Recurrent Neural Networks

Carlos Valle

Departamento de Informática
Universidad Técnica Federico Santa María

cvalle@inf.utfsm.cl

December 7, 2018

Overview

1. Introduction
2. Recurrent Neural Networks
3. Back-propagation Through Time
4. Bidirectional Networks
5. Ideas to improve an RNNs
6. Early recurrent network designs
7. Current recurrent network designs

Time dependence problems

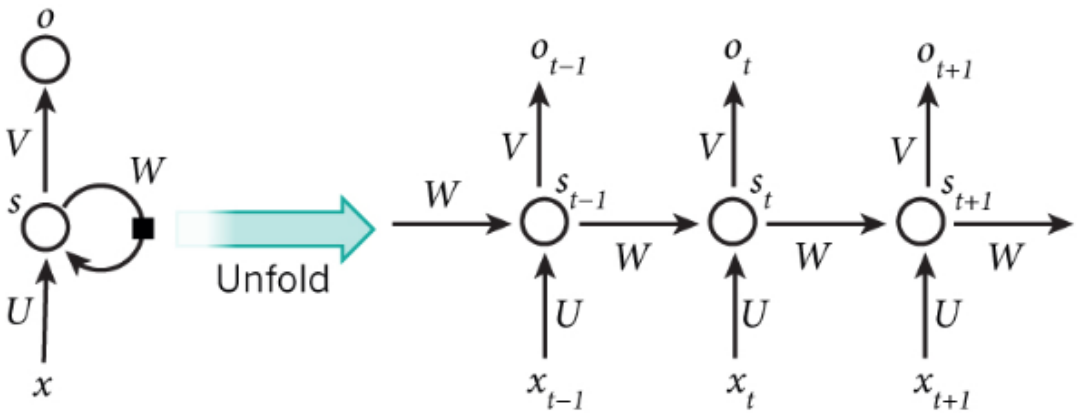
- In this chapter we will study learning algorithms for a sequence of values $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$.
- For example, captioning an audio, handwriting recognition, time series, etc. are some problems that can be modeled with sequences.
- In order to learn temporal dependencies among the elements of a sequence, we could pre-process data by using a time windows as input patterns.

Recurrent Networks (RNN)

- A recurrent neural network (RNN) is specialized for learning a sequence of values $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$.
- Unlike a feedforward neural network, an RNN allows cyclical connections.
- The key point is that the recurrent connections allow a 'memory' of previous inputs to persist in the network's internal state, and thereby influence the network output.
- "RNN with a sufficient number of hidden units can approximate any measurable sequence-to-sequence mapping to arbitrary accuracy" (Hammer, 2000).

Unfolding diagrams

- A useful way to visualize RNNs is to consider the update graph formed by 'unfolding' the network along the input sequence (as in the figure below).
- Note that the unfolded graph contains no cycles; otherwise the forward and backward pass would not be well defined.
- Viewing RNNs as unfolded graphs makes it easier to generalize to networks with more complex update dependencies.



RNN computations

- x_t is the input at time step t.
- s_t is the hidden state at time step t. It is known as the "memory" of the network.
- s_t is calculated based on the previous hidden state and the input at the current step: $s_t = f(Ux_t + Ws_{t-1})$.
- Here, f is the activation function of the hidden layer.
- s_{-1}, which is required to calculate the first hidden state, is typically initialized to all zeroes.
- o_t is the output at step t. $o_t = g(Vs_t)$. Where g is the activation function of the output layer.
- The total loss for a given sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$ is

$$L(\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t\}, \{y_1, y_2, \ldots, y_t\}) = \sum_t \ell(y_t | \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t\})$$

RNN remarks

- This involves performing a forward pass moving left to right through our illustration of the unrolled graph. Followed by a backward pass moving right to left through the graph. This is call *Back-propagation through time*
- You can think of the hidden state s_t as the memory of the network.
- s_t captures information about what happened in all the previous time steps.
- The output at step o_t is calculated solely based on the memory at time t.
- Unlike a traditional deep neural network, which uses different parameters at each layer, a RNN shares the same parameters (U, V, W above) across all steps.

RNN remarks (2)

- This reflects the fact that we are performing the same task at each step, just with different inputs.
- This greatly reduces the total number of parameters we need to learn.
- The model is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.
- Thus, learning a single, shared model allows generalization to sequence lengths that did not appear in the training set, and allows the model to be estimated with far fewer training examples than would be required without parameter sharing.

