You May Not Need Attention

Ofir Press , Noah A. Smith, Paul G. Allen
School of Computer Science & Engineering University of Washington ,
Allen Institute for Artificial Intelligence

fofirp,nasmithg@cs.washington.edu

Discussion Lead : Preeti Padelkar
Discussion Facilitator : Ehsan Amjadian

Abstract

The paper introduces a recurrent neural translation model and decoder. The eager translation model is low-latency, writing target tokens as soon as it reads the first source token, and uses constant memory during decoding.

It performs on par with the standard attention-based model of Bahdanau et al. (2014), and better on long sentences

It does not use attention and does not have a separate encoder

Motivation

- Eager translation model.
- simpler model that resembles the language model of Zaremba et al. (2014).
- Uses less model parameters.
- Less computational intense during inference.
- Without attention mechanism and the encoder and decoder are unified into a single module.

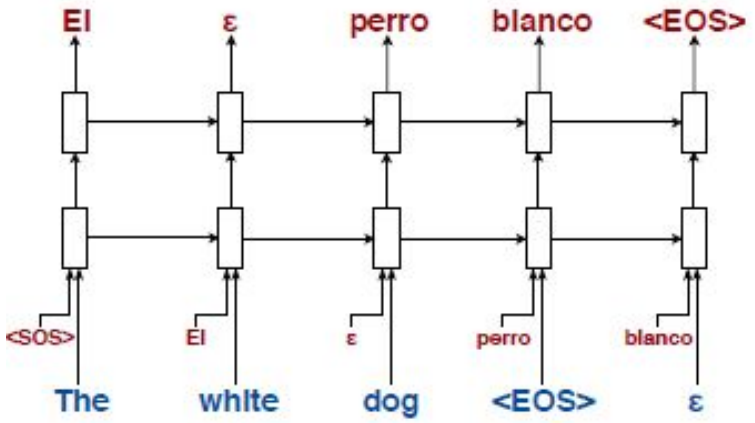


Figure 1: The eager model translating the sentence “The white dog” into Spanish. Source (target) tokens are in blue (red). **ε** is the padding token, which is removed during post processing. The diagram presents an eager translation model with two LSTM layers.

- The eager translation model uses a constant amount of memory, since it needs to use only one previous hidden state (rather than all previous hidden states) at every timestep.
- Our approach crams a prefix of a source sentence (and its resulting translation) into a dynamic memory vector, emitting target tokens immediately and every time another source word is read.

Data Preprocessing

- A source and target sequence as eager feasible if, for every aligned pair of words (si; tj), i j.
- Using the alignments inferred by an off-the-shelf alignment model (fast align; Dyer et al., 2013), then inserting the minimal number of **ε** (empty) tokens into the target sentence to achieve the desired property.

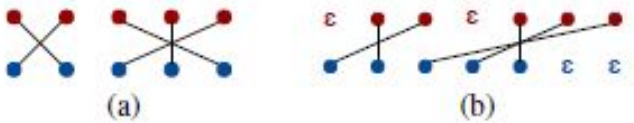


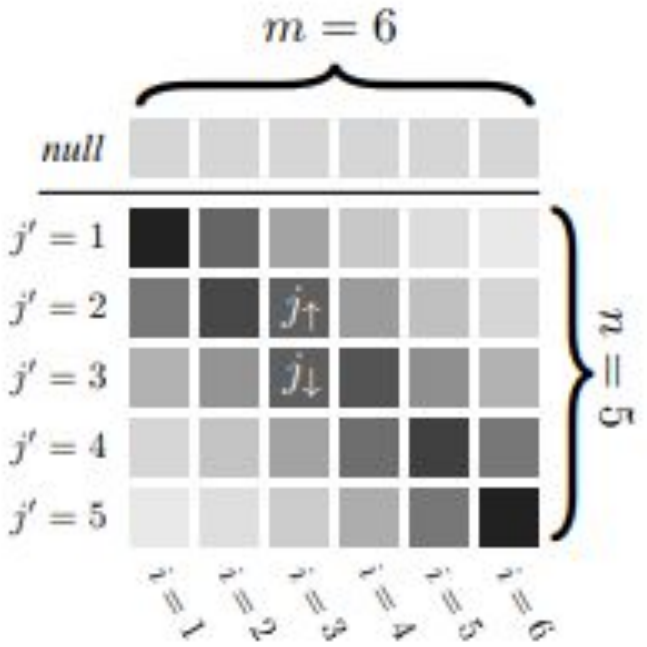
Figure 2: A source (blue) and target (red) sequence with their alignment before (a) and after (b) preprocessing to make the pair "eager feasible."

