CONTEXT: Convolutional Neural Network Code for Text Categorization

September 23, 2015

Revision history

- September, 2015: The following changes were made when v2.00 (corresponding to [2]) was released. Section 3 was added for semi-supervised learning, and the introduction was modified accordingly. Section 2.9 was added for new function `write_features`. Description of a *word-mapping file* was added to Section 1. NOTE3 on endian sensitivity was added to the end of the introduction.
- August 12, 2015: Sections 4.1, 4.2, and 4.3 were added to provide hints and tips on consumption of GPU memory and CPU memory during training and when to make data batches. Useful if you have many training/test data points or many target classes.

Introduction

This document describes how to use CONTEXT, the code for convolutional neural network (CNN) for text categorization available at `http://riejohnson.com/cnn_download.html`. The original purpose of releasing this code was to enable the reproduction of the experiments presented in "Effective use of word order for text categorization with convolutional neural networks" [1], though it also provides functions that were not used in [1]. CONTEXT v2.00 provides semi-supervised learning functions to enable the experiments in "Semi-supervised convolutional neural networks for text categorization via region embedding" [2]. Therefore, readers are assumed to be familiar with the concepts/terminology used in [1], and [2] for the semi-supervised section.

Unlike the CNN code designed for image data, CONTEXT efficiently handles *high-dimensional* and *sparse* documents of *variable sizes*. It applies CNN directly to one-hot vectors representing words (as opposed to requiring low-dimensional word vectors learned elsewhere), which leads to internally learning an *embedding* of *text regions* (or *context* as in the name) directly from one-hot vectors. The benefits of this approach compared with traditional supervised methods are discussed in [1]. The semi-supervised learning functionality is also based on region embedding learning. The merits of this approach compared with previous semi-supervised methods are discussed in [2].

In a basic supervised learning setting, training and testing a CNN using CONTEXT is done in the following two steps.

1. **Data preparation**: Generate data files used for CNN training and testing as input. In particular, to speed up training, we generate region vectors (vectors that represent small regions of documents) used in the convolution layer for training and testing, and write to files in advance, instead of making them on the fly during training/testing.
2. **CNN training and testing**: Train a CNN and optionally test it on development/test data as training proceeds, using the files generated above as input. Also, the trained CNN can be saved to a file and applied to other data.

The code interface for supervised learning is described in Sections 1 and 2, additional functions and parameters for semi-supervised learning are introduced in Section 3, and Section 4 discusses things to consider for obtaining good performance.

NOTE1 To use CONTEXT, your system must be equipped with a CUDA-capable GPU such as Tesla K20, and CUDA needs to be installed. `README` should be referred to for more details of hardware/software requirements and how to build executables.

NOTE2 The following sections refer to sample scripts located in the `sample` directory. The sample scripts are only intended for describing the interface using small data, and their settings are not optimized for performance. The scripts in the `test` directory are for reproducing the experiments in [1] and [2]. These scripts should be referred to for realistic examples on the real-world data.

NOTE3 Binary files produced by CONTEXT are endian sensitive and cannot be shared among the systems with different endianness.

Contents

1 Data preparation for CNN training: `prepText`

An executable `prepText` prepares input data for training and testing CNNs. It runs on CPU and does not use GPU.

Usage: `prepText` *action param1 param2* $\cdots$

The first argument *action* must be one of the following:

- `gen_vocab`: Generate a vocabulary file used as input to `gen_regions` and other functions of `prepText`.
- `gen_regions`: Generate a region vector file, a target file, and a word-mapping file used as input to CNN training.
- `show_regions`: Show the content of a region vector file.
- `gen_nbw`: Generate a NB weight file, used as input to `gen_nbwfeat`.
- `gen_nbwfeat`: Generate a NB-weighted feature file, used in the seq2-bow$n$-CNN experiments in [1].
- `gen_b_feat`: Generate a bag-of-n-gram file, used in the baseline experiments with SVM and fully-connected neural networks in [1].

The rest of the arguments depend on the *action* and they are described below.

1.1 `gen_vocab` (vocabulary file generation)

As the first step of data preparation, we generate a vocabulary file from training data. That is, we extract words (or n-grams) from text data.

Tokenized text file (input) Text files given to `gen_vocab` as input should contain documents, one per line, and each document should be already tokenized so that tokens are delimited by space. See `sample/data/s-test.tok` for an example.

Vocabulary file (output) The vocabulary file generated by `gen_vocab` contains words (or n-grams), one per line. Each word is optionally followed by its frequency; in that case, a tab character is used as a delimiter between the word and the frequency.

To reduce vocabulary size Typically, it is not necessary to use all the words in the training data to obtain good text categorization performance, while retaining many words is resource consuming. `gen_vocab` provides several ways to reduce the size of vocabulary such as only retaining the most frequent words up to `max_vocab_size`, only retaining the words occurring no fewer than `min_word_count`, removing stopwords listed in `stopword_fn`, and removing the words that include numerical characters (`RemoveNumbers`).

In the tables below, required parameters are marked with $*$; all other parameters are optional.

Parameters for `prepText gen_vocab` (vocabulary file generation).		
*	`input_fn=`	Path to the input token file.
*	`vocab_fn=`	Path to the file that the vocabulary is written to.
	`stopword_fn=`	Path to a stopword file. The words in this file will be excluded.
	`min_word_count=`	Minimum word counts to be included in the vocabulary file. (Default:No limit)
	`max_vocab_size=`	Maximum number of words to be included in the vocabulary file. The most frequent ones will be included. (Default:No limit)
	`LowerCase`	Convert upper-case to lower-case characters.
	`UTF8`	Convert UTF8 en dash, em dash, single/double quotes to ascii characters.
	`RemoveNumbers`	Exclude words that contain numbers.
	`WriteCount`	Write word counts as well as the words to the vocabulary file.
	`n=`	n for n-grams. E.g., if n=3, only tri-grams are included.

Example 1 (`sample/sample.sh`). *Extract words (delimited by space) from* `data/s-train.tok` *and convert them to lower cases while only retaining the most frequent 10000 words:*

```
prepText gen_vocab   input_fn=data/s-train.tok vocab_fn=data/s.vocab \
                     LowerCase max_vocab_size=10000
```

1.2 `gen_regions` (region file and target file generation)

Next, we generate a region file and target file, which serve as input files for CNN training and testing.

Tokenized text file (input) The input text file format is the same as `gen_vocab` above. The file should contain one document per line, and each document should be already tokenized so that tokens are delimited by space.

Label file (input) Each line of a label file should contain classification labels for each document, and the order of the documents must be the same as the text file. In case of multi-label classification (i.e., more than one label can be assigned to each document), the labels should be delimited by a vertical line |. The labels (any string) must be declared in a label dictionary described below.

Text/label file naming conventions The text file and the corresponding label file must have the same pathname stem with different file extensions, e.g., text file `data/s-train.tok` and label file `data/s-train.cat`.

Label dictionary file (input) The labels used in the label file above must be declared in a label dictionary file. The label dictionary file should contain one label per line. See `sample/data/s-cat.dic` for example.

