ConText Version 4: Neural Networks for Text Categorization

July 18, 2017

Revision history

- July, 2017: Entirely revised (including the title) when v4.00 (corresponding to [JZ17]) was released.
- May, 2016: Revised when v3.00 (corresponding to [JZ16b]) was released.
- September, 2015: Revised when v2.00 (corresponding to [JZ15b]) was released.

ConText This document describes how to use ConText version 4, neural network code for text categorization. ConText provides an implementation of neural networks for text categorization described in [JZ15a, JZ15b, JZ16b, JZ17]:

- Shallow CNNs (convolutional[1] neural networks) [JZ15a]
- Shallow CNNs enhanced with *unsupervised embeddings* (embeddings trained in an unsupervised manner) [JZ15b]
- Deep pyramid CNNs (DPCNN) enhanced with unsupervised embeddings [JZ17]
- LSTMs (long short-term memory networks) enhanced with unsupervised embeddings [JZ16b]

The shallow CNN is suitable for training with a small amount of training data, e.g., 25000 training documents, and DPCNN is suitable for training with a large amount of training data, e.g., one million training documents. If a good amount of in-domain unlabeled data is available, use of unsupervised embeddings is likely to improve the model accuracy. For this purpose, the labeled training data can also be used as unlabeled data ignoring the labels if it is sufficiently large.

ConText efficiently handles *high-dimensional* and *sparse* documents of *variable sizes without* shortening or padding to a fixed length.

Prerequisite General knowledge of classification and convolutional neural networks is assumed. Please at least read "Deep pyramid convolutional neural networks for text categorization" [JZ17] to get familiar with the concepts/terminology used in this document. It is also desirable to be familiar with [JZ15a, JZ15b], which are referred to by [JZ17]. LSTM users are assumed to be familiar with "Supervised and Semi-supervised Text Categorization using LSTM for Region Embeddings" [JZ16b].

Road map Section 1 gives an overview and introduces terminology and concepts used in the rest of the document. A table of shell scripts for reproducing the experiments in [JZ15a, JZ15b, JZ16b, JZ17] is in Section 1.6. There are two executables, `prepText` (data preparation) and `reNet` (neural network training/testing). Sections 2 and 3 describe the interface of `prepText` and `reNet`, respectively, with focus on simpler cases that do not use unsupervised embeddings. Section 4 describes how to use `prepText` and `reNet` for training unsupervised embeddings and integrating the trained unsupervised embeddings into a neural network. Finally, some tips are provided in Section 5.

[1] 'Con' of ConText comes from 'convolution' as well as 'context'.

NOTE To use this code, 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.

Contents

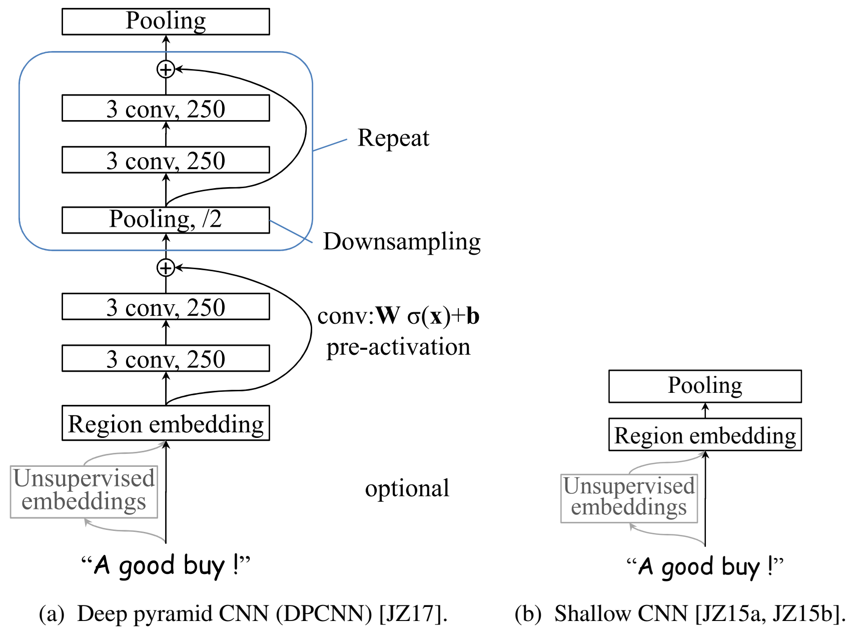
1 Overview **3**
- 1.1 Training without unsupervised embeddings . . . 3
- 1.2 Training with unsupervised embeddings . . . 4
  - 1.2.1 Side layers . . . 5
  - 1.2.2 Multi-dataset input . . . 5
- 1.3 Handling of variable-sized documents . . . 5
- 1.4 Training data batches for large data . . . 5
- 1.5 Endianness . . . 5
- 1.6 Shell scripts in the package . . . 6

2 Data preparation: `prepText` **6**
- 2.1 `prepText gen_vocab`: vocabulary file generation . . . 7
- 2.2 `prepText gen_regions`: region file and target file generation . . . 8

