Dynet Documentation

Release 1.0

Clab

Dec 14, 2016

Contents

DyNet (formerly known as cnn) is a neural network library developed by Carnegie Mellon University and many others. It is written in C++ (with bindings in Python) and is designed to be efficient when run on either CPU or GPU, and to work well with networks that have dynamic structures that change for every training instance. For example, these kinds of networks are particularly important in natural language processing tasks, and DyNet has been used to build state-of-the-art systems for syntactic parsing, machine translation, morphological inflection, and many other application areas.

DyNet can be installed according to the instructions below:

CHAPTER 1

Building/Installing

How to build DyNet and link it with your programs

1.1 Prerequisites

DyNet relies on a number of external libraries including Boost, CMake, Eigen, and Mercurial (to install Eigen). Boost, CMake, and Mercurial can be installed from standard repositories, for example on Ubuntu Linux:

```
sudo apt-get install libboost-all-dev cmake mercurial
```

Or on OSX, first make sure the Apple Command Line Tools are installed, then get Boost, CMake, and Mercurial with either homebrew or macports:

```
xcode-select --install
brew install boost cmake hg
sudo port install boost cmake mercurial
```

To compile DyNet you also need the development version of the Eigen library. **If you use any of the released versions, you may get assertion failures or compile errors.** If you don't have Eigen installed already, you can get it easily using the following command:

```
hg clone https://bitbucket.org/eigen/eigen/ -r 346ecdb
```

The *-r NUM* specified a revision number that is known to work. Adventurous users can remove it and use the very latest version, at the risk of the code breaking / not compiling.

1.2 Building

To get and build DyNet, clone the repository

```
git clone https://github.com/clab/dynet.git
```

then enter the directory and use `cmake <http://www.cmake.org/>`__ to generate the makefiles

```
cd dynet
mkdir build
cd build
cmake .. -DEIGEN3_INCLUDE_DIR=/path/to/eigen
```

Then compile, where "2" can be replaced by the number of cores on your machine

```
make -j 2
```

To see that things have built properly, you can run

```
./examples/xor
```

which will train a multilayer perceptron to predict the xor function.

1.3 Compiling/linking external programs

When you want to use DyNet in an external program, you will need to add the `dynet` directory to the compile path:

```
-I/path/to/dynet
```

and link with the DyNet library:

```
-L/path/to/dynet/build/dynet -ldynet
```

1.4 Debugging build problems

If you have a build problem and want to debug, please run

```
make clean
make VERBOSE=1 &> make.log
```

then examine the commands in the `make.log` file to see if anything looks fishy. If you would like help, send this `make.log` file via the "Issues" tab on GitHub, or to the dynet-users mailing list.

1.5 GPU/MKL support and build options

1.5.1 GPU (CUDA) support

`dynet` supports running programs on GPUs with CUDA. If you have CUDA installed, you can build DyNet with GPU support by adding `-DBACKEND=cuda` to your cmake options. This will result in three libraries named "libdynet," "libgdynet," and "libdynetcuda" being created. When you want to run a program on CPU, you can link to the "libdynet" library as shown above. When you want to run a program on GPU, you can link to the "libgdynet" and "libdynetcuda" libraries.

```
-L/path/to/dynet/build/dynet -lgdynet -ldynetcuda
```

(Eventually you will be able to use a single library to run on either CPU or GPU, but this is not fully implemented yet.)

1.5.2 MKL support

DyNet can leverage Intel's MKL library to speed up computation on the CPU. As an example, we've seen 3x speedup in seq2seq training when using MKL. To use MKL, include the following cmake option:

```
-DMKL=TRUE
```

If CMake is unable to find MKL automatically, try setting *MKL_ROOT*, such as

```
-DMKL_ROOT="/path/to/MKL"
```

If either *MKL* or *MKL_ROOT* are set, CMake will look for MKL.

By default, MKL will use all CPU cores. You can control how many cores MKL uses by setting the environment variable *MKL_NUM_THREADS* to the desired number. The following is the total time to process 250 training examples running the example encdec (on a 6 core Intel Xeon E5-1650):

```
encdec.exe --dynet-seed 1 --dynet-mem 1000 train-hsm.txt dev-hsm.txt
```

MKL_NUM_THREADS	Cores Used	Time(s)
<Without MKL>	1	28.6
1	1	13.3
2	2	9.5
3	3	8.1
4	4	7.8
6	6	8.2

As you can see, for this particular example, using MKL roughly doubles the speed of computation while still using only one core. Increasing the number of cores to 2 or 3 is quite beneficial, but beyond that there are diminishing returns or even slowdown.

1.5.3 Non-standard Boost location

`dynet` requires Boost, and will find it if it is in the standard location. If Boost is in a non-standard location, say `$HOME/boost`, you can specify the location by adding the following to your CMake options:

```
-DBOOST_ROOT:PATHNAME=$HOME/boost -DBoost_LIBRARY_DIRS:FILEPATH=$HOME/boost/lib
-DBoost_NO_BOOST_CMAKE=TRUE -DBoost_NO_SYSTEM_PATHS=TRUE
```

Note that you will also have to set your `LD_LIBRARY_PATH` to point to the `boost/lib` directory. Note also that Boost must be compiled with the same compiler version as you are using to compile DyNet.

1.5.4 Building for Windows

DyNet has been tested to build in Windows using Microsoft Visual Studio 2015. You may be able to build with MSVC 2013 by slightly modifying the instructions below.

First, install Eigen following the above instructions.

Second, install Boost for your compiler and platform. Follow the instructions for compiling Boost or just download the already-compiled binaries.

To generate the MSVC solution and project files, run cmake, pointing it to the location you installed Eigen and Boost (for example, at c:\libs\Eigen and c:\libs\boost_1_61_0):

```
mkdir build
cd build
cmake .. -DEIGEN3_INCLUDE_DIR=c:\libs\Eigen -DBOOST_ROOT=c:\libs\boost_1_61_0 -DBOOST_
↪LIBRARYDIR=c:\libs\boost_1_61_0\lib64-msvc-14.0 -DBoost_NO_BOOST_CMAKE=ON -G"Visual␣
↪Studio 14 2015 Win64"
```

This will generate *dynet.sln* and a bunch of **.vcxproj* files (one for the DyNet library, and one per example). You should be able to just open *dynet.sln* and build all. **Note: multi-process functionality is currently not supported in Windows, so the multi-process examples ('*-mp') will not be included in the generated solution**

The Windows build also supports CUDA with the latest version of Eigen (as of Oct 28, 2016), with the following code change:

- TensorDeviceCuda.h: Change *sleep(1)* to *Sleep(1000)*

CHAPTER 2

Installing the Python DyNet module.

(for instructions on installing on a computer with GPU, see below)

Python bindings to DyNet are supported for both Python 2.x and 3.x.

2.1 TL;DR

(see below for the details)

```
# Installing Python DyNet:

pip install cython  # if you don't have it already.
mkdir dynet-base
cd dynet-base
# getting dynet and eigen
git clone https://github.com/clab/dynet.git
hg clone https://bitbucket.org/eigen/eigen -r 346ecdb  # -r NUM specified a known␣
↪working revision
cd dynet
mkdir build
cd build
# without GPU support:
cmake .. -DEIGEN3_INCLUDE_DIR=../eigen -DPYTHON=`which python`
# or with GPU support:
cmake .. -DEIGEN3_INCLUDE_DIR=../eigen -DPYTHON=`which python` -DBACKEND=cuda

make -j 2 # replace 2 with the number of available cores
cd python
python setup.py install  # or `python setup.py install --user` for a user-local␣
↪install.

# this should suffice, but on some systems you may need to add the following line to␣
↪your
# init files in order for the compiled .so files be accessible to Python.
```

```
# /path/to/dynet/build/dynet is the location in which libdynet.dylib resides.
export DYLD_LIBRARY_PATH=/path/to/dynet/build/dynet/:$DYLD_LIBRARY_PATH
```

2.2 Detailed Instructions

First, get DyNet:

```
cd $HOME
mkdir dynet-base
cd dynet-base
git clone https://github.com/clab/dynet.git
cd dynet
git submodule init # To be consistent with DyNet's installation instructions.
git submodule update # To be consistent with DyNet's installation instructions.
```

Then get Eigen:

```
cd $HOME
cd dynet-base
hg clone https://bitbucket.org/eigen/eigen/ -r 346ecdb
```

(*-r NUM* specifies a known working revision of Eigen. You can remove this in order to get the bleeding edge Eigen, with the risk of some compile breaks, and the possible benefit of added optimizations.)

We also need to make sure the `cython` module is installed. (you can replace `pip` with your favorite package manager, such as `conda`, or install within a virtual environment)

```
pip install cython
```

To simplify the following steps, we can set a bash variable to hold where we have saved the main directories of DyNet and Eigen. In case you have gotten DyNet and Eigen differently from the instructions above and saved them in different location(s), these variables will be helpful:

```
PATH_TO_DYNET=$HOME/dynet-base/dynet/
PATH_TO_EIGEN=$HOME/dynet-base/eigen/
```

Compile DyNet.

This is pretty much the same process as compiling DyNet, with the addition of the `-DPYTHON=` flag, pointing to the location of your Python interpreter.

If Boost is installed in a non-standard location, you should add the corresponding flags to the `cmake` commandline, see the DyNet installation instructions page.

```
cd $PATH_TO_DYNET
PATH_TO_PYTHON=`which python`
mkdir build
cd build
cmake .. -DEIGEN3_INCLUDE_DIR=$PATH_TO_EIGEN -DPYTHON=$PATH_TO_PYTHON
make -j 2
```

Assuming that the `cmake` command found all the needed libraries and didn't fail, the `make` command will take a while, and compile DyNet as well as the Python bindings. You can change `make -j 2` to a higher number, depending on the available cores you want to use while compiling.