Forward Pass

- Consider a length T input sequence $\mathbf{x}$ presented to an RNN with I input units, H hidden units, and K output units.
- Let x_m^t be the value of input m at time t, and let a_j^t and $b_j^t = f_j(a_j^t)$ be respectively the network input to unit j at time t and the activation of unit j at time t.
- $f_j(\cdot)$ is the activation function of the unit j.
- For the hidden units we have

$$a_h^{(t)} = \sum_{m=1}^{M} w_{mh} x_m^{(t)} + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{(t-1)}$$

Forward Pass (2)

- Note that this requires initial values b_h^0 to be chosen for the hidden units, corresponding to the initial state of the network (before it receives any input), usually set to zero.
- The network inputs to the output units can be calculated at the same time as the hidden activations:

$$a_k^{(t)} = \sum_{h=1}^{H} w_{hk} b_h^{(t)}$$

Back-propagation Through Time (BPTT)

- BPTT consist in applying classical BP to the unfolded network.
- Recall that the influence from the hidden layer in BP is computed as

$$\delta_h = f'(a_j) \sum_{k=1}^{K} \delta_k^{(t)} w_{hk}$$

- The only difference is that for RNNs the loss function depends on the activation function of the hidden layer that not only influence on the output layer but also influence on the hidden layer at the next step. This is

$$\delta_h^t = f'(a_j^t) \left(\sum_{k=1}^{K} \delta_k^t w_{hk} + \sum_{h'=1}^{H} \delta_{h'}^{t+1} w_{hh'} \right), \tag{1}$$

where the influence from an output node a_j is given by

$$\delta_j^t := \frac{\partial E}{\partial a_j^t}$$

Back-propagation Through Time (BPTT)

- Note that $\delta_j^{(T+1)} = 0$ since no error is received from beyond the end of the sequence.
- Considering that the same weights are reused at every timestep, we sum over the whole sequence to get the derivatives with respect to the network weights:

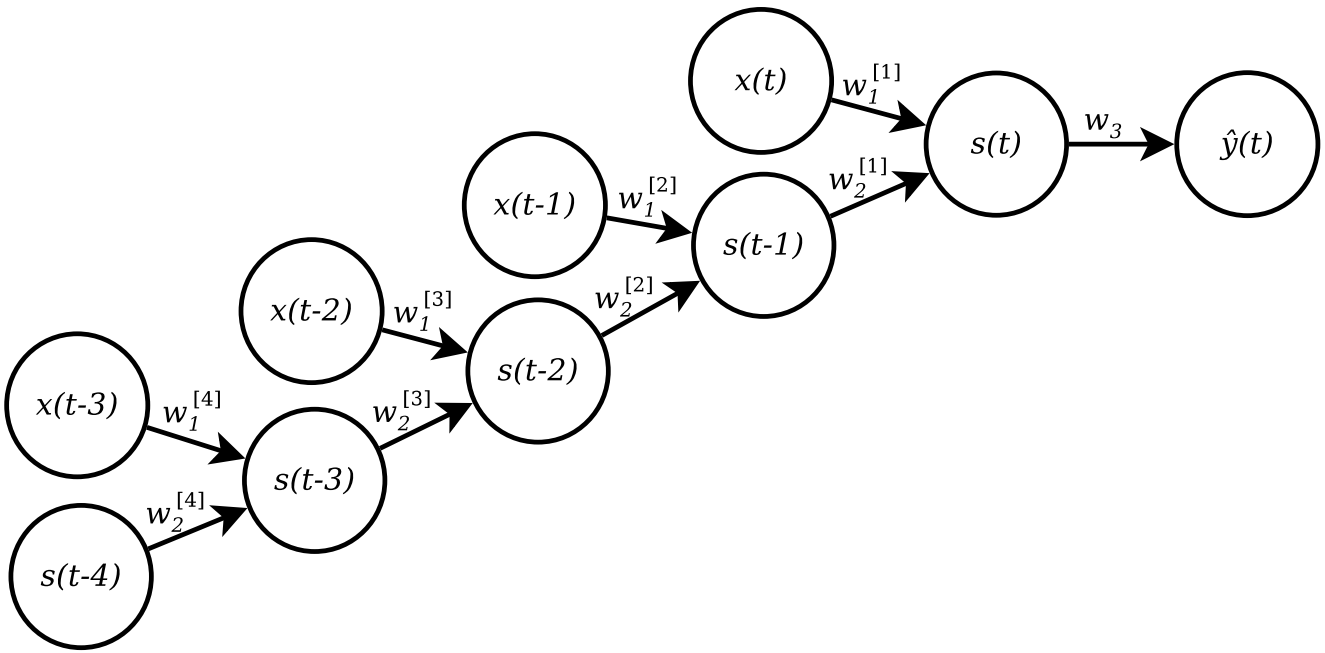
$$\frac{\partial E}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial E}{\partial a_j^t} \frac{\partial a_j^t}{\partial w_{ij}} = \sum_{t=1}^{T} \delta_j^t \frac{\partial a_j^t}{\partial w_{ij}}$$