Vocabulary file (input) The vocabulary file should be generated by `gen_vocab` above. To use a vocabulary file generated by some other means, note that a tab character (0x09) is regarded as a delimiter, i.e., in each line, a tab character and anything that follows are ignored. Also note that the case option must be consistent; e.g., if `gen_region` specifies `LowerCase` (which converts upper-case letters to lower-case), then the contents of the vocabulary file must also be all lower-case.

Region file (output) `gen_regions` generates a region file that contains region vectors with the specified region size, stride, and padding, in binary format. To generate the region file, a file extension `.xsmatvar` (which indicates that the file is in a sparse and variable-sized format) is automatically attached to the pathname stem specified by `region_fn_stem=`.

Target file (output) `gen_regions` generates a target file that contains classification targets in the format that the executable for CNN training can read. To generate the target file, a file extension either `.y` or `.ysmat` (which indicates the file format) is automatically attached to the pathname stem specified by `region_fn_stem=`.

Word-mapping file (output) `gen_regions` generates a *word-mapping file* that contains a mapping between words (or n-grams) and dimensions of the region vectors. To generate the word-mapping file, a file extension `.xtext` is automatically attached to the pathname stem specified by `region_fn_stem=`. This file is used by v2.00 (or higher) to ensure the consistency of the mapping at the time of training and application of the models.

Multi-label/no-label documents By default, single-label classification is assumed, and therefore, each document is expected to have exactly one label; the process terminates with an error on documents with more than one label or no label. To allow multi-label and no-label documents, specify `MultiLabel`.

Parameters for `prepText gen_regions` (region file generation).		
*	`input_fn=`	Input filename without extension.
*	`text_fn_ext=`	Filename extension of the tokenized text file.
*	`label_fn_ext=`	Filename extension of the label file. Required for target file generation.
*	`label_dic_fn=`	Path to the label dictionary file (input). The file should list the labels used in the label files, one per each line. Required for target file generation.
*	`vocab_fn=`	Path to the vocabulary file generated by `gen_vocab` (input).
*	`region_fn_stem=`	Pathname stem of the region vector file, target file, and word-mapping file (output). To make the pathnames of these files, the respective extensions will be attached. The name must not contain a plus sign `+`.
	`y_ext=`	Filename extension of the target file (output). `.y` | `.ysmat`. Use `.ysmat` (binary sparse format) if the number of classes is large. (Default: `.y` (text format))
*	`patch_size=`	Region size.
	`patch_stride=`	Region stride. (Default:1)
	`padding=`	Padding size. Region size minus one is recommended. (Default:0)
	`Bow`	Generate region vectors for a bow-convolutional layer. See Section 2.2.2 of [1] for bow-convolution. Shortened from `Bow-convolution`, which still works.
	`VariableStride`	Take variable strides; see Section 3.1 of [1].
	`LowerCase`	Convert upper-case to lower-case characters. On/off of this switch must be consistent with `gen_vocab`.
	`UTF8`	Convert UTF8 en dash, em dash, single/double quotes to ascii characters. On/off of this switch must be consistent with `gen_vocab`.
	`MultiLabel`	Allow multiple labels per data point for multi-label classification. Data points that do not have any label are also allowed.
	`AllowZeroRegion`	Do not ignore empty regions.
	`RegionOnly`	Generate a region file only. Do not generate a target file.
	`batch_id=`	Batch ID, e.g., `1of5` (the first batch out of 5), `2of5` (the second batch out of 5). Specify this when making multiple sets of files (*batches*) for one large dataset. See Section 2.3 for how to use this parameter.

Example 2 (`sample/sample.sh`). *Generate a region vector file* `data/s-train-p3.xsmatvar`*, a target file* `data/s-train-p3.y`*, and a word-mapping file* `data/s-train-p3.xtext` *from a tokenized text file* `data/s-train.tok` *and a label file* `data/s-train.cat`*, with region size 3, stride 1, and padding 2.*

```
prepText gen_regions   input_fn=data/s-train text_fn_ext=.tok label_fn_ext=.cat \
                       vocab_fn=s.vocab LowerCase \
                       label_dic_fn=data/s-cat.dic \
                       patch_size=3 patch_stride=1 padding=2 \
                       region_fn_stem=data/s-train-p3
```

1.3 Other functions of `prepText`

The parameters for the other functions are similar to those described above. To display help, enter

```
prepText   show_regions   or
prepText   gen_nbw        or
prepText   gen_nbwfeat    or
prepText   gen_b_feat
```

For example usage, see the scripts `test/train_imdb_seq2_bown.sh` for `gen_nbw` and `gen_nbwfeat`, and `sample/sample-1nn.sh` for `gen_b_feat`.

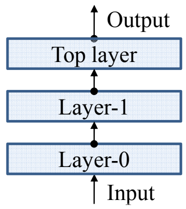

Figure 1: Single-connection example.

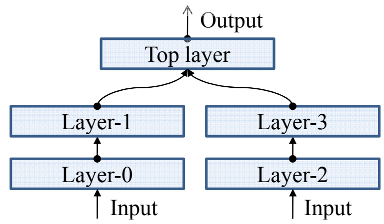

Figure 2: Multi-connection example.

2 CNN training and testing: `conText`

`conText` is an executable for training and testing CNNs using GPU.

Usage: `conText` *gpu# action param1 param2* $\cdots$

gpu# The first argument specifies the GPU ID number. Specify `-1` if the default GPU should be used,

action The second argument *action* must be one of the following:

- `cnn`: Train a CNN. Optionally, test the CNN on development/test data as training proceeds. This function can also be used to train a fully-connected neural network or a linear model.
- `cnn_predict`: Apply the trained model to new data.
- `write_features`: Write internal features to a file.

param1*, *param2, $\cdots$ Parameters can be specified via either command line arguments or a parameter file or a combination of both. If an argument starts with an `@`, the string following `@` is regarded as a pathname to a parameter file. See `sample/param/sample.param` for an example of the parameter file. Use of a parameter file is sometimes convenient as parameters can be organized and commented.

Example 3 (`sample/sample-paramfile.sh`)**.** *Use parameters in* `param/sample.param` *and set* `random_seed=1`.

```
conText -1 cnn   random_seed=1 @param/sample.param
```

To describe the interface, the following term/concepts will be used.

- **Hidden layer**: In this code, each *hidden layer* performs a sequence of operations: weighting and activation (computation of $\boldsymbol{\sigma}(\mathbf{Wx}+\mathbf{b})$), pooling (optional), and response normalization (optional), in that order. (Note that an alternative is to treat each function as an individual layer, e.g., a pooling layer and a response normalization layer, and so on.) We assign unique id numbers to hidden layers starting from zero, e.g., layer-0, layer-1, $\cdots$.
- **Top layer**: A top layer is a special layer that only performs weighting ($\mathbf{Wx} + b$) to generate output.
- **Single-connection architecture**: With a *single-connection* architecture, data flows from layer-0 to layer-1, from layer-1 to layer2, and so on and from the last hidden layer to the top layer, as illustrated in Figure 1.
- **Multi-connection architecture**: A *multi-connection* architecture allows each layer to be connected to more than one layer; e.g., see Figure 2.

