the data we've loaded using the *DataSets.show* method, which is a useful feature for debugging.

```
data.show()
```

The previous command will display this to the terminal:

```
dev:
features:
        item: vec.shape: (10000,) dim: 1682 <class 'antk.core.loader.HotIndex'>
        user: vec.shape: (10000,) dim: 943 <class 'antk.core.loader.HotIndex'>
labels:
        ratings: (10000, 1) <type 'numpy.ndarray'>
train:
features:
        item: vec.shape: (80000,) dim: 1682 <class 'antk.core.loader.HotIndex'>
        user: vec.shape: (80000,) dim: 943 <class 'antk.core.loader.HotIndex'>
labels:
        ratings: (80000, 1) <type 'numpy.ndarray'>
```

For this data there are 10,000 ratings in dev and test, and 80,000 ratings in train. Notice that the data type of *item* and *user* above is *HotIndex*. This is a data structure for storing one hot vectors, with a field for a vector of indices into a one hot matrix and the column size of the one hot matrix. This will be important as we intend to use the *lookup* function, which takes *HotIndex* objects for its *data* argument, makes a placeholder associated with this data and uses the *dim* attribute of the *HotIndex* data to create a **tf.Variable** tensor with the correct dimension. The output is an **embedding_lookup** using the placeholder and variable tensors created.

This model does better with the target ratings centered about the mean so let's center the ratings.

```
data.train.labels['ratings'] = loader.center(data.train.labels['ratings'])
data.dev.labels['ratings'] = loader.center(data.dev.labels['ratings'])
```

Todo

Make a plain text file named mf.config using the text below. We will use this to make the tensorflow computational graph:

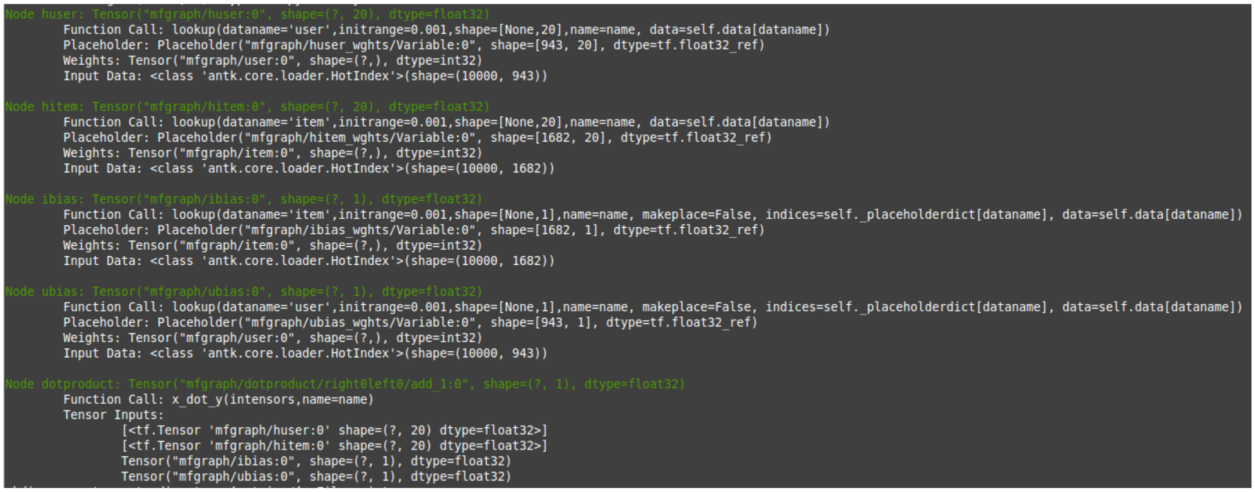
```
dotproduct x_dot_y()
    -huser lookup(dataname='user', initrange=0.001, shape=[None, 100])
    -hitem lookup(dataname='item', initrange=0.001, shape=[None, 100])
    -ibias lookup(dataname='item', initrange=0.001, shape=[None, 1])
    -ubias lookup(dataname='user', initrange=0.001, shape=[None, 1])
```