3 Neural network training and testing: `reNet` **9**
- 3.1 `reNet train`: Neural network training . . . 10
  - 3.1.1 Network parameters . . . 10
  - 3.1.2 Layer parameters . . . 12
  - 3.1.3 Output . . . 14
  - 3.1.4 Multi-dataset input . . . 15
  - 3.1.5 Training data batches . . . 15
  - 3.1.6 Warm-start from a saved model . . . 15
- 3.2 `reNet predict`: applying a trained network to new data . . . 16
- 3.3 GPU-related settings . . . 16
  - 3.3.1 Device memory handler . . . 16
  - 3.3.2 To control computation on GPU . . . 17

4 Using unsupervised embeddings **17**
- 4.1 Creation of an artificial auxiliary task using unlabeled data . . . 17
- 4.2 Training of unsupervised embedding . . . 18
- 4.3 Training with labeled data and unsupervised embedding . . . 18
- 4.4 Integration of external word vectors . . . 19

5 Tips **20**
- 5.1 Miscellaneous tips . . . 20
- 5.2 GPU memory consumption during training . . . 20
- 5.3 CPU memory consumption during training . . . 20



(a) Deep pyramid CNN (DPCNN) [JZ17]. (b) Shallow CNN [JZ15a, JZ15b].

Figure 1: Deep pyramid CNN (DPCNN) and shallow CNN for text categorization. Copied from [JZ17]. $\oplus$ indicates addition.

1 Overview

This section describes the steps taken for training neural networks using ConText and introduces terminology and concepts referred to in the rest of this document.

1.1 Training without unsupervised embeddings

Let us start with a simple case where we do not use unsupervised embeddings. Using ConText, training and testing of neural networks of Figure 1 (ignoring unsupervised embeddings in the gray boxes) take the following steps.

1. Generate a vocabulary file from training data.
   Input: a tokenized text file.

2. Generate files used as input to neural network training, which are

   - *Region file*: a file that contains features in the form of *sparse region vectors* (described below).
   - *Target file*: a file that contains classification label information.
   - *Word-mapping file*: a file that shows the mapping between the words and the dimensions of sparse region vectors.

   Input: a vocabulary file, a tokenized text file, a label file, and a label dictionary.

3. Train a neural network while optionally testing the classification error rate on the validation data and/or test data. Trained neural networks can be saved in a file.
   Input: a region file, a target file, and a word-mapping file of training data (and optionally test data).

4. Optionally, save the trained neural network to a file, and apply it to test data.

There are a few things unique to CONTEXT in this flow.

Sparse region vectors A CNN for text typically starts with converting each word in a document to a dense and continuous vector (*word embedding*). By contrast, the first layer of the *Shallow CNN* [JZ15a, JZ15b] as well as *Deep pyramid CNN (DPCNN)* [JZ17] (Figure 1) performs *text region embedding*, which generalizes word embedding to the embedding of text regions covering one or more words. Essentially, a *region embedding layer* converts each region of text centered around each word in a document to a dense and continuous vector via an embedding function $f(\mathbf{x})$. The first step of doing so is to make input $\mathbf{x}$, a straightforward vector representation of text regions. The following three types are considered in Section 2.2 of [JZ17] and implemented in CONTEXT:

- sequential input: the concatenation of one-hot vectors representing the words in the region
- bow input: a bag-of-word vector
- bag-of-n-gram vector, e.g., a bag of word uni, bi, and trigrams contained in the region

In this document, we call these vectors *sparse region vectors*. Setting the region size to 1, the sparse region vector of all types becomes a one-hot vector, and thus region embedding becomes word embedding.

A region embedding layer with the sequential input is equivalent to a convolution layer applied to a sequence of one-hot vectors representing a document. [JZ15a, JZ15b] took this viewpoint and called a region embedding layer a *convolution layer*. That is why we call the networks of [JZ15a, JZ15b] 'Shallow CNNs', but from the region embedding view point, they are 'a region embedding layer + a pooling layer'.

Generating a *region file* containing *sparse region vectors* prior to training/testing Although conceptually generation of sparse region vectors is part of a region embedding layer, CONTEXT separates it out from a region embedding layer and performs it prior to neural network training or testing. That is, a *region file* that contains sparse region vectors is generated in advance and used as input to neural network training/testing. This is for speeding up training, and this is why a feature file is called *region file* in this document.

1.2 Training with unsupervised embeddings

In Figure 1, [JZ15b, JZ16b, JZ17] have shown that *unsupervised embeddings* of text regions, which are region embeddings trained in an unsupervised manner, improves text categorization accuracy when used for producing additional input to CNNs or LSTMs. Training of unsupervised region embeddings with CONTEXT takes the following two steps.

1. **Generation of an artificial auxiliary task**: On unlabeled data, generate features and labels for an artificial auxiliary task of predicting neighboring text regions, which results in a region file, a target file, and a word-mapping file.
2. **Training for the auxiliary task**: Train a neural network with one hidden layer, using the files above as input, and the save the hidden layer in a file, which will serve as an unsupervised embedding function in the final neural network training.

Conceptually, the hidden layer of this network is a region embedding layer, and we again separate out the process of sparse region vector generation from the layer so that it is done prior to the training.

The trained unsupervised embedding function is integrated into a neural network such as Shallow CNNs or DPCNN via a *side layer*.

1.2.1 Side layers

In CONTEXT, a *side layer* is a special layer that takes input only from input data, not from other layers, and passes its output only to layer-0, which is a bottom layer; see the gray boxes with 'Unsupervised embeddings' in Figure 1 for example. That is, when a side layer is attached, layer-0 takes two types of input, sparse region vectors and the output of the side layer, and it conceptually treats them as concatenated.