	FR→EN	EN→FR	DE→EN	EN→DE
ε Proportion	23%	14%	23%	20%

Table 1: Average percentage of ε in the target sentences of the four language directions. Initial padding tokens and padding tokens inserted to make the source and target sequence lengths equal are not counted.

Off-the-shelf alignment

- Given a source sentence f with length n, first generate the length of the target sentence, m.
- Generate an alignment, a = {a1,a2,...,am}, that indicates which source word each target word will be a translation of.
- Finally, generate the m output words, where each translation ei depends only on fai.



Model

- At each timestep, our model first embeds the current input word (in the source language) and the previously selected output word (in the target language) into dense representations both of dimension E.

- These vectors are concatenated and the resulting vector is then fed into a multi-layered LSTM containing 2E units at each layer.

- The output of the LSTM is transformed into a vector of size E using a fully connected layer. The output of that fully connected layer is transformed into a distribution over the target vocabulary using an output embedding matrix and the softmax function.

Aligned Batching

- Concatenate all the source sentences into a source string and all the target sentences into a target string, keeping them in the same order.

- As in language modeling training, the last hidden state from the (i - 1)th batch becomes the initial hidden state of the ith batch.

Decoding

1. Padding limit:
   - We place an upper limit on the number of padding symbols emitted, by forcing the probability of **ε** to zero after the limit is reached.
   - Initial padding symbols are not counted towards this limit.

2. Source padding injection (SPI):
   - Once the EOS token in the source language is read, the decoder assigns a high probability to the end of-sequence (EOS) token.
   - It enables the model to output a sentence that is longer than the input sentence. Without it, the generated sentence length is capped by the source sentence length (and is exactly equal to source sentence length minus the number of generated padding tokens).

add_epsilon.py (pre processing step)

```
for a in range(len(parsedLines)):
    l = parsedLines[a]
    assert l != None
    src_split =
read_single_line(src.readline())

    trg_split =
read_single_line(trg.readline())

    lenS, lenT = len(src_split),
len(trg_split)
    cost = 0
```

```
for i in range(3):
        trg_split.insert(0, START_PAD)
        cost += 1

    for pair in l:
        i, j = pair
        if i > j:
            diff = i - j
            if diff > cost:
                local_cost = diff - cost
```

```
for d in range(local_cost):
      trg_split.insert(j + cost, SRC_EPSILON)
      cost += 1
if lenS > lenT + cost:
        trg_split.extend([SRC_EPSILON] * (lenS - lenT - cost)) # pad at the end,
                # so that both src and trg sequences are of the same size.
   elif lenT + cost > lenS:
        src_split.extend([TRG_EPSILON] * (lenT + cost - lenS))

    if a < num_valid:
        src_out_val.write(" ".join(src_split) + '\n')
        trg_out_val.write(" ".join(trg_split) + '\n')
   else:
        src_out_train.write(" ".join(src_split) + '\n')
        trg_out_train.write(" ".join(trg_split) + '\n')
```

Locked Dropout

```
class LockedDropout(nn.Module):
    ...

    def forward(self, x, dropout=0.5):
        if not self.training or not dropout:
            return x
        m = x.data.new(1, x.size(1), x.size(2)).bernoulli_(1 - dropout)
        mask = Variable(m, requires_grad=False) / (1 - dropout)
        mask = mask.expand_as(x)
        return mask * x
```