You now have a working Python binding inside of `build/dynet`. To verify this is working:

```
cd $PATH_TO_DYNET/build/python
python
```

then, within Python:

```
import dynet as dy
print dy.__version__
model = dy.Model()
```

In order to install the module so that it is accessible from everywhere in the system, run the following:

```
cd $PATH_TO_DYNET/build/python
python setup.py install --user
```

The `--user` switch will install the module in your local site-packages, and works without root privileges. To install the module to the system site-packages (for all users), or to the current *virtualenv* (if you are on one), run `python setup.py install` without this switch.

You should now have a working python binding (the dynet module).

Note however that the installation relies on the compiled dynet library being in `$PATH_TO_DYNET/build/dynet`, so make sure not to move it from there.

Now, check that everything works:

```
cd $PATH_TO_DYNET
cd pyexamples
python xor.py
python rnnlm.py rnnlm.py
```

Alternatively, if the following script works for you, then your installation is likely to be working:

```
from dynet import *
model = Model()
```

If it doesn't work and you get an error similar to the following:

```
ImportError: dlopen(/Users/sneharajana/.python-eggs/dyNET-0.0.0-py2.7-macosx-10.11-
↪intel.egg-tmp/_dynet.so, 2): Library not loaded: @rpath/libdynet.dylib
Referenced from: /Users/sneharajana/.python-eggs/dyNET-0.0.0-py2.7-macosx-10.11-intel.
↪egg-tmp/_dynet.so
Reason: image not found``
```

then you may need to run the following (and add it to your shell init files):

export DYLD_LIBRARY_PATH=/path/to/dynet/build/dynet/:$DYLD_LIBRARY_PATH

2.3 Windows Support

You can also use Python on Windows by following similar steps to the above. For simplicity, we recommend using a Python distribution that already has Cython installed. The following has been tested to work:

1. Install WinPython 2.7.10 (comes with Cython already installed).
2. Run CMake as above with `-DPYTHON=/path/to/your/python.exe`.
3. Open a command prompt and set `VS90COMNTOOLS` to the path to your Visual Studio "Common7/Tools" directory. One easy way to do this is a command such as:

```
set VS90COMNTOOLS=%VS140COMNTOOLS%
```

4. Open dynet.sln from this command prompt and build the "Release" version of the solution.
5. Follow the rest of the instructions above for testing the build and installing it for other users

Note, currently only the Release version works.

2.4 GPU/MKL Support

2.4.1 Installing/running on GPU

For installing on a computer with GPU, first install CUDA. The following instructions assume CUDA is installed.

The installation process is pretty much the same, while adding the `-DBACKEND=cuda` flag to the `cmake` stage:

```
cmake .. -DEIGEN3_INCLUDE_DIR=$PATH_TO_EIGEN -DPYTHON=$PATH_TO_PYTHON -DBACKEND=cuda
```

(if CUDA is installed in a non-standard location and `cmake` cannot find it, you can specify also `-DCUDA_TOOLKIT_ROOT_DIR=/path/to/cuda`.)

Now, build the Python modules (as above, we assume Cython is installed):

After running `make -j 2`, you should have the files `_dynet.so` and `_gdynet.so` in the `build/python` folder.

As before, `cd build/python` followed by `python setup.py install --user` will install the module.

In order to use the GPU support, you can either:

- Use `import _gdynet as dy` instead of `import dynet as dy`
- Or, (preferred), `import dynet` as usual, but use the commandline switch `--dynet-gpu` or the GPU switches detailed here when invoking the program. This option lets the same code work with either the GPU or the CPU version depending on how it is invoked.

2.4.2 Running with MKL

If you've built DyNet to use MKL (using `-DMKL` or `-DMKL_ROOT`), Python sometimes has difficulty finding the MKL shared libraries. You can try setting `LD_LIBRARY_PATH` to point to your MKL library directory. If that doesn't work, try setting the following environment variable (supposing, for example, your MKL libraries are located at `/opt/intel/mkl/lib/intel64`):

```
export LD_PRELOAD=/opt/intel/mkl/lib/intel64/libmkl_def.so:/opt/intel/mkl/lib/intel64/
↪libmkl_avx2.so:/opt/intel/mkl/lib/intel64/libmkl_core.so:/opt/intel/mkl/lib/intel64/
↪libmkl_intel_lp64.so:/opt/intel/mkl/lib/intel64/libmkl_intel_thread.so:/opt/intel/
↪lib/intel64_lin/libiomp5.so
```

And get the basic information to create programs and use models:

CHAPTER 3

DyNet Tutorial

An illustration of how models are trained (for a simple logistic regression model) is below:

First, we set up the structure of the model.

Create a model, and an SGD trainer to update its parameters.

```
Model mod;
SimpleSGDTrainer sgd(mod);
```

Create a "computation graph," which will define the flow of information.

```
ComputationGraph cg;
```

Initialize a 1x3 parameter vector, and add the parameters to be part of the computation graph.

```
Expression W = parameter(cg, mod.add_parameters({1, 3}));
```

Create variables defining the input and output of the regression, and load them into the computation graph. Note that we don't need to set concrete values yet.

```
vector<dynet::real> x_values(3);
Expression x = input(cg, {3}, &x_values);
dynet::real y_value;
Expression y = input(cg, &y_value);
```

Next, set up the structure to multiply the input by the weight vector, then run the output of this through a logistic sigmoid function logistic regression).

```
Expression y_pred = logistic(W*x);
```

Finally, we create a function to calculate the loss. The model will be optimized to minimize the value of the final function in the computation graph.

```
Expression l = binary_log_loss(y_pred, y);
```

We are now done setting up the graph, and we can print out its structure:

```
cg.print_graphviz();
```

Now, we perform a parameter update for a single example. Set the input/output to the values specified by the training data:

```
x_values = {0.5, 0.3, 0.7};
y_value = 1.0;
```

"forward" propagates values forward through the computation graph, and returns the loss.

```
dynet::real loss = as_scalar(cg.forward(l));
```

"backward" performs back-propagation, and accumulates the gradients of the parameters within the "Model" data structure.

```
cg.backward(l);
```

"sgd.update" updates parameters of the model that was passed to its constructor. Here 1.0 is the scaling factor that allows us to control the size of the update.

```
sgd.update(1.0);
```

Note that this very simple example that doesn't cover things like memory initialization, reading/writing models, recurrent/LSTM networks, or adding biases to functions. The best way to get an idea of how to use DyNet for real is to look in the `example` directory, particularly starting with the simplest `xor` example.

3.1 Python Tutorials

Guided examples in Python can be found in the jupyter tutorials.

CHAPTER 4

Command Line Options

All programs using DyNet have a few command line options. These must be specified at the very beginning of the command line, before other options.

- `--dynet-mem NUMBER`: DyNet runs by default with 512MB of memory each for the forward and backward steps, as well as parameter storage. You will often want to increase this amount. By setting NUMBER here, DyNet will allocate more memory. Note that you can also individually set the amount of memory for forward calculation, backward calculation, and parameters by using comma separated variables `--dynet-mem FOR,BACK,PARAM`. This is useful if, for example, you are performing testing and don't need to allocate any memory for backward calculation.
- `--dynet-weight-decay NUMBER`: Adds weight decay to the parameters, which modifies each parameter w such that *w *= (1-weight_decay)* after every update. This is similar to L2 regularization, but different in a couple ways, which are noted in detail in the "Unorthodox Design" section.
- `--dynet-gpus NUMBER`: Specify how many GPUs you want to use, if DyNet is compiled with CUDA. Currently, only one GPU is supported.
- `--dynet-gpu-ids X,Y,Z`: Specify the GPUs that you want to use by device ID. Currently only one GPU is supported, but if you use this command you can select which one to use.

CHAPTER 5

Operations

5.1 Operation Interface

The following functions define DyNet "Expressions," which are used as an interface to the various functions that can be used to build DyNet computation graphs. Expressions for each specific function are listed below.

struct *#include <expr.h>* Expressions are the building block of a Dynet computation graph. [long description]

Public Functions

`dynet::expr::Expression::`**`Expression`**(ComputationGraph *_pg_, VariableIndex _i_)

Base expression constructor.

[long description]

Parameters

- `pg`: [description]
- `i`: [description]

5.2 Input Operations

These operations allow you to input something into the computation graph, either simple scalar/vector/matrix inputs from floats, or parameter inputs from a DyNet parameter object. They all requre passing a computation graph as input so you know which graph is being used for this particular calculation.

Expression `dynet::expr::`**`input`**(ComputationGraph &_g_, real _s_)

Scalar input.

Create an expression that represents the scalar value s

Return An expression representing s

Parameters

- `g`: Computation graph
- `s`: Real number

Expression `dynet::expr::input`(ComputationGraph &*g*, **const** real **ps*)
Modifiable scalar input.

Create an expression that represents the scalar value *ps. If *ps is changed and the computation graph recalculated, the next forward pass will reflect the new value.

Return An expression representing *ps

Parameters

- `g`: Computation graph
- `ps`: Real number pointer

Expression `dynet::expr::input`(ComputationGraph &*g*, **const** Dim &*d*, **const** std::vector<float> &*data*)
Vector/matrix/tensor input.

Create an expression that represents a vector, matrix, or tensor input. The dimensions of the input are defined by `d`. So for example > `input(g,{50},data)`: will result in a 50-length vector > `input(g,{50,30},data)`: will result in a 50x30 matrix and so on, for an arbitrary number of dimensions. This function can also be used to import minibatched inputs. For example, if we have 10 examples in a minibatch, each with size 50x30, then we call > `input(g,Dim({50,30},10),data)` The data vector "data" will contain the values used to fill the input, in column-major format. The length must add to the product of all dimensions in d.