An unsupervised embedding can be integrated into a neural network by attaching a side layer to layer-0 and initializing it with the embedding layer trained on unlabeled data. As is done in [JZ15b, JZ16b, JZ16a, JZ17], it is useful to have more than one unsupervised embedding that use diverse region sizes and diverse sparse region vector representations. For this purpose, multiple side layers can be attached to layer-0.

1.2.2 Multi-dataset input

We again separate the process of sparse region vector generation from a side layer and do it prior to training. Note that the sparse region vectors required for a side layer can be different from the sparse region vectors for layer-0 (the bottom layer). This is because they may have different region sizes, or they may use different vocabularies (e.g., words vs. $\{1,2,3\}$-grams of words). Also, more than one side layer can be attached to layer-0, and they may require different sparse region vectors from each other. In this situation, a neural network needs to take *multi-dataset input* with information indicating which layer should use which region file. CONTEXT provides `dsno` parameter (described later) for passing such information.

1.3 Handling of variable-sized documents

Naturally, the length of documents is not a fixed value. CONTEXT treats such variable-sized documents as variable-sized without any shortening or padding to a fixed length. A document is internally represented as a matrix whose columns correspond to locations in the document (or time steps in a sequence), and the number of columns can be different from document to document while the number of rows must be fixed for all documents at each layer. However, when the internal data reaches the last weight layer (called *top layer* in this document), which serves as a linear model for classification, the number of columns for each document must be 1 so that documents are represented by document vectors of a fixed length. This can be achieved by pooling with a fixed number of pooling units (followed by flattening if the number is greater than 1), as discussed in Section 2.2.3 of [JZ15a].

1.4 Training data batches for large data

When training data is large (e.g., one million documents), it is convenient to divide it into several batches (e.g., 10 batches). Moreover, this batch option is recommended if the amount of training data is so large that

- it consumes too much CPU memory if loaded entirely, or
- one epoch (going through the entire training data) takes a long time and you want to save/test the models or reduce the step-size before reaching the end of the training data.

That is, while the entire training data (i.e., region files etc. for training data) is loaded into the CPU memory by default, with training data batches, only one batch at a time is loaded into the CPU memory, which reduces the CPU memory consumption to $1/k$ with k batches. While testing can be done only at the end of each epoch by default, with training data batches, testing can be done at the end of each batch. However, note that each batch should be a good sample of the feature/class distributions; therefore, each batch has to be sufficiently large, and division into batches must be done randomly.

1.5 Endianness

Non-text files generated by this code (e.g., model files, embedding files, and feature files) are Endian sensitive and cannot be shared by the systems with different Endianness.

1.6 Shell scripts in the package

The table below shows the shell scripts included in the package, located at `examples/`. To get started, it is recommended to pick one of these scripts suitable for the purpose, instead of starting from a scratch, and modify it for your own data.

In the table below, the scripts with a citation in the second column contain the network configurations and meta-parameters used in the experiments in the corresponding publication. Those experiments used real-world data, of which small datasets, IMDB and Elec, are included in this package. The scripts with "toy data" are only for providing a quick example, and their network configurations and meta-parameters are not optimal for the data.

Script	Source	Description
sample.sh	With toy data	For confirming installation.
sample-save-predict.sh		Applying a saved model to test data.
sample-multilab.sh		Multi-label classification.
sample-wordvec.sh		Use of external word vectors.
dpcnn-without-unsemb.sh	[JZ17]	DPCNN without unsupervised embeddings.
dpcnn-with-1unsemb.sh	With toy data	DPCNN with one unsupervised embedding.
dpcnn-with-4unsemb.sh	[JZ17]	DPCNN with four unsupervised embeddings.
dpcnn-functions.sh		Functions called from the scripts above (dpcnn*.sh).
shcnn-seq-imdb-elec.sh	[JZ15a] seq	Shallow CNN w/ sequential input.
shcnn-seq2-imdb-elec.sh	[JZ15a] seq2	Shallow CNN w/ sequential input with two region sizes.
shcnn-seq2-bown-imdb-elec.sh	[JZ15a] seq2-bow$n$	Shallow CNN w/ sequential input and NB-weighted n-grams.
shcnn-unsup-imdb-elec.sh	[JZ15b] unsup-tv	Shallow CNN w/ unsupervised embedding with bow input.
shcnn-parsup-imdb-elec.sh	[JZ15b] parsup-tv	Shallow CNN w/ partially-supervised embedding.
shcnn-unsup3-imdb-elec.sh	[JZ15b] unsup3-tv	Shallow CNN w/ unsupervised embed. with bag of n-grams.
shcnn-3unsemb-imdb-elec.sh	[JZ15b] all three	Shallow CNN w/ 3 unsupervised embeddings.
lstm-imdb-elec.sh	[JZ16b]	LSTM without unsupervised embeddings.
lstm-2unsemb-imdb-elec.sh	[JZ16b]	LSTM w/ 2 unsupervised LSTM embeddings.
lstm-5unsemb-imdb-elec.sh	[JZ16b]	LSTM w/ 5 unsupervised LSTM/CNN embeddings.
shcnn-5unsemb-imdb-elec.sh	[JZ16b]	Shallow CNN w/ 5 unsupervised LSTM/CNN embeddings.

