Moses statistical machine translation system

Moses is a statistical machine translation system that allows you to automatically train translation models for any language pair. All you need is a collection of translated texts (parallel corpus). Once you have a trained model, an efficient search algorithm quickly finds the highest probability translation among the exponential number of choices.

Features

- Moses offers two types of translation models: *phrase-based* and *tree-based*
- Moses features *factored translation models*, which enable the integration linguistic and other information at the word level
- Moses allows the decoding of *confusion networks* and *word lattices*, enabling easy integration with ambiguous upstream tools, such as automatic speech recognizers or morphological analyzers
- The *Experiment Management System* makes using Moses much easier.

The released software includes a command line executable which can used for decoding. The source code for the decoder, can be downloaded from github. Download the latest release [1] or the current snapshot from github.

The development of Moses is mainly supported by the European Union under the following projects:

- EuroMatrix and TC-STAR (Framework 6)
- EuroMatrixPlus, LetsMT, META-NET, MosesCore and MateCat (Framework 7)

It has received additional support from

- University of Edinburgh, Scotland
- Charles University, Prague, Czech Republic
- Fondazione Bruno Kessler, Trento, Italy
- RWTH Aachen, Germany
- University of Maryland, College Park, United States
- Massachusetts Institute of Technology, United States
- US funding agencies DARPA, NSF, and Department of Defence

Moses is licensed under the LGPL. [2]

The minimum software requirements are:

- Moses (obviously!)
- GIZA++, for word-aligning your parallel corpus
- IRSTLM, SRILM, OR KenLM for language model estimation.

KenLM is included in Moses and the default in the Moses tool-chain. IRSTLM and KenLM are LGPL licensed (like Moses) and therefore available for commercial use. [3]

GIZA++: Training of statistical translation models. GIZA++ is an extension of the program GIZA (part of the SMT toolkit EGYPT) which was developed by the Statistical Machine Translation team during the summer workshop in 1999 at the Center for Language and Speech Processing at Johns-Hopkins University (CLSP/JHU). GIZA++ includes a lot of additional features. The extensions of GIZA++ were designed and written by Franz Josef Och. [4]

IRSTLM. The IRST Language Modeling (IRSTLM) Toolkit features algorithms and data structures suitable to estimate, store, and access very large n-gram language models. This software has been integrated into a popular open source Statistical Machine Translation decoder called Moses, and is compatible with language models created with other tools, such as the SRILM Tooolkit. [5]

SRILM. The SRI Language Modeling (SRILM) toolkit offers tools for building and applying statistical language models for use in speech recognition, statistical tagging and segmentation, and machine translation. The SRILM toolkit is used in the Moses SMT system for Language modeling support. [6]

KenLM estimates unpruned language models with modified Kneser-Ney smoothing. The builder is disk-based: you specify the amount of RAM to use and it performs disk-based merge sort when necessary. It's faster than SRILM and IRSTLM and scales to much larger models. [7]

Corpus Preparation

To train a translation system we need parallel data (text translated into two different languages) which is aligned at the sentence level.

To prepare the data for training the translation system, we have to perform the following steps:

- *tokenisation*: This means that spaces have to be inserted between (e.g.) words and punctuation.
- *truecasing*: The initial words in each sentence are converted to their most probable casing. This helps reduce data sparsity.
- *cleaning*: Long sentences and empty sentences are removed as they can cause problems with the training pipeline, and obviously mis-aligned sentences are removed.

Language Model Training

The language model (LM) is used to ensure fluent output, so it is built with the target language.

For faster loading need binarise the input file using KenLM. Note that can also use IRSTLM which also has a binary format that Moses supports.

Training the Translation System

To do training the translation model, need run word-alignment (using GIZA++), phrase extraction and scoring, create lexicalised reordering tables and create your Moses configuration file, all with a single command.

If you have a multi-core machine it's worth using the -cores argument to encourage as much parallelisation as possible.