Return An expression representing data

Parameters

- `g`: Computation graph
- `d`: Dimension of the input matrix
- `data`: A vector of data points

Expression `dynet::expr::input`(ComputationGraph &*g*, **const** Dim &*d*, **const** std::vector<float> **pdata*)
Updatable vector/matrix/tensor input.

Similarly to input that takes a vector reference, input a vector, matrix, or tensor input. Because we pass the pointer, the data can be updated.

Return An expression representing *pdata

Parameters

- `g`: Computation graph
- `d`: Dimension of the input matrix
- `pdata`: A pointer to an (updatable) vector of data points

Expression `dynet::expr::input`(ComputationGraph &*g*, **const** Dim &*d*, **const** std::vector<unsigned int> &*ids*, **const** std::vector<float> &*data*, float *defdata* = 0.f)
Sparse vector input.

This operation takes input as a sparse matrix of index/value pairs. It is exactly the same as the standard input via vector reference, but sets all non-specified values to "defdata" and resets all others to the appropriate input values.

Return An expression representing data

Parameters

- `g`: Computation graph
- `d`: Dimension of the input matrix
- `ids`: The indexes of the data points to update
- `data`: The data points corresponding to each index
- `defdata`: The default data with which to set the unspecified data points

Expression `dynet::expr::parameter`(ComputationGraph &*g*, Parameter *p*)

Load parameter.

Load parameters into the computation graph.

Return An expression representing p

Parameters

- `g`: Computation graph
- `p`: Parameter object to load

Expression `dynet::expr::const_parameter`(ComputationGraph &*g*, Parameter *p*)

Load constant parameters.

Load parameters into the computation graph, but prevent them from being updated when performing parameter update.

Return An expression representing the constant p

Parameters

- `g`: Computation graph
- `p`: Parameter object to load

Expression `dynet::expr::lookup`(ComputationGraph &*g*, LookupParameter *p*, unsigned *index*)

Look up parameter.

Look up parameters according to an index, and load them into the computation graph.

Return An expression representing p[index]

Parameters

- `g`: Computation graph
- `p`: LookupParameter object from which to load
- `index`: Index of the parameters within p

Expression dynet::expr::**lookup**(ComputationGraph &*g*, LookupParameter *p*, **const** unsigned **pindex*)

Look up parameters with modifiable index.

Look up parameters according to the *pindex, and load them into the computation graph. When *pindex changes, on the next computation of forward() the values will change.

Return An expression representing p[*pindex]

Parameters

- g: Computation graph
- p: LookupParameter object from which to load
- pindex: Pointer index of the parameters within p

Expression dynet::expr::**const_lookup**(ComputationGraph &*g*, LookupParameter *p*, unsigned *index*)

Look up parameter.

Look up parameters according to an index, and load them into the computation graph. Do not perform gradient update on the parameters.

Return A constant expression representing p[index]

Parameters

- g: Computation graph
- p: LookupParameter object from which to load
- index: Index of the parameters within p

Expression dynet::expr::**const_lookup**(ComputationGraph &*g*, LookupParameter *p*, **const** unsigned **pindex*)

Constant lookup parameters with modifiable index.

Look up parameters according to the *pindex, and load them into the computation graph. When *pindex changes, on the next computation of forward() the values will change. However, gradient updates will not be performend.

Return A constant expression representing p[*pindex]

Parameters

- g: Computation graph
- p: LookupParameter object from which to load
- pindex: Pointer index of the parameters within p

Expression dynet::expr::**lookup**(ComputationGraph &*g*, LookupParameter *p*, **const** std::vector<unsigned> &*indices*)

Look up parameters.

The mini-batched version of lookup. The resulting expression will be a mini-batch of parameters, where the "i"th element of the batch corresponds to the parameters at the position specified by the "i"th element of "indices"

Return An expression with the "i"th batch element representing p[indices[i]]

Parameters

- g: Computation graph

- p: LookupParameter object from which to load
- indices: Index of the parameters at each position in the batch

Expression dynet::expr::**lookup**(ComputationGraph &*g*, LookupParameter *p*, **const** std::vector<unsigned> **pindices*)

Look up parameters.

The mini-batched version of lookup with modifiable parameter indices.

Return An expression with the "i"th batch element representing p[*pindices[i]]

Parameters

- g: Computation graph
- p: LookupParameter object from which to load
- pindices: Pointer to lookup indices

Expression dynet::expr::**const_lookup**(ComputationGraph &*g*, LookupParameter *p*, **const** std::vector<unsigned> &*indices*)

Look up parameters.

Mini-batched lookup that will not update the parameters.

Return A constant expression with the "i"th batch element representing p[indices[i]]

Parameters

- g: Computation graph
- p: LookupParameter object from which to load
- indices: Lookup indices

Expression dynet::expr::**const_lookup**(ComputationGraph &*g*, LookupParameter *p*, **const** std::vector<unsigned> **pindices*)

Look up parameters.

Mini-batched lookup that will not update the parameters, with modifiable indices.

Return A constant expression with the "i"th batch element representing p[*pindices[i]]

Parameters

- g: Computation graph
- p: LookupParameter object from which to load
- pindices: Lookup index pointers.

Expression dynet::expr::**zeroes**(ComputationGraph &*g*, **const** Dim &*d*)

Create an input full of zeros.

Create an input full of zeros, sized according to dimensions d.

Return A "d" dimensioned zero vector

Parameters

- g: Computation graph
- d: The dimensions of the input

Expression dynet::expr::**random_normal**(ComputationGraph &*g*, **const** Dim &*d*)
Create a random normal vector.

Create a vector distributed according to normal distribution with mean 0, variance 1.

Return A "d" dimensioned normally distributed vector

Parameters

- g: Computation graph
- d: The dimensions of the input

Expression dynet::expr::**random_bernoulli**(ComputationGraph &*g*, **const** Dim &*d*, real *p*, real *scale* = 1.0f)
Create a random bernoulli vector.

Create a vector distributed according to bernoulli distribution with parameter p.

Return A "d" dimensioned bernoulli distributed vector

Parameters

- g: Computation graph
- d: The dimensions of the input
- p: The bernoulli p parameter
- scale: A scaling factor for the output ("active" elements will receive this value)

Expression dynet::expr::**random_uniform**(ComputationGraph &*g*, **const** Dim &*d*, real *left*, real *right*)
Create a random uniform vector.

Create a vector distributed according to uniform distribution with boundaries left and right.

Return A "d" dimensioned uniform distributed vector

Parameters

- g: Computation graph
- d: The dimensions of the input
- left: The left boundary
- right: The right boundary

5.3 Arithmetic Operations

These operations perform basic arithemetic over values in the graph.

Expression dynet::expr::**operator-**(**const** *Expression* &*x*)
Negation.

Negate the passed argument.

Return The negation of x

Parameters

- x: An input expression

Expression dynet::expr::**operator+**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Expression addition.

Add two expressions of the same dimensions.

Return The sum of x and y

Parameters

- x: The first input
- y: The second input

Expression dynet::expr::**operator+**(**const** *Expression* &*x*, real *y*)
Scalar addition.

Add a scalar to an expression

Return An expression equal to x, with every component increased by y

Parameters

- x: The expression
- y: The scalar

Expression dynet::expr::**operator+**(real *x*, **const** *Expression* &*y*)
Scalar addition.

Add a scalar to an expression

Return An expression equal to y, with every component increased by x

Parameters

- x: The scalar
- y: The expression

Expression dynet::expr::**operator-**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Expression subtraction.

Subtract one expression from another.

Return An expression where the ith element is x_i minus y_i

Parameters

- x: The expression from which to subtract
- y: The expression to subtract

Expression dynet::expr::**operator-**(real *x*, **const** *Expression* &*y*)
Scalar subtraction.

Subtract an expression from a scalar

Return An expression where the ith element is x_i minus y

Parameters

- x: The scalar from which to subtract
- y: The expression to subtract

Expression dynet::expr::**operator-**(**const** *Expression* &*x*, real *y*)
Scalar subtraction.

Subtract a scalar from an expression

Return An expression where the ith element is x_i minus y

Parameters

- x: The expression from which to subtract
- y: The scalar to subtract

Expression dynet::expr::**operator***(**const** *Expression* &*x*, **const** *Expression* &*y*)
Matrix multiplication.

Multiply two matrices together. Like standard matrix multiplication, the second dimension of x and the first dimension of y must match.

Return An expression x times y

Parameters

- x: The left-hand matrix
- y: The right-hand matrix

Expression dynet::expr::**operator***(**const** *Expression* &*x*, float *y*)
Matrix-scalar multiplication.

Multiply an expression component-wise by a scalar.

Return An expression where the ith element is x_i times y

Parameters

- x: The matrix
- y: The scalar

Expression dynet::expr::**operator***(float *y*, **const** *Expression* &*x*)
Matrix-scalar multiplication.

Multiply an expression component-wise by a scalar.

Return An expression where the ith element is x_i times y

Parameters

- x: The scalar
- y: The matrix

Expression dynet::expr::**operator/**(**const** *Expression* &*x*, float *y*)
Matrix-scalar division.

Divide an expression component-wise by a scalar.

Return An expression where the ith element is x_i divided by y

Parameters

- x: The matrix

- `y`: The scalar

Expression `dynet::expr::`**`affine_transform`**(**const** std::initializer_list<*Expression*> &*xs*)
Affine transform.

