Stanford CS224N NLP with Deep Learning

Winter 2021

Lecture 6 – Simple and LSTM RNNs

Using RNNs

Training

Given a corpus $x_1, \ldots, x_T$, we compute the output distribution $y^{(t)}$ for every step t (the probability distribution for *each word*).
We can use a loss function like cross-entropy loss between the distributions and the true words.

This can be considered a form of teacher-forcing, since all data points use the first $n-1$ words from the corpus sequence and predict the n^{th} (and not the predicted sequence from x_1).

Computationally speaking, the gradient descent step is usually done in batches to avoid infeasibly large costs.

Gradient Descent

Recall, however, that in the computation of y, the weight matrix W_h occurs repeatedly. We need to find the derivative of $J(\theta)$ w.r.t this repeated matrix, which we can obtain as

$$\frac{\partial J}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J}{\partial W_h}\bigg|_i,$$

where i ranges over the occurrences of W_h.
Note that the gradients in the sum are *not* all equal, since the upstream gradient differs for each (coming from the next occurrence of W_h).

This algorithm is called *backpropagation through time*. It is sometimes *truncated*, *i.e.*, run up to a limited number of steps backwards.

Text Generation with RNN LMs

Like n-gram LMs, text is generated by RNN LMs by repeated sampling. The only difference is that sequential outputs must be passed back into the RNN to obtain the next ones.

RNNs can also be used for other tasks, like text classification, question answering, machine translation, and so on.

Evaluating LMs

Language models are generally evaluated by the *perplexity* metric, which is the geometric mean of the inverse probabilities

$$\text{perp} = \prod_{t=1}^{T} \left(\frac{1}{P(x^{t+1} \mid x^1 \cdots x^t)} \right)^{\frac{1}{T}},$$

which is equivalent to the exponential of the cross-entropy loss $\exp(J(\theta))$.

Problems with RNNs

RNNs frequently face the problem of *vanishing gradients*. When loss is back-propagated by the chain rule, if the successive partial derivatives are small, the gradient falls to zero.
For example, if

$$h^{(t)} = W_h h^{(t+1)} + W_x x^{(t)} + b_1,$$

then

$$\frac{\partial h^{(t)}}{\partial h^{(t+1)}} = W_h.$$

This means that

$$\frac{\partial J^{(i)}(\theta)}{\partial h^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \prod_{t=j+1}^{i} \frac{\partial h^{(t)}}{\partial h^{(t-1)}} = \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} W_h^{i-j}.$$

Now, if the sum of the eigenvalues λ_i of W_h is less than 1, then this expression reduces to

$$\sigma_{i=1}^{n} c_i \lambda_i^n q_i,$$

which will be negligible.

What this entails is that model weights are updated more w.r.t nearby effects and not long-term effects.

We might also face the opposite issue – *exploding gradients*. If the gradient is too large, then the update step becomes too big, which can cause bad updates. This is fixed by *gradient clipping*; if the gradient is higher than some threshold, it is scaled down before applying it.

Fixing the vanishing gradient problem means that RNNs need to be enabled to preserve information over many timesteps.

Long Short-Term Memory RNNs (LSTMs)

This is a type of RNN that was intended to solve the vanishing gradients problem.

It maintains two vectors at each time step t, the hidden state $h^{(t)}$ and the cell state $c^{(t)}$. Both of these are n-dimensional vectors. The cell state is intended to store long-term information.

First, the LSTM cell computes the forget, input and output gate values:

$$\begin{aligned} f^{(t)} &= \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f), \\ i^{(t)} &= \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i), \\ o^{(t)} &= \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o), \end{aligned}$$

which control what is forgotten from the cell state, what is added to the cell state, and what is added to the hidden state respectively.
Then, we find the new cell content

$$\tilde{c}^{(t)} = \tanh(W_c h^{(t-1)} + U_x^{(t)} + b_c),$$

and use this and the input vector to compute the new cell state

$$c^{(t)} = f^{(t)} * c^{(t-1)} + i^{(t)} * \tilde{c}^{(t)}$$

and from this, the new hidden state

$$h^{(t)} = o^{(t)} * \tanh(c^{(t)}).$$

The LSTM architecture makes it easier for the RNN to preserve information over many timesteps. In practice, this leads to a memory that lasts approximately 100 steps (as compared to about 7 for vanilla RNNs).

The hidden states can be considered as *contextual representations* of the words in the sentence. However, this architecture only incorporates the left context. *Bidirectional LSTMs* (or, more generally, RNNs) run two sequences in either direction and concatenate their hidden steps at each timestep. This is intended to create contextual representations incorporating context in both directions.