The python syntax highlighting illustrates the fact that the node specifications in a .config file are just python function calls with two things omitted, the first argument which is a tensor or list of tensors, and the last argument which is the name of the tensor output which defines it's unique variable scope. The first argument is derived from the structure of the config spec, inferred by a marker symbol which we have chosen as '-'. The input is the list of tensors or the single tensor in the spec at the next level below a node call. Tabbing is optional. It may be easier to read a config file with tabbing if you are using node functions without a long sequence of arguments. The second omitted argument, the name, is whatever directly follows the graph markers.

Now we make an *AntGraph* object.

```
with tf.variable_scope('mfgraph'):
ant = config.AntGraph('mf.config',
                      data=data.dev.features,
                      marker='-',
                      develop=True)
```

When you run the code now you will get a complete print of the tensors made from the config file because we have set the **develop** argument to **True**.

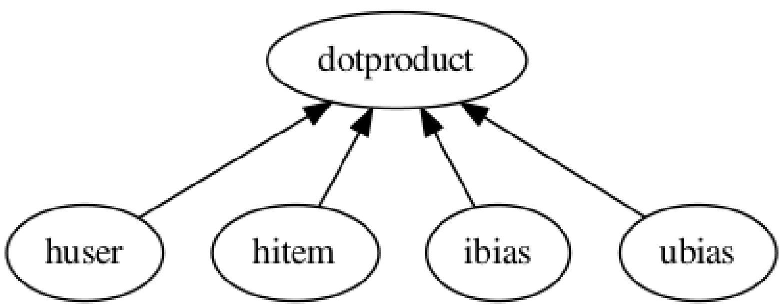


```
Node huser: Tensor("mfgraph/huser:0", shape=(?, 20), dtype=float32)
        Function Call: lookup(dataname='user',initrange=0.001,shape=[None,20],name=name, data=self.data[dataname])
        Placeholder: Placeholder("mfgraph/huser_wghts/Variable:0", shape=[943, 20], dtype=tf.float32_ref)
        Weights: Tensor("mfgraph/user:0", shape=(?,), dtype=int32)
        Input Data: <class 'antk.core.loader.HotIndex'>(shape=(10000, 943))

Node hitem: Tensor("mfgraph/hitem:0", shape=(?, 20), dtype=float32)
        Function Call: lookup(dataname='item',initrange=0.001,shape=[None,20],name=name, data=self.data[dataname])
        Placeholder: Placeholder("mfgraph/hitem_wghts/Variable:0", shape=[1682, 20], dtype=tf.float32_ref)
        Weights: Tensor("mfgraph/item:0", shape=(?,), dtype=int32)
        Input Data: <class 'antk.core.loader.HotIndex'>(shape=(10000, 1682))

Node ibias: Tensor("mfgraph/ibias:0", shape=(?, 1), dtype=float32)
        Function Call: lookup(dataname='item',initrange=0.001,shape=[None,1],name=name, makeplace=False, indices=self._placeholderdict[dataname], data=self.data[dataname])
        Placeholder: Placeholder("mfgraph/ibias_wghts/Variable:0", shape=[1682, 1], dtype=tf.float32_ref)
        Weights: Tensor("mfgraph/item:0", shape=(?,), dtype=int32)
        Input Data: <class 'antk.core.loader.HotIndex'>(shape=(10000, 1682))

Node ubias: Tensor("mfgraph/ubias:0", shape=(?, 1), dtype=float32)
        Function Call: lookup(dataname='user',initrange=0.001,shape=[None,1],name=name, makeplace=False, indices=self._placeholderdict[dataname], data=self.data[dataname])
        Placeholder: Placeholder("mfgraph/ubias_wghts/Variable:0", shape=[943, 1], dtype=tf.float32_ref)
        Weights: Tensor("mfgraph/user:0", shape=(?,), dtype=int32)
        Input Data: <class 'antk.core.loader.HotIndex'>(shape=(10000, 943))

Node dotproduct: Tensor("mfgraph/dotproduct/right0left0/add_1:0", shape=(?, 1), dtype=float32)
        Function Call: x_dot_y(intensors,name=name)
        Tensor Inputs:
                [<tf.Tensor 'mfgraph/huser:0' shape=(?, 20) dtype=float32>]
                [<tf.Tensor 'mfgraph/hitem:0' shape=(?, 20) dtype=float32>]
                Tensor("mfgraph/ibias:0", shape=(?, 1), dtype=float32)
                Tensor("mfgraph/ubias:0", shape=(?, 1), dtype=float32)
```