This performs an affine transform over an arbitrary (odd) number of expressions held in the input initializer list xs. The first expression is the "bias," which is added to the expression as-is. The remaining expressions are multiplied together in pairs, then added. A very common usage case is the calculation of the score for a neural network layer (e.g. b + Wz) where b is the bias, W is the weight matrix, and z is the input. In this case xs[0] = b, xs[1] = W, and xs[2] = z.

Return An expression equal to: xs[0] + xs[1]*xs[2] + xs[3]*xs[4] + ...

Parameters

- `xs`: An initializer list containing an odd number of expressions

Expression `dynet::expr::`**`sum`**(**const** std::initializer_list<*Expression*> &*xs*)
Sum.

This performs an elementwise sum over all the expressions in xs

Return An expression where the ith element is equal to xs[0][i] + xs[1][i] + ...

Parameters

- `xs`: An initializer list containing expressions

Expression `dynet::expr::`**`average`**(**const** std::initializer_list<*Expression*> &*xs*)
Average.

This performs an elementwise average over all the expressions in xs

Return An expression where the ith element is equal to (xs[0][i] + xs[1][i] + ...)/|xs|

Parameters

- `xs`: An initializer list containing expressions

Expression `dynet::expr::`**`sqrt`**(**const** *Expression* &*x*)
Square root.

Elementwise square root.

Return An expression where the ith element is equal to sqrt(x_i)

Parameters

- `x`: The input expression

Expression `dynet::expr::`**`erf`**(**const** *Expression* &*x*)
Gaussian error function.

Elementwise calculation of the Gaussian error function

Return An expression where the ith element is equal to erf(x_i)

Parameters

- `x`: The input expression

Expression dynet::expr::**tanh**(**const** *Expression* &*x*)
Hyperbolic tangent.

Elementwise calculation of the hyperbolic tangent

Return An expression where the ith element is equal to tanh(x_i)

Parameters

- x: The input expression

Expression dynet::expr::**exp**(**const** *Expression* &*x*)
Natural exponent.

Calculate elementwise y_i = e^{x_i}

Return An expression where the ith element is equal to e^{x_i}

Parameters

- x: The input expression

Expression dynet::expr::**square**(**const** *Expression* &*x*)
Square.

Calculate elementwise y_i = x_i^2

Return An expression where the ith element is equal to x_i^2

Parameters

- x: The input expression

Expression dynet::expr::**cube**(**const** *Expression* &*x*)
Cube.

Calculate elementwise y_i = x_i^3

Return An expression where the ith element is equal to x_i^3

Parameters

- x: The input expression

Expression dynet::expr::**lgamma**(**const** *Expression* &*x*)
Log gamma.

Calculate elementwise y_i = ln(gamma(x_i))

Return An expression where the ith element is equal to ln(gamma(x_i))

Parameters

- x: The input expression

Expression dynet::expr::**log**(**const** *Expression* &*x*)
Logarithm.

Calculate the elementwise natural logarithm y_i = ln(x_i)

Return An expression where the ith element is equal to ln(x_i)

Parameters

- x: The input expression

Expression dynet::expr::**logistic**(**const** *Expression* &*x*)
Logistic sigmoid function.

Calculate elementwise y_i = 1/(1+e^{x_i})

Return An expression where the ith element is equal to y_i = 1/(1+e^{x_i})

Parameters

- x: The input expression

Expression dynet::expr::**rectify**(**const** *Expression* &*x*)
Rectifier.

Calculate elementwise the recitifer (ReLU) function y_i = max(x_i,0)

Return An expression where the ith element is equal to max(x_i,0)

Parameters

- x: The input expression

Expression dynet::expr::**softsign**(**const** *Expression* &*x*)
Soft Sign.

Calculate elementwise the softsign function y_i = x_i/(1+|x_i|)

Return An expression where the ith element is equal to x_i/(1+|x_i|)

Parameters

- x: The input expression

Expression dynet::expr::**pow**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Power function.

Calculate an output where the ith element is equal to x_i^y_i

Return An expression where the ith element is equal to x_i^y_i

Parameters

- x: The input expression
- y: The exponent expression

Expression dynet::expr::**min**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Minimum.

Calculate an output where the ith element is min(x_i,y_i)

Return An expression where the ith element is equal to min(x_i,y_i)

Parameters

- x: The first input expression
- y: The second input expression

Expression dynet::expr::**max**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Maximum.

Calculate an output where the ith element is max(x_i,y_i)

Return An expression where the ith element is equal to max(x_i,y_i)

Parameters

- x: The first input expression
- y: The second input expression

Expression dynet::expr::**max**(**const** std::initializer_list<*Expression*> &*xs*)
Max.

This performs an elementwise max over all the expressions in xs

Return An expression where the ith element is equal to max(xs[0][i], xs[1][i], ...)

Parameters

- xs: An initializer list containing expressions

Expression dynet::expr::**dot_product**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Dot Product.

Calculate the dot product sum_i x_i*y_i

Return An expression equal to the dot product

Parameters

- x: The input expression
- y: The input expression

Expression dynet::expr::**cmult**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Componentwise multiply.

Do a componentwise multiply where each value is equal to x_i*y_i. This function used to be called cwise_multiply.

Return An expression where the ith element is equal to x_i*y_i

Parameters

- x: The first input expression
- y: The second input expression

Expression dynet::expr::**cdiv**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Componentwise multiply.

Do a componentwise multiply where each value is equal to x_i/y_i

Return An expression where the ith element is equal to x_i/y_i

Parameters

- x: The first input expression
- y: The second input expression

Expression dynet::expr::**colwise_add**(**const** *Expression* &*x*, **const** *Expression* &*bias*)
Columnwise addition.

Add vector "bias" to each column of matrix "x"

Return An expression where bias is added to each column of x

Parameters

- x: An MxN matrix
- bias: A length M vector

5.4 Probability/Loss Operations

These operations are used for calculating probabilities, or calculating loss functions for use in training.

Expression dynet::expr::**softmax**(**const** *Expression* &*x*)
Softmax.

The softmax function, which sets each element to be e^{x[i]}/{sum_j e^{x[j]}}.

Return A vector after calculating the softmax

Parameters

- x: A vector

Expression dynet::expr::**log_softmax**(**const** *Expression* &*x*)
Log softmax.

The log of the softmax function, which sets each element to be log(e^{x[i]}/{sum_j e^{x[j]}}).

Return A vector after calculating the log softmax

Parameters

- x: A vector

Expression dynet::expr::**log_softmax**(**const** *Expression* &*x*, **const** std::vector<unsigned> &*restriction*)
Restricted log softmax.

The log softmax function calculated over only a subset of the vector elements. The elements to be included are set by the restriction variable. All elements not included in restriction are set to negative infinity.

Return A vector with the log softmax over the specified elements

Parameters

- x: A vector over which to calculate the softmax
- restriction: The elements over which to calculate the softmax

Expression dynet::expr::**logsumexp**(**const** std::initializer_list<*Expression*> &*xs*)
Log, sum, exp.

The elementwise "logsumexp" function that calculates $ln(\sum_i e^{xs_i})$, used in adding probabilities in the log domain.

Return The result.

Parameters

- `xs`: Expressions with respect to which to calculate the logsumexp.

Expression `dynet::expr::`**`pickneglogsoftmax`**(**const** *Expression* &*x*, unsigned *v*)

Negative softmax log likelihood.

This function takes in a vector of scores `x`, and performs a log softmax, takes the negative, and selects the likelihood corresponding to the element `v`. This is perhaps the most standard loss function for training neural networks to predict one out of a set of elements.

Return The negative log likelihood of element `v` after taking the softmax

Parameters

- `x`: A vector of scores
- `v`: The element with which to calculate the loss

Expression `dynet::expr::`**`pickneglogsoftmax`**(**const** *Expression* &*x*, unsigned **pv*)

Modifiable negative softmax log likelihood.

This function calculates the negative log likelihood after the softmax with respect to index `*pv`. This computes the same value as the previous function that passes the index `v` by value, but instead passes by pointer so the value `*pv` can be modified without re-constructing the computation graph. This can be used in situations where we want to create a computation graph once, then feed it different data points.

Return The negative log likelihood of element `*pv` after taking the softmax

Parameters

- `x`: A vector of scores
- `pv`: A pointer to the index of the correct element

Expression `dynet::expr::`**`pickneglogsoftmax`**(**const** *Expression* &*x*, **const** std::vector<unsigned> &*v*)

Batched negative softmax log likelihood.

This function is similar to standard pickneglogsoftmax, but calculates loss with respect to multiple batch elements. The input will be a mini-batch of score vectors where the number of batch elements is equal to the number of indices in `v`.

Return The negative log likelihoods over all the batch elements

Parameters

- `x`: An expression with vectors of scores over N batch elements
- `v`: A size-N vector indicating the index with respect to all the batch elements

Expression `dynet::expr::`**`pickneglogsoftmax`**(**const** *Expression* &*x*, **const** std::vector<unsigned> **pv*)

Modifiable batched negative softmax log likelihood.

This function is a combination of modifiable pickneglogsoftmax and batched pickneglogsoftmax: `pv` can be modified without re-creating the computation graph.

Return The negative log likelihoods over all the batch elements

Parameters

- `x`: An expression with vectors of scores over N batch elements
- `pv`: A pointer to the indexes

Expression `dynet::expr::hinge`(**const** *Expression* &*x*, unsigned *index*, float *m* = 1.0)

Hinge loss.

This expression calculates the hinge loss, formally expressed as: hinge$(x, index, m) = \sum_{i \neq index} \max(0, m - x[index] + x[i])$.

