Artificial Neural Networks
Lecture Notes

Prof. Dr. Sen Cheng

Winter Semester 2019/20

12 Recurrent neural networks

Feedforward (ffw) networks process information one item at a time. This mode of processing assumes that each item (x_i, y_i) is independent of other items (x_j, y_j) in the dataset, where $i \neq j$. This assumption does not work well, if the data are sequential, such that an item (x_t, y_t) at one time point t is strongly correlated with the preceding or following item(s). These sequential dependencies occur quite often in data:

1. *Time-series data*. Almost all variables that change in time have strong sequential correlations, e.g., stock prices, gross domestic product, blood pressure, atmospheric temperature, etc. The task is usually to predict some future value(s) in the time-series.

2. Text data. The specific sequence, in which words are arranged, matters a lot for the meaning of a sentence, e.g. "I see everything that I eat." is quite different from "I eat everything that I see." Computer code can be considered text data for our purposes. Networks are used for many different kinds of tasks such as: predicting the next character/word, translation, speech-to-text, handwriting-recognition, figure captioning, sentiment analysis.

3. Biological data such as DNA sequences.

There are several ways, in which sequential data can be processed with a ffw network. One could simply concatenate the entire sequence into one single input and process that input in a ffw network. However, that would dramatically increase the dimensionality of the input and, thus, the size of the network required to model the data. If we knew that the sequential correlations had a certain finite distance, e.g., one item only affects the next τ items in the sequence, then we could concatenate only the $\tau + 1$ elements that affect each other

$$\begin{bmatrix} x_{t-\tau} \\ \vdots \\ x_{t-1} \\ x_t \end{bmatrix} \tag{1}$$

and provide them simultaneously as inputs. (Note, sequences with $\tau = 1$ are called *Markov chains*). However, the concatenation approach works only if one knows how many timesteps τ into the future are affected by the current state, τ is finite, and if τ is fixed for the entire sequence. These conditions are not satified in many applications. For example, if you wanted to predict the next word in a sentence, and you are presented the words one-by-one in a sequence. The next word does not only depend on the current word, but also on many preceeding words. Exactly how many is not predetermined and can change a lot based on the sentence, context, speaker, etc.

Finally, one could ignore the information that is contained in the sequence all together and extract some summary information from the data. For instance, instead of representing a text as a sequence of words, one could represent it as a frequency distribution over words, i.e., the input is a vector whose elements represent words and the value indicates the number of times a word occurs in the text. This approach is known as *bag of words*.

While these workarounds might work well enough in some cases, they don't work in general. That's why we need *recurrent neural networks (RNN)*. They are defined structurally by having loops in their connections, this enables information to flow from one unit back to the same unit, which in turn enables the network to sustain activity/ store

Figure 1: Ripples on water surface. Foto by Brocken Inaglory. Licensed under Creative Commons BY-SA 3.0.

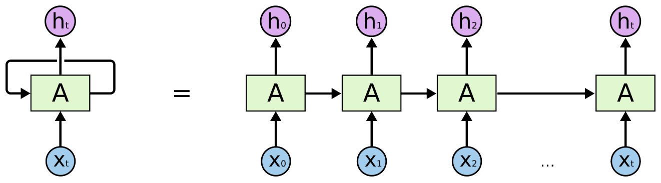


Figure 2: RNN unrolled in time.

information in the absence of external inputs. This is the property that allows RNN to integrate information over time. Similar to ripples on a water surface recording a history of rain drops falling in (Fig. 1), RNNs can be thought of as a *short-term memory* that stores a history of its recent inputs. Ffw networks lack this short-term memory. Note, however, that all neural networks, including ffw networks, store memory of training inputs in their weights. This can be thought of as *long-term memory*.

RNNs have been shown to be *Turing complete*.

12.1 Backpropagation through time

A recurrent network evolves from timestep t to $t+1$

$$h_t = \phi\left(w_{fw}x_t + w_{rec}h_{t-1}\right) \tag{2}$$

This corresponds to propagating activity from layer $l-1$ to l if the network is unrolled in time (Fig. 2). We can pretend that the network state at a particular timepoint t constitutes a layer in a network, which is connected to the next layer with weights w_t. Of course all the $w_t = w_{rec}$, but we can view this as weight sharing in a network.

