Deep Learning

MInDS @ Mines

Sequence Data

So far we've considered data that can be described as a whole or we have described data as a whole. Here "describing as a whole" refers to the existence of certain attributes in the data without consideration of any order in which the data appears. This approach has been effective so far with most of our applications but in some applications, such as text, the order in which the data appears has an impact on the resulting output. We would therefore like to include the order in which the data appears in some way. Data that includes an order representation of its components is called *sequence data*. A simple way of representing sequence data is by identifying all the possible values for a given component and then using a boolean vector to denote which of those values exists. For example, if our dataset contains only four characters, "ATCG", we can represent the first sequence in our data, "AACG" as,

$$x_1 = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, x_2 \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, x_3 = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, x_4 \begin{bmatrix}0\\0\\0\\1\end{bmatrix}, \quad (1)$$

where the first value represents whether the character is an "A", second is "T", third is "C", and fourth is "G" and we denote the i^{th} component of the data as x_i.[1]

[1] This can extend to words in text where we are aware of all words in the text's vocabulary.

This sequence representation is great in representing our data but now it is more complex in working with our previously used machine learning models. For an example in natural language processing, we can represent a sentence as a list of these vectors but each sentence will have a different number of words and therefore and different vector size which makes them incompatible with previously covered methods.

Recurrent Neural Networks

Recurrent Neural Networks (RNN)s are neural networks where the connections between neurons can pass over a sequence. This allows the neural network to "remember" previous occurrences of data by passing the results from the current state into the future run. The result from x_i is passed in to compute the result for x_{i+1}.

A recurrent unit can produce two outputs; a, the activation output, and h, the hidden output that can be passed on to the next hidden layer or can be the predicted target y. We will refer to h and y interchangeably. A single RNN will also store three weight matrixes; W_{ax}, W_{aa}, and W_{ha}, where W_{ij} represents the weight matrix applied to i to determine j. We can calculate

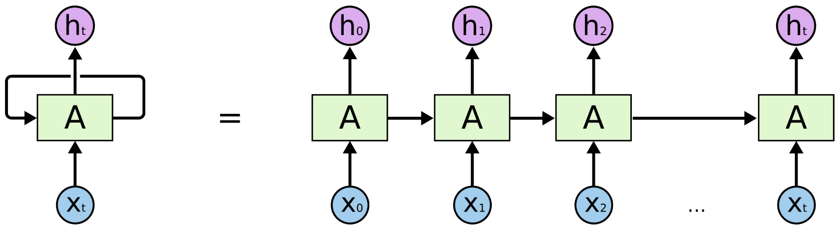

Figure 1: A graphical depiction of how an RNN works. Note the indexes on the x's here denote the order in the sequence of a single data sample. Source

the resulting values of a recurrent unit as,

$$\mathsf{a}_t = f(\mathsf{W}_{ax}\mathsf{x}_t + \mathsf{W}_{aa}\mathsf{a}_{t-1} + \mathsf{b}_a), \tag{2}$$

$$\mathsf{h}_t = f(\mathsf{W}_{ha}\mathsf{a}_t + \mathsf{b}_h). \tag{3}$$

Since the activation output of the layer applied to the first item in the sequence is passed in as an input to the calculation of the second item's activation, we are now utilizing the sequence order in our model. Note that the three weight matrixes are shared across all values. This allows the network to learn a pattern that depends on the order of the words or relationship between them regardless of where they generally appear in the sequence.

f is the activation function for the layer which we usually set to the hyperbolic tangent function for RNNs.

RNNs can handle data that appears in a sequence but this is not limited to the input data. If we refer to a sequence as "many" and an individual data point as "one", an RNN can receive one input and produce one output, it can receive one input and produce a sequence of many values, receive many inputs and output one value, or receive many inputs and output many values. Some example applications of RNNs in these areas are,

A simpler variation of RNNs exists that does not distinguish between h and a and instead sets $h = a$ resulting in,

$$\mathsf{h}_t = f(\mathsf{W}_{hx}\mathsf{x}_t + \mathsf{W}_{hh}\mathsf{h}_{t-1} + \mathsf{b}_h),$$

$$\mathsf{W}_{hx} = \mathsf{W}_{ax}, \mathsf{W}_{hh} = \mathsf{W}_{ha}$$

- One to one: image classification
- One to many: image captioning
- Many to one: Sentiment analysis of text
- Many to many: Text prediction or text translation

