Homework 5 Solution

Fall 2019

Exercises

1. A convolution-based time delay network (TDN) applies convolution across a 1D temporal sequence. The convolution operation allows a network to share parameters across time but in a *shallow* manner. The output of convolution is a sequence where each member of the output is a function of a small number of *neighboring members* of the input. The idea of parameter sharing manifests in the application of the same convolution kernel at each time step.

   Recurrent neural networks (RNN) share parameters in a different way. Each member of the output is a function of the previous members of the output. Each member of the output is produced using the *same* update rule applied to the previous outputs. This recurrent formulation results in the sharing of parameters through a very *deep* computational graph. When an RNN is trained to predict the future from the past, it typically learns to use its hidden internal states to represent a lossy summary of task-relevant aspects of inputs up to the current time step.

   As a consequence of the differences between the two types of networks, their training speeds and representational powers are different. RNNs (at least the type-A RNNs discussed in lecture) can simulate Turing machines, while a TDN cannot. However, RNNs are also (generally speaking) slower to train, as TDNs' training process can be parallelized while an RNN cannot (for instance, for a type-A RNN, forward propagation for the whole input sequence has to be completed before backpropagation can be executed).

2. Following the notation used in the problem statement, we have the following expressions for the gradient of the loss w.r.t. the weight matrices $\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{V}$:

$$\nabla_{\boldsymbol{V}} \mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial o_i^{(t)}} \right) \nabla_{\boldsymbol{V}^{(t)}} o_i^{(t)} = \sum_t (\nabla_{\boldsymbol{o}^{(t)}} \mathcal{L}) \boldsymbol{h}^{(t-1)T} \tag{1}$$

$$\nabla_{\boldsymbol{W}} \mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} = \sum_t \operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) (\nabla_{\boldsymbol{h}^{(t)}} \mathcal{L}) \boldsymbol{h}^{(t-1)T} \tag{2}$$

$$\nabla_{\boldsymbol{U}} \mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} = \sum_t \operatorname{diag}\left(1 - (\boldsymbol{h}^{(t)})^2\right) (\nabla_{\boldsymbol{h}^{(t)}} \mathcal{L}) \boldsymbol{x}^{(t)T} \tag{3}$$

3. (i) Figure 1 illustrates the difference between the teacher forcing method's training and test time setups with an example having input sequence with two time steps.

   We see that at time $t = 2$, the model is trained to maximize the conditional probability of $y^{(2)}$ given both the input $\boldsymbol{x}$ sequence so far and the previous $\boldsymbol{y}$ value from the training set. Maximum likelihood thus specifies that during training, rather than feeding the model's own output back into itself, these connections should be fed with the target values specifying what the correct output should be.

   When the model is deployed, the true output is generally *not* known. So at test time, we approximate the correct output $\boldsymbol{y}^{(t)}$ with the model's output $\boldsymbol{o}^{(t)}$, and feed the output back into the model.

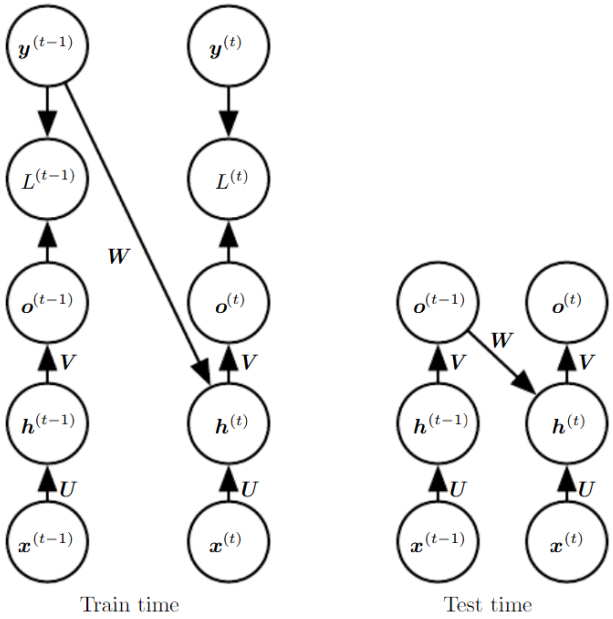

Figure 1: Teacher Forcing

(ii) The disadvantage of strict teacher forcing arises if the network is going to be later used in a *closed-loop* mode, with the network outputs (or samples from the output distribution) fed back as input. In this case, the fed-back inputs that the network sees during training could be quite different from the kind of inputs that it will see at test time. One way to mitigate this problem is to train with both *teacher-forced* inputs and *free-running* inputs, for example by predicting the correct target a number of steps in the future through the unfolded recurrent output-to-input paths. In this way, the network can learn to take into account input conditions (such as those it generates itself in the free-running mode) not seen during training and how to map the state back toward one that will make the network generate proper outputs after a few steps. Another approach to mitigate the gap between the inputs seen at training time and the inputs seen at test time randomly chooses to use generated values or actual data values as input. This approach exploits a curriculum learning strategy to gradually use more of the generated values as input.