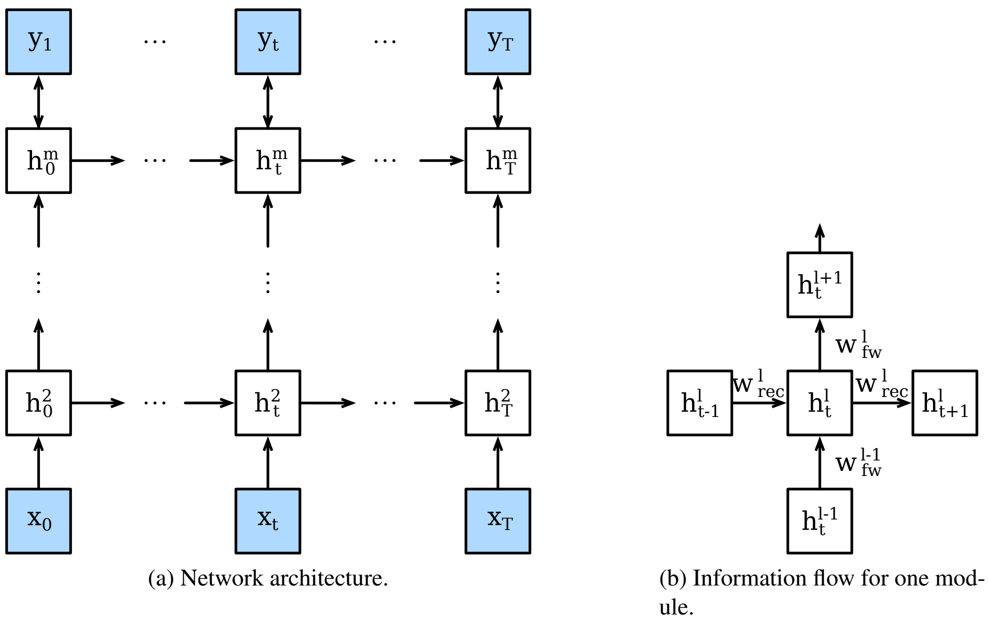
We can therefore apply the backpropation algorithm to this unrolled network. This is called *Backpropagation Through Time (BPTT)*. It is like regular backpropagation through a ffw network, except that the recurrent weights are shared. Therefore, the weight changes computed iteratively for every layer Δw_t should not be applied immediately. Instead, they are stored and all applied at the end: $\Delta w_{rec} = \sum_t \Delta w_t$.

12.2 Deep RNN

In addition, RNN architecture can be deep by adding hidden layers to the processing hierarchy (Fig. 3). We denote the activity of the l-th layer at timestep t as

$$h_t^l = \phi\left(w_{fw}^{l-1}h_t^{l-1} + w_{rec}^l h_{t-1}^l\right) \tag{3}$$

That means that the input comes from two sources, the ffw pathway (Fig. 3, vertical arrows) and the recurrent pathway



(a) Network architecture.

(b) Information flow for one module.

Figure 3: Unrolled deep RNN.

(Fig. 3, horizontal arrows).

Therefore, in the backward pass errors have to flow back through both pathways.

$$\delta_t^l = \phi'\left(a_t^l\right) \odot \left[\left(w_{fw}^l\right)^T \delta_t^{l+1} + \left(w_{rec}^l\right)^T \delta_{t+1}^l\right] \tag{4a}$$

The ffw error in the output layer $l = m$ is determined by the gradient of the loss function.

$$\delta_t^m = \phi'\left(a_t^m\right) \odot \left[\nabla L(h_t^m) + (w_{rec}^m)^T \delta_{t+1}^m\right] \tag{4b}$$

The network potentially generates output in multiple time steps, and not just in the last one. Depending on the problem, the external inputs or training labels might be missing at some timepoints in the sequence. If that is the case, the missing input or output terms in Eq. 4 are ignored.

