CSE 4502/5717 Big Data Analytics

Lecture 20, April 9, 2018

1 The need for minibatches:

One of the reasons for partitioning the input into minibatches is to be able to update the network parameters frequently. Another reason is to employ matrix-matrix multiplications (instead of matrix-vector multiplications).

Consider levels $(l-1)$ and level l in a feed-forward network, $(2 \leq l \leq L)$. Let the number of neurons in each level be n. Let $\boldsymbol{a}^l = (a_1^l \ a_2^l \ \ldots \ a_n^l)^T$ be the activation vector for the nodes in level l (for $2 \leq l \leq L$). We have shown that $\boldsymbol{a}^l = \boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$ (for $2 \leq l \leq L$). Here $\boldsymbol{b}^l = (b_1^l \ b_2^l \ \ldots \ b_n^l)^T$ and

$$\boldsymbol{W}^l = \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots & w_{1n}^l \\ w_{21}^l & w_{22}^l & \cdots & w_{2n}^l \\ \vdots & & & \\ w_{n1}^l & w_{n2}^l & \cdots & w_{nn}^l \end{bmatrix}.$$

In other words, given $\boldsymbol{a}^{l-1}$, we can compute $\boldsymbol{a}^l$ using a matrix-vector multiplication (and a vector-vector addition that takes relatively much less time).

Note that the above discussion assumes that we process one example at a time. Let b be the size of the minibatch. When we move from level $(l-1)$ to level l, we can compute $\boldsymbol{a}^l$ values for all the b examples in the minibatch together. Let the examples in the minibatch be $E_1, E_2, \ldots, E_b$. Also let the activation vector for level l and example i be denoted as $\boldsymbol{a}^{(l,i)}$, for $2 \leq l \leq L$ and $1 \leq i \leq b$.

Given $\boldsymbol{a}^{(l-1,1)}, \boldsymbol{a}^{(l-1,2)}, \cdots, \boldsymbol{a}^{(l-1,b)}$, we can compute $\boldsymbol{a}^{(l,1)}, \boldsymbol{a}^{(l,2)}, \cdots, \boldsymbol{a}^{(l,b)}$ using a matrix-matrix multiplication. Specifically, if

$$\boldsymbol{A}^l = \begin{bmatrix} \boldsymbol{a}^{(l,1)} & \boldsymbol{a}^{(l,2)} & \cdots & \boldsymbol{a}^{(l,b)} \end{bmatrix},$$

and

$$\boldsymbol{B}^l = \begin{bmatrix} \boldsymbol{b}^{(l,1)} & \boldsymbol{b}^{(l,2)} & \cdots & \boldsymbol{b}^{(l,b)} \end{bmatrix},$$

for $2 \leq l \leq L$, then, $\boldsymbol{A}^l = \boldsymbol{W}^l \times \boldsymbol{A}^{l-1} + \boldsymbol{B}^l$.

Given $\boldsymbol{A}^{l-1}$, we can compute $\boldsymbol{A}^l$ by multiplying the $(n \times n)$ matrix $\boldsymbol{W}^l$ and the $(n \times b)$ matrix $\boldsymbol{A}^{l-1}$. The results will be a $(n \times b)$ matrix that can be added with $\boldsymbol{B}^l$ in $O(nb)$ time. The matrix-matrix multiplication can be done in $O(n^2 b)$ time using a simple algorithm yielding a total run time of $O(n^2 b)$.

However, we can do better using any of the fast known algorithms for matrix multiplication. For example, we can use the fact that we can multiply two $k \times k$ matrices in $O(k^{2.373})$ time as follows: Let $q = \frac{n}{b}$. Partition the matrix $\boldsymbol{W}^l$ as:

$$\begin{bmatrix} \boldsymbol{W}_{11}^l & \boldsymbol{W}_{12}^l & \cdots & \boldsymbol{W}_{1q}^l \\ \boldsymbol{W}_{21}^l & \boldsymbol{W}_{22}^l & \cdots & \boldsymbol{W}_{2q}^l \\ \vdots & & & \\ \boldsymbol{W}_{q1}^l & \boldsymbol{W}_{q2}^l & \cdots & \boldsymbol{W}_{qq}^l \end{bmatrix}$$

where $\boldsymbol{W}_{ij}$ is of size $(b \times b)$, for $1 \leq i, j \leq q$.

Also, partition $\boldsymbol{A}^{l-1}$ as:

$$\begin{bmatrix} \boldsymbol{A}_1^{l-1} \\ \boldsymbol{A}_2^{l-1} \\ \vdots \\ \boldsymbol{A}_q^{l-1} \end{bmatrix}$$

where each $\boldsymbol{A}_i^{l-1}$ is of size $(b \times b)$, for $1 \leq i \leq q$.