First, Section 2.1 describes the interface for training a CNN with single-connection architecture. CNN with a multi-connection architecture and other types only requires a few additional parameters, and they are described in Sections 2.2–2.7.

2.1 `cnn` (training and testing): single-connection CNN

There are two types of parameters, *layer parameters* which specifies parameters for each layer, and *network parameters* which are not specific to layers.

2.1.1 Network parameters

The parameters below are for controlling the overall (as opposed to layer-specific) aspect of training.

To control input		
*	`trnname=`	Training data name. Region/target filename without extension.
*	`tstname=`	Test/development data name. Region/target filename without extension. It must not contain a plus sign. Required for testing unless `NoTest` is specified.
*	`datatype=sparse`	Dataset type. Sparse input. Other options are described later.
	`data_dir=`	Directory where data files are.
*	`x_ext=.xsmatvar`	Region filename extension. Other types are described later.
*	`y_ext=`	Target filename extension. `.y` (text format) | `.ysmat.` (sparse format)
	`NoTest`	Do no testing. Training only.
To control output		
	`save_fn=`	Path to the file the model is saved to for warm-start later.
	`save_interval=`	s. The model is saved to a file every s epochs. (Default: no save)
	`test_interval=`	t. Evaluate performance on test/development data every t epochs. (Default: 1)
	`evaluation_fn=`	Write performance results to this file in the csv format.
	`ExactTrainingLoss`	Compute and display training loss. If this switch is off, estimated loss is shown instead.
	`LessVerbose`	Do not display too much information.
	`inc=`	d. Show progress after going through d data points. Useful when data is large or when training is time-consuming.
To define network architecture.		
*	`layers=`	Number of hidden layers.
To control training.		
*	`num_iterations=`	Number of epochs.
*	`loss=`	Loss function. `Square` | `Log` | `BinLogi2` `Square`: $(p-y)^2/2$ where p is prediction and y is target. `Log` (softmax and log loss) `BinLogi2` (binary logistic loss for $y \in \{0,1\}$): $\log(1+\exp(-(2y-1)p))$
	`mini_batch_size=`	Mini-batch size for training. (Default:100)
	`step_size_scheduler=`	Step-size scheduler type. `Few` (reduce it a few times) is recommended. (Default: no scheduling)
	`step_size_decay=`	Use this with `step_size_scheduler=Few`. Step-size is reduced by multiplying this value when it is reduced.
	`step_size_decay_at=`	Use this with `step_size_scheduler=Few`. Step-size is reduced after this many epochs. To reduce it more than once, use an underbar (_) as a delimiter, e.g., `80_90` reduces it after the 80-th and 90-th epochs.
Other parameters.		
	`random_seed=`	Seed for random number generation.
	`Regression`	Specify this if the task is regression. (Default: classification)
	`MultiLabel`	Specify this if each data point can be assigned multiple labels. (Default: single-label classification).
	`test_mini_batch_size=`	Mini-batch size for parallel processing for testing. (Default: 100).

Example 4 (`sample/sample.sh`). *Perform 20 epochs of SGD with mini-batch size 100 while reducing the step-size by multiplying 0.1 after 15 epochs.*

```
···  num_iterations=20 mini_batch_size=100 \
     step_size_scheduler=Few step_size_decay_at=15 step_size_decay=0.1
```

2.1.2 Layer parameters

The parameters below can be specified for a specific layer by attaching the layer id ℓ (ℓ =`0`, `1`, $\cdots$, `top_`) in front of the keywords, e.g., `0step_size=0.1` for layer-0 or `top_reg_L2=1e-4` for the top layer. Without a layer id, the parameter is regarded as a default parameter and applied to all the layers; i.e., the parameter with a layer id supersedes the one without the id. For example,

```
layers=3 reg_L2=0.0001 0reg_L2=0 top_reg_L2=0.01
```

is equivalent to

```
layers=3 0reg_L2=0 1reg_L2=0.0001 2reg_L2=0.0001 top_reg_L2=0.01
```

Number of neurons and activation type.		
*	[ℓ]`nodes=`	Number of weight vectors (or neurons).
	[ℓ]`activ_type=`	Activation type. `None` | `Log` | `Rect` | `Softplus` | `Tanh` . (Default:`None`) `None`: $\sigma(x) = x$ `Log` (sigmoid): $\sigma(x) = 1/(1 + \exp(-x))$ `Rect` (rectifier): $\sigma(x) = \max(x, 0)$ `Softplus`: $\sigma(x) = \log(1 + \exp(x))$ `Tanh`: $\sigma(x) = (\exp(2x) - 1)/(\exp(2x) + 1)$
Pooling.		
	[ℓ]`pooling_type=`	Pooling type. `Max` | `Avg` | `L2` | `None`. (Default:`None` (no pooling)). To perform pooling, either `num_pooling` or `pooling_size` is required, but not both.
	[ℓ]`num_pooling=`	Number of pooling regions. Fixed-sized output is produced; see Section 2.2.3 of [1]. For example, `num_pooling=5` makes five pooling regions of equal size without overlapping, which leads to output of size 5 (5 'pixels').
	[ℓ]`pooling_size=`	Pooling size, as in CNN for images. Variable-sized output is produced if input is variable-sized. Use this with care since a fully-connected layer such as the top layer requires fixed-sized input. Example: `sample/sample-cc.sh`
	[ℓ]`pooling_stride=`	Pooling stride. Required if `pooling_size` is specified.
To control training.		
	[ℓ]`dropout=`	Dropout rate [4]. (Default:No dropout). Dropout is applied to the input to the layer. Dropout cannot be applied to sparse input.
	[ℓ]`reg_L2=`	L2 regularization parameter. Useful value: 1e-4, 1e-3, maybe 1e-5
	[ℓ]`reg_L2const=`	c: max-norm regularization parameter. Linearly scale to c when the 2-norm of a weight vector exceeds c. `reg_L2` and `reg_L2const` are mutually exclusive.
	[ℓ]`init_weight=`	x: scale of weight initialization. Weights will be initialized by Gaussian distribution with zero mean with standard deviation x. If `InitWeightUniform` is on, initial weights will be in $[-x, x]$. (Default:0.01)
	[ℓ]`InitWeightUniform`	Initialize weights with uniform distribution. (Default:Gaussian distribution)
	[ℓ]`init_intercept=`	Initial values of intercepts. (Default:0)
	[ℓ]`NoIntercept`	No intercept.
	[ℓ]`RegularizeIntercept`	Regularize intercepts. (Default: intercepts are unregularized)

SGD parameters.		
*	[ℓ]step_size=	Initial step-size (learning rate) for SGD.
	[ℓ]momentum=	Momentum for SGD. Useful value: 0.9.

Example 5 (sample/sample.sh). *In the following, layer-0 is with 500 neurons, rectifier activation, and max-pooling with one pooling unit; in all layers, the L2 regularization parameter is 1e-4 and the initial step-size is 0.25; and the top layer performs dropout with rate 0.5.*

```
··· 0nodes=500 0pooling_type=Max 0num_pooling=1 0activ_type=Rect \
    reg_L2=1e-4 step_size=0.25 top_dropout=0.5
```

Response normalization Optionally, response normalization can be applied, similar to [3]. The pooling result indexed by j is multiplied with $(\gamma + \alpha \sum_i \mathbf{v}_i^2)^{-\beta}$ where i moves within the window of size w surrounding j. If resnorm_type=Cross (cross-neuron), the sum is taken over neurons in a window defined on a ring (where the first neuron is considered to be next to the last neuron), and in particular, if width w is no smaller than the number of neurons, the sum is taken over all the neurons.