We can get a visual representation of the graph with another line:

```
ant.display_graph()
```

When you run this code a graphviz dot pdf image of the graph you have composed should pop up on the screen (assuming you have graphviz installed). This pdf file will show up in the pics folder with the name **no_name.pdf**. There are of course parameters for specifying the name and location where you want the picture to go. The dot specification will be located in the same place as the picture and be named **no_name.dot** unless you have specified a name for the file.



Shown in the graph picture above the `x_dot_y` function takes a list of tensors as its first argument. The first two tensors are matrices whose rows are dot producted resulting in a vector containing a scalar for each row. The second two tensors are optional biases. For this model, giving a user and item bias helps a great deal. When `lookup` is called more than once in a config file using the same *data* argument the previously made placeholder tensor is used, so here *ibias* depends on the same placeholder as *hbias* and *ubias* depends on the same placeholder as *huser*, which is what we want.

The `AntGraph` object, *ant* is a complete record of the tensors created in graph building. There are three accessible fields, `tensordict`, `placeholderdict`, and `tensor_out`, which are a dictionary of non-placeholder tensors made during graph creation, a dictionary of placeholder tensors made during graph creation and the tensor or list of tensors which is the output of the top level node function. These should be useful if we want to access tensors post graph creation.

Okay let's finish making this model:

```
y = ant.tensor_out
y_ = tf.placeholder("float", [None, None], name='Target')
ant.placeholderdict['ratings'] = y_ # put the new placeholder in the placeholderdict for training
objective = (tf.reduce_sum(tf.square(y_ - y)) +
             0.1*tf.reduce_sum(tf.square(ant.tensordict['huser'])) +
             0.1*tf.reduce_sum(tf.square(ant.tensordict['hitem'])) +
```

```
              0.1*tf.reduce_sum(tf.square(ant.tensordict['ubias'])) +
              0.1*tf.reduce_sum(tf.square(ant.tensordict['ibias'])))
dev_rmse = tf.sqrt(tf.div(tf.reduce_sum(tf.square(y - y_)), data.dev.num_examples))

model = generic_model.Model(objective, ant.placeholderdict,
          mb=500,
          learnrate=0.01,
          verbose=True,
          maxbadcount=10,
          epochs=100,
          evaluate=dev_rmse,
          predictions=y)
```

Notice that the `tensordict` enables easy access to *huser*, *hitem*, *ubias*, *ibias*, which we want to regularize to prevent overfitting. The `Model` object we are creating *model* needs the fields *objective*, *placeholderdict*, *predictions*, and *targets*. If you don't specify the other parameters default values are set. *objective* is used as the loss function for gradient descent. *placeholderdict* is used to pair placeholder tensors with matrices from a dataset dictionary with the same keys. *targets*, and *predictions* are employed by the loss function during evaluation, and by the prediction function to give outputs from a trained model.

Training is now as easy as:

```
model.train(data.train, dev=data.dev)
```

You should get about 0.92 RMSE.

