Focusing on Fast Neural Network Inference

Anonymous ACL submission

Abstract

This paper describe the submissions to the efficiency track for GPUs by members of the University of Edinburgh, Adam Mickiewicz University, Tilde and University of Alicante. We focus on efficient implementation of the recurrent deep-learning model as implemented in Amun, the fast inference engine for neural machine translation. We improve the performance with an efficient mini-batching and maxi-batching algorithm. Translation speed was also reduced by fusing the softmax operation with the beam seach algorithm.

1 Introduction

As neural machine translation (NMT) models have become the new state-of-the-art, the challenge is to make their deployment efficient and economical. This is the challenge that this shared task is shining a spotlight on.

It is tempting to use a general purpose deep-learning toolkit such as Tensorflow or Pytorch to train and do the the inference where faster translation could be obtained by tuning parameters within the toolkit, and choosing fast models.

We take an opposing approach by using and enhancing a custom inference engine, Amun (Junczys-Dowmunt et al., 2016), which we developed on the premise that fast deep-learning inference is an issue that deserves dedicated tools which are not compromised by other objectives such as training or support for multiple models. As well as delivering on the useful goal of fast inference, it can serve as a test-bed for novel ideas on neural network inference, and is useful as a means to explore the upper limit of the possible speed for a particular model and hardware. That is, Amun is an inference-only engine that supports a limited number of NMT models that put fast inference on modern GPU above all other considerations.

We submitted two systems to this year's shared task for the efficient translation on GPU. Our first submission was tailored to be as fast as possible while being above the baseline BLEU score. Our second submission trade some of the speed of first submission to return good quality translation.

We will describe below our experimental setup as well as the enhancements to Amun that we feel enabled it to achieve its outstanding speed.

2 Improvements

We desribe the main enhancements to Amun since the original 2016 publication.

2.1 Batching

The use of mini-batching is critical for fast model inference. The size of the batch is determined by the number of inputs sentences to the encoder in an encoder-decoder model. However, the number of batches during decoding can vary as some sentences have completed translating or the beam search add more hypotheses to the batch.

It is tempting to ignore these considerations, for example, by always decoding with a constant batch and beam size and ignoring hypotheses which are not needed but this comes at a cost of lower translation speed due to wasteful processing.

The Amun engine implements an efficient batching algorithm that takes into account the actual number of hypotheses that needs to be decoded at each decoding step, Figure 1.

Algorithm 1 Mini-batching

```
procedure BATCHING(encoded sentences i)
    Create batch b from i
    while b ≠ ∅ do
        b' ← DecodeAndBeamSearch(b)
        b ← ∅
        for all hypo h in b' do
            if h ≠ </s> then
                Add h to b
            end if
        end for
    end while
end procedure
```

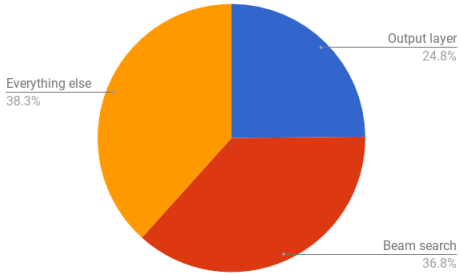

Figure 1: Proportion of time spent during translation

2.2 Softmax and Beam Search Fusion

Most NMT more predict a large number of classes in their output layer, corresponding to the number of words or subword units in their target language. For example, Sennrich et al. (2016) experimented with target vocabulary sizes of 60,000 and 90,000 sub-word units. This makes the output layer of NMT models very computationally expensive. Figure **??** shows the breakdown of amount of time during translation our NMT system; nearly 70% of the time is involved in the output layer. We describe the steps to fuse the operations to improve efficiency.

The output layer of most deep learning models consist of the following steps

1. multiplication of the weight matrix with the input vector $p = wx$
2. addition of a bias term to the resulting scores $p = p + b$
3. applying the activation function, most commonly softmax $p_i = \exp(p_i) / \sum \exp(p_i)$
4. a beam search for the best (or n-best) output classes $\text{argmax}_i\, p_i$

In models with a small number of classes such as binary classification, the computational

effort required is trivial and fast but this is not the case for the large number of classes found in typical NMT models.

We leave step 1 for future work and focus on the last three steps, the outline for which are shown in Algorithm 2. For brevity, we show the algorithm for 1-best, a beam search of the n-bests is a simple extension of this.

As can be seen, the matrix p is iterated over five times - once to add the bias, three times to calculate the softmax, and once to search for the best classes. We propose fusing the three functions into one kernel, a popular optimization technique (Guevara et al., 2009), making use of the following observations.