[ℓ]resnorm_type=	None | Cross. (Default:None (do nothing))
[ℓ]resnorm_width=	w: Width w above.
[ℓ]resnorm_one=	γ in the formula above. (Default:1)
[ℓ]resnorm_alpha=	α in the formula above.
[ℓ]resnorm_beta=	β in the formula above.

Region size, stride, and padding with *dense input* When datatype=sparse or sparse_multi, the input files are required to be in the special format that makes use of sparseness (i.e., most vector components are zero), called *sparse input*. With datatype=sparse or sparse_multi, the first layer takes *sparse input*. Other layers all take *dense input*. With *sparse input*, region size/stride and padding are determined at the time of region file generation using prepText gen_regions, not by layer parameters. Region size/stride and padding in the layer with *dense input* are specified via layer parameters as follows.

[ℓ]patch_size=	Region size in the convolution layer with *dense input*.
[ℓ]patch_stride=	Region stride in the convolution layer with *dense input*.
[ℓ]padding=	Padding size at the edge in the convolution layer with *dense input*.

If input is fixed-sized, and if patch_size is omitted, the layer is considered to be fully-connected. An example of a fully-connected network will be discussed in Section 2.7.3

2.1.3 Output

Evaluation results (single-label classification) Optionally, classification evaluation results can be written in the CSV format to a file specified via evaluation_fn. Lines of this file look as follows:

```
ite,5,0.226053, test-loss,0.217621, perf:err,0.34
ite,10,0.119096, test-loss,0.164462, perf:err,0.22
```

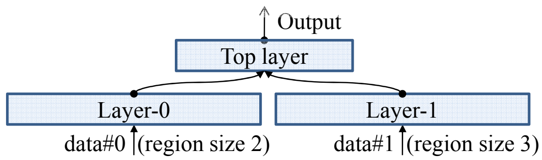

Figure 3: Multi-connection architecture defined in `sample/sample-multiconn.sh`.

The numbers in each line above are the number of epochs, training loss approximation, test loss, and error rate (1 - accuracy). The training loss approximation is the average of data point loss, which is obtained on each data point as a side product of forward propagation during the designated epoch. It is approximate since the network state keeps changing during the epoch. The regularization term is not included. To obtain the real training loss at the expense of slightly longer processing, specify `ExactTrainingLoss`, and it will be computed and shown in the evaluation file, for example, as follows.

```
ite,5,0.226053, trn-loss,0.20709,0.0725723, test-loss,0.217621, perf:err,0.34
```

There are two numbers following `trn-loss`. The first number is the training loss (averaged over data points) excluding the regularization term. The second number is the L2 regularization term.

Evaluation results (multi-label classification) The description above was for single-label classification cases. If the task is multi-label classification, indicated by `MultiLabel`, error rate is omitted and only test loss is shown.

stdout While evaluation is done in the interval specified by `test_interval`, the loss approximation is written to stdout every epoch. If, for example, `inc=100`, the loss approximation is written to stdout also every time training goes through 100 more data points, which is convenient with large data. Also, the network and layer setting is written to stdout at the beginning of training.

Model files When `save_interval` and `save_fn` are specified, models are written to files in binary format. The model files can be used for warm-start of training and making predictions on new data.

2.2 `cnn`: a multi-connection CNN

Training of a multi-connection CNN requires additional parameters to specify the connection between layers and between input data and layers. We use the parameter setting in the script `sample/sample-multiconn.sh` as a running example. The network architecture of this CNN is shown in Figure 3. The multi-connection parameters in this script are as follows:

```
extension=multi  layers=2  conn0=0-top  conn1=1-top
datatype=sparse_multi  data_ext0=p2  data_ext1=p3
0dataset_no=0  1dataset_no=1
```

and they are explained below.

To enable multi-connection architecture `extension=multi` must be specified.

To define connection The connections between layers in Figure 3 are defined by:

```
layers=2  conn0=0-top  conn1=1-top
```

`layers=2` indicates that there are two hidden layers (as in the single-connection parameter). The numbers on the right to the equal sign are layer id's, and `top` indicates the top layer. `conn0=0-top` indicates that the data flows from layer-0 to the top layer, and `conn1=1-top` indicates that the data flows from layer-1 to the top layer. The numbers following `conn` must be sequential starting from 0.

Example 6. *The order of the* `conn` *parameters does not matter. The connection in Figure 2 can be specified by:*

`layers=4 conn0=0-1 conn1=2-3 conn2=1-top conn3=3-top` *or*
`layers=4 conn0=0-1 conn1=1-top conn2=2-3 conn3=3-top`

Although both specifies the same network connection, the results may differ since the order of random initialization of weights may be different.

To use multiple datasets A multi-connection network may have more than one bottom layer. Since regions of the bottom layers are defined at the time of region file generation (`prepText gen_regions`), in order to allow multiple bottom layers to have different region definitions from each other, we need to be able to specify multiple region files as input to training. This is done by first enabling multi-dataset option by:

`datatype=sparse_multi`

and then specifying the filename stems of the datasets, in this example, as follows:

`data_dir=data trnname=s-train- tstname=s-test- data_ext0=p2 data_ext1=p3`

This indicates that the pathname stems of the two datasets are as follows.

	for training	for testing
dataset-0	`data/s-train-p2`	`data/s-test-p2`
dataset-1	`data/s-train-p3`	`data/s-test-p3`

Note that actual pathnames to region/target/word-mapping files are obtained by further concatenating the file extensions such as `.xsmatvar` and `.y`. Finally, we specify which layer takes which data as input in the format of ℓ`dataset_no=`d, which indicates layer-ℓ takes dataset-d as input. For example,

`0dataset_no=0 1dataset_no=1`

specifies that layer-0 takes dataset-0 and layer-1 takes dataset-1 as input.

2.3 `cnn`: data batches for large training data

When training data is large, it is convenient to divide it to several *batches* (see Section 4.3 for when to use batches). This can be done by doing the following:

1. Divide a tokenized text file and a label file for training data into k batches (e.g., $k = 5$).
2. After generating a vocabulary file using `prepText gen_vocab` as before, invoke `prepText gen_regions` k times to generate a region and a target file for each of k batches, using `batch_id` to indicate the position in the batches. `prepText gen_regions` uses this parameter to name the region/target files appropriately.
3. Train a CNN with `num_batches=`k to indicate that the training data consists of k batches. Note that in the case of multiple batches, going through one batch is counted as *one epoch*; therefore, for example, `num_batches=5 num_iterations=100` results in going through the entire data (all five batches) 20 times instead of 100, and with `step_size_decay_at=4`, the step size is reduced after going through about four fifth of the data.