Comparison

Current Model

- 4 LSTM layers with 1,000 units for our model, and embeddings of size 500.
- The model is regularized during training using dropout on both the LSTM and the word embeddings, as done by Merity et al. (2017).
- Training with a batch size of 200, and we backpropagation through time for 60 tokens.
- We use SGD and start with a learning rate of 20. We check the perplexity on the validation set every 6,500 updates and halve the learning rate if it does not improve.
- The padding limit, source padding injection value, and beam size that we use for inference on the test set are the ones that perform best on the development set.

Reference model

- As a reference model we use the Open- NMT (Klein et al., 2017) implementation of Bahdanau et al. (2014).
- The model has 2 LSTM layers in the encoder and two in the decoder, all with 1,000 units, and embeddings of size 500. This resulted in a model containing a similar number of parameters to our model.
- The optimization algorithm used is SGD, with a starting rate of 1, which is halved every 10,000 steps if there is no improvement in development set perplexity.

Experiments

- Both our model and the reference model are trained until there is no improvement on the development set for 50,000 updates. The reference model took 13 hours to train on a single GPU while our models took around 38 hours.
- Eager model can process approximately three times the amount of source tokens per second as the OpentNMT reference model, training takes longer because the eager model requires more epochs to converge.
- We compute BLEU (Bilingual Evaluation Understudy Score) scores on the detokenize outputs using SacreBLEU (Post, 2018).

Results

	Source Sentence Length	Number of Sentences	Reference Model BLEU	Eager Model BLEU
FR→EN	1–20	864	**26.22**	23.74
	21–40	1312	**29.50**	29.20
	41–60	659	**28.71**	27.77
	61–80	152	27.66	**27.89**
	81+	16	22.10	**27.44**
DE→EN	1–20	963	**22.94**	20.12
	21–40	1275	**23.07**	22.95
	41–60	414	**23.06**	22.53
	61–80	76	23.02	**23.51**
	81+	9	21.24	**24.73**

Table 3: BLEU performance by source sentence length on the FR→EN and DE→EN test sets.

		FR→EN	EN→FR	DE→EN	EN→DE
Start εs	0	24.42	19.97	20.11	11.50
	1	25.76	24.81	20.81	15.81
	2	27.10	25.63	21.45	16.53
	3	27.98	**26.98**	21.39	17.36
	4	28.30	26.37	22.00	17.52
	5	**28.47**	25.49	**22.59**	**17.97**
Ref. Model		28.56	27.20	23.01	18.89

Table 2: BLEU performance on the test sets for the reference model and our model with zero to five initial ε padding tokens (as defined in Sec. 2)

On FR -> EN and EN -> FR the model is at most 0.8% lower in terms of BLEU than the reference model. On the harder DE -> EN and EN -> DE tasks, the eager model is at most 4.8% worse than the reference model.

References

- Off the shelf alignment paper - http://www.aclweb.org/anthology/N13-1073
- https://github.com/ofirpress/YouMayNotNeedAttention
- Regularizing and Optimizing LSTM Language Models https://arxiv.org/pdf/1708.02182.pdf

2 min break

Discussion Points

- Why the difference between Eng to Fr and the other way around?
- Why do they use drop out inside the LSTM timestep layers?
- Would the eager model work with the Transformer XL architecture?
- Why use variable sequence length during training?
  - `bptt = args.bptt if np.random.random() < 0.95 else args.bptt / 2.`
- Why does the model despite being simpler take longer to train?

  The alignments that fast_align generates are not perfect, and sometimes epsilon padding symbols appear in wrong places or are sometimes missing. Having to deal with not only translating but also predicting when to emit a padding symbol makes the eager model's objective much harder than the one that the standard model must learn (just translation).