(iii) The method itself originates from the method of maximum likelihood, in which during training the model receives the ground truth output $y^{(t)}$ as input at time $t+1$. We can see this by examining a sequence with two time steps. The conditional maximum likelihood criterion is

$$log\, p(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} | \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}) \tag{4}$$

$$= log\, p(\boldsymbol{y}^{(2)} | \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}) + p(\boldsymbol{y}^{(1)} | \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}) \tag{5}$$

4. (i) The parameter sharing used in RNN is equivalent to assuming that the conditional distribution of the hidden layer state variables at time $t+1$ given their values at time t is *stationary*.

(ii) The three mechanisms:

(I) We can use self-delimiting neural networks (SLIM RNNs) for computationally efficient inference of large neural networks by using the activity of the "halt neuron" to monitor the change of weights at the start of a sequence, and then focusing on only a small part of the network. For training a SLIM RNN, the special "halt" symbol is inserted at the end of each sequence.

(II) Another option is to introduce an extra Bernoulli output to the model that represents the decision to either continue generation or halt generation at each time step. This approach is more general than the approach of adding an extra symbol to the vocabulary, because it may be applied to any RNN, rather than only RNNs that output a sequence of symbols. For

example, it may be applied to an RNN that emits a sequence of real numbers. The new output unit is usually a sigmoid unit trained with the cross-entropy loss. In this approach the sigmoid is trained to maximize the log-probability of the correct prediction as to whether the sequence ends or continues at each time step.

(III) Another way to determine the sequence length τ is to add an extra output to the model that predicts the integer τ itself. The model can sample a value of τ and then sample τ steps worth of data. This approach requires adding an extra input to the recurrent update at each time step so that the recurrent update is aware of whether it is near the end of the generated sequence. This extra input can either consist of the value of τ or can consist of $\tau - t$, the number of remaining time steps. Without this extra input, the RNN might generate sequences that end abruptly, such as a sentence that ends before it is complete. This approach is based on the decomposition

$$P(\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(\tau)}) = P(\tau)P(\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(\tau)}|\tau). \tag{6}$$

5. The block diagram of an LSTM cell is shown in Figure 2.

LSTM recurrent networks have LSTM cells (as shown in the block diagram) that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but also has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_i^{(t)}$, which has a linear self-loop similar to leaky units. Here, however, the self-loop weight (or the associated time constant) is controlled by a *forget gate* unit $f_i^{(t)}$ (for time step t and cell i), which sets this weight to a value between 0 and 1 via a sigmoid unit:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right), \tag{7}$$

where $\boldsymbol{x}^{(t)}$ is the current input vector and $\boldsymbol{h}^{(t)}$ is the current hidden layer vector, containing the outputs of all the LSTM cells, and $\boldsymbol{b}^f, \boldsymbol{U}^f, \boldsymbol{W}^f$ are respectively biases, input weights, and recurrent weights for the forget gates. The LSTM cell internal state is thus updated as follows, but with a conditional self-loop weight $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right), \tag{8}$$

where $\boldsymbol{b}, \boldsymbol{U}$ and $\boldsymbol{W}$ respectively denote the biases, input weights, and recurrent weights into the LSTM cell. The external *input gate* unit $g_i^{(t)}$ is computed similarly to the forget gate (with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right). \tag{9}$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off, via the *output gate* $q_i^{(t)}$, which also uses a sigmoid unit for gating:

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)}, \tag{10}$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right), \tag{11}$$

which has parameters $\boldsymbol{b}^o, \boldsymbol{U}^o, \boldsymbol{W}^o$ for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s_i^{(t)}$ as an extra input (with its weight) into the three gates of the i-th unit, as shown in Figure 2. This would require three additional parameters.