Return The hinge loss of candidate `index` with respect to margin `m`

Parameters

- `x`: A vector of scores
- `index`: The index of the correct candidate
- `m`: The margin

Expression `dynet::expr::hinge`(**const** *Expression* &*x*, **const** unsigned **pindex*, float *m* = 1.0)

Modifiable hinge loss.

This function calculates the hinge loss with with respect to index `*pindex`. This computes the same value as the previous function that passes the index `index` by value, but instead passes by pointer so the value `*pindex` can be modified without re-constructing the computation graph. This can be used in situations where we want to create a computation graph once, then feed it different data points.

Return The hinge loss of candidate `*pindex` with respect to margin `m`

Parameters

- `x`: A vector of scores
- `pindex`: A pointer to the index of the correct candidate
- `m`: The margin

Expression `dynet::expr::hinge`(**const** *Expression* &*x*, **const** std::vector<unsigned> &*indices*, float *m* = 1.0)

Batched hinge loss.

The same as hinge loss, but for the case where `x` is a mini-batched tensor with `indices.size()` batch elements, and `indices` is a vector indicating the index of each of the correct elements for these elements.

Return The hinge loss of each mini-batch

Parameters

- `x`: A mini-batch of vectors with `indices.size()` batch elements
- `indices`: The indices of the correct candidates for each batch element
- `m`: The margin

Expression `dynet::expr::hinge`(**const** *Expression* &*x*, **const** std::vector<unsigned> **pindices*, float *m* = 1.0)

Batched modifiable hinge loss.

A combination of the previous batched and modifiable hinge loss functions, where vector `*pindices` can be modified.

Return The hinge loss of each mini-batch

Parameters

- `x`: A mini-batch of vectors with `indices.size()` batch elements
- `pindices`: Pointer to the indices of the correct candidates for each batch element
- `m`: The margin

Expression `dynet::expr::`**`sparsemax`**(**const** *Expression* &*x*)

Sparsemax.

The sparsemax function (Martins et al. 2016), which is similar to softmax, but induces sparse solutions where most of the vector elements are zero. **Note:** This function is not yet implemented on GPU.

Return The sparsemax of the scores

Parameters

- `x`: A vector of scores

Expression `dynet::expr::`**`sparsemax_loss`**(**const** *Expression* &*x*, **const** std::vector<unsigned> &*target_support*)

Sparsemax loss.

The sparsemax loss function (Martins et al. 2016), which is similar to softmax loss, but induces sparse solutions where most of the vector elements are zero. It has a gradient similar to the sparsemax function and thus is useful for optimizing when the sparsemax will be used at test time. **Note:** This function is not yet implemented on GPU.

Return The sparsemax loss of the labels

Parameters

- `x`: A vector of scores
- `target_support`: The target correct labels.

Expression `dynet::expr::`**`sparsemax_loss`**(**const** *Expression* &*x*, **const** std::vector<unsigned> **ptarget_support*)

Modifiable sparsemax loss.

Similar to the sparsemax loss, but with ptarget_support being a pointer to a vector, allowing it to be modified without re-creating the compuation graph. **Note:** This function is not yet implemented on GPU.

Return The sparsemax loss of the labels

Parameters

- `x`: A vector of scores
- `ptarget_support`: A pointer to the target correct labels.

Expression `dynet::expr::`**`squared_norm`**(**const** *Expression* &*x*)

Squared norm.

The squared norm of the values of x: $\sum_i x_i^2$.

Return The squared norm

Parameters

- x: A vector of values

Expression dynet::expr::**squared_distance**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Squared distance.

The squared distance between values of x and y: $\sum_i (x_i - y_i)^2$.

Return The squared distance

Parameters

- x: A vector of values
- y: Another vector of values

Expression dynet::expr::**l1_distance**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Squared distance.

The L1 distance between values of x and y: $\sum_i |x_i - y_i|$.

Return The squared distance

Parameters

- x: A vector of values
- y: Another vector of values

Expression dynet::expr::**huber_distance**(**const** *Expression* &*x*, **const** *Expression* &*y*, float *c* = 1.345f)
Huber distance.

The huber distance between values of x and y parameterized by c, $\sum_i L_c(x_i, y_i)$ where:

$$L_c(x, y) = \begin{cases} lr\frac{1}{2}(y - x)^2 & \text{for } |y - f(x)| \leq c, \\ c\,|y - f(x)| - \frac{1}{2}c^2 & \text{otherwise.} \end{cases}$$

Return The huber distance

Parameters

- x: A vector of values
- y: Another vector of values
- c: The parameter of the huber distance parameterizing the cuttoff

Expression dynet::expr::**binary_log_loss**(**const** *Expression* &*x*, **const** *Expression* &*y*)
Binary log loss.

The log loss of a binary decision according to the sigmoid sigmoid function $-\sum_i (y_i * ln(x_i) + (1 - y_i) * ln(1 - x_i))$

Return The log loss of the sigmoid function

Parameters

- x: A vector of values
- y: A vector of true answers

Expression dynet::expr::**pairwise_rank_loss**(**const** *Expression* &*x*, **const** *Expression* &*y*, real *m* = 1.0)

Pairwise rank loss.

A margin-based loss, where every margin violation for each pair of values is penalized: $\sum_i max(x_i - y_i + m, 0)$

Return The pairwise rank loss

Parameters

- x: A vector of values
- y: A vector of true answers
- m: The margin

Expression dynet::expr::**poisson_loss**(**const** *Expression* &*x*, unsigned *y*)

Poisson loss.

The negative log probability of y according to a Poisson distribution with parameter x. Useful in Poisson regression where, we try to predict the parameters of a Possion distribution to maximize the probability of data y.

Return The Poisson loss

Parameters

- x: The parameter of the Poisson distribution.
- y: The target value

Expression dynet::expr::**poisson_loss**(**const** *Expression* &*x*, **const** unsigned **py*)

Modifiable Poisson loss.

Similar to Poisson loss, but with the target value passed by pointer so that it can be modified without reconstructing the computation graph.

Return The Poisson loss

Parameters

- x: The parameter of the Poisson distribution.
- py: A pointer to the target value

5.5 Flow/Shaping Operations

These operations control the flow of information through the graph, or the shape of the vectors/tensors used in the graph.

Expression dynet::expr::**nobackprop**(**const** *Expression* &*x*)

Prevent backprop.

This node has no effect on the forward pass, but prevents gradients from flowing backward during the backward pass. This is useful when there's a subgraph for which you don't want loss passed back to the parameters.

Return The new expression

Parameters

- x: The input expression

Expression dynet::expr::**reshape**(**const** *Expression* &*x*, **const** Dim &*d*)

Reshape to another size.

This node reshapes a tensor to another size, without changing the underlying layout of the data. The layout of the data in DyNet is column-major, so if we have a 3x4 matrix

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} \end{pmatrix}$$

```
    and transform it into a 2x6 matrix, it will be rearranged as:
```

$$\begin{pmatrix} x_{1,1} & x_{3,1} & x_{2,2} & x_{1,3} & x_{3,3} & x_{2,4} \\ x_{1,2} & x_{1,2} & x_{3,2} & x_{2,3} & x_{1,4} & x_{3,4} \end{pmatrix}$$

```
   **Note:** This is O(1) for forward, and O(n) for backward.
```

Return The reshaped expression

Parameters

- x: The input expression
- d: The new dimensions

Expression dynet::expr::**transpose**(**const** *Expression* &*x*)

Transpose a matrix.

Get the transpose of the matrix. **Note:** This is O(1) if either the row or column dimension is 1, and O(n) otherwise.

Return The transposed expression

Parameters

- x: The input expression

Expression dynet::expr::**select_rows**(**const** *Expression* &*x*, **const** std::vector<unsigned> &*rows*)

Select rows.

Select a subset of rows of a matrix.

Return An expression containing the selected rows

Parameters

- x: The input expression
- rows: The rows to extract

Expression dynet::expr::**select_rows**(**const** *Expression* &*x*, **const** std::vector<unsigned> **prows*)

Modifiable select rows.

Select a subset of rows of a matrix, where the elements of prows can be modified without re-creating the computation graph.

Return An expression containing the selected rows

Parameters

- x: The input expression

- `prows`: The rows to extract

Expression `dynet::expr::`**`select_cols`**(**const** *Expression* &*x*, **const** std::vector<unsigned> &*cols*)
Select columns.

Select a subset of columns of a matrix. select_cols is more efficient than select_rows since DyNet uses column-major order.

Return An expression containing the selected columns

Parameters

- `x`: The input expression
- `columns`: The columns to extract

Expression `dynet::expr::`**`select_cols`**(**const** *Expression* &*x*, **const** std::vector<unsigned> **pcols*)
Modifiable select columns.

Select a subset of columns of a matrix, where the elements of pcols can be modified without re-creating the computation graph.

Return An expression containing the selected columns

Parameters

- `x`: The input expression
- `pcolumns`: The columns to extract

Expression `dynet::expr::`**`sum_batches`**(**const** *Expression* &*x*)
Sum over minibatches.

Sum an expression that consists of multiple minibatches into one of equal dimension but with only a single minibatch. This is useful for summing loss functions at the end of minibatch training.

Return An expression with a single batch

Parameters

- `x`: The input mini-batched expression

Expression `dynet::expr::`**`pick`**(**const** *Expression* &*x*, unsigned *v*)
Pick element.

Pick a single element from an expression.

Return The value of x[v]

Parameters

- `x`: The input expression
- `v`: The index of the element to select

Expression `dynet::expr::`**`pick`**(**const** *Expression* &*x*, **const** std::vector<unsigned> &*v*)
Pick multiple elements.

