Unimproved MT tuning and decoding

Jonathan Graehl

Better line search directions for MERT

- MERT over forests picks optimal corpus-BLEU weights given a starting point and a search direction (if the forests are pruned, periodic retranslating and merging is necessary)
- With many features, you need to try many directions. Usually: orthogonal (vary only one feature) and a handful of random.
- Idea: pick better search directions

A better line search direction

- Use per-sentence best (anti-)oracle+model score translation (like MIRA hope+fear).
- The weight vector that gives the maximum separation between a bad translation and a good translation is just (good features-bad features). Displacing the current weight vector by that will increase the separation.
- Both model->hope and fear->hope directions seem reasonable (the direction is just the difference in the 1-best feature vectors). We try both (generally we try many directions and make a greedy choice for the best BLEU).
- It takes time to compute (anti-)oracles, so we randomly select small batches of sentences and average the feature differences in each batch to generate a target weight vector.

Objections?

- Why are we performing an expensive exact infinite-line search using a heuristic direction?
- Maybe computing an expected-BLEU gradient direction would be more reasonable.
- MERT has no smoothing. When using large feature sets, we should add regularization.
- We need to still include random or orthogonal directions in case sparse features aren't (yet) represented in model/hope/fear.

Does it work?

No.

(tuning) oracle BLEU weight	0 (baseline)	0.1	1	10	100
test BLEU	22.01	22.13	22.06	21.85	22.08

Tuning a urdu Hiero system with 10 dense features, there was no improvement. Convergence also wasn't any faster.

Orthogonal directions and random directions are used in all cases.

Since the preference for high model score vs. good or bad BLEU score is scale-dependent, I tried a wide range of weights for the contribution of the loss to the fear/hope objective.

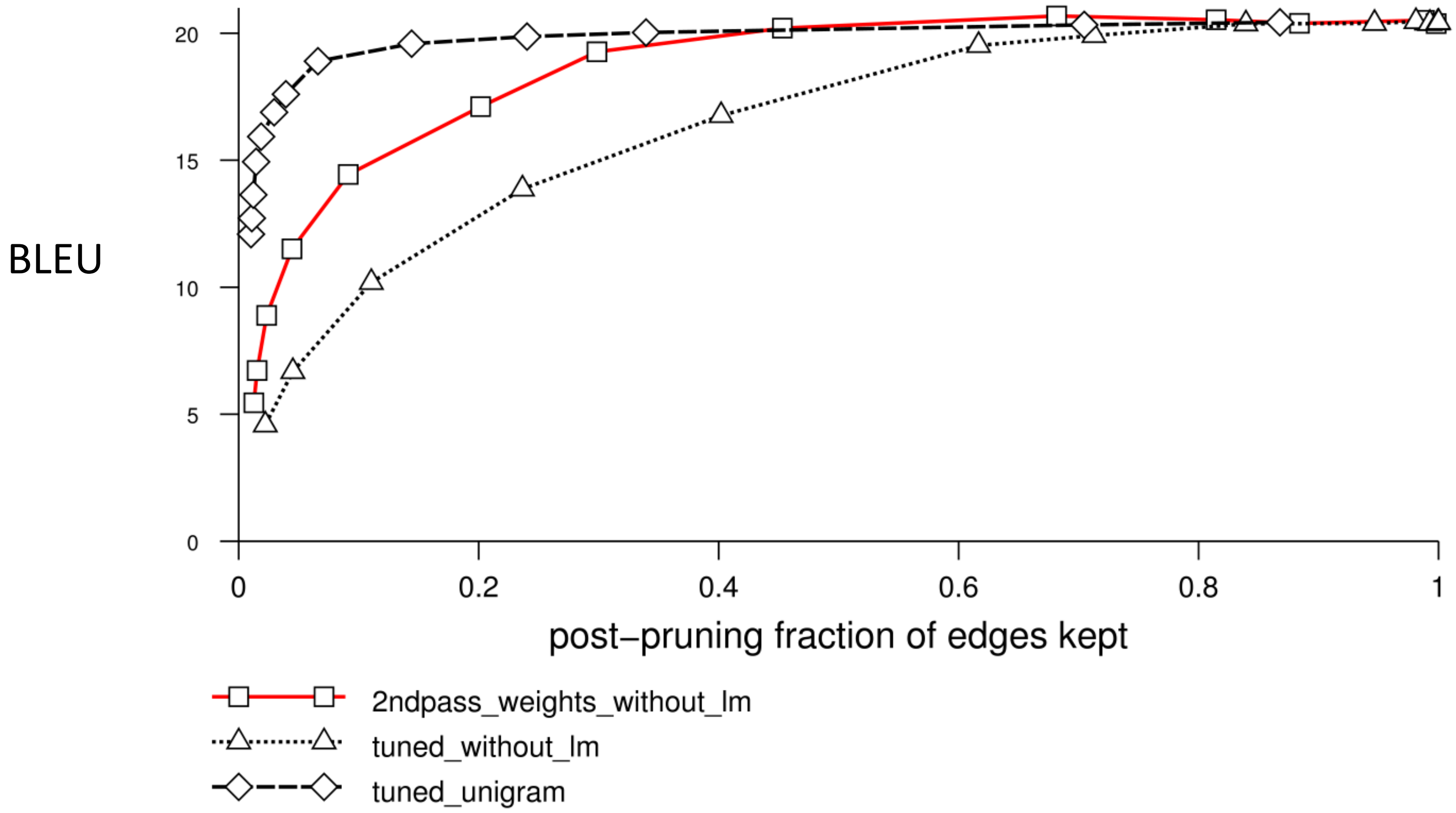
Still possibly worth trying: more, sparser features. Different direction-averaging oracle batch sizes. More batches. Regularization.

