Q1 Recurrent Neural Networks

30 Points

The questions in this section are based on material covered in Week 13.

Q1.1

15 Points

Suppose we have the following small recurrent neural network:

$$h^{(t)} = \text{RELU}(w_1 x_1^{(t)} + w_2 x_2^{(t)} + u h^{(t-1)} + b_h)$$
$$y^{(t)} = \sigma(v h^{(t)} + b_y)$$

(The superscript notation (t) is used to avoid confusion when we also want to use subscripts for our parameters; $h^{(t)}$ means the same thing as h_t in the slides.)

Suppose we have the following loss function:

$$L(y) = -\ln(y)$$

What is the partial derivative of the loss $L(y^{(1)})$ with respect to the parameter w_1, ie. $\frac{\partial L(y^{(1)})}{\partial w_1}$, in this network? Show your work and simplify the expression as much as possible. You can assume that all values are positive for RELU purposes.

Given,

$$h^{(t)} = \text{RELU}(w_1 x_1^{(t)} + w_2 x_2^{(t)} + u h^{(t-1)} + b_h)$$
$$y^{(t)} = \sigma(v h^{(t)} + b_y)$$
$$L(y) = -\ln(y)$$

We can rewrite the network using new additional variables:

$$a^{(t)} = w_1 x_1^{(t)} \qquad b^{(t)} = w_2 x_2^{(t)}$$
$$c^{(t)} = U h^{(t-1)} \qquad d^{(t)} = a^t(t) + b^{(t)} + c^{(t)} + b_h$$
$$h^{(t)} = \text{RELU}(d^{(t)})$$

$e^{(t)} = vh^{(t)} \qquad f^{(t)} = e^{(t)} + b_y$
$y^{(t)} = \sigma(f^{(t)}) \qquad L(y) = -\ln(y^{(t)})$

$a^{(1)} = w_1 x_1^{(1)}, b^{(1)} = w_2 x_2^{(1)}, c^{(1)} = Uh^{(0)}$
$d^{(1)} = w_1 x_1^{(1)} + w_2 x_2^{(1)} + Uh^{(0)} + b_h$
$h^{(1)} = \text{RELU}(d^{(1)}) = w_1 x_1^{(1)} + w_2 x_2^{(1)} + Uh^{(0)} + b_h$
[Assuming all values are positive as given]

$e^{(1)} = vh^{(1)} = vw_1 x_1^{(1)} + vw_2 x_2^{(1)} + vUh^{(0)} + vb_h$
$f^{(1)} = e^{(1)} + b_y = vw_1 x_1^{(1)} + vw_2 x_2^{(1)} + vUh^{(0)} + vb_h + b_y$
$y^{(1)} = \sigma(f^{(1)}) = \frac{1}{1+e^{-f^{(1)}}}$

$L(y^{(1)}) = -\ln(y^{(1)})$

$\frac{\partial L(y^{(1)})}{\partial w_1} = \frac{\partial L(y^{(1)})}{\partial y^{(1)}} \cdot \frac{\partial y^{(1)}}{\partial f^{(1)}} \cdot \frac{\partial f^{(1)}}{\partial e^{(1)}} \cdot \frac{\partial e^{(1)}}{\partial h^{(1)}} \cdot \frac{\partial h^{(1)}}{\partial d^{(1)}} \cdot \frac{\partial d^{(1)}}{\partial a^{(1)}} \cdot \frac{\partial a^{(1)}}{\partial w_1}$ [using chain rule]

$$= -\frac{1}{y^{(1)}} \cdot \sigma(f^{(1)})(1 - \sigma(f^{(1)})) \cdot 1 \cdot v \cdot x_1^{(1)}$$

$$\frac{\partial L(y^{(1)})}{\partial w_1} = -\frac{v \cdot x_1^{(1)}}{y^{(1)}} \cdot \sigma(f^{(1)})(1 - \sigma(f^{(1)}))$$

How many paths are there in the computation graph of this network from the parameter w_1 to the loss $L(y^{(2)})$? What are the partial derivatives $\frac{\partial L(y^{(2)})}{\partial w_1}$ along each of those paths? Show your work and simplify the expressions as much as possible.

We can rewrite the network using new additional variables:

$a = w_1 x_1^{(1)} \qquad c = w_2 x_2^{(1)} \qquad d = Uh^{(0)}$
$f = a + c + d + b_h \qquad h^{(1)} = \text{RELU}(f)$

$e = w_1 x_1^{(2)} \qquad g = w_2 x_2^{(2)} \qquad j = Uh^{(1)}$

$$k = e + g + j + b_h \qquad h^{(2)} = \text{RELU}(k)$$

$$m = vh^{(2)} \qquad n = m + b_y$$
$$y^{(2)} = \sigma(n) \qquad L(y^{(2)}) = -\ln(y^{(2)})$$

There are two paths in the computation graph of this network from the parameter w_1 to the loss $L(y^{(2)})$ with different derivations of partial derivative $\frac{\partial L(y^{(2)})}{\partial w_1}$ for each path.