There are additional shell scripts at `examples/other-sh/`, which are mainly for the RCV1 experiments in [JZ15a, JZ15b, JZ16b]. The text files of RCV1 are not included in the package due to the copyright issues.

Some scripts (`*3unsemb*.sh`, `*5unsemb*.sh`, and `*parsup*.sh`) optionally download files of the *little-endian* format (Intel convention) such as embedding files. These files cannot be used in systems with *Big Endian* (Motorola convention). Please read the comments in the scripts for alternatives in that case.

NOTE To run the shell scripts either at `examples/` or `examples/other-sh/`, the current directory must be set to `examples/`.

2 Data preparation: `prepText`

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

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

The first argument *action* takes the following values:

`gen_vocab`	Generate a vocabulary file used as input to region file generation.

`gen_regions`	Generate a region file, a target file, and a word-mapping file used as input to neural network training and testing.
`gen_regions_unsup`	From unlabeled data, generate training data for an artificial auxiliary task for the purpose of unsupervised embedding training.
`merge_vocab`	Merge vocabulary files. Meant for creating a vocabulary file of mixed types such as a vocabulary set of uni-, bi- and trigrams of words.
`adapt_word_vectors`	Used for converting external word vectors (pre-trained elsewhere) to the format usable in CONTEXT.

Among these, `gen_regions_unsup` and `adapt_word_vectors` are related to unsupervised embeddings, and so they are discussed later in Section 4.

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

2.1 `prepText gen_vocab`: vocabulary file generation

`prepText gen_vocab` extracts 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 `examples/data/s-test.txt.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 a token file (input). If it ends with `.lst`, the file should contain a list of pathnames of token files, one per line.
*	`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. (Default:1)

Example 1. *Extract words (delimited by space) from* `data/s-train.txt.tok` *and convert them to lower cases*

while only retaining the most frequent 10000 words:

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

2.2 `prepText gen_regions`: region file and target file generation

Next, we generate a region file, target file, and a word-mapping file, which serve as input files for neural network 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 tokenized 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 |.
See `examples/data/s-multilab-test.cat` for example. The labels should be strings without white space, and they must be declared in a label dictionary described below.

Text/label file naming conventions The tokenized text file and the corresponding label file must have the same pathname stem with different file extensions, `.txt.tok` and `.cat`, respectively, by default, e.g., text file `data/s-train.txt.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 `examples/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 sparse region vectors with the specified region size, stride, and padding. To generate the region file, a file extension `.xsmatbcvar` (which indicates that the file is in a sparse, $\{0, 1\}$, compact, 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 neural network trainer `reNet` can read. To generate the target file, a file extension `.y` 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 for ensuring 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.
*	`vocab_fn=`	Path to the vocabulary file generated by `gen_vocab` (input).
*	`region_fn_stem=`	Pathname stem of the region 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 `+`.
*	`patch_size=`	Region size. For LSTM, set this to 1.
	`patch_stride=`	Region stride. (Default:1)
	`padding=`	Padding size. (Default:0)
	`Bow`	Sparse region vectors are generated in the bag-of-word representation.
	`VariableStride`	Take variable strides; see Section 3.1 of [JZ15a].
	`RegionOnly`	Generate a region file only. Do not generate a target file.
	`text_fn_ext=`	Filename extension of the tokenized text file. (Default:`.txt.tok`)
	`label_fn_ext=`	Filename extension of the label file. Used only when target file generation is required. (Default:`.cat`)
	`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.
	`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.
	`NoSkip`	Do not remove empty regions.
	`batch_id=`	Batch ID. This must be in the format of i`of`k such as `1of5` (the first batch out of 5) and `2of5` (the second batch out of 5). See Sections 1.4 and 3.1.5 for batches.

3 Neural network training and testing: `reNet`

`reNet` is an executable for training and testing neural networks using GPU.

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

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

memory_size Description of this argument is delayed to Section 3.3.1.

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

- `train`: Train a neural network. Optionally, test the network on validation/test data as training proceeds.
- `predict`: Apply the trained network to new data.

***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.

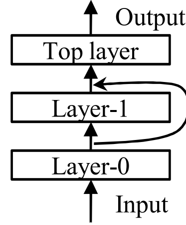


Figure 2: To indicate the layer connection as in this figure, specify `layers=2 conn=0-1-top,0-top`.

Hidden layers In CONTEXT, a neural network consists of two types of layers, *hidden layers* and a *top layer* (output layer). Hidden layers are defined by users, with unique and consecutive layer numbers starting from zero, e.g., layer-0, layer-1, $\cdots$. Each layer is defined with the type of layer such as a weight layer or a pooling layer.

Top layer (output layer) A top layer is a special layer that only performs weighting ($\mathbf{Wx} + \mathbf{b}$) to generate output. It is automatically attached on top of the user-defined hidden layers.

3.1 `reNet train`: Neural network training

There are two types of parameters, *layer parameters* which specifies parameters for each layer, and *network parameters*.

3.1.1 Network parameters

The parameters below are network parameters, which control the overall aspect of network definition and training.