This took about 1.5 hours using 2 cores on a powerful laptop (Intel i7-2640M, 8GB RAM, SSD). Once it's finished there should be a moses.ini file in the directory ~/working/train/model. You can use the model specified by this ini file to decode (i.e. translate), but there's a couple of problems with it. The first is that it's very slow to load, but we can fix that by binarising the phrase table and reordering table, i.e. compiling them into a format that can be load quickly. The second problem is that the weights used by Moses to weight the different models against each other are not optimised - if you look at the moses.ini file you'll see that they're set to default values like 0.2, 0.3 etc. To find better weights we need to tune the translation system, which leads us on to the next step...

Tuning

This is the slowest part of the process. Tuning requires a small amount of parallel data, separate from the training data. Launch the tuning process, the end result of tuning is an ini file with trained weights, which should be in ~/working/mert- work/moses.ini.

Testing

Now we can run Moses and get our translation. At this stage, your probably wondering how good the translation system is. To measure this, we use another parallel data set (the test set) distinct from the ones we've used so far.

The model that we've trained can then be filtered for this test set, meaning that we only retain the entries needed translate the test set. This will make the translation a lot faster.

You can test the decoder by first translating the test set (takes a wee while) then running the BLEU script on it. [2]

The phrase-based decoder in Moses, with using a simple model

A Simple Translation Model

Let us begin with a look at the toy phrase-based translation model that is available for download at http://www.statmt.org/moses/download/sample-models.tgz. Unpack the tar ball and enter the directory sample-models/phrase-model.

The model consists of two files:

- phrase-table the phrase translation table, and
- moses.ini the configuration file for the decoder.

Let us look at the first line of the phrase translation table (file phrase-table).

The translation tables are the main knowledge source for the machine translation decoder. The decoder consults these tables to figure out how to translate input in one language into output in another language.

Being a phrase translation model, the translation tables do not only contain single word entries, but multi-word entries. These are called phrases, but this concept means nothing more than an arbitrary sequence of words, with no sophisticated linguistic motivation.

Next - Running the Decoder

We run the decoder.

Trace

There are two switches that force the decoder to reveal more about its inner workings: -report-segmentation and -verbose.

The trace option reveals which phrase translations were used in the best translation found by the decoder.

Verbose

Now for the next switch, -verbose (short -v), that displays additional run time information. The verbosity of the decoder output exists in three levels. The default is 1. Moving on to -v 2 gives additional statistics for each translated sentences.

The stack sizes after each iteration of the stack decoder. An iteration is the processing of all hypotheses on one stack: After the first iteration (processing the initial empty hypothesis), 10 hypothesis that cover one German word are placed on stack 1, and 2 hypotheses that cover two foreign words are placed on stack 2. Note how this relates to the 12 translation options.

During the beam search a large number of hypotheses are generated (453). Many are discarded early because they are deemed to be too bad (272), or pruned at some later stage (0), and some are recombined (69). The remainder survives on the stacks.

The most verbose output -v 3 provides even more information.

Before decoding, the phrase translation table is consulted for possible phrase translations. For some phrases, we find entries, for others we find nothing.

The pair of numbers next to a phrase is the coverage, pC denotes the log of the phrase translation probability, after c the future cost estimate for the phrase is given.

Future cost is an estimate of how hard it is to translate different parts of the sentence.

Tuning for Quality

The key to good translation performance is having a good phrase translation table. But some tuning can be done with the decoder. The most important is the tuning of the model parameters.

The probability cost that is assigned to a translation is a product of probability costs of four models:

- phrase translation table,
- language model,
- reordering model, and
- word penalty.

Each of these models contributes information over one aspect of the characteristics of a good translation:

- The phrase translation table ensures that the English phrases and the German phrases are good translations of each other.
- The language model ensures that the output is fluent English.
- The distortion model allows for reordering of the input sentence, but at a cost: The more reordering, the more expensive is the translation.
- The word penalty ensures that the translations do not get too long or too short.

Each of the components can be given a weight that sets its importance. Mathematically, the cost of translation is:

p(e|f) = phi(f|e)^weight_phi * LM(e)^weight_lm * D(e,f)^weight_d * W(e)^weight_w

The probability p(e|f) of the English translation e given the foreign input f is broken up into four models, phrase translation phi(f|e), language model LM(e), distortion model D(e,f), and word penalty W(e) = exp(length(e)). Each of the four models is weighted by a weight.