Possible confound: I didn't implement the hope/fear decoding myself; however, it's been verified to at least partially work: it yields kbests with better (hope) or worse (fear) BLEU.

Faster LM rescoring of TM forests

We don't need the whole TM forest to get the best translation using the ngram LM (inside-outside pruning):

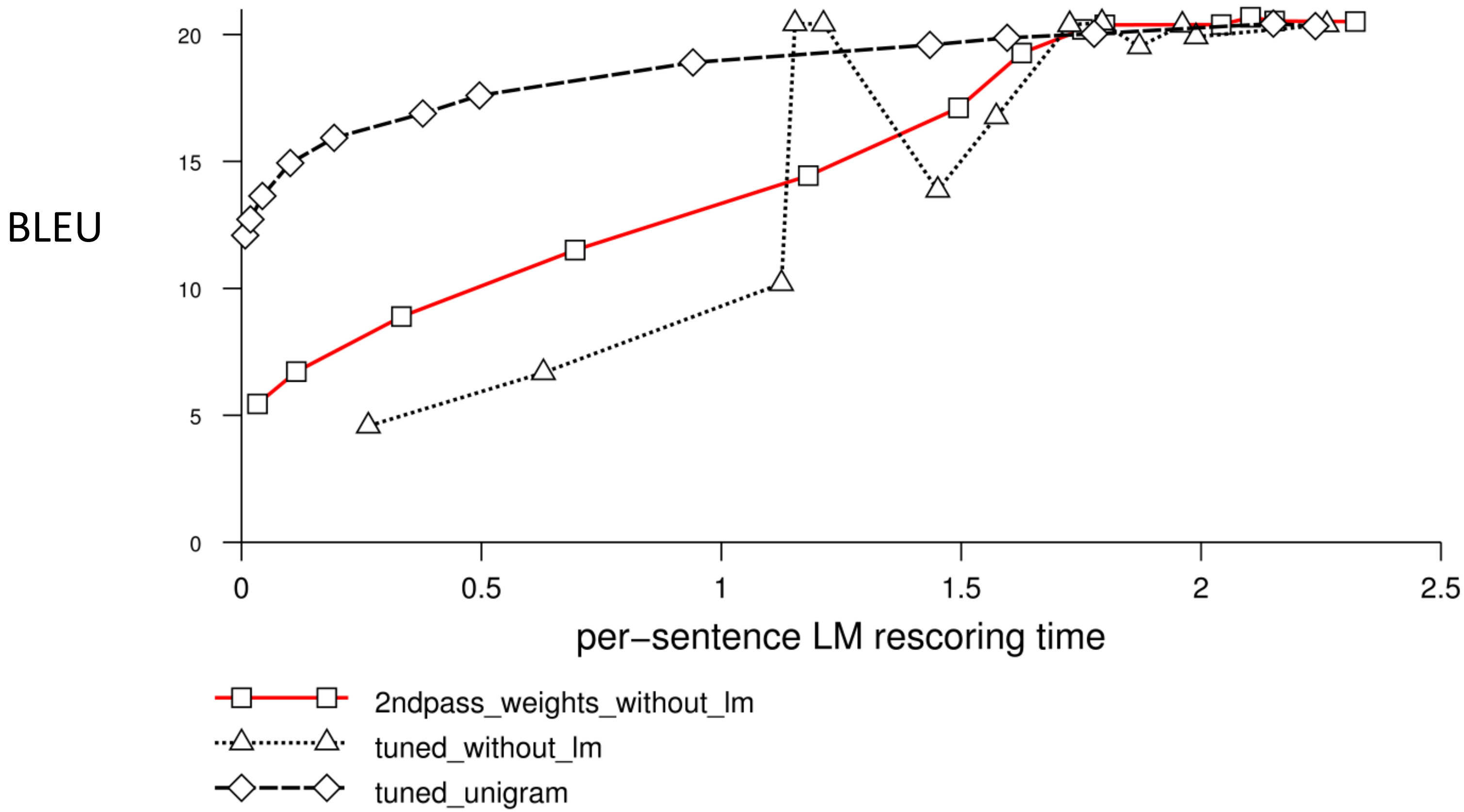
In multipass (TM –> TM+3gram), tuning for 1st pass –> BLEU (y) lost to pruning (x).