To define network architecture.		
*	`layers=`	Number of hidden layers.
	`conn=`	This parameter defines the connections among layers using '`-`' as a connector and '`,`' as a delimiter. Hidden layers are identified with consecutive layer numbers starting from 0, and the top layer is represented by '`top`'. For example, `layers=2 conn=0-1-top,0-top` indicates the connection shown in Figure 2. When omitted, data flows in the order of the layer numbers.
	`ConcatConn`	Effective only when there is a layer that receives data from more than one layer. If this switch is turned on, the outputs of two or more layers going into one layer are concatenated; otherwise, they are added (`AdditiveConn`).
To control training input		
*	`trnname=`	Input training filename without extension.
*	`tstname=`	Input test filename without extension. Required for testing unless `NoTest` is specified. Two test sets can be specified using + (a plus sign) as a delimiter. For example, `validation+test` specifies two test sets whose filenames (without extension) are `validation` and `test`.
*	`datatype=sparse`	Data type. `sparse` indicates sparse input. Other types are for image processing and not described in this document.
	`data_dir=`	Directory where data files are.
	`dsno`i`=`	For multi-dataset input. Explained in Section 3.1.4.
	`x_ext=`	Region filename extension. Specify the extension of the region file as generated by `prepText`, which is `.xsmatbcvar` except for the case of unsupervised embedding training. (Default:`.xsmatbcvar`)

	Parameter	Description
	`y_ext=`	Target filename extension. Specify the extension of the target file as generated by `prepText`, which is `.y` except for the case of unsupervised embedding training. (Default: `.y`)
	`NoTest`	Do no testing. Training only.
	`num_batches=`	Number of training data batches. Explained in Section 3.1.5. (Default:1)
To control training output		
	`save_fn=`	Path to the file the model is saved to.
	`save_interval=`	For example, with `save_after=40` and `save_interval=5`, a model is saved to a file after 40 epochs, 45 epochs, and so on.
	`save_after=`	See above. (Default: 0)
	`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.
	`inc=`	d. Show progress after going through d data points. Useful when data is large or when training is time-consuming.
To control training.		
*	`num_epochs=`	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 $y \in \{0,1\}$): $\log(1+\exp(-(2y-1)p))$
	`max_loss=`	When estimated training loss goes over this value, training is terminated. Use this to detect 'explosion', typically caused by setting the learning rate to a too large value.
	`optim=`	`SGD` | `Rmsp` | `AdaDelta` `SGD`: mini-batch SGD with or without momentum. `Rmsp`: rmsprop [TH12]. `AdaDelta` [Zei12]. (Default:`SGD`).
	`momentum=`	Momentum. Useful value: 0.9. Effective only when `optim=SGD`.
	`mini_batch_size=`	Mini-batch size for training. (Default:100)
	`ss_scheduler=`	Step-size scheduler type. `Few` (reduce it a few times) is recommended. (Default: no scheduling)
	`ss_decay=`	Use this with `ss_scheduler=Few`. Step-size is reduced by multiplying this value when it is reduced. Useful value:0.1
	`ss_decay_at=`	Use this with `ss_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. Useful value: $\frac{4}{5}$`num_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).
	`SaveDataMem`	Reduce the consumption of the CPU memory by loading test data into the CPU memory only when it is needed so that test data and training data do not occupy the CPU memory at the same time. This saves the CPU memory at the expense of file I/O overhead.

Example 2. *Perform 20 epochs of training with mini-batch size 100 while reducing the step-size by multiplying 0.1 after 15 epochs.*

```
···   num_epochs=20 mini_batch_size=100 \
      ss_scheduler=Few ss_decay_at=16 ss_decay=0.1
```

3.1.2 Layer parameters

`reNet` provides distinct types of hidden layer such as an LSTM layer and a pooling layer. Different types of layers require different layer parameters.