There are a few antk functionalities we can take advantage of to make our code more compact. Any node_op function that creates trainable weights has a parameter for adding l2 regularization to the weights of the model. We just change our config as below and we can eliminate the four extra lines in the definition of **objective**.

```
dotproduct x_dot_y()
    -huser lookup(dataname='user', initrange=0.001, l2=0.1, shape=[None, 100])
    -hitem lookup(dataname='item', initrange=0.001, l2=0.1, shape=[None, 100])
    -ibias lookup(dataname='item', initrange=0.001, l2=0.1, shape=[None, 1])
    -ubias lookup(dataname='user', initrange=0.001, l2=0.1, shape=[None, 1])
```

Also, we have a function for RMSE, and we can evaluate the mean absolute error using the **save_tensors** argument to the generic_model constructor. Our code now looks like this:

```
y = ant.tensor_out
y_ = tf.placeholder("float", [None, None], name='Target')
ant.placeholderdict['ratings'] = y_ # put the new placeholder in the graph for training
objective = node_ops.se(y_ - y)
dev_rmse =  node_ops.rmse(y, y_)
dev_mae = node_ops.mae(y, y_)

model = generic_model.Model(objective, ant.placeholderdict,
          mb=500,
          learnrate=0.01,
          verbose=True,
          maxbadcount=10,
          epochs=100,
          evaluate=dev_rmse,
          predictions=y,
          save_tensors={'dev_mae': dev_mae})
model.train(data.train, dev=data.dev)
```

If you don't wan't to evaluate a model during training, for instance if you are doing cross-validation, you can just hand the `train` method a training set and omit the dev set. Note that here there must be keys in either the `DataSet` `features`, or `labels` dictionaries, that match with the keys from the `placeholderdict` which is handed to the `Model` constructor. In our case we have placed a placeholder with the key *ratings* in the `placeholdedict` corresponding to the *ratings* key in our *data* `DataSet`. So our `placeholderdict` is:

```
{'item': <tensorflow.python.framework.ops.Tensor object at 0x7f0bea7b43d0>,
 'user': <tensorflow.python.framework.ops.Tensor object at 0x7f0bea846e90>,
 'ratings': <tensorflow.python.framework.ops.Tensor object at 0x7f0bea77fc90>}
```

Now we have a trained model that does pretty well but it would be nice to automate a hyper-parameter search to find the best we can do (should be around .91).

We can change our mf.config file to accept variables for hyperparameters by substituting hard values with variable names prefixed with a '$':

```
dotproduct x_dot_y()
     -huser lookup(dataname='user', initrange=$initrange, l2=$l2, shape=[None, $kfactors])
     -hitem lookup(dataname='item', initrange=$initrange, l2=$l2, shape=[None, $kfactors])
     -ibias lookup(dataname='item', initrange=$initrange, l2=$l2, shape=[None, 1])
     -ubias lookup(dataname='user', initrange=$initrange, l2=$l2, shape=[None, 1])
```

Now we have to let the `AntGraph` constructor know what to bind these variables to with a *variable_bindings* argument. So change the constructor call like so.