Example 7 (`sample/sample-batch.sh`). *Example of two batches.*

```
prepText gen_regions   input_fn=s-train1 batch_id=1of2 \
                       region_fn_stem=sb_train ···
prepText gen_regions   input_fn=s-train2 batch_id=2of2 \
                       region_fn_stem=sb_train ···
conText gpu# cnn num_batches=2 trnname=sb-train ···
```

2.4 `cnn`: warm-start from a saved CNN

Training can be resumed by warm-starting from a CNN saved to a file. To save a CNN to a file, `save_fn` and `save_interval` should be specified as described 2.1. With `fn_for_warmstart` specified, training warm-starts from the model saved in the designated file.

To warm-start from a saved model		
*	`fn_for_warmstart=`	Path to the model file (input), saved during training using `cnn` via `save_fn=`.

Note that the network configuration such as the number of neurons and pooling setting are saved in the model file, and they cannot be changed when training warm-starts. Training parameters such as the regularization parameter and step-sizes need to be specified when warm-starting, and they can be different from before. See `sample/sample-warmstart.sh` to see which parameters should be omitted and which parameter should not at the time of warm-start. Also note that suspending training and resuming training by warm-start would result in a slightly different model from the model trained without interruption. This is mainly because random number generation for mini batch selection is reset at the time of warm-starting.

Example 8 (`sample/sample-warmstart.sh`). *Train a CNN and save the model after 20 epochs, and warm-start training from the saved model.*

```
conText -1 cnn   save_interval=20 save_fn=output/s-train-p3.mod ···
conText -1 cnn   fn_for_warmstart=output/s-train-p3.mod.ite20 ···
```

Note that the model filename is a concatenation of `save_fn`*,* `.ite`*, and the number of epochs (20 in this example).*

2.5 `cnn`: AdaDelta

The default training algorithm is SGD as described above. Optionally, AdaDelta [5] can also be used.

AdaDelta		
*	`AdaDelta`	Enable AdaDelta training
*	[ℓ]`rho=`	ρ in Algorithm 1 of [5]. Useful value: 0.95
*	[ℓ]`epsilon=`	ϵ in Algorithm 1 of [5].

When AdaDelta is enabled, the SGD parameters (`step_size` and `momentum`) are ignored. However, the step-size scheduling is in effect, e.g., if `step_size_scheduler=Few`, `step_size_decay=0.1` and `step_size_decay_at=80`, then the amount of weight update by AdaDelta is reduced to one tenth after 80 epochs. It appears that this technique also stabilizes weight updating by AdaDelta as well as SGD and sometimes improves performance. See `sample/sample-adad.sh` for example.

2.6 `cnn`: layer-by-layer training of single-connection networks

`focus_layers` specifies which layers should be updated during the training. This parameter is valid only on single-connection networks.

`focus_layers=`	Layers whose weights should be updated. (Default:All) For example, `focus_layers=0-1_0-1-2` indicates that layer-0 and layer-1 should be updated for the number of epochs specified by `num_iterations` and designated step-size scheduling first, and then layer-0, 1, and 2 should be updated for `num_iterations` epochs with the designated step-sized scheduling. This parameter is valid only on single-connection networks.

2.7 `cnn`: examples

[1] only experimented with a few types of networks, and there are other network architectures that can be explored using this code. Sample scripts are provided for some of them.

2.7.1 CNN with a convolution layer above a convolution layer

`sample/sample-cc.sh` trains a CNN with a convolution layer above a convolution layer.

2.7.2 CNN with a fully-connected layer above a convolution layer

`sample/sample-cf.sh` trains a CNN with a fully-connected layer above a convolution layer.

2.7.3 Fully-connected neural networks

`sample/sample-1nn.sh` and `sample/sample-2nn.sh` train a fully-connected neural network of one layer and two layers, respectively, using bag-of-word vectors as input.

2.7.4 Linear models

`sample/sample-linear.sh` trains a linear model using bag-of-word vectors as input.

2.8 `cnn_predict` (applying a CNN to new data)

`conText cnn_predict` applies a model saved during training to new data and write prediction values to a file.

Prediction file (binary output) By default, `cnn_predict` writes prediction values to a file in the following format.

offset	length	type	description
0	4	int	s: Length of float. Typically 4.
4	4	int	c: Number of classes
8	4	int	d: Number of data points
12	$s \times c \times d$	float array	Prediction values sorted by data points first and classes next. That is, the first c values are for the first data point, and the next c values are for the second data point, and so on.

Prediction file (text output) Optionally, `cnn_predict` writes prediction values in the text format, one data point per line. However, this could be very inefficient in space/time when the number of data points or the number of classes is large.

`conText cnn_predict` **(to apply a CNN to test data)**

*	`model_fn=`	Path to the model file (input), saved during training using `cnn` via `save_fn=` parameter. Note that the model filename is a concatenation of `save_fn` and `.ite`n where n is the number of epochs, e.g., `model.ite100` if `save_fn=model`.
*	`prediction_fn=`	Path to the prediction file (output).
*	`tstname=`	Test data name.
*	`datatype=`	Dataset type. `sparse` | `sparse_multi`. This must be the same as training.
	`data_dir=`	Directory where data files are.
*	`x_ext=`	Feature filename extension. This must be the same as training.
	`test_mini_batch_size=`	Mini-batch size for parallel processing for testing. (Default:100)
	`WriteText`	Write prediction values in the text format. (Default: binary format)
	`extension=`	If the model uses multi-connection architecture, specify `extension=multi`.
	`data_ext`i`=`	If `datatype=sparse_multi`, specify data name extensions `data_ext0`, `data_ext1`,$\cdots$, similar to training.

Example 9 (`sample/sample-save-predict.sh`). *Single-connection example.*

```
conText gpu#  cnn           save_fn=output/s-train-p3.mod  save_interval=10  ···

conText gpu#  cnn_predict   model_fn=output/s-train-p3.mod.ite20 \
                            prediction_fn=output/s-test-p3.pred.bin \
                            datatype=sparse  tstname=s-test-p3  data_dir=data \
                            x_ext=.xsmatvar
```

Example 10 (`sample/sample-multiconn-save-predict.sh`). *Multi-connection example.*

```
conText gpu#  cnn_predict   model_fn=output/s-train-p2p3.mod.ite20 \
                            prediction_fn=output/s-test-p2p3.pred.bin \
                            extension=multi  datatype=sparse_multi \
                            tstname=s-test-  data_ext0=p2  data_ext1=p3 \
                            x_ext=.xsmatvar  data_dir=data
```

2.9 `write_features` (write internal features to a file)

`write_features` writes features produced internally in a CNN to a file. At the moment, this function is limited to writing features of *fixed-sized*. By default, the features produced as input to the top layer are written. Input to a middle layer (which takes input from some other layer) can also be requested via the `layer` parameter; however, if it is not a valid layer (e.g., a layer with variable-sized input), it would result in producing a file of size 0.

Output file format The output file is an array of floating-point values (`float` in C++ if the `-D` options of the makefile are unchanged), sorted by data points first and vector dimensions next. For example, if the feature vectors are 3-dimensional and there are 2 data points, the output file contains 24 bytes of six values,

$$f[0,0], f[1,0], f[2,0], f[0,1], f[1,1], f[2,1]$$

in this order, where $f[i,d]$ is the i-th component of the vector for the d-th data point.