The layer parameters 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 applicable layers while 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
```

However, as an exception, omission of the layer id is not allowed for `layer_type`, `name` (layer name), and `layer_fn` (layer initialization file).

Layer parameters shared by all types are as follows.

Layer parameters.		
ℓlayer_type=	Act	Activation. A component-wise function is applied.
	Dropout	Dropout.
	DenWei	Weighting. A DenWei layer can take dense input only (Den for dense).
	LstmF	Forward LSTM (left to right).
	LstmB	Backward LSTM (right to left).
	Lstm2	Bi-directional LSTM (forward LSTM + backward LSTM).
	Noop	Do nothing.
	Patch	Concatenate vectors belonging to (typically overlapping) small regions (i.e., patches). Dense input only. A Patch layer followed by a DenWei layer makes a convolution layer.
	Pooling	Pooling and optional response normalization.
	Weight+	Weighting. Sparse input is allowed. Optionally, it can perform pooling and response normalization in this order, after weighting and activation, as well as dropout before weighting and activation.
	WeightS+	Same as Weight+ except that it allows side layers attached to it.
ℓname=	Optionally, each layer can be named. The assigned name is used when network configuration is displayed in stdout.	
ℓlayer_fn=	Optionally, a layer can be initialized by a *layer file*, which was generated via save_layer_fn during previous training. Used for pre-training and integration of unsupervised embeddings.	
[ℓ]save_layer_fn=	x: Layer-ℓ is saved to a *layer file*, x.epo$n$.ReLayerℓ. where n is the number of epochs at the time of saving.	

`Weight[S]+` layers and `Lstm{F|B|2}` layers can take sparse input and so they can be a bottom layer. Other types of layer can take only dense input.

The table below describes the parameters specific to the layer types.

Act layers	
[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`: $\sigma(x) = (\exp(2x) - 1)/(\exp(2x) + 1)$
`DenWei` layers (weight layer with dense input)		
*	[ℓ]`nodes=`	Number of weight vectors, also called the number of feature maps.
*	[ℓ]`step_size=`	Step-size (learning rate). Effective and required only when `optim=SGD` or `Rmsp`.
	[ℓ]`Fixed`	Fix the weights. No update.
	[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)
	[ℓ]`rmsprop_decay=`	Coefficient of exponential moving average for rmsprop. Effective only when `optim=Rmsp` is on. (Default:0.95)
	[ℓ]`adad_rho=`	ρ in Algorithm 1 of [Zei12]. Effective only when `optim=AdaDelta`. Useful value: 0.95
	[ℓ]`adad_epsilon=`	ϵ in Algorithm 1 of [Zei12]. Effective only when `optim=AdaDelta`.
`Patch` layers. Note: `Patch`$\rightarrow$`DenWei` makes a 1D convolution layer. It can be also used for flattening.		
	[ℓ]`patch_size=`	Region size or the size of 1D convolution kernel.
	[ℓ]`patch_stride=`	Stride.
	[ℓ]`padding=`	Padding size at the beginning and the end of sequences.
	[ℓ]`ForceSameShape`	Force the output to have the same number of locations (or time step) as input, adjusting the padding size at the end if necessary.
`Pooling` layers.		
	[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 [JZ15a]. 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. This is like pooling size for images except that this is 1D pooling. Variable-sized output is produced.
	[ℓ]`pooling_stride=`	Pooling stride. Required if `pooling_size` is specified.
	[ℓ]`resnorm_type=`	`None` | `Text`. (Default: `None`). `None`: No normalization. `Text`: Let v_i be a vector at location i (or time step i) after pooling. v_i is normalized by multiplying with $(1 + \|v_i\|^2)^{-1/2}$.
`Dropout` layers		
	[ℓ]`dropout=`	Dropout rate [SHK+14]. (Default:No dropout).
`Weight+` layers and `WeightS+` layers		
	To control weighting, use the same parameters as `DenWei`. Optionally, this layer can be configured as a multi-functional layer, which performs dropout before weighting, and activation and pooling after weighting. To activate these additional operations, specify parameters for the respective layers (`Dropout`, `Act`, and/or `Pooling`).	

	[ℓ]dsno=	Dataset#. Used in the multi-dataset setting. See Sections 1.2.2 & 3.1.4.
	[ℓ]weight_fn=	Weight initialization file. Used for initializing a word embedding layer with word vectors pre-trained elsewhere. See Section 4.4.
LstmF, LstmB, and Lstm2 layers: To control weighting, use the parameters for DenWei.		
*	[ℓ]nodes=	Number of LSTM units, or the dimensionality of output vectors.
	[ℓ]dsno=	Dataset#. Used in the multi-dataset setting.
	[ℓ]chop_size=	s: during training, each document will be chopped into segments of maximum size s. Recommended values: 50, 100.
	[ℓ]NoGate_i	Do not use the input gate.
	[ℓ]NoGate_o	Do not use the output gate.
	[ℓ]NoGate_f	Do not use the forget gate. Not recommended.
	[ℓ]bprop_max=	Backpropagation goes back only this far. Not recommended. Provided only for baseline experiments. Use chopping instead.

Example 3. *In the following, layer-0 is a* Weight+ *layer with 500 feature maps, 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.*

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

3.1.3 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:

```
epoch,5,0.226053, perf:err,0.34
epoch,10,0.119096, perf:err,0.22
```

The numbers in each line above are the number of epochs, training loss approximation, 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. If there are two test sets, two error rates are shown after perf:err in the order of the test set filenames in the tstname parameter.

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

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 a special format. The model files can be used for making predictions on new data. For example, when save_fn=my_model, the model filename after 100 epochs will be my_model.epo100.ReNet.

3.1.4 Multi-dataset input

As described in Section 1.2.2, some neural networks require multi-dataset input. Information as to which layer should get input from which region file is specified via the `dsno` (dataset#) parameters, associating region files with the dataset#'s and associating with the dataset#'s with bottom layers. For example,

```
data_dir=data trnname=s-train- tstname=s-test- dsno0=p2 dsno1=p3
```

indicates that the pathname stems (excluding extensions such as `.y`) 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`

To indicate which layer takes which dataset, the layer parameter ℓ`dsno` needs to be specified. For example,