The weighting is provided to the decoder with the four parameters *weight-t, weight-l, weight-d,* and *weight-w*. The default setting for these weights is 1, 1, 1, and 0. These are also the values in the configuration file moses.ini.

Setting these weights to the right values can improve translation quality.

What is the right weight setting depends on the corpus and the language pair. Ususally, a held out development set is used to optimize the parameter settings. The simplest method here is to try out with a large number of possible settings, and pick what works best. Good values for the weights for phrase translation table (weight-t, short tm), language model (weight-l, short lm), and reordering model (weight-d, short d) are 0.1-1, good values for the word penalty (weight-w, short w) are -3-3. Negative values for the word penalty favor longer output, positive values favor shorter output.

Tuning for Speed

Let look at some additional parameters that help to speed up the decoder. Unfortunately higher speed usually comes at cost of translation quality. The speed-ups are achieved by limiting the search space of the decoder. By cutting out part of the search space, we may not be able to find the best translation anymore.

Translation Table Size

One strategy to limit the search space is by reducing the number of translation options used for each input phrase, i.e. the number of phrase translation table entries that are retrieved. While in the toy example, the translation tables are very small, these can have thousands of entries per phrase in a realistic scenario. If the phrase translation table is learned from real data, it contains a lot of noise. So, we are really interested only in the most probable ones and would like to elimiate the others.

The are two ways to limit the translation table size: by a fixed limit on how many translation options are retrieved for each input phrase, and by a probability threshold, that specifies that the phrase translation probability has to be above some value.

Hypothesis Stack Size (Beam)

A different way to reduce the search is to reduce the size of hypothesis stacks. For each number of foreign words translated, the decoder keeps a stack of the best (partial) translations. By

reducing this stack size the search will be quicker, since less hypotheses are kept at each stage, and therefore less hypotheses are generated.

From a user perspective, search speed is linear to the maximum stack size.

Note that the number of hypothesis entered on stacks is getting smaller with the stack size.

As we have previously described with translation table pruning, we may also want to use the relative scores of hypothesis for pruning instead of a fixed limit. The two strategies are also called histogram pruning and threshold pruning.

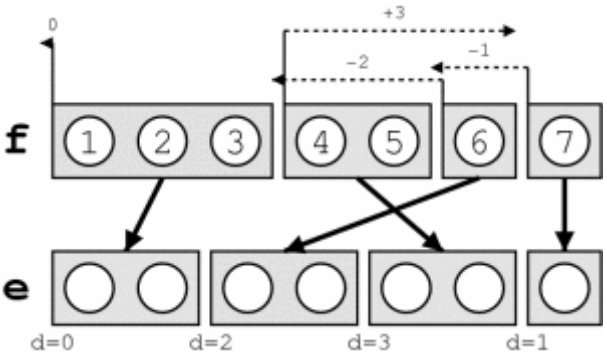
With small stack sizes or small thresholds we risk search errors, meaning the generation of translations that score worse than the best translation according to the model.

Limit on Distortion (Reordering)

The basic reordering model implemented in the decoder is fairly weak. Reordering cost is measured by the number of words skipped when foreign phrases are picked out of order.

Total reordering cost is computed by $D(e,f) = -\Sigma_i (d\_i)$ where d for each phrase i is defined as d = abs(last word position of previously translated phrase + 1 - first word position of newly translated phrase).

This is illustrated by the following graph:



This reordering model is suitable for local reorderings: they are discouraged, but may occur with sufficient support from the language model. But large-scale reorderings are often arbitrary and effect translation performance negatively.

By limiting reordering, we can not only speed up the decoder, often translation performance is increased. Reordering can be limited to a maximum number of words skipped (maximum d) with the switch -distortion-limit, or short -dl.

Setting this parameter to 0 means monotone translation (no reordering). If you want to allow unlimited reordering, use the value -1. [8]

Moses supports models that have become known as hierarchical phrase-based models and syntax-based models. These models use a grammar consisting of SCFG (Synchronous Context-Free Grammar) rules. In the following, we refer to these models as tree-based models.