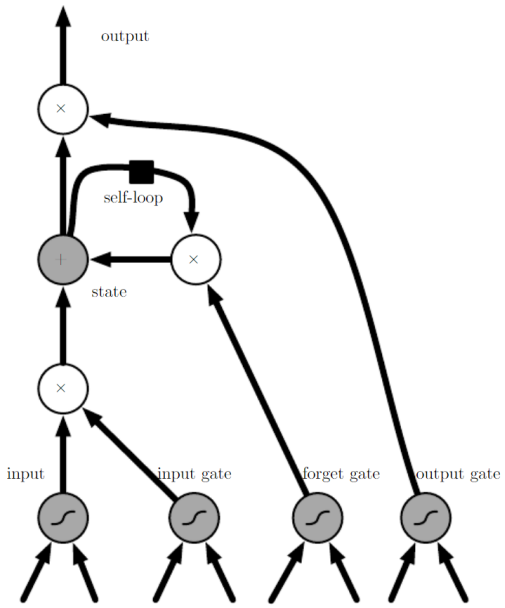

Figure 2: LSTM Cell Block Diagram

6. (i) Bidirectional RNNs were invented to address the need of having the model prediction $\boldsymbol{y}^{(t)}$ that may depend on the whole input sequence. For example, in speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of co-articulation and may even depend on the next few words because of the linguistic dependencies between nearby words: if there are two interpretations of the current word that are both acoustically plausible, we may have to look far into the future (and the past) to disambiguate them. This is also true of handwriting recognition and many other sequence-to-sequence learning tasks.

   (ii) Figure 3 illustrates the computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences $\boldsymbol{x}$ to target sequences $\boldsymbol{y}$, with loss $L^{(t)}$ at each step t. The $\boldsymbol{h}$ recurrence propagates information forward in time (toward the right), while the $\boldsymbol{g}$ recurrence propagates information backward in time (toward the left). Thus at each point t, the output units $\boldsymbol{o}^{(t)}$ can benefit from a relevant summary of the past in its $\boldsymbol{h}(t)$ input and from a relevant summary of the future in its $\boldsymbol{g}^{(t)}$ input.

   (iii) A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. A variably-sized sequence $x^{(1)}, ..., x^{(t)}$ can be mapped to a fixed-sized representation (the output $\boldsymbol{o}$), with a fixed set of parameters (the weight matrices $\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}$). Figure 4 illustrates a supervised learning case in which some target $\boldsymbol{y}$ is provided that is associated with the whole sequence.

   (iv) One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length τ, the depth (measured as the number of compositions of nonlinear operations) can be drastically reduced from τ to $\mathcal{O}(log\tau)$, which might help deal with long-term dependencies. An open question is how to best structure the tree. One option is to have a tree structure that does not depend on the data, such as a balanced binary tree. In some application domains, external methods can suggest the appropriate tree structure. For example, when processing natural language sentences, the tree structure for the recursive network can be fixed to the structure of the parse tree of the sentence provided by a natural language parser. Ideally, one would like the learner itself to discover and infer the tree structure that is appropriate for any given input.

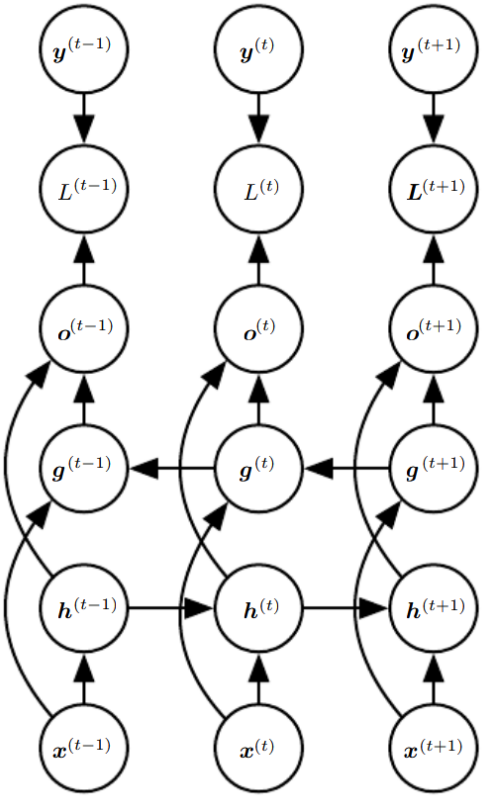

Figure 3: Bidirectional RNN Example

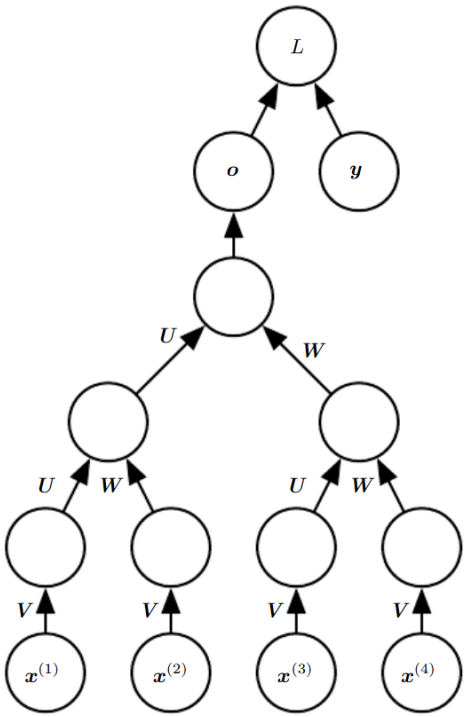

Figure 4: Recursive Neural Network Example