RNNs: When to apply BPTT and/or update weights?

Ask Question

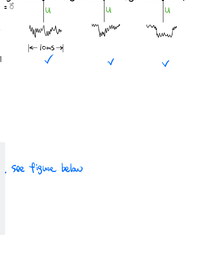
Asked 3 years, 11 months ago Active yesterday Viewed 7k times

16

I am trying to understand the high-level application of RNNs to sequence labeling via (among others) Graves' 2005 paper on phoneme classification.

To summarize the problem: We have a large training set consisting of (input) audio files of single sentences and (output) expert-labeled start times, stop times and labels for individual phonemes (including a few "special" phonemes such as silence, such that each sample in each audio file is labeled with some phoneme symbol.)

The thrust of the paper is to apply an RNN with LSTM memory cells in the hidden layer to this problem. (He applies several variants and several other techniques as comparison. I am for the moment ONLY interested in the unidirectional LSTM, to keep things simple.)



I believe I understand the architecture of the network: An input layer corresponding to 10 ms windows of the audio files, preprocessed in ways standard to audio work; a hidden layer of LSTM cells, and an output layer with a one-hot coding of all possible 61 phone symbols.

I believe I understand the (intricate but straightforward) equations of the forward pass and backward pass through the LSTM units. They are just calculus and the chain rule.

What I do not understand, after reading this paper and several similar ones several times, is *when exactly* to apply the backpropagation algorithm and *when exactly* to update the various weights in the neurons.

Two plausible methods exist:

1) Frame-wise backprop and update

```
Load a sentence.
Divide into frames/timesteps.
For each frame:
- Apply forward step
- Determine error function
- Apply backpropagation to this frame's error
- Update weights accordingly
At end of sentence, reset memory
Load another sentence and continue.
```

for Example, Chunk length = 3 steps, See figure below

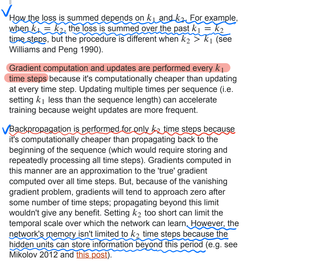
or,

2) Sentence-wise backprop and update:

```
Load a sentence.
Divide into frames/timesteps.
For each frame:
- Apply forward step
- Determine error function
At end of sentence:
- Apply backprop to average of sentence error functi
- Update weights accordingly
- Reset memory
Load another sentence and continue.
```

Note that this is a general question about RNN training using the Graves paper as a pointed (and personally relevant) example: When training RNNs on sequences, is backprop applied at every timestep? Are weights adjusted every timestep? Or, in a loose analogy to batch training on strictly feed-forward architectures, are errors accumulated and averaged over a particular sequence before backprop and weight updates are applied?

Or am I even more confused than I think?



1 Answer

26

I'll assume we're talking about recurrent neural nets (RNNs) that produce an output at every time step (if output is only available at the end of the sequence, it only makes sense to run backprop at the end). RNNs in this setting are often trained using truncated backpropagation through time (BPTT), operating sequentially on 'chunks' of a sequence. The procedure looks like this:

1. Forward pass: Step through the next k_1 time steps, computing the input, hidden, and output states.
2. Compute the loss, summed over the previous time steps (see below).
3. Backward pass: Compute the gradient of the loss w.r.t. all parameters, accumulating over the previous k_2 time steps (this requires having stored all activations for these time steps). Clip gradients to avoid the exploding gradient problem (happens rarely).
4. Update parameters (this occurs once per chunk, not incrementally at each time step).
5. If processing multiple chunks of a longer sequence, store the hidden state at the last time step (will be used to initialize hidden state for beginning of next chunk). If we've reached the end of the sequence, reset the memory/hidden state and move to the beginning of the next sequence (or beginning of the same sequence, if there's only one).
6. Repeat from step 1.

How the loss is summed depends on k_1 and k_2. For example, when $k_1 = k_2$, the loss is summed over the past $k_1 = k_2$ time steps, but the procedure is different when $k_2 > k_1$ (see Williams and Peng 1990).

