ECE-GY 9123 / Spring 2021

Homework 4 solutions

1. **(3 points)** *RNNs for images*. In this exercise, we will develop an RNN-type architecture for processing multi-dimensional data (such as RGB images). Here, hidden units are themselves arranged in the form of a grid (as opposed to a sequence). Each hidden unit is connected from the corresponding node from the input layer, as well as recurrently connected to its "top" and "left" neighbors. Here is a picture:

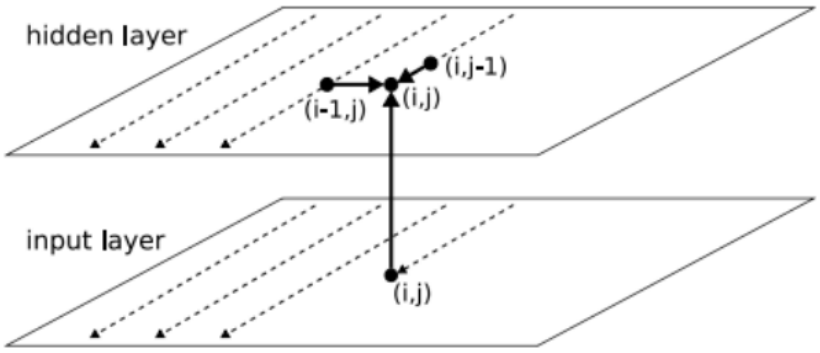

Figure 1: 2D RNNs

The equation for the hidden unit is given as follows (assume no bias parameters for simplicity):

$$h^{i,j} = \sigma\left(W_{in}x^{i,j} + W_{left}h^{i-1,j} + W_{top}h^{i,j-1}\right).$$

   a. Assume that the input grid is $n \times n$, each input $x^{i,j}$ is a c-channel vector, and the hidden state $h^{i,j}$ has dimension h. How many trainable parameters does this architecture have? You can assume zero padding as needed to avoid boundary effects.

   b. How many arithmetic operations (scalar adds and multiplies) are required to compute all the hidden activations? You can write your answer in big-Oh notation.

   c. Compare the above architecture with a regular convnet, and explain advantages/disadvantages.

Solution

   a. $2h^2 + hc$ (the weight matrix mapping input to hidden state is a $c \times h$ matrix, and the weight matrix mapping hidden states is $h \times h$).

   b. Each hidden state is defined in terms of 3 matrix-vector products, hence takes $O(h^2 + hc)$ scalar adds and multiplies to compute. Therefore, the overall number of trainable parameters is $O(h^2n^2 + hcn^2)$.

   c. Pros of 2D-RNN over convnets: i) Directionality (so extracted features are naturally spatially sensitive to top-to-down or left-to-right behavior; this could be either a pro or a

con depending on how you argue it). ii) Outputs are context-sensitive. Suitable for tasks such as image completion.

Cons: Much harder to train (due to unrolling). Not easily parallelizable (also due to unrolling).

2. **(1 points)** *Attention! My code takes too long.* In class, we showed that a regular self-attention layer takes $O(T^2)$ time to evaluate for an input with T tokens. Propose a way to reduce this running time to, say, $O(T)$, and comment on its possible pros vs cons.

   Solution

   There is no unique answer. One common approach is to perform some kind of *windowing*, where each token attends to all its *previous* tokens, or to a sliding window consisting of its previous t tokens, or to some fixed window of size w. The latter two options decrease the running time to $O(T)$ if the window size is small. See the last two patterns.

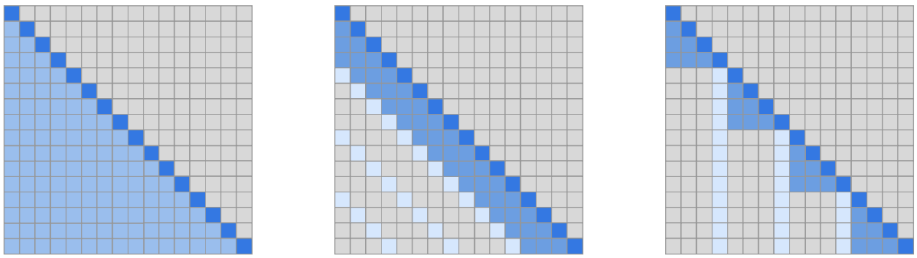

Figure 2: Attention patterns ($t = 3, w = 4$)

   Pros and cons: Easier to compute for sure. But loss of long term context/dependencies.

3. **(2 points)** *Making skip-gram training practical*. Suppose we have a dictionary containing d words, and we are trying to learn skip-gram embeddings of each word using a very large training corpus. Suppose that our embedding dimension is chosen to be h.

   a. Calculate the gradient of the cross-entropy loss for a *single* target-context training pair of words. What is the running time of calculating this gradient in terms of d and h?

   b. Here is a second approach. Argue that the gradient from part a can be written as a weighted average of some number of terms, and intuitively describe a *sampling*-based method to improve the running time of the gradient calculation.

   Solution

   a. The cross-entropy loss (as derived in class) for word pair (i, j) can be written as:

$$l = u_j^T v_i - \log\left(\sum_k \exp(u_k^T v_i)\right).$$

   Therefore, the gradient of l is the (partial) derivative with respect to all the u's and all the v's. Now, it is clear that l depends on v_i (and not the other columns of V), so we only need to calculate $\frac{\partial l}{\partial v_i}$. If $\hat{y}$ is the network prediction (calculated during the forward pass),

then the gradient can be written as:

$$\begin{aligned}\frac{\partial l}{\partial v_i} &= u_j - \frac{\sum_k \exp(u_k^T v_i) u_k}{\sum_k \exp(u_k^T v_i)} \\ &= u_j - U\text{softmax}(U v_i) \\ &= u_j - U\hat{y}.\end{aligned}$$

Now, for any $j' \neq j$, the gradient of l with respect to u_k can be written in terms of the backprop equation:

$$\begin{aligned}\frac{\partial l}{\partial u_{j'}} &= -\frac{\sum_k \exp(u_k^T v_i) v_i}{\sum_k \exp(u_k^T v_i)} \\ &= -V\text{softmax}(U v_i),\end{aligned}$$

and for the remaining term, we can write

$$\frac{\partial l}{\partial u_j} = v_i - V\text{softmax}(U v_i),$$

The running time is dominated by the softmax calculation in the gradient with respect to v_i, which takes $O(dh)$ time.

b. The gradient with respect to U can be written as:

$$\Delta = u_j - U\hat{y} = U(y - \hat{y}) = \frac{1}{d}\sum_{i=1}^{d} d(y_i - \hat{y}_i)u_i.$$

This is an average over d terms, and therefore (just as in SGD) can be approximated by uniform sampling. Therefore, choose a random subset $S \subset [1, \ldots, d]$ and write

$$\Delta \approx \frac{1}{|S|}\sum_{i \in S} d(y_i - \hat{y}_i)u_i.$$

This is an unbiased estimate of the gradient. If S is constant size this can be calculated in $O(1)$ time.

4. **(4 points)** Open the (incomplete) Jupyter notebook provided as an attachment to this homework in Google Colab (or other cloud service of your choice) and complete the missing items. Save your finished notebook in PDF format and upload along with your answers to the above theory questions in a single PDF.

   Solution

   A solution notebook has been posted on NYUClasses.