```
0dsno=0 1dsno=1
```

indicates that layer-0 takes dataset-0, and layer-1 takes dataset-1 as input.

3.1.5 Training data batches

As mentioned in Section 1.4, it is sometimes convenient to divide training data into several *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, 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 (e.g., `batch_id=1of5`). `prepText gen_regions` uses this parameter to name the region/target files appropriately.
3. Train a neural network 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_epochs=100` results in going through the entire data (all five batches) 20 times instead of 100, and for example, with `ss_decay_at=4`, the step size is reduced after going through about four fifth of the data.

3.1.6 Warm-start from a saved model

Training can be resumed by warm-starting from a network saved to a file. To save a model to a file, `save_fn` and `save_interval` (and optionally `save_after`) should be specified as described in Section 3.1.

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

The network architecture such as layer types and connection between layers as well as the loss function is 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. 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.

3.2 `reNet predict`: applying a trained network to new data

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

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

	`reNet predict` **(to apply a saved model to test data)**	
*	`model_fn=`	Path to the model file (input), saved during training. Note that the model filename is a concatenation of `save_fn` and `.epo`n`.ReNet` where n is the number of epochs, e.g., `model.epo100.ReNet` if `save_fn=model`.
*	`prediction_fn=`	Path to the prediction file (output).
*	`tstname=`	Test data name.
*	`datatype=sparse`	Dataset type.
	`dsno`i`=`	If there are multiple datasets, specify `dsno0`, `dsno1`,$\cdots$, in exactly the same way as training.
	`data_dir=`	Directory where data files are.
	`x_ext=`	Feature filename extension. This must be the same as training. (Default:`.xsmatbcvar`)
	`test_mini_batch_size=`	Mini-batch size for parallel processing for testing. (Default:100)
	`WriteText`	Write prediction values in the text format. (Default: non-text format)

3.3 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.

3.3.1 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: `reNet` *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.

Use of the device memory handler is particularly recommended with LSTM, as it often makes it two times faster. In other cases, the amount of speed up may be small.

Note that when device memory is very tight, use of the device memory handler may cause device memory shortage due to fragmentation, and the device memory handler should be disabled in that case.

Example 4. *The following example uses GPU#1 and the device memory handler with 4.5GB device memory.*