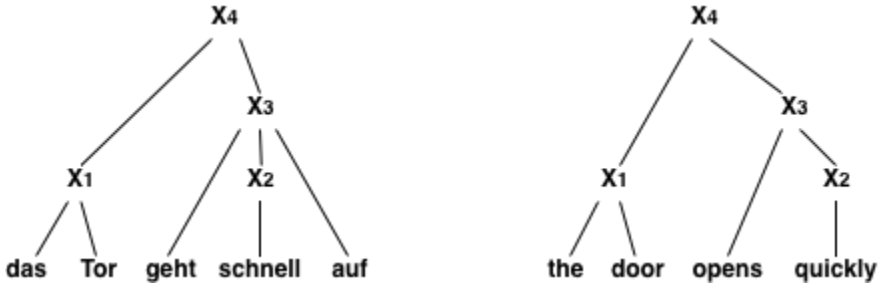
Tree-Based Models

Traditional phrase-based models have as atomic translation step the mapping of an input phrase to an output phrase. Tree-based models operate on so-called grammar rules, which include variables in the mapping rules.

The variables in these grammar rules are called non-terminals, since their occurrence indicates that the process has not yet terminated to produce the final words (the terminals). Besides a generic non-terminal X, linguistically motivated non-terminals such as NP (noun phrase) or VP (verb phrase) may be used as well in a grammar (or translation rule set).

We call these models tree-based, because during the translation a data structure is created that is a called a tree.

When applying these rules in the given order, we produce the translation The door opens quickly in the following fashion:



First the simple phrase mappings (1) Das Tor to The door and (2) schnell to quickly are carried out. This allows for the application of the more complex rule (3) geht X_1 auf to opens X_1. Note that at this point, the non-terminal X, which covers the input span over schnell is replaced by a known translation quickly. Finally, the glue rule (4) X_1 X_2 to X_1 X_2 combines the two fragments into a complete sentence.

Formally, such context-free grammars are more constraint than the formalism for phrase-based models. In practice, however, phrase-based models use a reordering limit, which leads to linear decoding time. For tree-based models, decoding is not linear with respect to sentence length, unless reordering limits are used.

Current research in tree-based models has the expectation to build translation models that more closely model the underlying linguistic structure of language, and its essential element: recursion. This is an active field of research.

Chart Decoding

Phrase-Based decoding generates a sentence from left to right, by adding phrases to the end of a partial translation. Tree-based decoding builds a chart, which consists of partial translation for all possible spans over the input sentence.

Currently Moses implements a CKY+ algorithm for arbitrary number of non-terminals per rule and an arbitrary number of types of non-terminals in the grammar.

Decoder Parameters

The most important consideration in decoding is a speed/quality trade-off. If you want to win competitions, you want the best quality possible, even if it takes a week to translate 2000 sentences. If you want to provide an online service, you know that users get impatient, when they have to wait more than a second.

Beam Settings

The chart decoder has an implementation of CKY decoding using cube pruning. The latter means that only a fixed number of hypotheses are generated for each span. This number can be changed with the option cube-pruning-pop-limit (or short cbp). The default is 1000, higher numbers slow down the decoder, but may result in better quality.

Another setting that directly affects speed is the number of rules that are considered for each input left hand side. It can be set with ttable-limit.

Limiting Reordering

The number of spans that are filled during chart decoding is quadratic with respect to sentence length. But it gets worse. The number of spans that are combined into a span grows linear with sentence length for binary rules, quadratic for trinary rules, and so on. In short, long sentences become a problem. A drastic solution is the size of internal spans to a maximum number.

This sounds a bit extreme, but does make some sense for non-syntactic models. Reordering is limited in phrase-based models, and non-syntactic tree-based models (better known as

hierarchical phrase-based models) and should limit reordering for the same reason: they are just not very good at long-distance reordering anyway.

The limit on span sizes can be set with max-chart-span. In fact its default is 10, which is not a useful setting for syntax models.

