Evaluating Tensorflow

Reid Pryzant[*]
Department of Computer Science
Stanford University
`rpryzant@stanford.edu`

Abstract

Easy-to-use software lets researchers build translation models without full control or knowledge of their inner workings. This constrains the diversity and effectiveness of modern machine translation systems. We work to alleviate this problems by thoroughly examining the efficacy of Tensorflow's machine translation abstractions. We observe that our novel dataset is too noisy to be useful, and that Tensorflow's abstractions do not work as well as advertised.

1 Introduction

Until recently, all commercial translation systems used finely articulated phrase-based translation algorithms. These systems learned language-specific features and transformations, and were highly adapted for particular language pairs, sometimes to an unrecognizable extent.

Modern deep learning techniques are sometimes advertised as a solution to this hyper-adaptation; they provide a common end-to-end architecture that can be applied to arbitrary translation problems. Ease of implementation is a force behind the rise of NMT systems . Software packages like Tensorflow, pytorch, Keras, etc. provide useful low- and high-level abstractions and primitives for potential deep learning. But there are unforseen pitfalls to taking these abstractions at face value; they might not work as well as advertised.

With this project we hope to informing the broader community as to the quality and reliability of Tensorflow abstractions by conducting a comparative investigation in neural machine translation.

2 Neural Machine Translation

In the following sections we describe neural machine translation (NMT), our model of choice.

2.1 Language Models

Language modeling is indispensable for machine translation and, by extension, neural machine translation. It is by modeling the language of the source and target sentences that an NMT system produces fluent translations.

In particular, a language model is a statistical model that is capable of judging how "fluent" a sentence is in some language. The mechanism by which such a model works is by specifying a probability distribution over the words of a sentence, and assigning higher probabilities to sequences that, when taken together, are more fluent. This begs the question of what "fluency" is, but this is outside the scope of this project.

Consider a sentence that consists of the following word sequence: $s = w_1, ..., w_m$. The probability assigned to this sentence by a language model will be,

[*]Thanks to Denny Britz, who crawled `daddicts.com` and provided advice throughout the project.

$$p(s) = \prod_{i=1}^{m} p(w_i | w_{<i})$$

Where "$w_{<i}$" means all the words in s before w_i. Intuitively, this equation is saying that the fluency of a sentence s is the product of probabilities for each $w_i \in s$, where the proability for each word is conditioned on all the words prior to it.

2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are effective tools for finding structure in sequential data. These models take a vector $\boldsymbol{x}_t$ at each timestep. They compute a hidden state vector $\boldsymbol{h}_t$ for each timestep by applying linear maps to the the previous hidden state $\boldsymbol{h}_{t-1}$ and the current input $\boldsymbol{x}_t$. The result is passed through a logistic sigmoid nonlinearity:

$$\boldsymbol{h}_t = \sigma\Big(\boldsymbol{W}^{(hx)}\boldsymbol{x}_t + \boldsymbol{W}^{(hh)}\boldsymbol{h}_{t-1}]\Big) \tag{1}$$

The RNN can generate a probability distribution over output symbols for each timestep by attaching a fully connected layer to the hidden states, and passing those logits through a softmax:

$$\boldsymbol{s}_t = \boldsymbol{W}^{(y)}\boldsymbol{h}_t \tag{2}$$
$$\hat{\boldsymbol{p}}_t = \mathrm{softmax}(\boldsymbol{s}_t) \tag{3}$$
$$\tag{4}$$

The softmax function transforms the vector of logits $\boldsymbol{s}_t$ into a probability distribution $\boldsymbol{p}_t$, which is defined for each output symbol $y \in Y$. The probability of each output symbol y is then

$$\boldsymbol{p}_i^{(y)} = \frac{e^{\boldsymbol{s}_t^{(y)}}}{\sum_{y' \in Y} e^{\boldsymbol{s}_t^{(y')}}} \tag{5}$$

Though RNNs can in theory model indefinitely long dependencies, training them is difficult because repeated application of the nonlinearity drives error signals to exponential decay as they propagate back through time.