Is it faster?

Barely (if you want a good translation):

In multipass (TM –> TM+3gram), tuning for 1st pass –> BLEU (y) lost to pruning (x).



Soft pruning

- We lose too much good stuff by using an inside-outside global beam that removes large portions of the LM-unscored forest
- Coarse-to-fine using lower order ngrams or fewer bits per word (parts of speech or other classes) may work (but not as well as [Petrov 08] claimed)
- Idea: spend more time (more items evaluated in cube pruning) on nodes with good –LM inside-outside viterbi scores, and less on worse-first-pass nodes (soft pruning, rather than 0% or 100% only).
- Baseline: N[i] = 200 if v[i]>threshold (per-word), 0 otherwise.
- Promise: N[i] ~ v[i]^β (normalized so average N[i]=200 or whatever)
- (N[i] is the number of cube pruning LM descendants explored for item i)
- (v[i] is the viterbi probability for item i: e^(λ•f) where f is the feature vector of the best derivation using i.)

Does it work?

- No.

β	0 (baseline)	0.01	0.1	10
BLEU	20.21	20.16	20.11	19.7
Ngram rescoring time (avg)	1.8s	1.9s	2.0s	1.1s

(Hiero Urdu 3gram)
Will it help with smaller baseline N[i] than 200?
Will it help with syntactic categories with/without per-span limits?

Thank you.

Appendix: FSA target string models

```
struct SameFirstLetter : public FsaFeatureFunctionBase<SameFirstLetter> {
  SameFirstLetter(std::string const& param) :
    FsaFeatureFunctionBase<SameFirstLetter>(1,singleton_sentence("END"))
                // 1 byte of state, scan final (single) symbol "END" to get final state cost
  {
    start[0]='a'; h_start[0]=0; Init();
  }
  int markov_order() const { return 1; }
  Featval Scan1(WordID w, void const* old_state, void *new_state) const {
    char cw=TD::Convert(w)[0];
    char co=*(char const*)old_state;
    *(char *)new_state = cw;
    return cw==co?1:0;
  }
  void print_state(std::ostream &o, void const* st) const {
    o<<*(char const*)st;
  }
  static std::string usage(bool param,bool verbose) {
    return FeatureFunction::usage_helper("SameFirstLetter",
                              "[no args]",
                              "1 each time 2 consecutive words start with the same letter",
                              param,verbose);
  }
};
global_ff_registry->Register(new FFactory<FeatureFunctionFromFsa<SameFirstLetter> >);
// creates the usual bottom-up forest rescoring state with unscored left words, right state
    from scored words.
```

(typed fixed length state, e.g. int)

```
struct ShorterThanPrev : FsaTypedBase<int,ShorterThanPrev> {
  ShorterThanPrev(std::string const& param)
    : FsaTypedBase<int,ShorterThanPrev>(-1,4,singleton_sentence(TD::se)) //
   start, h_start, end_phrase
    // h_start estimate state: anything <4 chars is usually shorter than previous
  { Init(); }
  static std::string usage(bool param,bool verbose) {
    return FeatureFunction::usage_helper(
      "ShorterThanPrev",
      "",
      "stupid example stateful (bigram) feature: 1 per target word that's shorter
   than the previous word (end of sentence considered '</s>')",
      param,verbose);
  }
  static inline int wordlen(WordID w) {
    return std::strlen(TD::Convert(w));
  }
  Featval ScanT1(SentenceMetadata const& /* smeta */,const Hypergraph::Edge& /*
   edge */,WordID w,int prevlen,int &len) const {
    len=wordlen(w);
    return (len<prevlen) ? 1 : 0;
  }
};
```

Ngram language model

```
template <class Accum>
void ScanAccum(SentenceMetadata const& /* smeta */,Hypergraph::Edge const& e
   ,WordID w, void const* old_st, void *new_st, Accum *a) const
{
    if (!ctxlen_) {
      Add(floored(pimpl_->WordProb(w,&empty_context)),a);
    } else {
      WordID ctx[ngram_order_];
      state_copy(ctx,old_st);
      ctx[ctxlen_]=TD::none;
      Featval p=floored(pimpl_->WordProb(w,ctx));
    FSALMDBG(e,"p("<<TD::Convert(w)<<"|"<<TD::Convert(ctx,ctx+ctxlen_)<<")="<<p);FSALMDBGnl(e);
      // states are srilm contexts so are in reverse order (most recent word is first, then 1-
    back comes next, etc.).
      WordID *nst=(WordID *)new_st;
      nst[0]=w; // new most recent word
      to_state(nst+1,ctx,ctxlen_-1); // rotate old words right
    #if LM_FSA_SHORTEN_CONTEXT
      p+=pimpl_->ShortenContext(nst,ctxlen_);
    #endif
      Add(p,a);
    }
}
```

Accum is a template so you can equally support feature vectors as a single features. You can implement: scan a whole sequence of words at once, possibly exceeding declared markov order (using higher order ngram scores along the way).