Handling Unknown Words

In a target-syntax model, unknown words that just copied verbatim into the output need to get a non-terminal label. In practice unknown words tend to be open class words, most likely names, nouns, or numbers. With the option unknown-lhs you can specify a file that contains pairs of non-terminal labels and their probability per line.

Optionally, we can also model the choice of non-terminal for unknown words through sparse features, and optimize their cost through MIRA or PRO. This is implemented by relaxing the label matching constraint during decoding to allow soft matches, and allowing unknown words to expand to any non-terminal.

Technical Settings

The parameter non-terminals is used to specify privileged non-terminals. These are used for unknown words (unless there is a unknown word label file) and to define the non-terminal label on the input side, when this is not specified.

Typically, we want to consider all possible rules that apply. However, with a large maximum phrase length, too many rule tables and no rule table limit, this may explode. The number of rules considered can be limited with rule-limit. Default is 5000.

Training Parameters

There are a number of additional decisions about the type of rules you may want to include in your model. This is typically a size / quality trade-off: Allowing more rule types increases the size of the rule table, but lead to better results. Bigger rule tables have a negative impact on memory use and speed of the decoder.

There are two parts to create a rule table: the extraction of rules and the scoring of rules. The first can be modified with the parameter --extract-options="..." of train-model.perl. The second with --score-options="...".

Here are the extract options:

- --OnlyDirect: Only creates a model with direct conditional probabilities *p(f|e)* instead of the default direct and indirect (*p(f|e)* and *p(e|f)*).
- --MaxSpan SIZE: maximum span size of the rule. Default is 15.
- --MaxSymbolsSource SIZE and --MaxSymbolsTarget SIZE: While a rule may be extracted from a large span, much of it may be knocked out by sub-phrases that are substituted by non-terminals. So, fewer actual symbols (non-terminals and words remain). The default maximum number of symbols is 5 for the source side, and practically unlimited (999) for the target side.
- --MinWords SIZE: minimum number of words in a rule. Default is 1, meaning that each rule has to have at least one word in it. If you want to allow non-lexical rules set this to zero. You will not want to do this for hierarchical models.
- --AllowOnlyUnalignedWords: This is related to the above. A rule may have words in it, but these may be unaligned words that are not connected. By default, at least one aligned word is required. Using this option, this requirement is dropped.
- --MaxNonTerm SIZE: the number of non-terminals on the right hand side of the rule. This has an effect on the arity of rules, in terms of non-terminals. Default is to generate only binary rules, so the setting is 2.
- --MinHoleSource SIZE and --MinHoleTarget SIZE: When sub-phrases are replaced by non-terminals, we may require a minimum size for these sub-phrases. The default is 2 on the source side and 1 (no limit) on the target side.

- --DisallowNonTermConsecTarget and --NonTermConsecSource. We may want to restrict if there can be neighboring non-terminals in rules. In hierarchical models there is a bad effect on decoding to allow neighboring non-terminals on the source side. The default is to disallow this -- it is allowed on the target side. These switches override the defaults.
- --NoFractionalCounting: For any given source span, any number of rules can be generated. By default, fractional counts are assigned, so probability of these rules adds up to one. This option leads to the count of one for each rule.
- --NoNonTermFirstWord: Disallows that a rule starts with a non-terminal.

Once rules are collected, the file of rules and their counts have to be converted into a probabilistic model. This is called rule scoring, and there are also some additional options:

- --OnlyDirect: only estimates direct conditional probabilities. Note that this option needs to be specified for both rule extraction and rule scoring.
- --NoLex: only includes rule-level conditional probabilities, not lexical scores.
- --GoodTuring: Uses Good Turing discounting to reduce actual accounts. This is a good thing, use it.

Training Syntax Models

Training hierarchical phrase models, i.e., tree-based models without syntactic annotation, is pretty straight-forward. Adding syntactic labels to rules, either on the source side or the target side, is not much more complex. The main hurdle is to get the annotation. This requires a syntactic parser.

Syntactic annotation is provided by annotating all the training data (input or output side, or both) with syntactic labels. The format that is used for this uses XML markup.

