Using RNNs

What do RNNs produce?

- **Encoding of the sentence** — can pass this a decoder or make a classification decision about the sentence
- **Encoding of each word** — can pass this to another layer to make a prediction (can also pool these to get a different sentence encoding)
- 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
  - DT NN VBD JJ
  - 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 directions’ token representations
- Token classification based on concatenation of both final outputs
**Training RNNs**

- Loss = negative log likelihood of probability of gold label (or use SVM or other loss)
- Backpropagate through entire network
- Example: sentiment analysis

**RNN Language Modeling**

- \( P(w|\text{context}) = \frac{\exp(w \cdot h_t)}{\sum_{w'} \exp(w' \cdot h_t)} \)
- equivalent to
- \( P(w|\text{context}) = \text{softmax}(W h_t) \)
- \( W \) is a (vocab size) \( \times \) (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 efficiently batch up training across time (one run of the RNN)

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

\[ \text{loss} = -\log P(w^*|\text{context}) \]

In PyTorch: simply add the losses together and call .backward()

Batched LM Training

- Why not one long chain?
- Batch dim
- torch.nn.LSTM / torch.nn.GRU: expect input in [seq len, batch, word dim] format
- Input: [4, 2, dim]
- Cannot parallelize across timesteps of RNN since output depends on previous timesteps
- Using larger batches is necessary to achieve maximum parallelism
**Other Implementation Details**

- `torch.nn.Embedding`: maps sequence of word indices to vectors
  - `[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

**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 `\n`

---

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
- Binary switch: tells us if we’re in a quote or not

“...You mean to imply that I have nothing to eat out of... on the contrary, I can supply you with everything even if you want to give dinner parties,” warmly replied Chichikov, who tried by every word he spoke to prove his own rectitude and therefore imagined Kutuzov to be enraged by the same desire.

Kotzov, shrugging his shoulders, replied with his subtle penetrating smile: “I meant merely to say what I said.”

Karpathy et al. (2015)

Stack: activation based on indentation

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)

Melis et al. (2017)