Thank You

Model.py

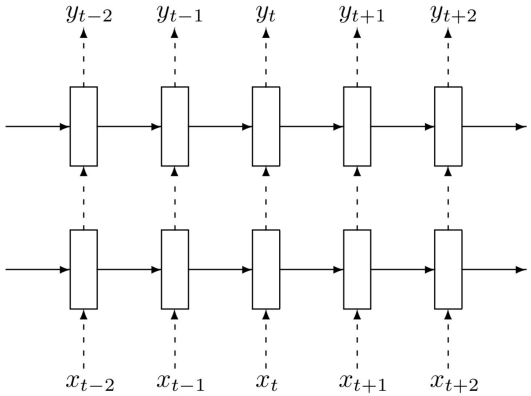
```
def forward(self, input, prev_targets, hidden, return_h=False):
        combined_targets = torch.cat((input.unsqueeze(-1),
prev_targets.unsqueeze(-1)), -1)
        emb = embedded_dropout(self.encoder, combined_targets,
dropout=self.dropoute if self.training else 0)
        emb = emb.view(input.shape[0],input.shape[1], -1)
        emb = self.lockdrop(emb, self.dropouti)

        for l, rnn in enumerate(self.rnns):
            raw_output, new_h = rnn(raw_output, hidden[l])
            new_hidden.append(new_h)
            raw_outputs.append(raw_output)
            if l != self.nlayers - 1:
                raw_output = self.lockdrop(raw_output, self.dropouth)
                outputs.append(raw_output)
```

```
hidden = new_hidden
output = self.lockdrop(raw_output, self.dropout)
 outputs.append(output)
out_size_orig0 = output.size(0)
out_size_orig1 = output.size(1)
output_c =
torch.tanh(self.combiner(output.view(output.size(0)*output.size(1),
output.size(2))))
output_c = output_c.view(output.size(0), output.size(1), -1)
output_c_dropped = self.lockdrop(output_c, self.dropoutcomb)
decoded = self.decoder(output_c_dropped)
result = decoded.view(out_size_orig0, out_size_orig1, decoded.size(2))
if return_h:
    return result, hidden, raw_outputs, outputs
return result, hidden
```

Github account : https://github.com/ofirpress/YouMayNotNeedAttention/

Main.py




```
// Load the data
corpus = data.Corpus(args.data)
eval_batch_size = 10
test_batch_size = 1
train_data_src, train_data_trg = batchify(corpus.train_src, corpus.train_trg,
args.batch_size, args)
val_data_src, val_data_trg = batchify(corpus.valid_src, corpus.valid_trg, eval_batch_size,
args)
test_data_src, test_data_trg = batchify(corpus.valid_src, corpus.valid_trg,
test_batch_size, args)  # test data is same as valid data, we just use different batch size
// Build the model
ntokens = len(corpus.dictionary)
model = model.RNNModel('LSTM', ntokens, args.emsize, args.nhid, args.nlayers, args.dropout,
args.dropouth, args.dropouti, args.dropoute, args.wdrop, args.dropoutcomb, args.tied)
weight = torch.ones(len(corpus.dictionary))
epsilon = corpus.dictionary.word2idx["@@@"]
criterion = nn.CrossEntropyLoss()
total_params = sum(x.size()[0] * x.size()[1] if len(x.size()) > 1 else x.size()[0] for x in
model.parameters())
```