After annotating the training data with syntactic information, you can simply run train-model.perl as before, except that the switches --source-syntax or --target-syntax (or both) have to be set.

Annotation Wrappers

To obtain the syntactic annotation, you will likely use a third-party parser, which has its own idiosyncratic input and output format. You will need to write a wrapper script that converts it into the Moses format for syntax trees.

Here provide wrappers (in scripts/training/wrapper) for the following parsers.

- Bitpar is available from the web site of the University of Munich. The wrapper is parse-de-bitpar.perl
- Collins parser is availble from MIT. The wrapper is parse-en-collins.perl

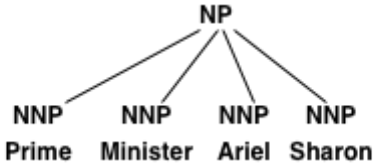
Relaxing Parses

The use of syntactic annotation puts severe constraints on the number of rules that can be extracted, since each non-terminal has to correspond to an actual non-terminal in the syntax tree.

Recent research has proposed a number of relaxations of this constraint. The program relax-parse (in training/phrase-extract) implements two kinds of parse relaxations: binarization and a method proposed under the label of syntax-augmented machine translation (SAMT) by Zollmann and Venugopal.

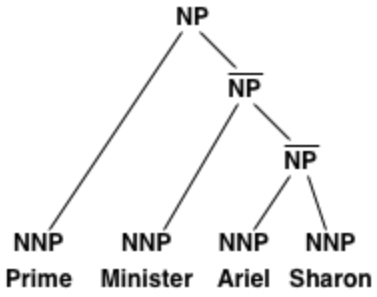
Readers familiar with the concept of binarizing grammars in parsing, be warned: We are talking here about modifying parse trees, which changes the power of the extracted grammar, not binarization as a optimization step during decoding.

The idea is the following: If the training data contains a subtree such as



then it is not possible to extract translation rules for Ariel Sharon without additional syntactic context. Recall that each rule has to match a syntactic constituent.

The idea of relaxing the parse trees is to add additional internal nodes that makes the extraction of additional rules possible. For instance left-binarization adds two additional nodes and converts the subtree into:



The additional node with the label `^NP` allows for the straight-forward extraction of a translation rule (of course, unless the word alignment does not provide a consistent alignment).

The program `relax-parse` allows the following tree transformations:

- --LeftBinarize and --RightBinarize: Adds internal nodes as in the example above. Right-binarization creates a right-branching tree.
- --SAMT 1: Combines pairs of neighboring children nodes into tags, such as DET+ADJ. Also nodes for everything except the first child (NP\\DET) and everything except the last child (NP/NN) are added.
- --SAMT 2: Combines any pairs of neighboring nodes, not only children nodes, e.g., VP+DET.
- --SAMT 3: not implemented.
- --SAMT 4: As above, but in addition each previously unlabeled node is labeled as FAIL, so no syntactic constraint on grammar constraint remains.

Note that you can also use both --LeftBinarize and --RightBinarize. Note that in this case, as with all the SAMT relaxations, the resulting annotation is not any more a tree, since there is not a single set of rule applications that generates the structure (now called a forest).

On-Disk Rule Table

The rule table may become too big to fit into the RAM of the machine. Instead of loading the rules into memory, it is also possible to leave the rule table on disk, and retrieve rules on demand. [9]

Optimizing Moses

Multi-threaded Moses

Moses supports multi-threaded operation, enabling faster decoding on multi-core machines. The current limitations of multi-threaded Moses are:

1. irstlm is not supported, since it uses a non-threadsafe cache
2. lattice input may not work - this has not been tested
3. increasing the verbosity of Moses will probably cause multi-threaded Moses to crash
4. Decoding speed will flatten out after about 16 threads. For more scalable speed with many threads, use Moses2

How much memory do I need during decoding?

The single-most important thing you need to run Moses fast is MEMORY. Lots of MEMORY. (For example, the Edinburgh group have servers with 144GB of RAM). The rest of this section is just details of how to make the training and decoding run fast.

Calculate total file size of the binary phrase tables, binary language models and binary reordering models.