The first path is directly through e :

$$\frac{\partial L(y^{(2)})}{\partial w_1} = \frac{\partial L(y^{(2)})}{\partial y^{(2)}} \cdot \frac{\partial y^{(2)}}{\partial n} \cdot \frac{\partial n}{\partial m} \cdot \frac{\partial m}{\partial h^{(2)}} \cdot \frac{\partial h^{(2)}}{\partial k} \cdot \frac{\partial k}{\partial e} \cdot \frac{\partial e}{\partial w_1}$$ [using chain rule]

$$= -\frac{1}{y^{(2)}} \cdot \sigma(n)(1 - \sigma(n)) \cdot 1 \cdot v \cdot 1 \cdot 1 \cdot x_1^{(2)}$$

$$\frac{\partial L(y^{(2)})}{\partial w_1} = -\frac{v \cdot x_1^{(2)}}{y^{(2)}} \cdot \sigma(n)(1 - \sigma(n))$$

The second path is through $h^{(1)}$:

$$\frac{\partial L(y^{(2)})}{\partial w_1} = \frac{\partial L(y^{(2)})}{\partial y^{(2)}} \cdot \frac{\partial y^{(2)}}{\partial n} \cdot \frac{\partial n}{\partial m} \cdot \frac{\partial m}{\partial h^{(2)}} \cdot \frac{\partial h^{(2)}}{\partial k} \cdot \frac{\partial k}{\partial j} \cdot \frac{\partial j}{\partial h^{(1)}} \cdot \frac{\partial h^{(1)}}{\partial f} \cdot \frac{\partial f}{\partial a} \cdot \frac{\partial a}{\partial w_1}$$ [using chain rule]

$$= -\frac{1}{y^{(2)}} \cdot \sigma(n)(1 - \sigma(n)) \cdot 1 \cdot v \cdot 1 \cdot 1 \cdot U \cdot 1 \cdot 1 \cdot x_1^{(1)}$$

$$\frac{\partial L(y^{(2)})}{\partial w_1} = -\frac{U \cdot v \cdot x_1^{(1)}}{y^{(2)}} \cdot \sigma(n)(1 - \sigma(n))$$

If we run this network for three time steps, calculating the loss $L(y^{(t)})$ and performing standard stochastic gradient descent at each time step $t = 1, 2, 3$, how many total updates does the parameter w_1 receive? Explain your answer (3-4 sentences).

Total number of updated that parameter w_1 receives is 6(six).
Only input x_1 is with parameter w_1 in each time step. So in the first time step, calculating loss and performing standard stochastic gradient descent updates the parameter w_1 one time.
In the second and third time steps, this updates parameter w_1 two and three times respectively. Thus giving the total updates received to be equal to six.

(For this problem, it is not required to submit a picture of the computation graph, but it is highly recommended to draw one for yourself to aid in answering the questions.)

(HINT: You may find it easier to introduce additional variables to represent the outputs of individual operations, ie. the nodes in the computation graph. If you do so, clearly state what each new variable represents.)

Q1.2 Batched Recurrent Networks

5 Points

Recall that training using mini-batch gradient descent usually gives a better training time/performance trade-off than stochastic and full-batch gradient descent. A mini-batch consists of b training inputs that are combined into a single input tensor. For a feedforward network, for example, if a single training input is a vector of length n, then a mini-batch is a matrix of shape $(b \times n)$.

Suppose we want to use mini-batch gradient descent to train a recurrent neural network. That is, we have b training sequences, where each sequence is a matrix of shape $(l_i \times m)$, where l_i is the length of the sequence and m is your word embedding size. There is no guarantee that the input sequences are all the same length. How can we combine them into a single input tensor? Describe a strategy to do this (2-3 sentences). Your answer should address the following issues:

- What is the shape of the mini-batch input tensor, and what does each dimension of the tensor mean?
- How do you assign values to the tensor's entries, and how does this solve the problem of different sequence lengths?
- How does your strategy affect the network's predictions and parameter updates?