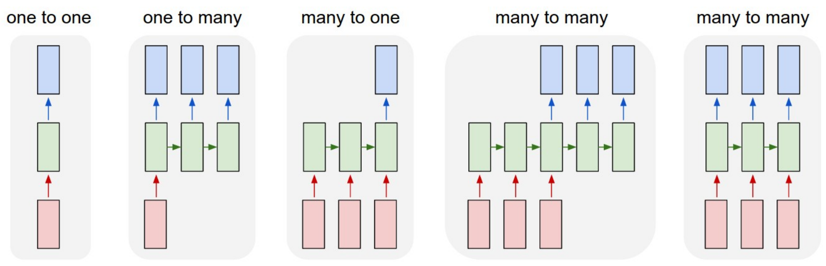

Figure 2: A visual representation of the various types of RNNs. Source

We can create deep RNNs by stacking the layers as we would with any other neural network type. This can improve the performance of the RNN.

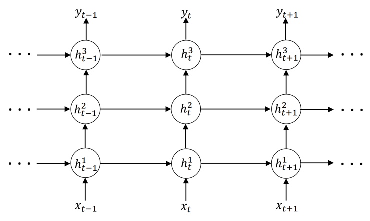

Figure 3: A visual representation of a deep RNN. Source

The Vanishing Gradient Problem

When a neural network contains a sufficiently large number of hidden layers, applying backpropagation to train the network will result in the weights of the layers closest to the output being updated by a larger portion than those at the beginning of the network. The vanishing gradient problem is where the weights of the initial layers are not updated significantly due to the gradient being very low by the time it propagates that far back.

When we train an RNN we program it by unfolding the network to appear like a feed forward neural network. [2] This results in an RNN being naturally a very deep network and it is susceptible to the vanishing gradient problem. To handle the vanishing gradient problem, we have two commonly used approaches that adapt the recurrent unit; Gated Recurrent Units (GRUs) and Long Short Term Memory cells (LSTMs).

[2] Applying backpropogation this way is referred to as backpropogation through time.

Another issue that both GRUs and LSTMs handle is when we have a long sequence, the basic RNN can't remember information from that far back.

Gated Recurrent Units (GRUs)

Gated Recurrent Units introduce two gates; the update gate and the reset gate, which control how we modify the hidden output being passed between sequence values. The gates learn values that allow us to store historical values for a longer period in the sequence. The update gate controls how much to update the passed through values whereas the reset gate controls how to reset the passed through value based on the current input. We represent the values for the update and reset gate respectively as,

$$\mathsf{z}_t = f(\mathsf{W}_{zh}\mathsf{h}_{t-1} + \mathsf{W}_{zx}\mathsf{x}_t + \mathsf{b}_z), \tag{4}$$

$$\mathsf{r}_t = f(\mathsf{W}_{rh}\mathsf{h}_{t-1} + \mathsf{W}_{rx}\mathsf{x}_t + \mathsf{b}_r). \tag{5}$$

f here is usually the sigmoid function. We can see that they are calculated in a similar fashion however each of their weights will be trained separately based on how we use the values in updating the passed through values. We can control the update as follows,

For a simplified notation we can use,

$$\mathsf{W}_{ab}\mathsf{b} + \mathsf{W}_{ac}\mathsf{c} = \mathsf{W}_a\left[\mathsf{b}, \mathsf{c}\right].$$

$$\tilde{\mathsf{h}}_t = g(\mathsf{W}_{hh}(\mathsf{r}_t * \mathsf{h}_{t-1}) + \mathsf{W}_{hx}\mathsf{x}_t + \mathsf{b}_h) \tag{6}$$

$$\mathsf{h}_t = (1 - \mathsf{z}_t) * \mathsf{h}_{t-1} + \mathsf{z}_t * \tilde{\mathsf{h}}_t \tag{7}$$

g here is usually the hyperbolic tangent function and $*$ represents element wise multiplication. We can see that we use r_t to calculate the reset value, $\tilde{\mathsf{h}}_t$, and then we use z_t to control how we weight that new value with the previous passed through value.