See `sample/sample-write-features.sh` and `sample/sample-multiconn-write-features.sh` for examples.

NOTE CONTEXT v2.00 or higher is required for using this function.

Parameters for `conText write_features`.		
*	`model_fn=`	Path to the model file (input), saved during training using `cnn` via `save_fn=` parameter. Note that the model filename is a concatenation of `save_fn` and `.ite`n where n is the number of epochs, e.g., `model.ite100` if `save_fn=model`.
*	`feature_fn=`	Path to the file to which the features will be written (output).
*	`tstname=`	Data name.
*	`datatype`	Dataset type. `sparse`|`sparse_multi`. This must be the same as training.
	`data_dir=`	Directory where data files are.
*	`x_ext=.xsmatvar`	Region filename extension.
	`layer=`	Internal features produced as input to this layer will be written to the designated file. Default: top layer. The layer must be a middle or the top layer with fixed-sized input. Otherwise, it would result in either an error or output of size 0.
	`extension=`	This must be the same as training.
	`data_ext`i	If `datatype=sparse_multi`, specify data name extensions `data_ext0`, `data_ext1`,$\cdots$, similar to training.

2.10 GPU-related settings

This section describes GPU-related parameters, which may be useful for speeding up training/testing in some cases. Basic knowledge of GPU and CUDA programming is assumed.

2.10.1 Device memory handler

Allocation of GPU device memory (cudaMalloc) is relatively expensive and avoidance of frequent device memory allocation sometimes speeds up processing. For this purpose, CONTEXT is equipped with a device memory handler which obtains a large amount of device memory from the system at the beginning and allocates to the callers when needed. The device memory handler is enabled by the *memory_size* parameter as follows:

Usage: `conText` *gpu#*[`:`*memory_size*] *action param1 param2* $\cdots$

memory_size specifies the size of device memory that the device memory handler should pre-allocate; it is in GB (2^{30} bytes) and can be a fraction. If *memory_size* is omitted or set to a non-positive value, the device memory handler is disabled. If *memory_size* is too small to satisfy a request by the application with the pre-allocated memory, the memory handler calls cudaMalloc so that the application will not fail. While use of the device memory handler may speed up processing (though typically by only a small amount), when device memory is very tight, it may cause device memory shortage due to fragmentation, and the device memory handler should be disabled in that case.

Example 11. *The following example uses GPU#1 and the device memory handler with 4.5GB device memory.*

`conText 1:4.5 cnn`$\cdots$

2.10.2 To control computation on GPU

The following parameters may be useful.

To control GPU parallel processing in `conText cnn` and `cnn_predict`	
`gpu_max_threads=`	Maximum number of threads per block to be generated on GPU. (Default: the maximum of the GPU)
`gpu_max_blocks=`	Maximum number of blocks to be generated on on GPU. (Default: the maximum of the GPU)
`NoCuspasreIndex`	Do not use cusparse for row indexing (Default: use cusparse). When mini-batch size is small, this sometimes speeds up training/testing significantly.

3 Semi-supervised CNN (using `prepText` and `conText`)

The purpose of providing the semi-supervised learning code is to enable the reproduction of the experiments presented in [2]. Therefore, readers are assumed to be familiar with the concepts/terminology used in [2]. Semi-superivsed CNN training described below uses the functions in the earlier sections, and readers are assumed to be familiar with at least Section 1 (data preparation) and Section 2.1 (single-connection CNN training).

Semi-supervised learning with CNNs described in [2] is done in two steps:

1. **Training with unlabeled data**: Obtain a region embedding from unlabeled data. This is done by training with unlabeled data for the artificial task of predicting adjacent regions from each region.
2. **Training with labeled data**: Train a supervised CNN with labeled data for the task of interest using the region embedding obtained above to produce additional input to the CNN.

In both steps, training can be done by `conText cnn`, which is described for use for supervised learning earlier in this document, and a few additional parameters useful/required for the semi-supervised learning setting will be described below.

NOTE1 CONTEXT v2.00 or higher is required for semi-supervised learning.

NOTE2 To apply the semi-supervised learning process described here to your data, it is recommended to start with one of the sample scripts and modify it for your data.

- Toy examples: `sample/sample-ss-*.sh`
- Real-world data examples: `test/semisup-*.sh`
  To run these scripts, you need to download `unlab_data.tar.gz` and extract it at `test/` so that the directory `test/unlab_data` will be created.

3.1 Training with unlabeled data (tv-embedding learning)

From unlabeled data, we learn a *tv-embedding* ('tv' stands for 'two-view', see [2] for definition) via training for predicting adjacent regions from each region. This is done by first generating training data from unlabeled data using `prepText gen_regions_unsup` (or `gen_regions_parsup`) and then performing training using `conText cnn`.

3.1.1 Training data generation from unlabeled data

The first step of tv-embedding learning is to generate training data from unlabeled data, for the task of predicting adjacent regions (target regions) using each region (feature region). `prepText gen_regions_unsup` is provided for this purpose. Using this function, feature regions are represented by either bag-of-word, bag-of-n-grams, or position-sensitive vectors (sequential). Target regions are represented by either bag-of-word or bag-of-n-gram vectors.

Optionally, `prepText gen_regions_parsup` can be also used, which represents target regions in a partially-supervised fashion. To use this function, first a CNN needs to be trained with labeled data and the trained CNN should be applied to unlabeled data to obtain the *embedded regions* derived from each region (i.e., output of supervised region embedding) using `conText write_embedded`. The embedded regions are considered to be representations of regions with respect to the task objective (e.g., sentiment analysis). The obtained embedded regions are used as input to `gen_regions_parsup` to represent target regions.

The following table describes parameters for `prepText gen_regions_unsup` and `gen_regions_parsup`.

Parameters for `prepText gen_regions_unsup` and `gen_regions_parsup`.		
*	`input_fn=`	Path to the input token file or the list of token files. If the filename ends with `.lst`, the file should be the list of token filenames. The input file(s) should contain one document per line, and each document should be tokens delimited by space.
*	`x_vocab_fn=`	Path to the vocabulary file generated by `gen_vocab`, used for X (features). To represent feature regions by bag-of-n-gram vectors, include n-grams in the vocabulary.
*	`x_type=`	Vector representation for X (features). `Bow` | `Seq`. (Default: Bow)
*	`region_fn_stem=`	Pathname stem of the region vector file, target file, and word-mapping file (output). To make the pathnames, the respective extensions will be attached.
*	`patch_size=`	Feature region size.
*	`patch_stride=`	Feature region stride. (Default: 1)
*	`padding=`	Feature padding size. Feature region size minus one is recommended.
*	`dist=`	Size of adjacent regions (target regions) used to produce Y (target). The same value as `patch_size` is recommended.
	`LowerCase`	Convert upper-case to lower-case characters.
	`UTF8`	Convert UTF8 en dash, em dash, single/double quotes to ascii characters.
	`MergeLeftRight`	Do not distinguish the target regions on the left and right.
Parameters for `prepText gen_regions_unsup` only.		
*	`y_vocab_fn=`	Path to the vocabulary file generated by `gen_vocab`, used for Y (target). To represent the target regions by bag-of-n-gram vectors, include n-grams in the vocabulary.
Parameters for `prepText gen_regions_parsup` only.		
*	`embed_fn=`	Pathname to the file produced by `conText write_embedded`.
*	`f_patch_size=`	Region size that was used by `write_embedded` to generate the file. It must be no greater than `dist`.
*	`f_patch_stride`	Region stride used by `write_embedded`.
*	`f_padding=`	Padding size used by `write_embedded`.
	`num_top`	k. To produce target (output) of each instance, only the k largest components of embedded regions will be retained, and the rest will be set to zero. Use this only when the components of embedded regions are guaranteed to be non-negative, e.g., when activation is rectifier, sigmoid, and so on. This speeds up tv-embedding training combined with *negative sampling*. Useful values: 5, 10.
	`scale_y`	Scale the target values so that the maximum value becomes approximately this value. Useful value: 1.