For example, The total size of these files is approx. 31GB. Therefore, a translation system using these models requires 31GB (+ roughly 500MB) of memory to run fast.

Faster Training

Parallel training

When word aligning, using mgiza with multiple threads significantly speed up word alignment.

Parallel Extraction

Once word alignment is completed, the phrase table is created from the aligned parallel corpus. There are 2 main ways to speed up this part of the training process.

Firstly, the training corpus and alignment can be split and phrase pairs from each part can be extracted simultaneously. This can be done by simply using the argument *-cores*.

Secondly, the Unix *sort* command is often executed during training. It is essential to optimize this command to make use of the available disk and CPU.

Language Model

Convert your language model to binary format. This reduces loading time and provides more control.

Loading on-demand

By default, language models fully load into memory at the beginning. If you are short on memory, you can use on-demand language model loading. The language model must be converted to binary format in advance and should be placed on LOCAL DISK, preferably SSD. For KenLM, you should use the trie data structure, not the probing data structure.

Suffix array

Suffix arrays store the entire parallel corpora and word alignment information in memory, instead of the phrase table. The parallel corpora and alignment file is often much smaller than the phrase table.

Therefore, it is more memory efficient to store the corpus in memory, rather than the entire phrase-table. This is usually structured as a suffix array to enable fast extraction of translations.

Translations are extracted as needed, usually per input test set, or per input sentence.

Moses support two different implementations of suffix arrays.

Cube Pruning

Cube pruning limits the number of hypotheses created for each stack (or chart cell in chart decoding). It is essential for chart decoding (otherwise decoding will take a VERY long time) and an option in phrase-based decoding.

Minimizing memory during training

TODO: MGIZA with reduced memory sntcoc

Minimizing memory during decoding

The biggest consumer of memory during decoding are typically the models. Here are some links on how to reduce the size of each.

- Language model
- Translation model
- Reordering model.

Phrase-table types

Moses has multiple phrase table implementations. The one that suits you best depends on the model you're using (phrase-based or hierarchical/syntax), and how much memory your server has.

Here is a complete list of the types:

- ***Memory*** - this read in the phrase table into memory. For phrase-based model and chart decoding. Note that this is much faster than Binary and OnDisk phrase table format, but it uses a lot of RAM.
- ***Binary*** - a phrase table is converted into a 'database'. Only the translations which are required are loaded into memory. Therefore, requiring less memory, but potentially slower to run. For phrase-based model
- ***OnDisk*** - reimplementation of Binary for chart decoding.
- ***SuffixArray*** - stores the parallel training data and word alignment in memory, instead of the phrase table. Extraction is done on the fly. Also have a feature where you can add parallel data while the decoder is running ('Dynamic Suffix Array'). For Phrase-based models.
- ***ALSuffixArray*** - Suffix array for hierarchical models.
- ***FuzzyMatch*** - Implementation of Koehn and Senellart (2010).
- ***Hiero*** - like SCFG, but translation rules are in standard Hiero-style format
- ***Compact*** - for phrase-based model. [10]

References

1. moses-smt/mosesdecoder. URL: https://github.com/moses-smt/mosesdecoder/tree/RELEASE-2.1.1
2. Welcome to Moses! URL: http://statmt.org/moses/
3. Baseline System. URL: http://www.statmt.org/moses/?n=Moses.Baseline
4. GIZA++. URL: http://www.statmt.org/moses/giza/GIZA++.html
5. IRSTLM. URL: https://hlt-mt.fbk.eu/technologies/irstlm
6. A. Stolcke. SRILM – an extensible language model-ing toolkit. In Proceedings International Conferenceon Spoken Language Processing, Denver, US, 2002, pp. 901–904.
7. Estimating Large Language Models with KenLM. URL: https://kheafield.com/code/kenlm/estimation/
8. Phrase-based Tutorial. URL: http://www.statmt.org/moses/?n=Moses.Tutorial
9. Syntax Tutorial. URL: http://www.statmt.org/moses/?n=Moses.SyntaxTutorial
10. Optimizing Moses. URL: http://www.statmt.org/moses/?n=Moses.Optimize