Pick multiple elements from an input expression

Return A vector of values {x[v[0]], x[v[1]], ...}

Parameters

- x: The input expression
- v: A vector of indicies to choose

Expression dynet::expr::**pick**(**const** *Expression* &*x*, unsigned **pv*)
Modifiable pick element.

Pick a single element from an expression, where the index is passed by pointer so we do not need to re-create the computation graph every time.

Return The value of x[*pv]

Parameters

- x: The input expression
- pv: Pointer to the index of the element to select

Expression dynet::expr::**pick**(**const** *Expression* &*x*, **const** std::vector<unsigned> **pv*)
Modifiable pick multiple elements.

Pick multiple elements from an input expression, where the indices are passed by pointer so we do not need to re-create the computation graph every time.

Return A vector of values {x[(*pv)[0]], x[(*pv)[1]], ...}

Parameters

- x: The input expression
- pv: A pointer to vector of indicies to choose

Expression dynet::expr::**pickrange**(**const** *Expression* &*x*, unsigned *v*, unsigned *u*)
Pick range of elements.

Pick a range of elements from an expression.

Return The value of {x[v],...,x[u]}

Parameters

- x: The input expression
- v: The beginning index
- u: The end index

Expression dynet::expr::**concatenate_cols**(**const** std::initializer_list<*Expression*> &*xs*)
Concatenate columns.

Perform a concatenation of the columns in multiple expressions. All expressions must have the same number of rows.

Return The expression with the columns concatenated

Parameters

- xs: The input expressions

Expression dynet::expr::**concatenate**(**const** std::initializer_list<*Expression*> &*xs*)
Concatenate rows.

Perform a concatenation of the rows in multiple expressions. All expressions must have the same number of columns.

Return The expression with the rows concatenated

Parameters

- xs: The input expressions

5.6 Noise Operations

These operations are used to add noise to the graph for purposes of making learning more robust.

Expression dynet::expr::**noise**(**const** *Expression* &*x*, real *stddev*)
Gaussian noise.

Add gaussian noise to an expression.

Return The noised expression

Parameters

- x: The input expression
- stddev: The standard deviation of the gaussian

Expression dynet::expr::**dropout**(**const** *Expression* &*x*, real *p*)
Dropout.

With a fixed probability, drop out (set to zero) nodes in the input expression, and **scale** the remaining nodes by 1/p. Note that there are two kinds of dropout:

- *Regular dropout:* where we perform dropout at training time and then scale outputs by p at test time.
- *Inverted dropout:* where we perform dropout and scaling at training time, and do not need to do anything at test time. DyNet implements the latter, so you only need to apply dropout at training time, and do not need to perform scaling and test time.

Return The dropped out expression

Parameters

- x: The input expression
- p: The dropout probability

Expression dynet::expr::**block_dropout**(**const** *Expression* &*x*, real *p*)
Block dropout.

Identical to the dropout operation, but either drops out *all* or *no* values in the expression, as opposed to making a decision about each value individually.

Return The block dropout expression

Parameters

- x: The input expression

- p: The block dropout probability

5.7 Tensor Operations

These operations are used for performing operations on higher order tensors.

5.8 Linera Algebra Operations

These operations are used for performing various operations common in linear algebra.

CHAPTER 6

Builders

Builders combine together various operations to implement more complicated things such as recurrent and LSTM networks

struct *#include <rnn.h>* interface for constructing an RNN, LSTM, GRU, etc. [long description] Subclassed by *dynet::SimpleRNNBuilder*

Public Functions

`dynet::RNNBuilder::`**`RNNBuilder`**`()`

Default constructor.

RNNPointer `dynet::RNNBuilder::`**`state`**`()` **const**

Get pointer to the current state.

Return Pointer to the current state

void `dynet::RNNBuilder::`**`new_graph`**`(`ComputationGraph *&cg*`)`

Initialize with new computation graph.

call this to reset the builder when you are working with a newly created ComputationGraph object

Parameters

- `cg`: Computation graph

void `dynet::RNNBuilder::`**`start_new_sequence`**`(`**const** std::vector<Expression> *&h_0* = {}`)`

Reset for new sequence.

call this before add_input and after new_graph, when starting a new sequence on the same hypergraph.

Parameters

- `h_0`: `h_0` is used to initialize hidden layers at timestep 0 to given values

Expression `dynet::RNNBuilder::`**`set_h`**`(`**const** RNNPointer *&prev*, **const** std::vector<Expression> *&h_new* = {}`)`

Explicitly set the output state of a node.

Return The hidden representation of the deepest layer

Parameters

- `prev`: Pointer to the previous state
- `h_new`: The new hidden state

Expression `dynet::RNNBuilder::set_s`(**const** RNNPointer &*prev*, **const** std::vector<Expression> &*s_new* = {})
Set the internal state of a node (for lstms/grus)

For RNNs without internal states (SimpleRNN, GRU...), this has the same behaviour as `set_h`

Return The hidden representation of the deepest layer

Parameters

- `prev`: Pointer to the previous state
- `s_new`: The new state. Can be `{new_c[0],...,new_c[n]}` or `{new_c[0],..., new_c[n], new_h[0],...,new_h[n]}`

Expression `dynet::RNNBuilder::add_input`(**const** Expression &*x*)
Add another timestep by reading in the variable x.

Return The hidden representation of the deepest layer

Parameters

- `x`: Input variable

Expression `dynet::RNNBuilder::add_input`(**const** RNNPointer &*prev*, **const** Expression &*x*)
Add another timestep, with arbitrary recurrent connection.

This allows you to define a recurrent connection to `prev` rather than to `head[cur]`. This can be used to construct trees, implement beam search, etc.

Return The hidden representation of the deepest layer

Parameters

- `prev`: Pointer to the previous state
- `x`: Input variable

void `dynet::RNNBuilder::rewind_one_step`()
Rewind the last timestep.

- this DOES NOT remove the variables from the computation graph, it just means the next time step will see a different previous state. You can rewind as many times as you want.

RNNPointer `dynet::RNNBuilder::get_head`(**const** RNNPointer &*p*)
Return the RNN state that is the parent of `p`

- This can be used in implementing complex structures such as trees, etc.

void `dynet::RNNBuilder::set_dropout`(float *d*)
Set Dropout.

Parameters

- d: Dropout rate

void dynet::RNNBuilder::**disable_dropout**()
Disable Dropout.

In general, you should disable dropout at test time

virtual Expression dynet::RNNBuilder::**back**() **const**
= 0Returns node (index) of most recent output.

Return Node (index) of most recent output

virtual std::vector<Expression> dynet::RNNBuilder::**final_h**() **const**
= 0Access the final output of each hidden layer.

Return Final output of each hidden layer

virtual std::vector<Expression> dynet::RNNBuilder::**get_h**(RNNPointer *i*) **const**
= 0Access the output of any hidden layer.

Return Output of each hidden layer at the given step

Parameters

- i: Pointer to the step which output you want to access

virtual std::vector<Expression> dynet::RNNBuilder::**final_s**() **const**
= 0Access the final state of each hidden layer.

This returns the state of each hidden layer, in a format that can be used in start_new_sequence (i.e. including any internal cell for LSTMs and the likes)

Return vector containing, if it exists, the list of final internal states, followed by the list of final outputs for each layer

virtual std::vector<Expression> dynet::RNNBuilder::**get_s**(RNNPointer *i*) **const**
= 0Access the state of any hidden layer.

See final_s for details

Return Internal state of each hidden layer at the given step

Parameters

- i: Pointer to the step which state you want to access

virtual unsigned dynet::RNNBuilder::**num_h0_components**() **const**
= 0Number of components in h_0

Return Number of components in h_0

virtual void dynet::RNNBuilder::**copy**(**const** *RNNBuilder* &*params*)
= 0Copy the parameters of another builder.

Parameters

- `params`: *RNNBuilder* you want to copy parameters from.

virtual void `dynet::RNNBuilder::`**`save_parameters_pretraining`**(**const** std::string *&fname*) **const**

This function saves all the parameters associated with a particular *RNNBuilder*'s derived class to a file.

This should not be used to seralize models, it should only be used to save parameters for pretraining. If you are interested in serializing models, use the boost serialization API against your model class.

Parameters

- `fname`: File you want to save your model to.

virtual void `dynet::RNNBuilder::`**`load_parameters_pretraining`**(**const** std::string *&fname*)

Loads all the parameters associated with a particular *RNNBuilder*'s derived class from a file.

This should not be used to seralize models, it should only be used to load parameters from pretraining. If you are interested in serializing models, use the boost serialization API against your model class.

Parameters

- `fname`: File you want to read your model from.

struct *#include <rnn.h>* This provides a builder for the simplest RNN with tanh nonlinearity. The equation for this RNN is : $h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$ Inherits from *dynet::RNNBuilder* **Public Functions**

`dynet::SimpleRNNBuilder::`**`SimpleRNNBuilder`**(unsigned *layers*, unsigned *input_dim*, unsigned *hidden_dim*, Model *&model*, bool *support_lags* = false)

Builds a simple RNN.

Parameters

- `layers`: Number of layers
- `input_dim`: Dimension of the input
- `hidden_dim`: Hidden layer (and output) size
- `model`: Model holding the parameters
- `support_lags`: Allow for auxiliary output?

Expression `dynet::SimpleRNNBuilder::`**`add_auxiliary_input`**(**const** Expression *&x*, **const** Expression *&aux*)

Add auxiliary output.