Since the length T of the sequences fed into the network are often quite long, unrolled RNN are usually deep networks and therefore suffer even more than ffw networks from the vanishing and exploding gradients problems. The gradient vanishes or explodes, depending on the largest eigenvalue λ of w_{rec} ($\lambda < 1$ vanishing; $\lambda > 1$ exploding). Therefore, learning in RNNs is particular unstable, and RNNs are difficult to train.

12.3 Long short-term memory (LSTM)

The vanishing/exploding gradient problem can be quite severe in RNN. That means that even though the network maintains short-term memory of the inputs, the errors do not propagate backwards through the network. The training of the recurrent network cannot take into account a long history of inputs, thus limiting the utility of the short-term memory. The solution is to increase the duration of the short-term memory like in the *long short-term memory (LSTM)* network (Hochreiter & Schmidhuber, 1997). A regular unit in an ANN performs a weighted summation of its inputs and then applies a nonlinear activation function. The unit maintains no memory of its previous activities. The short-term memory is maintained in a recurrent network because the previous activity of the layer is provided back to the layer as input. This activity has to serve two purposes, it has to maintain memory of the previous states and it has to signal some information to the other units in the network. All this is controlled by a single weight matrix w_{rec}^l. There is little control over the short-term memory process.

In an LSTM unit, short-term memory is much better controlled by separating the output of the unit from the memory maintanance. Memory of previous states is maintained in an internal state c_t (called a “memory cell”). This internal state is controlled by a forgetting “gate” and updated by external inputs, which is in turn controlled by the input gate (Fig. 4). The internal state passes through an activation function and an output gate to generate the output. All three “gates” are actually neural networks.

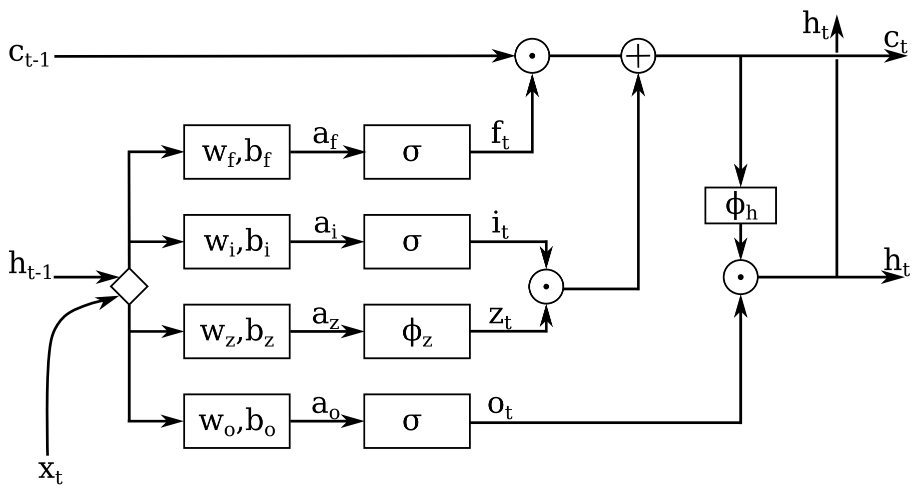


Figure 4: Long short-term memory (LSTM) unit.

12.3.1 Forward pass

Variables

- m, n: the number of input features and number of hidden units
- $x_t \in R^m$: input vector to the LSTM unit/ could be the output of a hidden layer below
- $h_t \in R^n$: hidden state vector also known as output vector of the LSTM unit
- $c_t \in R^n$: cell state vector
- $z_t \in R^n$: the candidate vector
- $f_t \in R^n$: forget gate's activation vector
- $i_t \in R^n$: input/update gate's activation vector
- $o_t \in R^n$: output gate's activation vector
- $w \in R^{n\times(n+m)}$, and $b \in R^n$: weight matrices and bias vector parameters, which need to be learned during training

Activation functions

- σ: activation function of gates, must be logistic function.
- ϕ_z: activation function of inputs, usually hyperbolic tangent function.
- ϕ_h: activation function of outputs, usually hyperbolic tangent function, but could also be identity function.