Long Short-Term Memory Networks (LSTMs) are a variant of the above RNN formulation. LSTMs can more effectively model long-term temporal dependencies by coping with the vanishing gradient problem inherent in their predecessor. LSTMs behave much like RNNs, but provide fine-grained control over how much of the previous hidden state is mixed into the new hidden state.

LSTMs have a pair of "memory" vectors, $\boldsymbol{c}_t$ and $\boldsymbol{h}_t$. First, $\boldsymbol{c}_t$, the *memory cell*, is a blending of the previous timestep's memory cell and a candidate $\widetilde{\boldsymbol{c}}$ that is proposed by the model:

$$\boldsymbol{c}_t = \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \widetilde{\boldsymbol{c}}_t \tag{6}$$
$$\widetilde{\boldsymbol{c}}_t = \tanh\Big(\boldsymbol{W}^{(c)}\boldsymbol{x}_t + \boldsymbol{U}^{(c)}\boldsymbol{h}_{t-1}\Big) \tag{7}$$

Note that the mixing in equation (3) is controlled by the *input gate* $\boldsymbol{i}_t$ and *forget gate* $\boldsymbol{f}_t$, both a function of the input and past hidden state:

$$\boldsymbol{i}_t = \sigma\Big(\boldsymbol{W}^{(i)}\boldsymbol{x}_t + \boldsymbol{U}^{(i)}\boldsymbol{h}_{t-1}\Big) \tag{8}$$
$$\boldsymbol{f}_t = \sigma\Big(\boldsymbol{W}^{(f)}\boldsymbol{x}_t + \boldsymbol{U}^{(f)}\boldsymbol{h}_{t-1}\Big) \tag{9}$$

Next, the second memory vector (the new *hidden state*) is a throttling of the new memory cell, with the degree of strangulation determined by an *output gate* $\boldsymbol{o}_t$:

$$\boldsymbol{o}_t = \sigma\Big(\boldsymbol{W}^{(o)}\boldsymbol{x}_t + \boldsymbol{U}^{(o)}\boldsymbol{h}_{t-1}\Big) \tag{10}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh(\boldsymbol{c}_t) \tag{11}$$

Finally, to improve the representational capacity of LSTMs, they may be

1. Stacked into layers where the hidden state of one layer is the next layer's input.
2. Run in both directions over the input, and the hidden states of each direction are concatenated before being sent downstream.

2.3 Neural Machine Translation

Neural machine translation aims to directly model the conditional log probability $\log p(y|x)$ of producing some translation $y = y_1, ..., y_m$ of a source sentence $x = x_1, ..., x_n$ by conditioning on the source. It goes about modeling this conditional probability through the encoder-decoder framework. In this approach, an $encoder$ network encodes the source sentence x into a single vector representation $\boldsymbol{e}$. The $decoder$ network conditions on this encoding and generates a translation y, one target word at a time. By doing this, the decoder is decomposing the conditional log probability into

$$\log p(y|x) = \sum_{t=1}^{m} \log p(y_t | y_{<t}, \boldsymbol{e}) \tag{12}$$

In practice, both the encoder and decoder are some breed of recurrent network, whether that be an RNN, LSTM, or GRU. We used LSTMS in these experiments. Furthermore, the encoder and decoder architecture can vary in several dimensions, including (but not limited to):

- $directionality$ - unidirectional or bidirectional.
- $depth$ - single or multilayer.
- $capacity$ - size of hidden state vectors, and number of cells in the network.

In more detail, at their lowermost layers, encoders recieve the *start symbol* `<start>`, followed by each token in the sentence, followed by the *end symbol* `<end>`. The model looks up the proper vector embeddings of these discrete symbols and feeds the embeddings into the encoder LSTM. Likewise, at the target side, the decoder LSTM outputs are projected into an embedding space, which are then used to look up output tokens with a final fully connected layer and a softmax. For these embedding layers to work, a vocabulary is selected, typically the top V most frequent words or subword units in the source and target corpus. We used words as the atomic units for our experiments. The embedding weights may be initialized with pre-trained vectors like those learned by word2vec or Glove. However, in this work, we initialized the embeddings randomly and learned them from scratch.