Using this strategy, the mini-batch input tensor will have shape = (b x maxSeqLen x m) where
b = batch size : iterates over batch training examples
m = word embedding size
maxSeqLen = maximum of all l_i's : length of longest sequence (so that all batch sequences can fit in tensor)

We can solve the problem of different sequence lengths by padding the shorted sequences with additional zeros. This way, all sequences will be of the same common dimension.
This strategy selects the last relevant output, hence it will not affect the network's predictions and parameter updates. Those positions will not affect the RNN computation on this input and will also make gradient for parameters at this position equal to 0.

Q1.3 Deep Recurrent Networks

5 Points

Suppose we have the following single-layer recurrent neural network:

$$\mathbf{h}_1^{(t)} = f_1(\mathbf{W}_1\mathbf{x}^{(t)} + \mathbf{U}_1\mathbf{h}_1^{(t-1)} + \mathbf{b}_1)$$
$$\mathbf{y}^{(t)} = f_y(\mathbf{V}\mathbf{h}_1^{(t)})$$

Unfortunately, this single-layer RNN is not giving good performance, so we want to improve it by adding a second layer and a residual connection from the output of the first layer to the output of the second layer. What are the equations for this new, two-layer RNN? Keep your variable names and notation as close as possible to those in the original network.

The supplied single-layer RNN's performance can be increased by stacking another layer, resulting in a deep RNN.
Some crucial points must be observed in order to get the equations of this two-layer deep RNN.
The first layer's hidden state information, as well as the current time step of the second layer, are used in the following time step.
Similarly, the 2nd layer's hidden state information is used in the next time step, as well as the current time step for output.

First layer hidden state equation:

$$h_1^{(t)} = f_1(W_1 x^{(t)} + U_1 h_1^{(t-1)} + b_1)$$

Second layer hidden state equation:

$$h_2^{(t)} = f_2(W_2 h_1^{(t)} + U_2 h_2^{(t-1)} + b_2)$$

Output equation:

$$y^{(t)} = f_y(V \cdot (h_1^{(t)} + h_2^{(t)}))$$

Q1.4 Bidirectional Recurrent Networks

5 Points

Suppose we have the following unidirectional recurrent neural network:

$$\overrightarrow{\mathbf{h}}_t = f_h(\mathbf{W}\mathbf{x}_t + \mathbf{U}\overrightarrow{\mathbf{h}}_{t-1} + \mathbf{b})$$
$$\mathbf{y}_t = f_y(\mathbf{V}\overrightarrow{\mathbf{h}}_t)$$

Unfortunately, this unidirectional, forward-only RNN is not giving good performance, so we want to improve it by adding a backward run of the same RNN to make the network bidirectional. What are the equations for this new, bidirectional recurrent network? Keep your variable names and notation as close as possible to those in the original network.

Converting the unidirectional, forward-only RNN into a bidirectional RNN can increase performance since we can process the input sequence in both directions in scenarios when inputs are accessible at all time-steps.

The equations to design such a bidirectional recurrent network will be as follows:

Equation for forward-run:

$$\overrightarrow{h_t} = f_h(W x_t + U \overrightarrow{h}_{t-1} + b)$$

Equation for backward-run:

$$\overleftarrow{h_t} = f_h(W x_t + U \overleftarrow{h}_{t+1} + b)$$

Hidden state concatenation and Final output prediction:

$$y_t = f_y(V \cdot (\overrightarrow{h_t} \oplus \overleftarrow{h_t}))$$

Q2 Attention Networks

15 Points

The questions in this section are based on material covered in Week 14.

Q2.1 Self Attention

5 Points

Suppose we have the following RNN encoder-decoder model:

$$\mathbf{h}^{(i)} = \tanh(\mathbf{W}_h\mathbf{x}^{(i)} + \mathbf{U}_h\mathbf{h}^{(i-1)})$$
$$\mathbf{d}^{(0)} = \mathbf{h}^{(n)}$$
$$\mathbf{e}_{enc}^{(ij)} = \mathbf{v}_{enc} \cdot \tanh(\mathbf{V}_{enc1}\mathbf{h}^{(i)} + \mathbf{V}_{enc2}\mathbf{d}^{(j-1)})$$
$$\mathbf{a}_{enc}^{(j)} = \text{softmax}(\mathbf{e}_{enc}^{(j)})$$
$$\mathbf{c}_{enc}^{(j)} = \sum_i \mathbf{a}_{enc}^{(ij)}\mathbf{h}^{(i)}$$
$$\mathbf{d}^{(j)} = \tanh(\mathbf{W}_d\mathbf{y}^{(j-1)} + \mathbf{U}_d\mathbf{d}^{(j-1)} + \mathbf{V}_d\mathbf{c}_{enc}^{(j)})$$

Unfortunately, this encoder-decoder with attention only from the decoder to the encoder is not giving good performance, so we want to improve it by adding self-attention to the decoder. What are the equations for this new, self-attentive encoder-decoder? Keep your variable names and notation as close as possible to those in the original network.