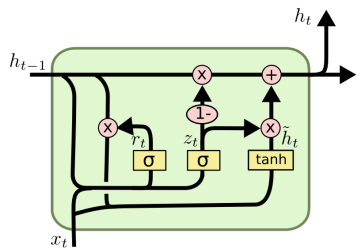

Figure 4: A visualization of GRUs. Source

Long Short Term Memory cells (LSTMs)

Long Short Term Memory cells (LSTMs) are similar to GRUs but they add an additional computation that can result in improved results. The key difference is the addition of another gat that controls how much we forget from memory versus how much we update using the input. Starting from the base RNN unit, LSTMs introduce forget, update/input, and output gates, and they also pass through 2 memory values, the intermediate cell memory **c** as well as the output, **h**. We first define the gate values as,

$$\mathsf{f}_t = f(\mathsf{W}_{fh}\mathsf{h}_{t-1} + \mathsf{W}_{fx}\mathsf{x}_t + \mathsf{b}_f), \tag{8}$$
$$\mathsf{i}_t = f(\mathsf{W}_{ih}\mathsf{h}_{t-1} + \mathsf{W}_{ix}\mathsf{x}_t + \mathsf{b}_i), \tag{9}$$
$$\mathsf{o}_t = f(\mathsf{W}_{oh}\mathsf{h}_{t-1} + \mathsf{W}_{ox}\mathsf{x}_t + \mathsf{b}_o). \tag{10}$$

Similar to GRUs, f here is usually the sigmoid function. Based on these values we can calculate the intermediate pass through cell memory and the output value,

$$\tilde{\mathsf{c}}_t = g(\mathsf{W}_{ah}(\mathsf{h}_{t-1}) + \mathsf{W}_{ax}\mathsf{x}_t + \mathsf{b}_a), \tag{11}$$
$$\mathsf{c}_t = \mathsf{f}_t * \mathsf{c}_{t-1} + \mathsf{i}_t * \tilde{\mathsf{c}_t}, \tag{12}$$
$$\mathsf{h}_t = \mathsf{o}_t * g(\mathsf{c}_t). \tag{13}$$

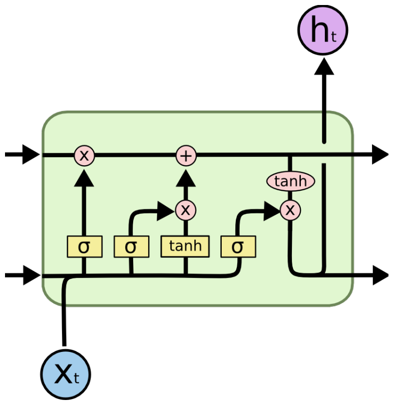

Figure 5: A visualization of LSTMs. Source

LSTMs may require more computation but can also provide superior results. More recent RNN implementations are all but guaranteed to incorporate LSTMs or GRUs.

A common variation on GRUs and LSTMs is called the ”peephole” GRU or ”peephole” LSTM. In this variation, the gates' values include the memory input with learned weights for it. An example calculation for a gate p would be,

$$\mathsf{p}_t = f(\mathsf{W}_{pc}\mathsf{c}_{t-1} + \mathsf{W}_{ph}\mathsf{h}_{t-1} + \mathsf{W}_{px}\mathsf{x}_t + \mathsf{b}_p). \tag{14}$$

Bidirectional RNNs

A bidirectional RNN is one where the values we pass through across the sequence are passed through in both directions; forwards and backwards. This means that information from future items in the sequence can be used to predict information related to the current values. This is especially useful in language as the remaining portions of a sentence can provide context about the previous part and vice versa. One issue with bidirectional RNNs is that you need the full sequence before you can start running the network. This prevents real time applications since you have to wait until the interaction completes before you can provide a result. It can be incorporated with pauses but would not be able to run until the interaction pauses.

The bidirectional approach can be applied to all the variations of RNNs including GRUs and LSTMs.