Once embeddings are retrieved, they are fed into the encoder network one at a time. The encoder network is initialized with a zero state, but the decoder's state is initialized with $\boldsymbol{e}$, the source encoding (unless we are using attention, which will be discussed in the next section). The decoder then runs until it produces an `<end>` token.

There are a few tricks that are not in the formal description of an NMT system, but are necessary to get it working well:

- Sentences in natural language are not of uniform length. So each source and target sentence in the corpus is padded with a special `<pad>` token.
- Padding creates distance between the end of the source and start of the target, though, so the padded source sequence is reversed before being fed into the encoder.

- Words outside the vocabulary are mapped to the `<unk>` token. There are a variety of unk-replacement strategies at decoding time. A convenient one uses attentional scores to copy words from the source. We didn't explore these, however, due to time and computational constraints.
- The decoder is an RNN, so it needs inputs to produce translations. Initializing with the encoder's final hidden state is not enough. Since it is supposed to model the decomposed conditional log probability of the target sentence, we feed in target symbols to the decoder as input. During training, we feed "true" target symbols at each timestep, i.e. at time t the decoder takes y_t and produces y_{t+1}. During test time, we don't have access to true symbols, so we select an output symbol for feeding (see next bullet).
- During testing we need to be able to translate unseen source sentences for which we have no targets. There are multiple ways to accomplish this. One, $greedy\ decoding$, is when we select the most likely word and use it as input for the next decoding step. Another, *beam search decoding* is when we keep a beam of likely candidates and push all of them through the decoder.

2.4 Attention Mechanisms

Instead of initializing the decoder with the final state of the encoder, an attention mechanism will allow the decoder to look over the entire history of encoder hidden states before producing each translated token. In other words, the attention mechanism provides a **random access memory** to source hidden states that can be referred to throughout the translation process. Though it slows down the inference process, such mechanisms afford interpretability and agency; models can focus on parts of the input before producing output tokens, which is similar to the way humans translate sentences.

There are several attention mechanisms that have been proposed for NMT. We opted to adopt Long et al.'s $global\ attention$, where the "focus" over encoder hidden states is a soft probability distribution that is differentiable (and thus may be backpropagated through) [1]. We chose this attention because it is the default Tensorflow behavior, and would provide a fairer comparison with our custom attention implementation. Note that because the decoder attends over the entire input sequence, tanslations are now conditional on the entire source sequence, not just the source encoding. This means that the network is modeling $p(y_t|y_{<t}, x)$.

The general approach of an attentional layer is as follows. Let h'_t be the *target hidden state*. This is the hidden state at the top layer of the decoder network. Let c_t be the *source context vector*. This vector is an attentional summary of the source sequence, i.e. the model had to attend to its encoder hidden states to produce this vector. The new hidden state and network output are:

$$\widetilde{h}_t = \tanh\Big(\boldsymbol{W}^{(c)}[h'_t : c_t]\Big) \tag{13}$$

$$p(y_t|y_{<t}, x) = \text{softmax}\Big(\boldsymbol{W}^{(s)}\widetilde{h}_t\Big) \tag{14}$$

There are several ways to compute the source context vector. We chose that of *global attention*, which considers all the states of the encoder. According to this formulation,

$$c_t = \sum_{i=0}^{m} a_t^i h_i \tag{15}$$

At a high level, c_t is the weighted sum of encoder hidden states. The weightings a_t^i are a probability distribution, where encoder hidden states that are more "similar" to the current decoder state are given higher probability:

$$a_t^i = \text{similarity}(h_t, h_i') \tag{16}$$
$$= \frac{\exp(\text{score}(h_i, h_t'))}{\sum_j \exp(\text{score}(h_j, h_t'))} \tag{17}$$

The scoring function tells you how similar two hidden state vectors are. We implemented two such scoring functions, *dot product* (17) and *bilinear* (18)

$$\text{score}_{dot}(h_i, h_t') = h_i^\intercal h_t' \tag{18}$$
$$\text{score}_{bilinear}(h_i, h_t') = h_i^\intercal \boldsymbol{W} h_t' \tag{19}$$

We hypothesize that the mediating matrix of the bilinear score function (and the interactions it affords) will lead to better performance.

