Using RNNs

What do RNNs produce?

‣ Encoding of each word (each $h_i$) — can pass this to another layer to make a prediction (can also pool these to get a different sentence encoding)

‣ Encoding of the sentence (final $h/c$) — can pass this a decoder or make a classification decision about the sentence

‣ RNN can be viewed as a transformation of a sequence of vectors into a sequence of context-dependent vectors

RNN Uses

‣ Transducer: make some prediction for each element in a sequence
  
  output $y = \text{score for each tag, then softmax}$

  the movie was great

‣ Acceptor/encoder: encode a sequence into a fixed-sized vector and use that for some purpose

  predict sentiment (matmul + softmax)

  translate

  paraphrase/compress

  the movie was great

Multilayer Bidirectional RNN

‣ Sentence classification based on concatenation of both final outputs

‣ Token classification based on concatenation of both directions’ token representations

the movie was great
What do LSTMs return in PyTorch?

- **Hidden/cell** states are a 2-tuple, tensors of size 2x1xdim here.
- **Outputs** are a single tensor of size 4x1xdim here.

Training RNNs

- **Loss** = negative log likelihood of probability of gold label (or use SVM or other loss).
- Backpropagate through entire network, RNN parameters get a gradient update from each timestep.
- Example: sentiment analysis.

RNN Language Modeling

- **Loss** = negative log likelihood of probability of gold predictions, summed over the tags.
- Loss terms filter back through network.
- Example: language modeling (predict next word given context).
RNN Language Modeling

\[
P(w|\text{context}) = \frac{\exp(w \cdot h_i)}{\sum_{w'} \exp(w' \cdot h_i)}
\]

equivalent to

\[
P(w|\text{context}) = \text{softmax}(Wh_i)
\]

- \(W\) is a (vocab size) x (hidden size) matrix; linear layer in PyTorch (rows are word embeddings)

Training RNNLMs

- Input is a sequence of words, output is those words shifted by one,
- Allows us to train on predictions across several timesteps simultaneously (similar to batching but this is NOT what we refer to as batching)

Training RNNLMs

- Total loss = sum of negative log likelihoods at each position
- In PyTorch: simply add the losses together and call .backward()
Batched LM Training

- torch.nn.LSTM / torch.nn.GRU: expect input in [seq len, batch, word dim] format, or in [batch, seq len, word dim] if you set batch_first = True executed in parallel
- Input: [4, 2, dim]
- Cannot parallelize across timesteps of RNN since output depends on previous timesteps, so using larger batches gives better parallelism

Other Implementation Details

- torch.nn.Embedding: maps sequence of word indices to vectors
- Input: [126, 285] -> [[0.1, -0.07, 1.2], [-2.3, 0.2, 1.4]]
- Moves from [sequence length] vector of indices -> [seq len, dim] tensor or [batch, sequence length] matrix -> [batch, seq len, dim tensor]

LM Evaluation

- Accuracy doesn’t make sense — predicting the next word is generally impossible so accuracy values would be very low
- Evaluate LMs on the likelihood of held-out data (averaged to normalize for length)

\[ \frac{1}{n} \sum_{i=1}^{n} \log P(w_i|w_1, \ldots, w_{i-1}) \]

- Perplexity: exp(average negative log likelihood). Lower is better
- Suppose we have probs 1/4, 1/3, 1/4, 1/3 for 4 predictions
- Avg NLL (base e) = 1.242 Perplexity = 3.464 <= geometric mean of denominators

Visualizing LSTM LMs
Visualizing LSTMs

- Train character LSTM language model (predict next character based on history) over two datasets: War and Peace and Linux kernel source code
- Visualize activations of specific cells (components of c) to understand them
- Counter: know when to generate 
- Binary switch: tells us if we’re in a quote or not
- Stack: activation based on indentation

The tale of the crossing of the Dnieper lies in the fact that it plainly and indubitably proved the falsity of all the plans for cutting off the enemy’s retreat and the soundness of the only possible line of action — the one Kutuzov and the general mass of the army demanded: namely, simply to follow the enemy up. The French crowdpushed at a continually increasing speed and all its energy was directed to reaching its goal. It fled like a wounded animal and it was impossible to block its path. This was shown not so much by the arrangements it made for crossing as by what took place at the bridges, when the bridges broke down, unarmed soldiers, people from Moscow and women with children who were with the French transport, all — carried on by the inevitable — pressed forward into boats and into the ice-covered water and did not surrender.

Karpathy et al. (2015)
Visualizing LSTMs

- Train character LSTM language model (predict next character based on history) over two datasets: War and Peace and Linux kernel source code
- Visualize activations of specific cells to see what they track
- Uninterpretable: probably doing double-duty, or only makes sense in the context of another activation

State-of-the-art LMs

- Good LSTM LMs have ~27M params, 4-5 layers
- Kneser-Ney 5-gram model with cache: PPL = 125.7
- LSTM: PPL ~ 60-80 (depending on how much you optimize it)
- LSTM character-level: PPL ~1.5 (205 character vocab)
- Better language models using transformers (will discuss after MT)