(HINT: Look at the diagram depicting self attention in slide deck 25.)

We can take attention to decoder the same way as attention to encoder. To make this new, self-attentive encoder-decoder, the equations are as follows:

$h^{(i)} = \tanh(W_h x^{(i)} + U_h h^{(i-1)})$
$d^{(0)} = h^{(n)}$

$e_{enc}^{(ij)} = v_{enc} \cdot \tanh(V_{enc1} h^{(i)} + V_{enc2} d^{(j-1)})$
$a_{enc}^{(j)} = \text{softmax}(e_{enc}^{(j)})$
$c_{enc}^{(j)} = \sum_i a_{enc}^{(ij)} h^{(i)}$

The new equations added are for $i \in [0, j-1]$:

$e_{dec}^{(ij)} = v_{dec} \cdot \tanh(V_{dec1} h^{(i)} + V_{dec2} d^{(j-1)})$
$a_{dec}^{(j)} = \text{softmax}(e_{dec}^{(j)})$
$c_{dec}^{(j)} = \sum_i a_{dec}^{(ij)} d^{(i)}$

For the decoder hidden state, the final calculations are made using:

$$d^{(j)} = \tanh(W_d y^{(j-1)} + U_d d^{(j-1)} + V_{d_1} c_{enc}^{(j)} + V_{d_2} c_{dec}^{(j)})$$

Q2.2 Multi-Head Attention

5 Points

Suppose we have a neural network with hidden size n that uses normal (single-head) attention. Unfortunately, this network is not giving good performance, so we want to improve it by switching to multi-head attention. How do we decide how many heads to use and how many dimensions each head should have? Are there any special considerations that we should be careful of when deciding these things? Briefly explain your answer (3-4 sentences).

Both the number of heads to use and the number of dimensions each head should have are hyperparameters that we need to tune.

The number of heads represents the types of different sorts of relevance (of a key/value to the query) that we wish to build for our system. These include relevance owing to status (singular/plural), direct dependency (key/value occurs as a result of query, or vice versa), and so on. The number of heads should be set such that it is not excessively high, as this increases the time required to train the algorithm. It should also not be too low, or the algorithm will be inefficient.

On the other hand, the dimensions of a head are learned by the algorithm itself. Since each head has its own W, U and V matrices that are multiplied by the query, key and value, the head can decrease or increase the weights on these dimensions according to what it has learned and generate optimal values.

Computation power and memory space can be addressed by the tuning of these hyperparameters in this way.

Also, hyperparameter tuning in neural net models involve retraining the entire network and running it on the development/validation set every time. To avoid this, simply use the hyperparameter values used in single-head, or use any other standard values.

Q2.3 Transformers and RNNs

5 Points

Recall that while a transformer-based encoder can be trained efficiently using parallelization, a transformer-based decoder is very slow and can't be parallelized. One way to take advantage of the transformer's strong performance without sacrificing too much computation time is to use a transformer-based encoder and an RNN decoder. How could you build such an encoder-decoder network (1-2 sentences)? Your answer should address the following:

- How do you initialize the decoder hidden state?
- How do you allow the decoder to copy words from the input sequence if needed?
- How do you train the encoder-decoder end-to-end?

This network can be designed in a way that allows parallelization at the encoder's end and implements RNN at the decoder's side. The hidden state in the RNN would be initialized to a base state, say 0, or set to the mean of encoder hidden states.

The decoder can generate words from the vocabulary or can copy them from the input using a coverage vector that helps to keep track of all the words that have been copied or translated, and attention weights to match translated words with those in the input. Attention simply needs the sequence of encoder hidden states.

The encoder-decoder model is trained end-to-end using back-propagation and gradient descent. The transformation or conversion functions used for initializing decoder states needs to be continuous and differentiable so that it doesn't break.

Q3 Late Penalty

0 Points

This question intentionally left blank.

Homework 5 Written

UNGRADED

STUDENT
Adithya Iyer

TOTAL POINTS
- / 45 pts

QUESTION 1

Recurrent Neural Networks		30 pts
1.1	(no title)	15 pts
1.2	Batched Recurrent Networks	5 pts
1.3	Deep Recurrent Networks	5 pts
1.4	Bidirectional Recurrent Networks	5 pts

QUESTION 2

Attention Networks		15 pts
2.1	Self Attention	5 pts
2.2	Multi-Head Attention	5 pts
2.3	Transformers and RNNs	5 pts

QUESTION 3

Late Penalty	0 pts