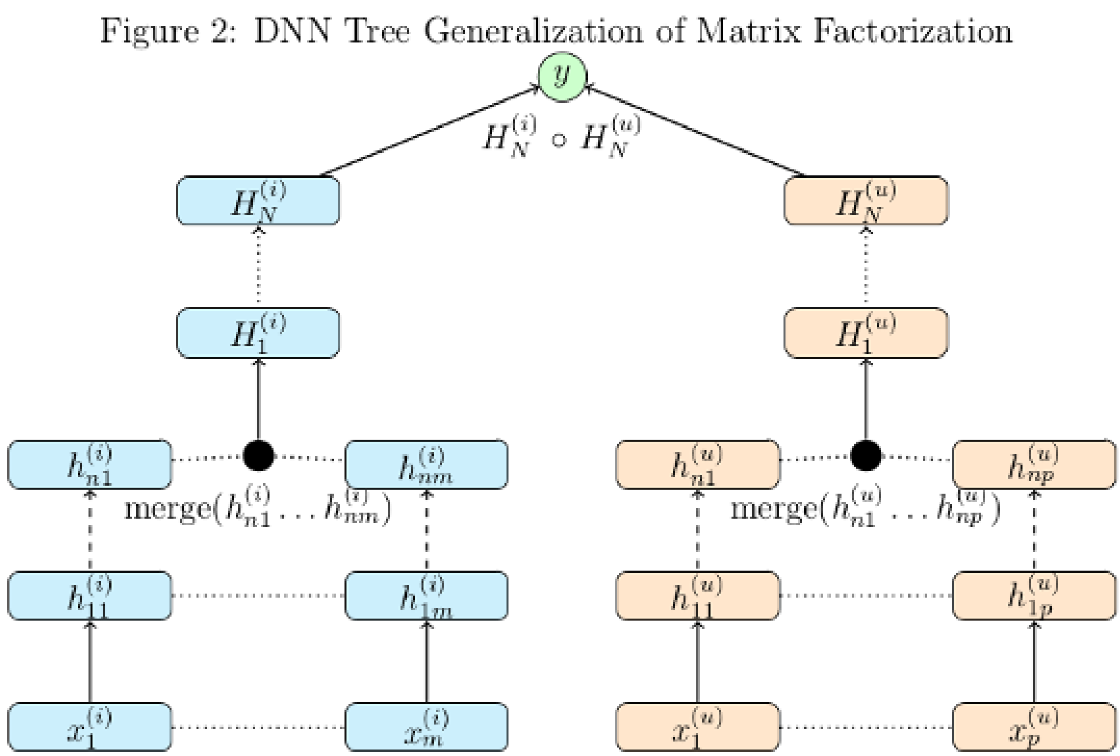
```
with tf.variable_scope('mfgraph'):
    ant = config.AntGraph('mf.config',
                            data=data.dev.features,
                            marker='-',
                            variable_bindings = {'kfactors': 100, 'initrange':0.001, 'l2':0.1})
```

Todo

Modify the code you've written to take command line arguments for the hyperparameters: *kfactors*, *initrange*, *mb*, *learnrate*, *maxbadcount*, *l2*, and *epochs*, and conduct a parameter search for the best model.

Part 2: Tree Model

To demonstrate the power and flexibility of using a config file we can make this more complex model below by changing a few lines of code and using a different config file:



We need to change the *read_data_sets* call to omit the optional *hashlist* parameter so we get more features from the data folder (if a *hashlist* parameter is not supplied, *read_data_sets* reads all files with name prefixes **features_** and **labels_**).

Todo

Make a new python file tree.py with the code below:

```
import tensorflow as tf
from antk.core import config
from antk.core import generic_model
from antk.core import loader
from antk.core import node_ops

data = loader.read_data_sets('ml100k', folders=['dev', 'train', 'item', 'user'])
data.show()
```

Now we have some user and item meta data which we can examine:

```
dev:
features:
        item: vec.shape: (10000,) dim: 1682 <class 'antk.core.loader.HotIndex'>
        user: vec.shape: (10000,) dim: 943 <class 'antk.core.loader.HotIndex'>
        words: (10000, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        time: (10000, 1) <type 'numpy.ndarray'>
labels:
        genre: (10000, 19) <type 'numpy.ndarray'>
        ratings: (10000, 1) <type 'numpy.ndarray'>
        genre_dist: (10000, 19) <type 'numpy.ndarray'>
item:
features:
        genres: (1682, 19) <type 'numpy.ndarray'>
        bin_doc_term: (1682, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        month: vec.shape: (1682,) dim: 12 <class 'antk.core.loader.HotIndex'>
        doc_term: (1682, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        tfidf_doc_term: (1682, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        year: (1682, 1) <type 'numpy.ndarray'>
labels:
train:
features:
        item: vec.shape: (80000,) dim: 1682 <class 'antk.core.loader.HotIndex'>
        user: vec.shape: (80000,) dim: 943 <class 'antk.core.loader.HotIndex'>
        words: (80000, 12734) <class 'scipy.sparse.csc.csc_matrix'>
        time: (80000, 1) <type 'numpy.ndarray'>
labels:
        genre: (80000, 19) <type 'numpy.ndarray'>
        ratings: (80000, 1) <type 'numpy.ndarray'>
        genre_dist: (80000, 19) <type 'numpy.ndarray'>
user:
features:
        occ: vec.shape: (943,) dim: 21 <class 'antk.core.loader.HotIndex'>
        age: (943, 1) <type 'numpy.ndarray'>
        zip: vec.shape: (943,) dim: 1000 <class 'antk.core.loader.HotIndex'>
        sex: vec.shape: (943,) dim: 2 <class 'antk.core.loader.HotIndex'>
labels:
```