2.5 Model Architecture

We conducted a **brief** hyperparameter search on a 100k phrase subset of our data with a max sequence length of 50. We were severely constrained by computational resources, but c'est la vie. Our final model consisted of an encoder-decoder network with 256-dimensional source and target embeddings, batch size of 128, 2-layer bidirectional LSTM encoder with 512 hidden units, and a 1-layer LSTM decoder with 512 hidden units. We implemented a beam-search decoder, but didn't finish it in time to conduct all of our experiments with it. Thus, it's performance is only presented in section (3.1). We trained with the Adam optimzer, a learning rate of 0.0003, and clipped gradients when their norms exceeded 5.0. We experimented with various learning rate annealing schedules, but found that this did not change model performance. We regularized the model with dropout at a rate of 0.2. All experiments were run after training for 14 epochs.

3 Experiments

We conducted three sets of experiments that investigated (1) the quality of our implementation and its various components, (2) the efficacy of tensorflow abstractions, and (3) the quality of our dataset.

3.1 Implementation

We implemented several handmade extensions to the vanilla sequence-to-sequence machine translation model (making use of no tensorflow abstractions beyond basic matrix operations). All of these should be tested. To this end, we conducted an ablation study on a subset of the IWSLT'15 EN-VI dataset. We reserved 100K sentence pairs for training, 7K for validation and test. Our results are as follows:

Vanilla	+dot attention	+bilinear attention	+bidirectional encoder	+beam decoding (beam size=10)
14.32	17.26	17.74	18.44	**19.53**

Table 1: Model performance on a 100k subset of EN-VI data.

It is apparent that attention provides the largest gains in performance, but that bidirectional encoders and beam search also helps. Note that this was the only experiment we had time to run with the beam search decoder.

3.2 Tensorflow

Tensorflow provides a plenitude of high-level abstractions for implementing NMT systems. These include,

- The RNN unrolling functions
  - `tf.nn.dynamic_rnn`
  - `tf.nn.bidirectional_dynamic_rnn`
  - `tf.contrib.rnn.python.ops.rnn.stack_bidirectional_dynamic_rnn`

  which may be used to implement the encoder with one function call.
- The decoder unrolling functions
  - `tf.contrib.seq2seq.attention_decoder_fn_train`
  - `tf.contrib.seq2seq.attention_decoder_fn_inference`
  - `tf.contrib.seq2seq.dynamic_rnn_decoder`

  Which can run both a decoder **and** an attention mechanism in a single function call.
- The seq2seq model functions
  - `tf.nn.seq2seq.embedding_rnn_seq2seq`
  - `tf.nn.seq2seq.embedding_attention_seq2seq`

  which can run the entire end-to-end seq2seq architecture in a single function call.

We gave our NMT system the ability to toggle between each of the above abstractions and from-scratch modules whose tensorflow usage was restricted to low-level matrix operations like concatenation, multiplication, etc.. We then trained and tested all togglings on 100k subsets of the aforementioned EN-VI dataset, as well as the ASPEC EN-JA dataset. Our results are shown in Figure 1.

Every time we toggled away from a tensorflow module, the performance of our model went up. This is puzzling because in theory, the exact same operations and derivatives should have been taken. We believe it may point to flaws in the Tensorflow source code, and in general, the fickle nature of complex systems.