Main.py (contd)

```
def evaluate(data_source_src, data_source_trg, batch_size=10):
    # Turn on evaluation mode which disables dropout.
    model.eval()
    total_eval_loss = 0
    ntokens = len(corpus.dictionary)
    hidden = model.init_hidden(batch_size)
    for i in range(0, data_source_src.size(0) - 1, args.bptt):
        data, prev_targets, targets = get_batch(data_source_src, data_source_trg, i, args, evaluation=True)
        output, hidden = model(data, prev_targets, hidden)
        output_flat = output.view(-1, ntokens)
        total_eval_loss += len(data) * criterion(output_flat, targets).data



        hidden = repackage_hidden(hidden)
    return total_eval_loss.item() / len(data_source_src)
```

```
def train(step_number, stored_loss, lr):
       # Turn on training mode which enables dropout.
        total_loss = 0
       start_time = time.time()
       ntokens = len(corpus.dictionary)
       hidden = model.init_hidden(args.batch_size)
       i = 0
       while i < train_data_src.size(0) - 1 - 1:
           bptt = args.bptt #if np.random.random() < 0.95 else args.bptt / 2.
           # Prevent excessively small or negative sequence lengths
          seq_len = bptt#max(5, int(np.random.normal(bptt, 5)))
        # There's a very small chance that it could select a very long sequence
length resulting in OOM
            model.train()
        data, prev_targets, targets = get_batch(train_data_src, train_data_trg, i,
args, seq_len=seq_len)
```

```
# Starting each batch, we detach the hidden state from how it was previously produced . If we didn't, the model would try backpropagating all the way to start of the dataset.
        hidden = repackage_hidden(hidden)
        optimizer.zero_grad()
        output, hidden, rnn_hs, dropped_rnn_hs = model(data, prev_targets, hidden, return_h=True)
        raw_loss = criterion(output.view(-1, ntokens), targets)
        loss = raw_loss

        # Activiation Regularization
        loss = loss + sum( dropped_rnn_h.pow(2).mean() for dropped_rnn_h in dropped_rnn_hs[-1:])
        # Temporal Activation Regularization (slowness)
        loss = loss + sum( (rnn_h[1:] - rnn_h[:-1]).pow(2).mean() for rnn_h in rnn_hs[-1:])
        loss.backward()

        # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
        torch.nn.utils.clip_grad_norm_(model.parameters(), args.clip)
        optimizer.step()
        total_loss += raw_loss.data
        if step_number % args.log_interval == 0 and step_number > 0:
             cur_loss = total_loss.item() / args.log_interval
            elapsed = time.time() - start_time     #timer doesnt stop while validating, so this will be wrong  ,  if there was a validation call since the                                                                last log print
```

```
 total_loss = 0
 start_time = time.time()
if step_number % args.update_interval == 0 and step_number > 0:
            val_loss = evaluate(val_data_src, val_data_trg, eval_batch_size)
            save_checkpoint(model, optimizer, args.save, suffix=str(step_number))  # just for debug
                 if step_number > args.start_decaying_lr_step:


           if math.exp(val_loss) < math.exp(stored_loss):
      stored_loss = val_loss
          else:
                lr *= 0.5
               print('Lowering LR to: ' + str(lr))
              for param_group in optimizer.param_groups:
                       param_group['lr'] = lr
 step_number += 1
        i += seq_len
        del data, targets, loss, raw_loss
        del output, rnn_hs, dropped_rnn_hs
    return step_number, stored_loss, lr
```

Embedded Regularize

```
def embedded_dropout(embed, words, dropout=0.1, scale=None):
  if dropout:
    mask =
embed.weight.data.new().resize_((embed.weight.size(0),
1)).bernoulli_(1 - dropout).expand_as(embed.weight) / (1 -
dropout)
    masked_embed_weight = mask * embed.weight
  else:
    masked_embed_weight = embed.weight
  if scale:
    masked_embed_weight =
scale.expand_as(masked_embed_weight) * masked_embed_weight
  padding_idx = embed.padding_idx
  if padding_idx is None:
      padding_idx = -1
  X = torch.nn.functional.embedding(words,
masked_embed_weight,
    padding_idx, embed.max_norm, embed.norm_type,
    embed.scale_grad_by_freq, embed.sparse)
  return X
```

```
if __name__ == '__main__':
  V = 50
  h = 4
  bptt = 10
  batch_size = 2

  embed = torch.nn.Embedding(V, h)

  words = np.random.random_integers(low=0,
high=V-1, size=(batch_size, bptt))
  words = torch.LongTensor(words)

  origX = embed(words)
  X = embedded_dropout(embed, words)

  print(origX)
  print(X)
```