Drawback of the BPTT (2)

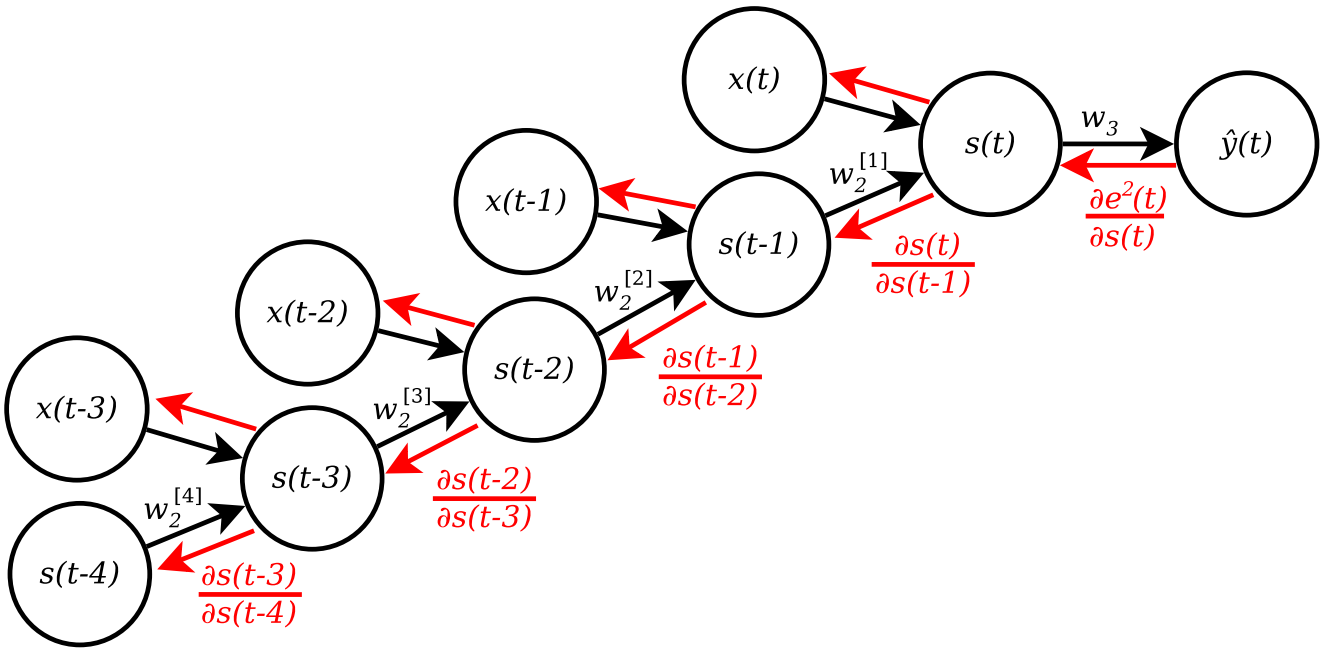
- $$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E}{\partial a^t} \frac{\partial a^t}{\partial W}$$
- $$\frac{\partial E_t}{\partial W} = \sum_{s=1}^{t} \frac{\partial E}{\partial a^t} \frac{\partial a^t}{\partial a^s} \frac{\partial a^t}{\partial W}$$
- $\frac{\partial a^t}{\partial a^s}$ measures the influence of the state at time s in the error obtained at time t.
- $\frac{\partial a^t}{\partial a^s} = \prod_{k=s+1}^{t} \frac{\partial a^k}{\partial a^{k-1}} = \prod_{k=s+1}^{t} J_k,$
- where J_k is the Jacobian.
- $||J_k||$ is the spectral radius (the absolute value of the largest eigenvalue).

Drawback of the BPTT (3)

- We can note that BPTT treat a RNN as a deep network with shared weights.
- Therefore, like BP, this algorithm will have the vanishing/exploding gradient descent problem.



Drawback of the BPTT (2)



Drawback of the BPTT (4)

- In linear networks:
- If $||J|| > 1$ the influence of the state at time s in the error obtained at time t exponentially grows (explodes).
- If $||J|| < 1$ the influence of the state at time s in the error obtained at time t exponentially decreases (vanishes).
- If $||J|| = 1$ the influence of the state at time s in the error obtained at time t is stably propagated. However, this is happening for each time s, hence, the network has a non selective memory.