Gradient computation and updates are performed every k_1 time steps because it's computationally cheaper than updating at every time step. Updating multiple times per sequence (i.e. setting k_1 less than the sequence length) can accelerate training because weight updates are more frequent.

Backpropagation is performed for only k_2 time steps because it's computationally cheaper than propagating back to the beginning of the sequence (which would require storing and repeatedly processing all time steps). Gradients computed in this manner are an approximation to the 'true' gradient computed over all time steps. But, because of the vanishing gradient problem, gradients will tend to approach zero after some number of time steps; propagating beyond this limit wouldn't give any benefit. Setting k_2 too short can limit the temporal scale over which the network can learn. However, the network's memory isn't limited to k_2 time steps because the hidden units can store information beyond this period (e.g. see Mikolov 2012 and this post).

Besides computational considerations, the proper settings for k_1 and k_2 depend on the statistics of the data (e.g. the temporal scale of the structures that are relevant for producing good outputs). They probably also depend on the details of the network. For example, there are a number of architectures, initialization tricks, etc. designed to mitigate the decaying gradient problem.

Your option 1 ('frame-wise backprop') corresponds to setting k_1 to 1 and k_2 to the number of time steps from the beginning of the sentence to the current point. Option 2 ('sentence-wise backprop') corresponds to setting both k_1 and k_2 to the sentence length. Both are valid approaches (with computational/performance considerations as above; #1 would be quite computationally intensive for longer sequences). Neither of these approaches would be called 'truncated' because backpropagation occurs over the entire sequence. Other settings of k_1 and k_2 are possible; I'll list some examples below.

References describing truncated BPTT (procedure, motivation, practical issues):

- Sutskever (2013). Training recurrent neural networks.
- Mikolov (2012). Statistical Language Models Based on Neural Networks.
  - Using vanilla RNNs to process text data as a sequence of words, he recommends setting k_1 to 10-20 words and k_2 to 5 words
  - Performing multiple updates per sequence (i.e. k_1 less than the sequence length) works better than updating at the end of the sequence
  - Performing updates once per chunk is better than incrementally (which can be unstable)
- Williams and Peng (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories.
  - Original (?) proposal of the algorithm
  - They discuss the choice of k_1 and k_2 (which they call h' and h). They only consider $k_2 \geq k_1$.
  - Note: They use the phrase "BPTT(h; h')" or 'the improved algorithm' to refer to what the other references call 'truncated BPTT'. They use the phrase 'truncated BPTT' to mean the special case where $k_1 = 1$.

Other examples using truncated BPTT:

- (Karpathy 2015). char-rnn.
  - Description and code
  - Vanilla RNN processing text documents one character at a time. Trained to predict the next character. $k_1 = k_2 = 25$ characters. Network used to generate new text in the style of the training document, with amusing results.
- Graves (2014). Generating sequences with recurrent neural networks.
  - See section about generating simulated Wikipedia articles. LSTM network processing text data as sequence of bytes. Trained to predict the next byte. $k_1 = k_2 = 100$ bytes. LSTM memory reset every $10,000$ bytes.
- Sak et al. (2014). Long short term memory based recurrent neural network architectures for large vocabulary speech recognition.
  - Modified LSTM networks, processing sequences of acoustic features. $k_1 = k_2 = 20$.
- Ollivier et al. (2015). Training recurrent networks online without backtracking.
  - Point of this paper was to propose a different learning algorithm, but they did compare it to truncated BPTT. Used vanilla RNNs to predict sequences of symbols. Only mentioning it here to to say that they used $k_1 = k_2 = 15$.
- Hochreiter and Schmidhuber (1997). Long short-term memory.
  - They describe a modified procedure for LSTMs

share cite improve this answer follow

edited Apr 13 '17 at 12:44 Community ♦ 1

answered Jun 22 '16 at 10:04 user20160 23.1k ● 1 ● 46 ● 74

This is an outstanding answer, and I wish I had the standing in this forum to award a substantial bounty to it. Especially useful are the concrete discussion of k1 vs k2 to contextualize my two cases against more general usage, and numeric examples of same. – Novak Jun 22 '16 at 23:14