The product of $\boldsymbol{W}^l$ and $\boldsymbol{A}^{l-1}$ can now be seen to be:

$$\begin{bmatrix} \sum_{j=1}^{q} \boldsymbol{W}_{1j}^{l} \times \boldsymbol{A}_{j}^{l-1} \\ \sum_{j=1}^{q} \boldsymbol{W}_{2j}^{l} \times \boldsymbol{A}_{j}^{l-1} \\ \vdots \\ \sum_{j=1}^{q} \boldsymbol{W}_{qj}^{l} \times \boldsymbol{A}_{j}^{l-1} \end{bmatrix}.$$

In other words, we have to perform q^2 matrix-matrix multiplications. Each such matrix-matrix multiplication involves two $b \times b$ matrices. Thus the total time needed to compute $\boldsymbol{A}^l$, given $\boldsymbol{A}^{l-1}$, is $O(q^2 M(b))$, where $M(b)$ is the time needed to multiply two $b \times b$ matrices. If $M(b) = O(b^{2.373})$, then the above run time is $O(n^2 b^{0.373})$. This is theoretically better than the $O(n^2 b)$ run time that the simple algorithm will take. However, in practice, if b is not large enough then the simple algorithm could take less time.

2 Different kinds of neural networks

Several kinds of NNs have been identified by scientists. Some of these are: Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Generative Adversarial Networks (GANs).

2.1 Convolutional Neural Networks

A CNN has at least one layer in which convolutions are employed (instead of affine transformations).

If $x(.)$ and $k(.)$ are two continuous signals, the convolution of these two signals is defined as

$$s(t) = (x * k)(t) = \int_{-\infty}^{\infty} x(a)\ k(t-a)\ da.$$

If $x(.)$ and $k(.)$ are discrete, then the convolution operation is defined as:

$$s(t) = (x * k)(t) = \sum_{a=-\infty}^{\infty} x(a)\ k(t-a).$$

In the case of a CNN, $x(.)$ will be the input and $k(.)$ will have to be learnt. We refer to $k(.)$ as the kernel.

If the input is an image $I(.,.)$ we can convolve it with a kernel $K(.,.)$ to get:

$$S(i,j) = \sum_{m} \sum_{n} I(m,n)\ K(i-m, j-n).$$

Consider the following input:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

Let the kernel be:

$$\begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}.$$

In this case there will be 4 outputs one corresponding to each of a_{11}, a_{12}, a_{21}, and a_{22}, respectively. $O_{11} = a_{11}k_{11} + a_{12}k_{12} + a_{21}k_{21} + a_{22}k_{22}$; $O_{12} = a_{12}k_{11} + a_{13}k_{12} + a_{22}k_{21} + a_{23}k_{22}$; $O_{21} = a_{21}k_{11} + a_{22}k_{12} + a_{31}k_{21} + a_{32}k_{22}$; and $O_{22} = a_{22}k_{11} + a_{23}k_{12} + a_{32}k_{21} + a_{33}k_{22}$. See Figure 1.

A CNN has three important properties: 1) Sparsity; 2 Parameter sharing; and 3) Equivariance.

Sparsity: The above CNN is sparse relative to a complete bipartite graph. For instance, the out-degree of each node of level 1 will be 4 in a traditional network. However in a CNN this is not

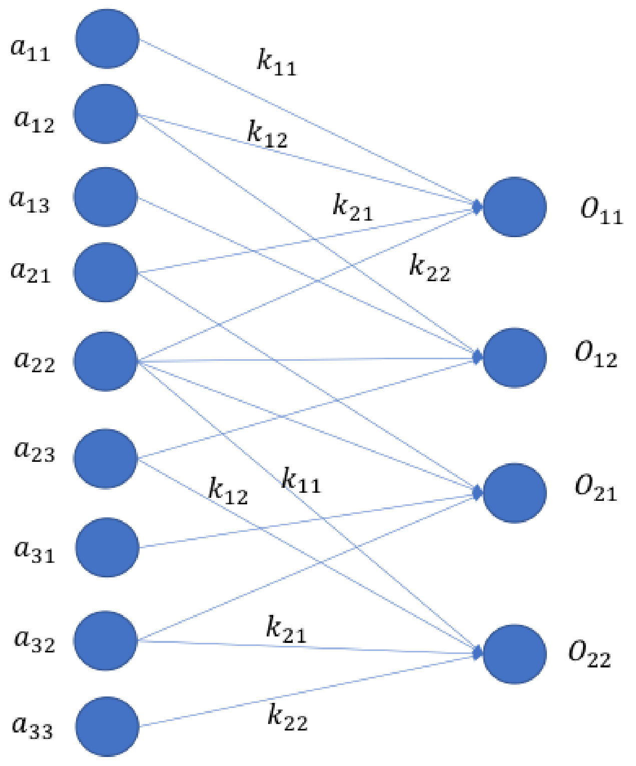

Figure 1: An example of a CNN. Not all the edge labels are shown.