The cell performs the usual weighted summation and activation function.

$$z_t = \phi_z\left(w_z\begin{bmatrix}h_{t-1}\\ x_t\end{bmatrix} + b_z\right) \tag{5}$$

where we use the notation x_t for the input to the LSTM unit (in a multi-layer network, the input would be the output of the lower layer h_t^{l-1}). However, the activity is not directly passed on to the output. Instead, it is gated by the input gate i_t before it is integrated into the new cell state. In addition, the new cell state, includes the previous cell state c_{t-1} gated by a forgetting gate f_t. Taken together, the new cell state is given by:

$$c_t = \underbrace{f_t \odot c_{t-1}}_{\text{memory}} + \underbrace{i_t \odot z_t}_{\text{new info.}} \tag{6a}$$

This internal state leaks into the output, controlled by the output gate o_t.

$$h_t = o_t \odot \phi_h(c_t) \tag{6b}$$

The gates are themselves neural networks that are learned.

$$f_t = \sigma\left(w_f \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_f\right) \tag{6c}$$

$$i_t = \sigma\left(w_i \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_i\right) \tag{6d}$$

$$o_t = \sigma\left(w_o \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_o\right) \tag{6e}$$

$$\tag{6f}$$

The initial values are $c_0 = 0$ and $h_0 = 0$.

Additional material:

- ”Understanding LSTM Networks” by Christopher Olah

12.3.2 Backward pass

We define

$$\delta k_t = \frac{\partial L}{\partial a_k^t} \tag{7}$$

$$\delta h_t = \frac{\partial L}{\partial h_t^l} \tag{8}$$

$$\delta c_t = \frac{\partial L}{\partial c_t^l} \tag{9}$$

where k stands for one of the three gates $k = f, i, o$ or the inputs $k = z$. Given all δ’s and gate values from layer l at time $t+1$, and from the next layer $l+1$ at time t, the errors in the current layer are

$$\delta c_t = o_t \odot \phi_h'(c_t) \odot \delta h_t + f_{t+1} \odot \delta c_{t+1} \tag{10a}$$

$$\delta h_t = \delta x_t^{l+1} + M_h\left[(w_f^l)^T \delta f_{t+1} + (w_i^l)^T \delta i_{t+1} + (w_z^l)^T \delta z_{t+1} + (w_o^l)^T \delta o_{t+1}\right] \tag{10b}$$

where $M_h = [I_{n\times n}, 0_{n\times m}]$ and δx_t^{l+1} are the errors from the next layer without the recurrent errors. The errors can be propagated further back towards the activations of the gates and the input.

$$\delta f_t = \sigma'(a_f) \odot c_{t-1} \odot \delta c_t \tag{10c}$$

$$\delta i_t = \sigma'(a_i) \odot z_t \odot \delta c_t \tag{10d}$$

$$\delta z_t = \phi_z'(a_z) \odot i_t \odot \delta c_t \tag{10e}$$

$$\delta o_t = \sigma'(a_o) \odot \phi_h(c_t) \odot \delta h_t \tag{10f}$$

Finally, the error propagated back to the lower layer is

$$\delta x_t^l = M_x\left[(w_f^l)^T \delta f_t + (w_i^l)^T \delta i_t + (w_z^l)^T \delta z_t + (w_o^l)^T \delta o_t\right] \tag{10g}$$

where $M_x = [0_{m\times n}, I_{m\times m}]$.

Once all the δ’s have been computed, the updates for the weight and biases are

$$\delta w_k = \sum_{t=0}^{T} \delta k_t \times \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} \tag{10h}$$

$$\delta b_k = \sum_{t=0}^{T} \delta k_t \tag{10i}$$

where $k = f, i, z, o$ and $h_{-1} = 0$.

12.3.3 Applications

Example by Andrej Karpathy: LSTM-network trained on Leo Tolstoy's "War and Peace" to predict the next character.