To use `gen_regions_parsup`, `conText write_embedded` should be done in advance to apply a CNN to unlabeled data and obtain embedded regions. Since at the moment the sole purpose of this function is to produce input to `gen_regions_parsup`, the format description of the output binary file is omitted. The table below describes parameters for `write_embedded`.

Parameters for `conText write_embedded`.		
*	`tstname=`	Data name. Region filename without extension. It must not contain a plus sign.
*	`datatype=sparse`	Dataset type. Sparse input.
	`data_dir=`	Directory where data files are.
*	`x_ext=.xsmatvar`	Region filename extension.
*	`model_fn=`	Path to the model file (input).
*	`embed_fn=`	Path to the file to which embedded regions will be written (output).

	`num_top=`	k: If specified, only the k largest components of the embedded regions are retained in each vector while the rest are set to zero. Use this only when the embedded regions are guaranteed to be non-negative, e.g., when activation is rectifier, sigmoid, and so on.

3.1.2 Training of tv-embedding

Using the training data generated from unlabeled data by `gen_regions_unsup` (or `gen_regions_parsup`), tv-embedding training is done using `conText cnn` in a manner similar to CNN training for supervised learning. Therefore, all the parameters for `conText cnn` introduced earlier are available; in addition, the following parameters are required/useful.

Additional parameters for `conText cnn`: useful (probably only) for tv-embedding training		
	`zero_Y_ratio=`	r: Sample *negative examples* so that only r times more negative examples than *positive examples* are retained. A *negative* (or *positive*) example is a pair of a region vector and a component of a target vector whose value is zero (or non-zero), respectively. Useful values: 5, 10.
	`zero_Y_weight=`	t: To compute loss, assign t times larger weights to negative examples than positive examples. Useful values: $1/r$ where `zero_Y_ratio=`r.
*	`save_lay0_fn=`	Layer-0 is saved to files specified by this parameter; more precisely, filenames are generated by concatenating `ite`n`.layer0` where n is the number of epochs. Files are generated at the timing designated by `save_interval`. When trained for tv-embedding learning, layer-0 embodies the region tv-embedding (a function to convert a region to a vector). The file can be later used to initialize a *side layer* to produce additional input to a CNN trained with labeled data.
	`NoCusparseIndex`	Do not use the cusparse library for indexing rows of sparse matrices. In the typical tv-embedding training setting (as in the sample scripts), this speeds up training a lot.

Note that for tv-embedding training `x_ext=.xsmat` and `y_ext=.ysmat` must be specified since those are the types of files `prepText gen_regions_unsup` (or `gen_regions_parsup`) produces. We save a tv-embedding (a function to convert a region to a vector) learned from unlabeled data to a file specified by `save_lay0_fn` and use it in the final training with labeled data as described next.

3.2 Training with labeled data and tv-embedding

The final step of semi-supervised learning is to train a CNN with labeled data, using a tv-embedding (obtained from unlabeled data and saved to a file via `save_lay0_fn`) to produce additional input to the CNN. In this code, we call the layer that produces additional input a *side layer*. Conceptually, a side layer is a convolution layer, which receives a document represented by a sequence of one-hot vectors as input and produces a vector for each small region of the document; the layer which the side layer is attached to receives the output of the side layer as input, in addition to the one-hot vector representation of the document. In this code, conversion of one-hot vectors to region vectors (which is done by a convolution layer, conceptually) is done by `prepText gen_regions` in advance; thus, input to a side layer needs to be prepared by `gen_regions` in advance as well.

To specify a training parameter for a side layer, we attach a *side layer identifier* at the beginning, in the form of

ℓ`side`i`_`

which refers to the i-th side layer ($i = 0, 1, \cdots$) attached to layer-ℓ. For example, if layer-0 has two side layers (i.e., layer-0 receives additional input from two side layers) and layer-1 has one side layer, then there are three valid side layer identifiers: `0side0_`, `0side1_`, and `1side0_`.

Additional parameters for `conText cnn`: to specify side layers	
ℓ`num_sides=`	Number of side layers attached to layer-ℓ. (Default: 0)
ℓ`side`i`_fn=`	The file to which `conText cnn` saved a tv-embedding via `save_lay0_fn`. The side layer warm-starts from this file.
ℓ`side`i`_dsno=`	The id of the dataset to be used as input to the side layer.
ℓ`side`i`_Fixed`	Do not update the weights.

3.3 To use word vectors obtained elsewhere

Use of word vectors obtained from unlabeled data is a form of semi-supervised learning. In the baseline experiments in [2], word vectors are integrated into CNN to produce additional input, using two methods; one takes the concatenation, and the other takes the average, of word vectors for the words in each region.

See `sample/sample-ss-wordvec*.sh` and `test/onehot-gn.sh` for details of these configurations. Here only an outline is given.

To enable conversion of one-hot vectors to word vectors in a side layer, the rows of side-layer weights need to be set to word vectors. For this purpose, `conText adapt_word_vectors` needs to be called before training, to convert word vectors to a *weight file* which `conText cnn` can use for initializing the weights. The parameters for `adapt_word_vectors` are shown in the table below.

To produce the concatenation of word vectors in a side layer, we need to prepare a region file with region size 1 as input to the side layer so that multiplication of the weights (initialized by word vectors) and a region vector results in conversion of a one-hot vector to a word vector. Specify how to do concatenation via `patch_size`, `patch_stride`, and `padding`; also specify `PatchLater` so that concatenation (i.e., generation of a patch) will be done after conversion of one-hot vectors to word vectors.

To produce the average of word vectors in a side layer, we need to prepare a region file with `Bow` option with an appropriate region size so that multiplication of the weights and a region vector results in the sum of word vectors in the region and specify `Avg` to convert the sum to the average.