the case. For example the out-degree of the node a_{11} is only one. Also, the in-degree of each node of level 2 in a traditional network is 9 and in the above CNN it is only 4.

Parameter Sharing: In the case of a traditional NN, each edge from level 1 to level 2 will have a unique weight. However in the CNN, edge weights are duplicated. All the edge weights are coming from the same kernel.

Equivariance: We say that a function $f(.)$ is equivariant with respect to another function $g(.)$, if $f(g(x)) = g(f(x))$, for all x. The convolution operation is equivariant with the shift operation. For instance if an object in an image is shifted, then its representation after convolution will shift by the same amount.

There are typically three layers in a CNN that are shown in Figure 2.

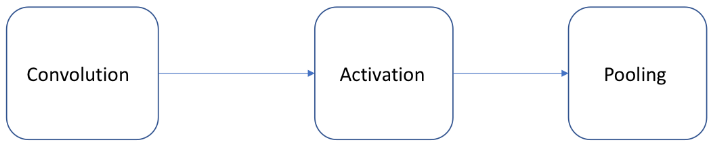

Figure 2: Layers in a CNN.

The first layer corresponds to the convolution operation. In the second layer an activation function is applied. In the pooling stage, the output of any node is replaced with a summary statistic of nearby outputs. This statistic, e.g., could be an average in a rectangular neighborhood. Other possibilities for this statistic include maximum, minimum, the L^2 norm, etc. Pooling is done to ensure invariance to small translations in the input. Practical CNNs could have millions of parameters.

2.2 Recurrent Neural Networks (RNNs)

In the case of a CNN, parameter sharing happens with the use of the same kernel to determine edge weights. In the case of a RNN, parameter sharing happens with recursion. Specifically, the output values at any point in time could depend on the output values from the previous time step. A RNN can be represented as a computational graph as shown in Figure 3.

In this figure, x stands for the input. An affine transformation U is applied on x and the transformed input serves as the input to the hidden layer h. An activation function could be applied at the hidden layer. The output from h is also fed back to h after an affine transformation V and after one unit of time delay (denoted as d in Figure 3). The output from h undergoes an affine transformation W before it reaches the output layer o.

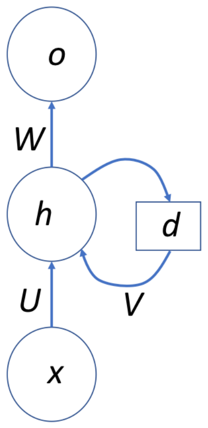

Figure 3: Computational graph of a RNN. In this figure d refers to unit delay.

A computational graph can be expanded as shown in Figure 4. In this figure, the output of the node labelled h^t is $\sigma(UX^t + Vh^{t-1} + b^t)$. We can train a RNN using back propagation after expanding it. CNNs have typically been employed for image applications whereas RNNs have been applied for sequence data.

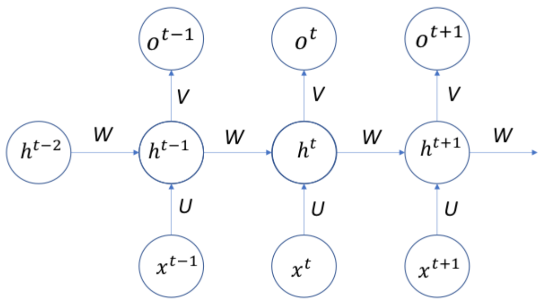

Figure 4: Expanding a computational graph corresponding to a RNN.

2.3 Generative Adversarial Networks (GANs)

A GAN has two networks: a generator and a discriminator. The generator learns to generate instances from a true data generating distribution and the discriminator evaluates the instances generated by the generator. They work in a competitive zero sum game framework. The generator is typically a deconvolutional network and the discriminator is a CNN. Both can be trained with back propagation. The discriminator is trained with examples whereas a generator is seeded with a random input.

GANs can be used to generate photorealistic images and have been used in video games, interior design, etc.