Sample text self-generated by the network:

Iteration 100:

```
tyntd-iafhatawiaoihrdemot  lytdws  e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tklrgd t o idoe ns,smtt   h ne etie h,hregtrs nigtike,aoaenns lng
```

Iteration 300:

```
"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

Iteration 500:

```
we counter. He stutn co des. His stanted out one ofler that concossions and was
to gearang reay Jotrets and with fre colt otf paitt thin wall. Which das stimn
```

Iteration 700:

```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.
```

Iteration 1200:

```
"Kite vouch!" he repeated by her
door. "But I would be done and quarts, feeling, then, son is people...."
```

Iteration 2000:

```
"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.
```

12.4 Gated recurrent unit (GRU)

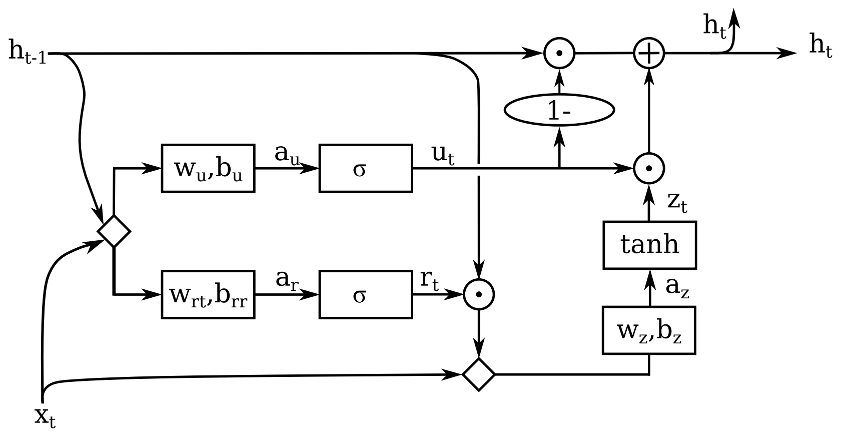


Figure 5: Gated recurrent unit (GRU).

The *gated recurrent unit (GRU)* was suggested by Cho et al. (2014) and is simpler than the LSTM. It maintains no internal cell state and combines the input gate and forget gate into a single update gate u_t (Fig. 5). It also introduces a new reset gate r_t that selectively acts on the previous output h_{t-1}. GRU has fewer parameters that need to be trained and appears to perform better in some cases, but in other cases LSTM perform better. So, it is not clear at the moment when to prefer GRU or LSTM.

12.4.1 Forward pass

$$u_t = \sigma\left(w_u \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_u\right) \tag{11a}$$

$$r_t = \sigma\left(w_r \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_r\right) \tag{11b}$$

$$z_t = \tanh\left(w_z \begin{bmatrix} r_t \odot h_{t-1} \\ x_t \end{bmatrix} + b_z\right) \tag{11c}$$

$$h_t = (1-u_t) \odot h_{t-1} + u_t \odot z_t \tag{11d}$$

Variables (same symbols as in LSTM mean the same things)

- $u_t \in R^n$: update gate's activation vector
- $r_t \in R^n$: reset gate's activation vector

12.4.2 Backward pass

Given δ's at $t+1$ and in layer $l+1$.

$$\delta h_t = (1-u_{t+1})(\delta x_t^{l+1} + \delta h_{t+1}) + M_h \left[w_u^T \delta u_{t+1} + w_r^T \delta r_{t+1}\right] + r_{t+1} \odot M_h w_z^T \delta z_{t+1} \tag{12}$$

where $M_h = [I_{n\times n}, 0_{n\times m}]$.

$$\delta z_t = \tanh'(a_z) \odot u_t \odot \delta h_t \tag{13a}$$

$$\delta u_t = \sigma'(a_u) \odot (z_t - h_{t-1}) \odot \delta h_t \tag{13b}$$

$$\delta r_t = \sigma'(a_r) \odot h_{t-1} \odot M_h w_z^T \delta z_t \tag{13c}$$

$$\tag{13d}$$

Finally, the error propagated back to the lower layer is

$$\delta x_t^l = M_x \left[w_u^T \delta u_t + w_r^T \delta r_t + w_z^T \delta z_t\right] \tag{13e}$$

where $M_x = [0_{m\times n}, I_{m\times m}]$.

Once all the δ's have been computed, the updates for the weight and biases are

$$\delta w_k = \sum_{t=0}^{T} \delta k_t \times \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} \tag{13f}$$

$$\delta b_k = \sum_{t=0}^{T} \delta k_t \tag{13g}$$

where $k = r, u, z$ and $h_{-1} = 0$.

References

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *ArXiv14061078 Cs Stat*.

Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780.