Firstly, softmax and exp are monotonic functions, therefore, we can move the search for the best class from FIND-BEST to SOFTMAX, at the start of the kernel.

Secondly, we are only interested in the probabilities of the best classes. Since they are now known at the start of the kernel, we compute softmax only for those classes.

Thirdly, the calculation of max and sum can be accomplished in one loop by adjusting sum whenever a higher max is found during the looping:

$$\begin{aligned} sum &= e^{x_t - max_b} + \sum_{i=0\ldots t-1} e^{x_i - max_b} \\ &= e^{x_t - max_b} + \sum_{i=0\ldots t-1} e^{x_i - max_a + \Delta} \\ &= e^{x_t - max_b} + e^{\Delta} \times \sum_{i=0\ldots t-1} e^{x_i - max_a} \end{aligned}$$

where max_a is the original maximum value, max_b is the new, higher maximum value, ie. $max_b > max_a$, and $\Delta = max_a - max_b$. The outline of our function is shown in Algorithm 3.

In fact, a well known optimization is to skip softmax altogether and calculate the argmax

Algorithm 2 Original softmax and beam Search Algorithm

procedure ADDBIAS(vector p, bias vector b)
  for all p_i in p **do**
    $p_i \leftarrow p_i + b_i$
  end for
end procedure

procedure SOFTMAX(vector p)
  ▷ calculate max for softmax stability
  $max \leftarrow -\infty$
  for all p_i in p **do**
    if $p_i > max$ **then**
      $max \leftarrow p_i$
    end if
  end for
  ▷ calculate denominator
  $sum \leftarrow 0$
  for all p_i in p **do**
    $sum \leftarrow sum + \exp(p_i - max)$
  end for
  ▷ calculate softmax
  for all p_i in p **do**
    $p_i \leftarrow \frac{\exp(p_i - max)}{sum}$
  end for
end procedure

procedure FIND-BEST(vector p)
  $max \leftarrow -\infty$
  for all p_i in p **do**
    if $p_i > max$ **then**
      $max \leftarrow p_i$
      $best \leftarrow i$
    end if
  end for
  return $max, best$
end procedure

Algorithm 3 Fused softmax and beam search

procedure FUSED-KERNEL(vector p, bias vector b)
  ▷ add bias, calculate max & $argmax$
  $max \leftarrow -\infty$
  $sum \leftarrow 0$
  for all p_i in p **do**
    $p'_i \leftarrow p_i + b_i$
    if $p'_i > max$ **then**
      $\Delta \leftarrow max - p'_i$
      $sum \leftarrow \Delta \times sum + 1$
      $max \leftarrow p'_i$
      $best \leftarrow i$
    else
      $sum \leftarrow sum + \exp(p'_i - max)$
    end if
  end for
  return $\frac{1}{sum}, best$
end procedure

over the input vector, Algorithm 4. This is only possible for beam size 1 and when we are not interested in returning the softmax probabilities.

2.3 Half-Precision

Reducing the number of bits needed to store floating point values from 32-bits to 16-bits promises to increase translation speed through faster calculations and reduced bandwidth usage. 16-bit floating point operations are supported by the GPU hardware and software available in the shared task.

In practise, however, efficiently using half-precision value requires a comprehensive redvelopment of the GPU code. We therefore make do with using the GPU's Tensor Core fast matrix multiplication routines which transparently converts 32-bit float point input matrices to 16-bit values and output a 32-bit float point product of the inputs.

Algorithm 4 Find 1-best only

procedure FUSED-KERNEL-1-BEST(vector p, bias vector b)
  $max \leftarrow -\infty$
  for all p_i in p **do**
    if $p_i + b_i > max$ **then**
      $max \leftarrow p_i + b_i$
      $best \leftarrow i$
    end if
  end for
  return $best$
end procedure

3 Experimental Setup

Both of our submitted systems uses the sequence-to-sequence model similar to that described in Sennrich et al. (2016), containing a bidirectional RNN in the encoder and a two-layer RNN in the decoder. We use BPE to adjust the vocabulary size.

We used a variety of GPUs to train the models but all testing was done on an Nvidia V100. Translation quality was measured using BLEU, specifically multi-bleu as found in the Moses toolkit. The validation and test sets provided by the shared task organisers were used to measure translation quality, but a 50,000 sentence subset of the training data was used to measure translation speed to obtain longer, more accurate measurements.

3.1 GRU-based system

Our first system uses gated recurrent units (GRU) throughout, trained with Marian, but submitted both systems using the Amun inference engine.