```
reNet 1:4.5 train···
```

3.3.2 To control computation on GPU

The following parameters may be useful.

To control GPU parallel processing in `reNettrain` **and** `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.

4 Using unsupervised embeddings

Recommendation Start with a provided shell script such as `examples/shcnn-unsup-imdb-elec.sh` (shallow CNNfor small training data) or `examples/dpcnn-with-*unsemb.sh` (DPCNN for large training data) and modify it for your data.

It was shown in [JZ15b, JZ16b, JZ16a, JZ17] that classification accuracy can be improved by using unsupervised text region embeddings (embeddings of text regions trained in an unsupervised manner) to produce additional features to shallow CNNs, LSTMs, and DPCNN.

Using CONTEXT, unsupervised embedding training is done in two steps:

1. **Creation of an artificial auxiliary task**: From unlabeled data, generate training data for the artificial task of predicting adjacent regions from regions. This is done by `prepText gen_regions_unsup`.

2. **Training with unlabeled data**: Train a neural network of one hidden layer using the training data above and save the hidden layer to a file, which will be used an unsupervised embedding function that produces additional features for a neural network. This is done by `reNet train`.

4.1 Creation of an artificial auxiliary task using unlabeled data

The following table describes parameters for `prepText gen_regions_unsup`.

Parameters for `prepText gen_regions_unsup`**.**		
*	`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.
*	`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=`	Format of X (features). `Bow` | `Seq`. (Default: `Bow`)

*	`region_fn_stem=`	Pathname stem of the feature file (X) and the target file (Y), and word-mapping file (output). To make the pathnames, the respective extensions will be attached. Note that both X and Y are sparse region vectors.
	`x_ext=`	Feature file extension. Use `.xsmatbcvar` for LSTM and use the default for others. (Default:`.xsmatbc`).
	`y_ext=`	Target file extension. Use `.ysmatbcvar` for LSTM and use the default for others. (Default:`.ysmatbc`).
*	`patch_size=`	Feature region size.
	`patch_stride=`	Feature region stride. (Default: 1)
	`padding=`	Feature padding size. (Default: 0)
*	`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.
	`RightOnly`	Use this for training a forward LSTM (left to right) so that the region to the right (future) of the current time step is regarded as a target region.
	`LeftOnly`	Use this for training a backward LSTM (right to left) so that the region to the left (past) of the current time step is regarded as a target region.
*	`y_vocab_fn=`	Path to the vocabulary file generated by `gen_vocab`, used for Y (target).

4.2 Training of unsupervised embedding

Unsupervised embedding training is done by `reNet train`. In addition to the parameters introduced earlier, the following parameters are required/useful.

Parameters for `reNet train`: useful (maybe only) for unsupervised 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.
`0save_layer_fn=`	Layer-0, which embodies the unsupervised embedding, is saved in this file.
`NoCusparseIndex`	Do not use the cusparse library for indexing rows of sparse matrices. In the typical unsupervised embedding training setting (as in the sample scripts), this speeds up training a lot.

Also note that `x_ext` and `y_ext` must be set correctly so that they point to the files produced by `prepText gen_regions_unsup`.

4.3 Training with labeled data and unsupervised embedding

As described in Section 1.2.1, the unsupervised embedding function trained above is integrated into a neural network for classification via a *side layer*. To specify parameters for a side layer, we attach a *side layer identifier* at the beginning of the parameters, in the form of

`0side`i`_`

which refers to the i-th side layer ($i = 0, 1, \cdots$) attached to layer-0.

Parameters for `reNet train` for controling side layers	
`num_sides=`	Number of side layers attached to layer-0. (Default: 0)
`UpdateSide`	Update weights in the side layers as training proceeds. Not recommended.
`0side`i`_layer_fn=`	The file that contains unsupervised embedding, generated via `0save_layer_fn` during unsupervised embedding training. The side layer warm-starts from this file.
`0side`i`_dsno=`	Dataset#. Explained in Section 3.1.4.

NOTE By default, side layer weights (i.e., unsupervised embeddings) are fixed during the training. This was done in [JZ15b, JZ16b, JZ16a, JZ17] and is recommended. To update them (fine-tuning), `UpdateSide` needs to be specified.

4.4 Integration of external word vectors

Use of pre-trained word vectors such as publicly available word vectors requires the following steps.

1. Define a vocabulary. This can be either the vocabulary of the word vectors or a vocabulary generated from data of the domain of interest.
2. Using the vocabulary file above, generate a region file with region size 1 by `prepText gen_regions` and word-mapping file, which defines the correspondence between the dimensions of region vectors and words.
3. Transform the pre-trained word vectors to a *weight file* which `reNet train` can use for initializing weights. This is done by `prepText adapt_word_vectors`, using the word-mapping file above to *align* the order of word vectors in the weight file to the order of the words in the region file.
4. To train a neural network, use the weight file above to initialize the weights of a word embedding layer via the [ℓ]`weight_fn` parameter. Use the region file above as input to the word embedding layer.

Parameters for `prepText adapt_word_vectors` to convert word vectors to a weight file.		
*	`wordvec_bin_fn`	Path to a binary file containing word vectors and words in the word2vec binary format (input). Either this or `wordvec_txt_fn` below is required. Not both.
*	`wordvec_txt_fn`	Path to a text file containing word vectors and words in the word2vec text format (input). Either this or `wordvec_bin_fn` above is required. Not both.
*	`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 one-hot vectors generated by `gen_regions`.
*	`weight_fn=`	Path to the weight file to be generated (output). It must end with `dmatc`.
	`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.
Parameters for a `Weight+` layer for initializing weights by a weight file.		
	[ℓ]`weight_fn`	Path to a weight file that contains initial weights.

5 Tips

5.1 Miscellaneous tips

Here are things to consider for obtaining good performance.

- **Training parameters** In addition to finding the right network configuration for the intended task, it is also important to find the right values for the following meta-parameters.
  - **Step-size (learning rate)** (`step_size`)
  - **Number of epochs** (`num_epochs`)
  - **Step-size scheduling**: In [JZ15a, JZ15b, JZ16b, JZ16a, JZ17], the step-size was reduced by multiplying 0.1 when training was 80% done. This can be done by setting `ss_scheduler=Few ss_decay=0.1` and setting `ss_decay_at=` to 80% of `num_epochs`.
  - **Optimization algorithm** (`optim`): In [JZ15a, JZ15b, JZ16b, JZ16a, JZ17], mini-batch SGD with momentum 0.9 was used for CNNs. For LSTM, rmsprop was also used in some cases. In our experience, use of rmsprop did not always lead to the best accuracy even though it converged fast.
  - **Early stopping**: If overfitting is observed, early stopping should be done based on the validation performance.
- **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.

5.2 GPU memory consumption during training

In a typical environment, GPU memory (*device memory*) is relatively small compared with CPU memory (*host memory*). When `reNet` 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 and therefore what parameters affect the consumption of GPU memory.

- **Weights**.
  A larger number of weight vectors (specified by `nodes`) consumes more GPU memory. A larger vocabulary consumes more GPU memory since the dimensionality of weight vectors in the region embedding layer depends on it. The sequential input with region size k requires k times more weights than the bow input. A very large number of target classes consumes a large amount of GPU memory since the number of weight vectors in the top layer is equal to the number of classes.
- **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.

However, consumption of GPU device memory does not depend on training data size or test data size.

5.3 CPU memory consumption during training

In contrast to GPU memory, consumption of CPU memory is affected by the size of training data and test data.

Training data By default, training data is read from the input files 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 3.1.5 for how to use batches.

Test data If optional testing is done, test data is entirely read into in the CPU memory.

`SaveDataMem` **option** When the `SaveDataMem` switch is turned on, test data is loaded into the CPU memory only when it is needed so that test data and training data do not occupy the CPU memory at the same time. This saves the CPU memory at the expense of file I/O overhead.

References

[JZ15a] Rie Johnson and Tong Zhang. Effective use of word order for text categorization with convolutional neural networks. In *Proceedings of NAACL HLT*, 2015.

[JZ15b] Rie Johnson and Tong Zhang. Semi-supervised convolutional neural networks for text categorization via region embedding. In *Proceedings of NIPS*, 2015.

[JZ16a] Rie Johnson and Tong Zhang. Convolutional neural networks for text categorization: Shallow word-level vs. deep character-level. *arXiv:1609.00718*, 2016.

[JZ16b] Rie Johnson and Tong Zhang. Supervised and semi-supervised text categorization using LSTM for region embeddings. In *Proceedings of ICML*, 2016.

[JZ17] Rie Johnson and Tong Zhang. Deep pyramid convolutional neural networks for text categorization. In *Proceedings of ACL*, 2017.

[SHK+14] 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(Jun):1929–1958, 2014.

[TH12] Tijman Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.

[Zei12] Matthew D. Zeiler. ADADELTA: An adaptive learning rate method. *arXiv:1212.5701*, 2012.