	Parameters for `conText adapt_word_vectors`	
	`wordvec_bin_fn`	Path to a binary file containing word vectors and words in the word2vec format (input). Either this or a pair of text files (`wordvec_txt_vec_fn` and `wordvec_txt_words_fn` below) is required. Note that `conText` reads this file in the endian-sensitive fashion. To share word vectors among the systems with different endianness, the text format below can be used instead.
	`wordvec_txt_vec_fn=`	Path to the word vector text file (input). One line per a vector. Vector components should be delimited by space.
	`wordvec_txt_words_fn=`	Path to the word vector vocabulary file (input) corresponding to the word vector text file (`wordvec_txt_vec_fn`). One line per word.
*	`word_map_fn=`	Path to the word-mapping file (`*.xtext`) generated by `prepText gen_regions` (input). Word vectors will be sorted in the order of this file so that the dimensions of the resulting weights will correctly correspond to the dimensions of the region vectors generated by `gen_regions`.
*	`weight_fn=`	Path to the weight file to be generated (output).

`rand_param=`	x: scale of initialization. If the word-mapping file (`word_map_fn`) contains words for which word vectors are *not* given, the word vectors for these unknown words will be randomly set by Gaussian distribution with zero mean with standard deviation x. (Default:0)
`random_seed=`	Seed of random number generation.
`IgnoreDupWords`	Ignore it if there are duplicated words associated with word vectors. If this is not turned on, the process will be terminated on the detection of duplicated words.

4 Hints and Tips

Here are things to consider for obtaining good performance.

- **seq-CNN vs. bow-CNN**. With `Bow-convolution` turned on when generating region files, region files for bow-CNN is generated, which represents regions by bag-of-word vectors; otherwise, seq-CNN is trained which represents regions by concatenation of one-hot vectors. Roughly speaking, if on your data, linear models with bag-of-n-gram vectors outperform linear models with bag-of-word vectors, then seq-CNN with a small region size (e.g., 3, 4, or 5) is likely to be effective on that data. Otherwise, bow-CNN with a large region size (e.g., 10, 15, or 20) is likely to be effective on that data. In [1], an example application of the former is sentiment classification and the latter is news topic classification.

- **Step-size scheduling**. In the experiments in [1], the number of epochs was fixed to 100 and the step-size was always reduced by multiplying 0.1 after 80 epoch (`num_iterations=100 step_size_scheduling=Few step_size_decay=0.1 step_size_decay_at=80`). This technique (reducing the step-size once or twice towards the end of training), also used in [3], often improves performance while it never hurts it.

- **Initial step-size**. It is important to set the initial step-size `step_size` to a good value. To search for a good value on the development data, typically, we tested $0.5, 0.25, 0.1, 0.05, \cdots$, while fixing momentum to 0.9 and mini-batch size to 100 (`momentum=0.9 mini_batch_size=100`) and step-size scheduling to the setting above.

- **Number of epochs**. The number of training data points used in [1] was around 25K. 50–100 epochs seemed appropriate for this size of data. If your training data is much larger, fewer epochs may suffice since there is more data to go through.

- **Case shifting**. Be consistent on case shifting. If you generated a vocabulary file with the `LowerCase` switch turned on, then generate regions files with the `LowerCase` switch on.

- The scripts in the `test` directory are for reproducing the experiments reported in [1], and they provide realistic examples on the real-world data (movie reviews, product reviews, and news). Note that the sample scripts at the `sample` directory are only for showing how to use CONTEXT, and their parameter settings are not intended for the optimum performance.

4.1 GPU memory consumption during training (`conText cnn`)

In a typical environment, GPU memory (*device memory*) is relatively small compared with CPU memory (*host memory*). When `conText` runs out of GPU memory, it terminates with an error message saying "out of memory", e.g.,

```
!cuda error!:  (Detected in AzPmem::alloc extra)
AzCuda::throwIfError:  AzPmat::reform_noinit 8.2e+08
cudaGetErrorString returned out of memory
```

The following describes what are loaded into GPU memory during training by `conText cnn` and therefore what parameters affect the consumption of GPU memory.

- **Weights**.
  A larger number of neurons (specified by `nodes`) consumes more GPU memory. A larger number of target classes also consumes more GPU memory since the number of weight vectors in the top layer is equal to the number of classes.

- **Target and prediction values of the entire test data** for optional testing during training.
  While `conText cnn` (training) can optionally do testing as training proceeds, it is not meant for *large* development/test data, where *large* means either *many data points* and/or *many target classes* (e.g., 3K). To test on large test data, `conText cnn_predict` should be used instead. That is, if your test data is large, avoid testing while training (which can be done by specifying `NoTest` and omitting `tstname`), save a trained model to a file (using `save_fn` and `save_interval`), and apply the saved model to the test data using `conText cnn_predict`.

- **Data in a mini batch** and work areas for processing a mini batch.
  A larger mini batch size (`mini_batch_size`) consumes more GPU memory. Also, a larger `num_pooling` increases the size of feature vectors used in the layer above, which are in GPU memory.

- **With CONTEXT v1.00, when `y_ext=.y`, the entire training target** (but not features) is loaded into GPU memory in the dense format (2D array: #(classes)×#(training data points)). When #(classes) is large, this is problematic. There are two solutions to this issue:
  - (**Recommended**) Use CONTEXT v1.01 (or higher), which only loads training target of a mini batch in the sparse format, irrespective of the `y_ext` setting, i.e., GPU memory consumption does not depend on the number of training data points.
  - With CONTEXT v1.00, specify `y_ext=.ysmat` in both `conText cnn` and `prepText gen_regions`. Then internal handling of target during training will be the same as v1.01 described above.

4.2 CPU memory consumption during training (`conText cnn`)

Training data Training data is read from the region file and target file and stays in the CPU memory during training. If training data is too large to fit in your CPU memory, divide the data into several *batches*. By doing so, only one of the batches is loaded into CPU memory; thus, for example, having k batches reduces CPU memory consumption to $1/k$. Note, however, that each batch must be a good sample of the feature/class distributions; therefore, division into batches should be done randomly. See Section 2.3 and `sample/sample-batch.sh` for how to use batches.

Test data If optional testing is done, test data also stays in the CPU memory. `conText cnn` (training) is not meant for large test data. `conText cnn_predict` should be used for large test data, if necessary, multiple times. That is, if your test data is large, avoid testing while training (which can be done by specifying `NoTest` and omitting `tstname`), save a trained model to a file (using `save_fn` and `save_interval`), and apply the saved model to the test data using `conText cnn_predict`.

4.3 When to make training data batches (`conText cnn`)

The batch option should be used if the number of training data is so large that

- it does not fit in the CPU memory, or
- one epoch (going through the entire training data) takes very long and you want to save the models or reduce the step-size before reaching the end of the training data.

Note that GPU memory consumption does not depend on the number of training data points, if the version is v1.01 (or higher), or if `y_ext=.ysmat` is used with v1.00 as described in Section 4.1. Also note that each batch must be a good sample of the feature/class distributions; therefore, division into batches should be done randomly. See Section 2.3 and `sample/sample-batch.sh` for how to use batches.

References

[1] Rie Johnson and Tong Zhang. Effective use of word order for text categorization with convolutional neural networks. In *Proceedings of NAACL HLT*, 2015.

[2] Rie Johnson and Tong Zhang. Semi-supervised convolutional neural networks for text categorization via region embedding. In *Proceedings of NIPS*, 2015.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of NIPS*, 2012.

[4] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15, 2014.

[5] Matthew D. Zeiler. ADADELTA: An adaptive learning rate method. *arXiv:1212.5701*, 2012.