We experimented with varying the vocabulary size and the RNN state size before settling

	State dim		
Vocab size	256	512	1024
1.000	12.23		12.77
5,000	16.79		17.16
10,000	18.00		18.19
20,000	-		19.52
30,000	18.51	19.17	19.64

Table 1: Validation set BLEU (newstest2014) for GRU-based model

	Vocab size	
Beam size	40,000	50,000
1	23.45	23.32
2	24.15	24.04
3	24.48	
4	24.42	
5	24.48	

Table 2: Validation set BLEU for mLSTM-based model

for a vocabulary size of 30,000 (for both source and target language) and 256 for the state size, Table 1.

After further experimentation, we decided to use sentence length normalization and NVidia's Tensor Core matrix multiplication which increased translation quality as well as translation speed. The beam was kept at 1 throughout for the fastest possible inference.

3.2 mLSTM-based system

Our second system uses multiplicative-LSTM in the encoder and the first layer of a decder, and a GRU in the second layer, using an extension of the Nematus toolkit which supports such model. We trained 2 systems with differing vocabulary sizes and varied the beam sizes before choosing the the settings that produces good translation quality on the validation set, Table 2.

4 Result

4.1 Batching

The efficiency of Amun's batching algorithm can be seen by observing the time taken for each decoding step in a batch of sentence, Figure 2. Amun's decoding becomes faster as sentences finish translating. This contrast with the Marian inference engine, which uses a simpler

Figure 2: Time taken for each decoding step for a batch of 1280 sentences

batching algorithm, where the speed stays relatively constant throughout the decoding of the batch.

Using batching can increase the translation speed by over 20 times in Amun and shows little degradation with larger batch sizes, Figure 3. Just as importantly, however, it doesn't suffer degradation with large batch sizes unlike the simpler algorithm which slows down when the batch size is above 1000. This scalability issue is likely to increase with ever increasing core counts.

4.2 Softmax and Beam Search Fusion

Fusing the bias and softmax operations in the output layer with the beam search results in a speed improvement by 25%, Figure 4. Its relative improvement decrease marginally with in-

Figure 3: Speed v. batch size

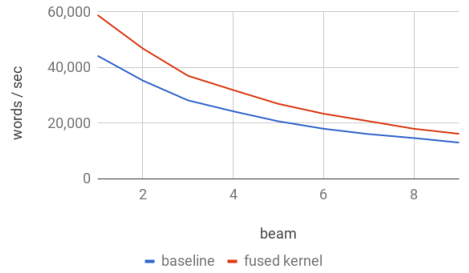

Figure 4: Using fused operation

creasing beam size.

Further insight can be gained by examining the time taken for each step in the output layer and beam search, Table 3. The fused operation only has to loop through the large cost matrix once, therefore, for low beam sizes its is comparable in speed to the simple kernel to add the bias. For higher beam sizes, the cost of maintaining the n-best list is felt.

4.3 Tensor Cores

By taking advantage of the GPU's hardware accelerated matrix multiplication, we can gain up to 20% in speed, Table 4.

5 Conclusion and Future Work

We have presented some of the improvement to Amun which are focused to improvement NMT inference.

We are also working to make deep-learning faster using more specialised hardware such as FPGAs. It would be interesting as future work to bring our focused approach to fast deep-learning inference to a more general training toolkit.

	Baseline	Fused
Beam size 1		
Multiplication	5.39	5.38 (+0%)
Add bias	1.26	
Softmax	1.69	2.07 (-86.6%)
Beam search	12.53	
Beam size 3		
Multiplication	14.18	14.16 (+0%)
Add bias	3.76	
Softmax	4.75	3.43 (-87.1%)
Beam search	18.23	
Beam size 9		
Multiplication	38.35	38.42 (+0%)
Add bias	11.64	
Softmax	14.4	17.5 (-72.1%)
Beam search	36.7	

Table 3: Time taken (sec) breakdown

Beam size	Baseline	Tensor Cores
1	39.97	34.54 (-13.6%)
9	145.8	116.8 (-20.0%)

Table 4: Time taken (sec) using Tensor Cores

References

Guevara, M., Gregg, C., Hazelwood, K. M., and Skadron, K. (2009). Enabling task parallelism in the cuda scheduler.

Junczys-Dowmunt, M., Dwojak, T., and Hoang, H. (2016). Is neural machine translation ready for deployment? a case study on 30 translation

directions. In *Proceedings of the 9th International Workshop on Spoken Language Translation (IWSLT)*, Seattle, WA.

Sennrich, R., Haddow, B., and Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.