The idea of this model is to have a deep neural network for each stream of user meta data and item meta data. The user and item dnn's are concatenated respectively and then fed to a user dnn and an item dnn. The outputs of these dnn's are dot producted to provide ratings predictions. We can succinctly express this model in a .config file.

Todo

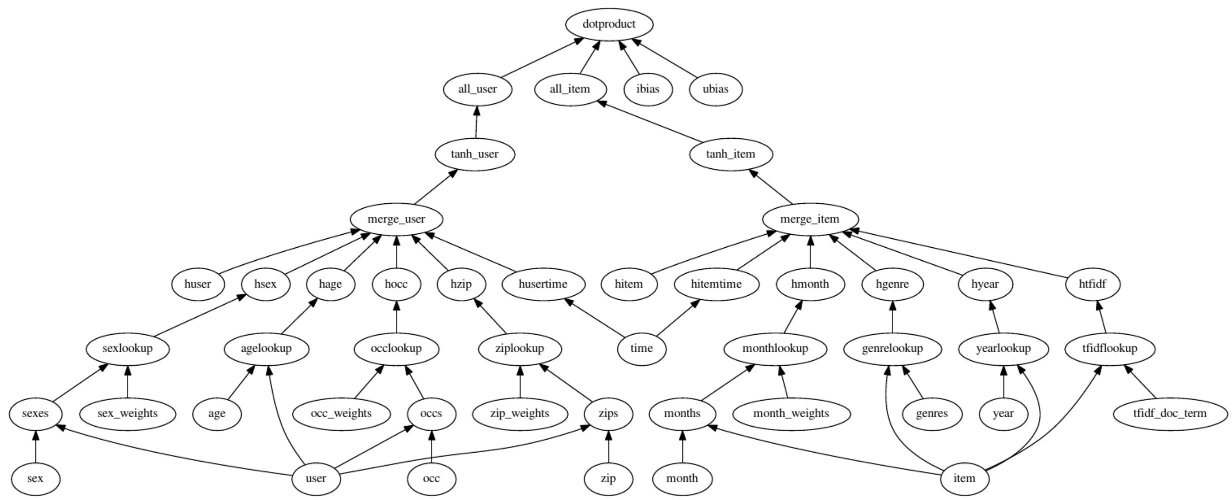
Make a plain text file called tree.config with the specs for our tree model.

```
dotproduct x_dot_y()
-all_user dnn([$kfactors,$kfactors,$kfactors], activation='tanh',bn=True,keep_prob=0.95)
--tanh_user tf.nn.tanh()
---merge_user concat($kfactors)
----huser lookup(dataname='user', initrange=$initrange, shape=[None, $kfactors])
----hage dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=0.95)
-----agelookup embedding()
------age placeholder(tf.float32)
------user placeholder(tf.int32)
----hsex dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=0.95)
```

```
-----sexlookup embedding()
------sex_weights weights('tnorm', [2, $kfactors])
------sexes embedding()
-------sex placeholder(tf.int32)
-------user placeholder(tf.int32)
----hocc dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=0.95)
-----occlookup embedding()
------occ_weights weights('tnorm', [21, $kfactors])
------occs embedding()
-------occ placeholder(tf.int32)
-------user placeholder(tf.int32)
----hzip dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=0.95)
-----ziplookup embedding()
------zip_weights weights('tnorm', [1000, $kfactors])
------zips embedding()
-------zip placeholder(tf.int32)
-------user placeholder(tf.int32)
----husertime dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=0.95)
-----time placeholder(tf.float32)
-all_item dnn([$kfactors,$kfactors,$kfactors], activation='tanh',bn=True,keep_prob=0.95)
--tanh_item tf.nn.tanh()
---merge_item concat($kfactors)
----hitem lookup(dataname='item', initrange=$initrange, shape=[None, $kfactors])
----hgenre dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=0.95)
-----genrelookup embedding()
------genres placeholder(tf.float32)
------item placeholder(tf.int32)
----hmonth dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=0.95)
-----monthlookup embedding()
------month_weights weights('tnorm', [12, $kfactors])
------months embedding()
-------month placeholder(tf.int32)
-------item placeholder(tf.int32)
----hyear dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=0.95)
-----yearlookup embedding()
------year placeholder(tf.float32)
------item placeholder(tf.int32)
----htfidf dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=0.95)
-----tfidflookup embedding()
------tfidf_doc_term placeholder(tf.float32)
------item placeholder(tf.int32)
----hitemtime dnn([$kfactors,$kfactors,$kfactors],activation='tanh',bn=True,keep_prob=0.95)
-----time placeholder(tf.float32)
-ibias lookup(dataname='item', shape=[None, 1], initrange=$initrange)
-ubias lookup(dataname='user', shape=[None, 1], initrange=$initrange)
```