Returns $h_t = \tanh(W_x x_t + W_h h_{t-1} + W_y y + b)$ where y is an auxiliary output TODO : clarify

Return The hidden representation of the deepest layer

Parameters

- `x`: Input expression
- `aux`: Auxiliary output expression

CHAPTER 7

Optimizers

The various optimizers that you can use to tune your parameters

struct *#include <training.h>* Stochastic gradient descent trainer. This trainer performs stochastic gradient descent, the goto optimization procedure for neural networks. In the standard setting, the learning rate at epoch t is $\eta_t = \frac{\eta_0}{1+\eta_{\text{decay}}t}$ Reference : reference needed Inherits from *dynet::Trainer* **Public Functions**

`dynet::SimpleSGDTrainer::`**`SimpleSGDTrainer`**(Model &*m*, real *e0* = 0.1, real *edecay* = 0.0)

Constructor.

Parameters

- `m`: Model to be trained
- `e0`: Initial learning rate
- `edecay`: Learning rate decay parameter.

struct *#include <training.h>* Stochastic gradient descent with momentum. This is a modified version of the SGD algorithm with momentum to stablize the gradient trajectory. The modified gradient is $\theta_{t+1} = \mu\theta_t + \nabla_{t+1}$ where μ is the momentum. Reference : reference needed Inherits from *dynet::Trainer* **Public Functions**

`dynet::MomentumSGDTrainer::`**`MomentumSGDTrainer`**(Model &*m*, real *e0* = 0.01, real *mom* = 0.9, real *edecay* = 0.0)

Constructor.

Parameters

- `m`: Model to be trained
- `e0`: Initial learning rate
- `mom`: Momentum
- `edecay`: Learning rate decay parameter

struct *#include <training.h>* Adagrad optimizer. The adagrad algorithm assigns a different learning rate to each parameter according to the following formula : $\delta_\theta^{(t)} = -\frac{\eta_0}{\epsilon+\sum_{i=0}^{t-1}(\nabla_\theta^{(i)})^2}\nabla_\theta^{(t)}$ Reference : Duchi et al., 2011 Inherits from *dynet::Trainer* **Public Functions**

`dynet::AdagradTrainer::`**`AdagradTrainer`**(Model *&m*, real *e0* = 0.1, real *eps* = 1e-20, real *edecay* = 0.0)

Constructor.

Parameters

- `m`: Model to be trained
- `e0`: Initial learning rate
- `eps`: Bias parameter ϵ in the adagrad formula
- `edecay`: Learning rate decay parameter

struct *#include <training.h>* AdaDelta optimizer. The AdaDelta optimizer is a variant of Adagrad where $\frac{\eta_0}{\sqrt{\epsilon+\sum_{i=0}^{t-1}(\nabla_\theta^{(i)})^2}}$ is replaced by $\frac{\sqrt{\epsilon+\sum_{i=0}^{t-1}\rho^{t-i-1}(1-\rho)(\delta_\theta^{(i)})^2}}{\sqrt{\epsilon+\sum_{i=0}^{t-1}(\nabla_\theta^{(i)})^2}}$, hence eliminating the need for an initial learning rate. Reference : ADADELTA: An Adaptive Learning Rate Method Inherits from *dynet::Trainer* **Public Functions**

`dynet::AdadeltaTrainer::`**`AdadeltaTrainer`**(Model *&m*, real *eps* = 1e-6, real *rho* = 0.95, real *edecay* = 0.0)

Constructor.

Parameters

- `m`: Model to be trained
- `eps`: Bias parameter ϵ in the adagrad formula
- `rho`: Update parameter for the moving average of updates in the numerator
- `edecay`: Learning rate decay parameter

struct *#include <training.h>* RMSProp optimizer. The RMSProp optimizer is a variant of Adagrad where the squared sum of previous gradients is replaced with a moving average with parameter ρ. Reference : reference needed Inherits from *dynet::Trainer* **Public Functions**

`dynet::RmsPropTrainer::`**`RmsPropTrainer`**(Model *&m*, real *e0* = 0.1, real *eps* = 1e-20, real *rho* = 0.95, real *edecay* = 0.0)

Constructor.

Parameters

- `m`: Model to be trained
- `e0`: Initial learning rate
- `eps`: Bias parameter ϵ in the adagrad formula
- `rho`: Update parameter for the moving average (`rho = 0` is equivalent to using Adagrad)
- `edecay`: Learning rate decay parameter

struct *#include <training.h>* Adam optimizer. The Adam optimizer is similar to RMSProp but uses unbiased estimates of the first and second moments of the gradient Reference : Adam: A Method for Stochastic Optimization Inherits from *dynet::Trainer* **Public Functions**

`dynet::AdamTrainer::`**`AdamTrainer`**(Model *&m*, float *e0* = 0.001, float *beta_1* = 0.9, float *beta_2* = 0.999, float *eps* = 1e-8, real *edecay* = 0.0)

Constructor.

Parameters

- `m`: Model to be trained
- `e0`: Initial learning rate
- `beta_1`: Moving average parameter for the mean
- `beta_2`: Moving average parameter for the variance
- `eps`: Bias parameter ϵ
- `edecay`: Learning rate decay parameter

struct *#include <training.h>* General trainer struct. Subclassed by *dynet::AdadeltaTrainer*, *dynet::AdagradTrainer*, *dynet::AdamTrainer*, *dynet::MomentumSGDTrainer*, *dynet::RmsPropTrainer*, *dynet::SimpleSGDTrainer* **Public Functions**

`dynet::Trainer::`**`Trainer`**(Model *&m*, real *e0*, real *edecay* = 0.0)

General constructor for a *Trainer*.

Parameters

- `m`: Model to be trained
- `e0`: Initial learning rate
- `edecay`: Learning rate decay

Public Members

bool `dynet::Trainer::`**`sparse_updates_enabled`**

Whether to perform sparse updates.

DyNet trainers support two types of updates for lookup parameters, sparse and dense. Sparse updates are the default. They have the potential to be faster, as they only touch the parameters that have non-zero gradients. However, they may not always be faster (particulary on GPU with mini-batch training), and are not precisely numerically correct for some update rules such as MomentumTrainer and *AdamTrainer*. Thus, if you set this variable to false, the trainer will perform dense updates and be precisely correct, and maybe faster sometimes.

CHAPTER 8

Examples

Here are some simple models coded in the examples of Dynet. Feel free to use and modify them.

8.1 Language models

Language modelling is one of the cornerstones of natural language processing. Dynet allows great flexibility in the creation of neural language models. Here are some examples.

template <class Builder**>**

struct *#include <rnnlm-batch.h>* This structure wraps any RNN to train a language model with minibatching.

Recurrent neural network based language modelling maximizes the likelihood of a sentence $\mathbf{s} = (w_1, \ldots, w_n)$ by modelling it as :

$L(\mathbf{s}) = p(w_1, \ldots, w_n) = \prod_{i=1}^{n} p(w_i | w_1, \ldots, w_{i-1})$

Where $p(w_i | w_1, \ldots, w_{i-1})$ is given by the output of the RNN at step i

In the case of training with minibatching, the sentences must be of the same length in each minibatch. This requires some preprocessing (see `train_rnnlm-batch.cc` for example).

Reference : Mikolov et al., 2010

Template Parameters

- `Builder`: This can be any RNNBuilder

Public Functions

`RNNBatchLanguageModel::`**`RNNBatchLanguageModel`**(Model *&model*, unsigned *LAYERS*, unsigned *INPUT_DIM*, unsigned *HIDDEN_DIM*, unsigned *VOCAB_SIZE*)

Constructor for the batched RNN language model.

Parameters

- `model`: Model to hold all parameters for training
- `LAYERS`: Number of layers of the RNN
- `INPUT_DIM`: Embedding dimension for the words
- `HIDDEN_DIM`: Dimension of the hidden states
- `VOCAB_SIZE`: Size of the input vocabulary

Expression `RNNBatchLanguageModel::getNegLogProb`(**const** vector<vector<int>> &*sents*, unsigned *id*, unsigned *bsize*, unsigned &*tokens*, ComputationGraph &*cg*)

Computes the negative log probability on a batch.

Return Expression for $ $\sum_{s\in\text{batch}} \log(p(s))$

Parameters

- `sents`: Full training set
- `id`: Start index of the batch
- `bsize`: Batch size (`id` + `bsize` should be smaller than the size of the dataset)
- `tokens`: Number of tokens processed by the model (used for loos per token computation)
- `cg`: Computation graph

void `RNNBatchLanguageModel::RandomSample`(**const** dynet::Dict &*d*, int *max_len* = 150, float *temp* = 1.0)

Samples a string of words/characters from the model.

This can be used to debug and/or have fun. Try it on new datasets!

Parameters

- `d`: Dictionary to use (should be same as the one used for training)
- `max_len`: maximu number of tokens to generate
- `temp`: Temperature for sampling (the softmax computed is $\frac{e^{\frac{r_t^{(i)}}{T}}}{\sum_{j=1}^{|V|} e^{\frac{r_t^{(j)}}{T}}}$). Intuitively lower temperature -> less deviation from the distribution (= more "standard" samples)

8.2 Sequence to sequence models

Dynet is well suited for the variety of sequence to sequence models used in modern NLP. Here are some pre-coded structs implementing the most common one.

template <class Builder>

struct *#include <encdec.h>* This structure is a "vanilla" encoder decoder model.

This sequence to sequence network models the conditional probability $p(y_1,\ldots,y_m|x_1,\ldots,x_n) = \prod_{i=1}^{m} p(y_i|\mathbf{e},y_1,\ldots,y_{i-1})$ where $\mathbf{e} = ENC(x_1,\ldots,x_n)$ is an encoding of the input sequence produced by a recurrent neural network.

Typically **e** is the concatenated cell and output vector of a (multilayer) LSTM.

Sequence to sequence models were introduced in Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation .