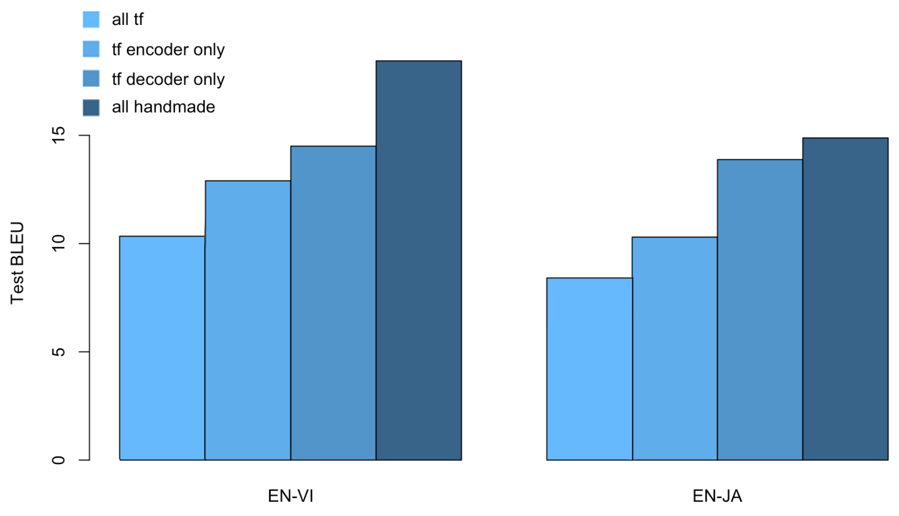

Figure 1: Model performance with gradually lessening reliance on tensorflow abstractions.

3.3 Dataset

The new dataset we made was rubbish. We trained all configurations of our NMT system on a 100K subset of the cleaned EN-JA corpus. The system had a test BLEU of 2.09. The training curves suggest that the model isn't overfitting or failing to converge, which suggests that our new dataset is either too noisy or contains too much of a variety of phrases and situations to be an effective tool for translation (see fig 2; val losses are below train losses because I'm feeding true targets into the decoder at validation time).

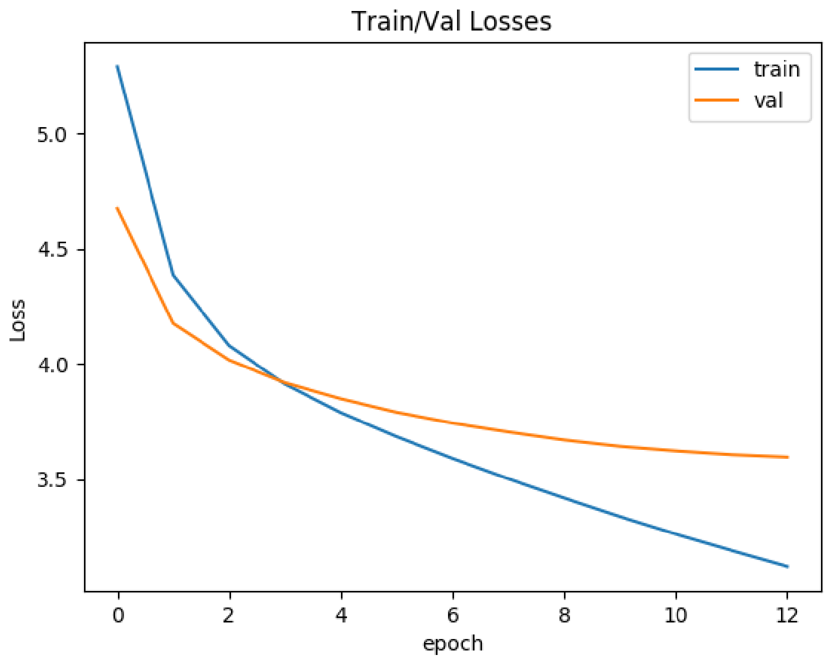

Figure 2: Training curves on our JA-EN dataset.

4 Conclusion

References

[1] Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning. ”Effective approaches to attention-based neural machine translation.” arXiv preprint arXiv:1508.04025 (2015).

[2] Breen, Jim. ”The EDICT Dictionary File”. Electronic Dictionary Research and Development Group. http://www.edrdg.org/jmdict/edict.html.

[3] Hagiwara, Masato, and Satoshi Sekine. ”Lightweight Client-Side Chinese/Japanese Morphological Analyzer Based on Online Learning.” COLING (Demos). 2014.