This model employs all the user and item meta-data we have at our disposal. The config file looks pretty complicated, and it is, but at least it fits on a screen and we can *read* the high level structure of the model. Imagine developing this model with straight python tensorflow code. This would be hundreds of lines of code and it would be much more difficult to *see* what was going on with the model. We can see what the model will look like without actually building the graph with the `config.testGraph` function.

```
config.testGraph('tree.config')
```



This looks like a pretty cool model! We should probably normalize the meta data features for training though.

```
data.train.labels['ratings'] = loader.center(data.train.labels['ratings'], axis=None)
data.dev.labels['ratings'] = loader.center(data.dev.labels['ratings'], axis=None)
data.user.features['age'] = loader.center(data.user.features['age'], axis=None)
data.item.features['year'] = loader.center(data.item.features['year'], axis=None)
data.user.features['age'] = loader.maxnormalize(data.user.features['age'])
data.item.features['year'] = loader.maxnormalize(data.item.features['year'])
```

All our other features besides time are categorical and so use lookups. I think I normalized time during data processing but it couldn't hurt to check. If you think it is a good idea you can whiten these data inputs to have zero mean and unit variance with some convenience functions from the loader module. Now we should build our graph. Notice that we have omitted the l2 variable in the config file. We are using dropout to regularize our output as an alternative, since this is a standard regularization technique for deep neural networks.

Remember we need a python dictionary of numpy matrices whose keys match the names of placeholder and lookup operations that will infer dimensions for the `AntGraph` constructor. So we need to add these lines:

```
datadict = data.user.features.copy()
datadict.update(data.item.features)
configdatadict = data.dev.features.copy()
configdatadict.update(datadict)
```

Now we can build the graph. We'll set **develop** to **False** because a lot of tensors are going to get made. If something goes wrong with a model this big set **develop** to **True** and pipe standard output to a file for analysis:

```
with tf.variable_scope('mfgraph'):
    ant = config.AntGraph('tree.config',
                            data=configdatadict,
                            marker='-',
                            variable_bindings = {'kfactors': 100, 'initrange':0.001},
                            develop=False)

y = ant.tensor_out
y_ = tf.placeholder("float", [None, None], name='Target')
ant.placeholderdict['ratings'] = y_  # put the new placeholder in the graph for training
objective = tf.reduce_sum(tf.square(y_ - y))
dev_rmse =  node_ops.rmse(y, y_)
```