Our implementation is more akin to the one from Sequence to sequence learning with neural networks .

Template Parameters

- `Builder`: This can theoretically be any RNNbuilder. It's only been tested with an LSTM as of now

Public Functions

`EncoderDecoder::EncoderDecoder()`
Default builder.

`EncoderDecoder::EncoderDecoder(`Model &*model*, unsigned *num_layers*, unsigned *input_dim*, unsigned *hidden_dim*, bool *bwd* = false`)`
Creates an *EncoderDecoder*.

Parameters

- `model`: Model holding the parameters
- `num_layers`: Number of layers (same in the ecoder and decoder)
- `input_dim`: Dimension of the word/char embeddings
- `hidden_dim`: Dimension of the hidden states
- `bwd`: Set to `true` to make the encoder bidirectional. This doubles the number of parameters in the encoder. This will also add parameters for an affine transformation from the bidirectional encodings (of size num_layers * 2 * hidden_dim) to encodings of size num_layers * hidden_dim compatible with the decoder

Expression `EncoderDecoder::encode(`**const** vector<vector<int>> &*isents*, unsigned *id*, unsigned *bsize*, unsigned &*chars*, ComputationGraph &*cg*`)`
Batched encoding.

Encodes a batch of sentences of the same size (don't forget to pad them)

Return Returns the expression for the negative (batched) encoding

Parameters

- `isents`: Whole dataset
- `id`: Index of the start of the batch
- `bsize`: Batch size
- `chars`: Number of tokens processed (used to compute loss per characters)
- `cg`: Computation graph

Expression `EncoderDecoder::encode(`**const** vector<int> &*insent*, ComputationGraph &*cg*`)`
Single sentence version of `encode`

Note : this just creates a trivial dataset and feed it to the batched version with batch_size 1. It's not very effective so don't use it for training.

Return Expression of the encoding

Parameters

- `insent`: Input sentence
- `cg`: Computation graph

Expression `EncoderDecoder::decode`(**const** Expression *i_nc*, **const** vector<vector<int>> &*osents*, int *id*, int *bsize*, ComputationGraph &*cg*)

Batched decoding.

[long description]

Return Expression for the negative log likelihood

Parameters

- `i_nc`: Encoding (should be batched)
- `osents`: Output sentences dataset
- `id`: Start index of the batch
- `bsize`: Batch size (should be consistent with the shape of `i_nc`)
- `cg`: Computation graph

Expression `EncoderDecoder::decode`(**const** Expression *i_nc*, **const** vector<int> &*osent*, ComputationGraph &*cg*)

Single sentence version of `decode`

For similar reasons as `encode`, this is not really efficient. USed the batched version directly for training

Return Expression for the negative log likelihood

Parameters

- `i_nc`: Encoding
- `osent`: Output sentence
- `cg`: Computation graph

vector<int> `EncoderDecoder::generate`(**const** vector<int> &*insent*, ComputationGraph &*cg*)

Generate a sentence from an input sentence.

Samples at each timestep ducring decoding. Possible variations are greedy decoding and beam search for better performance

Return Generated sentence (indices in the dictionary)

Parameters

- `insent`: Input sentence
- `cg`: Computation Graph

vector<int> `EncoderDecoder::generate`(Expression *i_nc*, unsigned *oslen*, ComputationGraph &*cg*)

Generate a sentence from an encoding.

You can use this directly to generate random sentences

Return Generated sentence (indices in the dictionary)

Parameters

- `i_nc`: Input encoding
- `oslen`: Maximum length of output
- `cg`: Computation graph

Mode advanced topics are below:

CHAPTER 9

Minibatching

How to perform minibatching to improve efficiency

CHAPTER 10

Multi-processing

How to perform processing on multiple threads

(TODO: create doc)

CHAPTER 11

Unorthodox Design

There are a couple design decisions about DyNet that are different from the way things are implemented in other libraries, or different from the way you might expect things to be implemented. The items below are a list of these unorthodox design decisions, which you should read to avoid being surprised. We also try to give some justification for these decisions (although we realize that this is not the only right way to do things).

11.1 Sparse Updates

By default, DyNet parameter optimizers perform sparse updates over `LookupParameters`. This means that if you have a `LookupParameters` object, use a certain subset of indices, then perform a parameter update, the optimizer will loop over the used subset, and not perform any updates over the unused values. This can improve efficiency in some cases: e.g. if you have embeddings for a vocabulary of 100,000 words and you only use 5 of them in a particular update, this will avoid doing updates over all 100,000. However, there are two things to be careful of. First, this means that some update rules such as ones using momentum such as `MomentumSGDTrainer` and `AdamTrainer` are not strictly correct (these could be made correct with some effort, but this would complicate the programming interface, which we have opted against). Also, on GPUs, because large operations are relatively cheap, it can sometimes be faster to just perform a single operation over all of the parameters, as opposed to multiple small operations. In this case, you can set the `sparse_updates_enabled` variable of your `Trainer` to `false`, and DyNet will perform a standard dense update, which is guaranteed to be exactly correct, and potentially faster on GPU.

11.2 Weight Decay

As described in the *Command Line Options*, weight decay is implemented through the option `--dynet-weight-decay`. If this value is set to `wd`, each parameter in the model is multiplied by `(1-wd)` after every parameter update. This weight decay is similar to L2 regularization, and is equivalent in the case of using simple SGD (`SimpleSGDTrainer`), but it is *not the same* when using any other optimizers such as `AdagradTrainer` or `AdamTrainer`. You can still try to use weight decay with these optimizers, and it might work, but if you really want to correctly apply L2 regularization with these optimizers, you will have to directly calculate the L2 norm of each of the parameters and add it to the objective function before performing your update.

11.3 Minibatching Implementation

Minibatching in DyNet is different than how it is implemented in other libraries. In other libraries, you can create minibatches by explicitly adding another dimension to each of the variables that you want to process, and managing them yourself. Instead, DyNet provides special *Operations* that allow you to perform input, lookup, or loss calculation over mini-batched input, then DyNet will handle the rest. The programming paradigm is a bit different from other toolkits, and may take a bit of getting used to, but is often more convenient once you're used to it.

11.4 Dropout Scaling

When using dropout to help prevent overfitting, dropout is generally applied at training time, then at test time all the nodes in the neural net are used to make the final decision, increasing robustness. However, because there is a disconnect between the number of nodes being used in each situation, it is important to scale the values of the output to ensure that they match in both situations. There are two ways to do this:

- **Vanilla Dropout:** At training time, perform dropout with probability `p`. At test time, scale the outputs of each node by `p`.
- **Inverted Dropout:** At training time, perform dropout with probability `p`, *and* scale the outputs by `1/p`. At test time, use the outputs as-is.

The first is perhaps more common, but the second is convenient, because we only need to think about dropout at training time, and thus DyNet opts to use the latter. See here for more details on these two methods.

And we welcome your contributions!

CHAPTER 12

Contibuting to Dynet

You are very welcome to contribute to Dynet, be it to correct a bug or add a feature. Here are some guidelines to guarantee consistency.

12.1 Coding Style

DyNet (the main version in C++) has certain coding style standards:

Overall Philosophy: DyNet is designed to minimize the computational overhead when creating networks. Try to avoid doing slow things like creating objects or copying memory in places that will be called frequently during computation graph construction.

Function Names: Function names are written in "snake_case".

const: Always use const if the input to a function is constant.

Pointer vs. Reference: When writing functions, use the following guidelines (quoted from here):

- Only pass a value by pointer if the value 0/NULL is a valid input in the current context.
- If a function argument is an out-value, then pass it by reference.
- Choose "pass by value" over "pass by const reference" only if the value is a POD (Plain Old Datastructure) or small enough (memory-wise) or in other ways cheap enough (time-wise) to copy.

Throwing Exceptions: When the user does something illegal, throw an exception. "assert" should never be used for something that might be triggered by a user. (As noted, there are still many places that don't follow this standard as of Dec. 13, 2016.)

12.2 Documentation

Dynet uses Doxygen for commenting the code and Sphinx for the general documentation.

If you're only documenting features you don't need to concern yourself with Sphinx, your doxygen comments will be integrated in the documentation automatically.

12.2.1 Doxygen guidelines

Please document any publicly accessible function you write using the doxygen syntax. You can see examples in the training file. The most important thing is to use `/*` style comments and `\command` style commands.

For ease of access the documentation is divided into *groups*. For now the groups are optimizers and operations. If you implement a function that falls into one of these groups, add `\ingroup [group name]` at the beginning of your comment block.

If you want to create a group, use `\defgroup [group-name]` at the beginning of your file. Then create a file for this group in sphinx (see next section).

Important : You can use latex in doxygen comments with the syntax `\f$ \f$`. For some reason since readthedocs updated their version of sphinx `\f[ \f]` doesn't work anymore so *don't use it* it breaks the build.

12.2.2 Sphinx guidelines

The sphinx source files are located in `doc/source`. They describe the documentation's organization using the reStructuredText Markup language.

Although reStructuredText is more powerful than Markdown it might feel less intuitive, especially when writing long documents. If needs be you can write your doc in Markdown and convert it using Pandoc.

For a tutorial on Sphinx see their tutorial.

Doxygen generated XML is integrated in sphinx files using the Breathe module. The only breathe command used now is `doxygengroup`. You shouldn't used commands for individual classes/functions/structs without a good reason. Most information should be put in the doxygen comments.

12.2.3 Building the docs

The documentation is automatically rebuilt by ReadTheDocs each time you push on Github.

If you want to build the documentation locally you'll need to install doxygen, sphinx and breathe and then run `build_doc.sh` from the `doc` folder.

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Index

D

E

R