Training this model will naturally take longer so we can set the evaluation schedule to be shorter than an epoch to check in on how things are doing. Also, we will need a smaller learnrate for gradient descent. So we can initialize a

Model object with the following hyper-parameters as a first approximation, and then train away...

```
model = generic_model.Model(objective, ant.placeholderdict,
                            mb=500,
                            learnrate=0.0001,
                            verbose=True,
                            maxbadcount=10,
                            epochs=100,
                            evaluate=dev_rmse,
                            predictions=y)
model.train(data.train, dev=data.dev, supplement=datadict, eval_schedule=1000)
```

Note: We added the supplement argument to *train* so that the placeholders related to meta-data could be added to the tensorflow feed dictionary with the backend function `get_feed_dict` employed by the *Model* constructor.

This model takes a while to train and from some poking around it is hard to find a set of hyperparameters that will approach the accuracy of a basic matrix factorization model. The hyperparameters I have provided should give about 0.93 RMSE which isn't good for this data set. We have a lot of things to try such as batch normalization, dropout, hidden layer size, number of hidden layers, activation functions, optimization strategies, subsets of the meta data to incorporate into the mode, and of course the standard learning rate and intitialization strategies.

Todo

Modify the code you've written to take arguments for the set of new hyperparameters, and optional optimization parameters from the *Model* API. Perform a parameter search to see if you can do better than basic MF.

3.3 Command Line Scripts

3.3.1 datatest.py

Tool for displaying data using *loader.read_data_sets*.

```
usage: datatest [-h] [-hashlist HASHLIST [HASHLIST ...]]
                [-cold | -subfolders SUBFOLDERS [SUBFOLDERS ...]]
                datadirectory
```

Positional arguments:

datadirectory Path to folder where data to be loaded and displayed is stored.

Options:

-hashlist List of hashes to read. Files will be read of the form "features_<hash>.ext" or"labels_<hash>.ext" where <hash> is a string in hashlist. If a hashlist is not specified all files of the form "features_<hash>.ext" or "labels_<hash>.ext" regardless what string <hash> is will be loaded.

-cold=False Extra loading and testing for cold datasets

-subfolders=('test', 'dev', 'train') List of subfolders to load and display.

3.3.2 normalize.py

Given the path to a file, Capitalization and punctuation is removed, except for infix apostrophes, e.g. "hasn't", "David's". The normalized text is saved with "_norm" appended to the file name before the extension. The normalized text is saved in the same directory as the original text. Beginning and end of sentence tokens are not provided by this normalization script.

```
usage: normalize [-h] filepath
```

Positional arguments:

filepath The path to the file including filename

3.4 Movie Lens Processing

3.4.1 generateTermDoc.py

```
usage: generateTermDoc [-h] datapath dictionary descriptions doc_term_file
```

Positional arguments:

datapath Path to folder where dictionary and descriptions are located, and created document term matrix will be saved.

dictionary Name of the file containing line separated words in vocabulary.

descriptions Name of the file containing line separated text descriptions.

doc_term_file Name of the file to save the created sparse document term matrix.

3.4.2 ml100k_item_process.py

Reads MovieLens 100k item meta data and converts to feature files. features_item_month.index: The produced files are: A file storing a *HotIndex* object of movie month releases.

features_item_year.mat: A file storing a numpy array of movie year releases.

features_item_genre.mat: A file storing a scipy sparse csr_matrix of one hot encodings for movie genre.

```
usage: ml100k_item_process [-h] datapath outpath
```

Positional arguments:

datapath The path to ml-100k dataset. Usually "some_relative_path/ml-100k

outpath The path to the folder to store the processed Movielens 100k item data feature files.

3.4.3 ml100k_user_process.py

Tool to process Movielens 100k user Metadata.

```
usage: ml100k_user_process [-h] datapath outpath
```

Positional arguments:

datapath Path to ml-100k

outpath	Path to save created files to.

CHAPTER FOUR

INDICES AND TABLES

- genindex
- modindex
- search

N

O

P

R

S

T

U

V

W

X