Drawback of the BPTT (5)

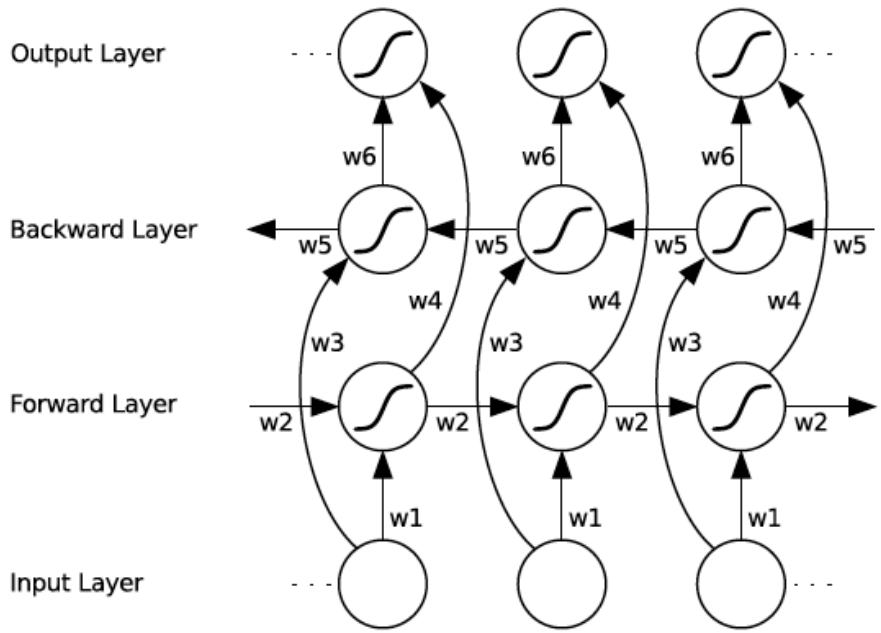
- In non linear networks: Pascanu et al. On the difficulty of training Recurrent Neural Networks (2013).
- $x_t = W_{rec} f(x_{t-1}) + u_t W_{in} + b$
- $$\frac{\partial a^t}{\partial a^s} = \prod_{k=s+1}^{t} \frac{\partial a^k}{\partial a^{k-1}} = W_{rec}^T \text{diag}(f'(a^{k-1}))$$
- $$||\frac{\partial a^k}{\partial a^{k-1}}|| \leq ||W_{rec}||\,||\text{diag}(f'(a^{k-1}))||$$
- If we assume that $||\text{diag}(f'(a^{k-1}))|| \leq \gamma$:
- $||W_{rec}|| < 1/\gamma$ is sufficient for vanishing gradients.
- $||W_{rec}|| > 1/\gamma$ is necessary for exploding gradients.
- tanh : $\gamma = 1$. sigmoid: $\gamma = 1/4$.

RNN issues for learning sequences

- Sometimes, for labeling some sequences it is beneficial to have access to future as well as past context.
- For example, when classifying a particular tweet, it is helpful to know the letters coming after it as well as those before.
- However, as we saw, standard RNNs process sequences in temporal order, ignoring future context.
- An obvious solution is to add a time-window of future context to the network input.
- However, as well as increasing the number of input weights, this approach suffers from the same problems as the time-window methods.

Bidirectional Networks (BRNN)

- BRNN present each training sequence forwards and backwards to two separate recurrent hidden layers, both of which are connected to the same output layer.
- This way they provide information from past and future to each element of the sequence associated to the same target.



Bidirectional networks (2)

- The forward pass for the BRNN hidden layers is the same as for a unidirectional RNN, however the output layer is not updated until both hidden layers have processed the entire input sequence.

Algorithm 1 BRNN Forward pass

1: **for** $t \leftarrow 1, \ldots, T$ **do**
2: Forward pass for the forward hidden layer, updating node values at each timestep.
3: **end for**
4: **for** $t \leftarrow T, \ldots, 1$ **do**
5: Forward pass for the backward hidden layer, updating node values at each timestep.
6: **end for**
7: **for** all$t$ **do**
8: Forward pass for the output layer, using node values from both hidden layers.
9: **end for**

Bidirectional networks (3)

- To perform the forward pass of this network, we use the BPTT, except that all the output layer nodes are calculated first, then fed back to the two hidden layers in opposite directions:

Algorithm 2 BRNN Backward pass

1: **for** all t **do**
2: Backward pass for the output layer, storing δ's at each timestep.
3: **end for**
4: **for** $t \leftarrow T, \ldots, 1$ **do**
5: BPTT backward pass for the forward hidden layer, using the stored δ's from the output layer.
6: **end for**
7: **for** $t \leftarrow 1, \ldots, T$ **do**
8: BPTT backward pass for the backward hidden layer, using the stored δ's from the output layer.
9: **end for**

BRNN remarks

- The BRNNs have shown outstanding results in protein structure prediction.
- Obviously, these networks are not designed for all kind of tasks.
- If the input sequences are spatial and not temporal there is no reason to distinguish between past and future inputs.
- However BRNNs can also be applied to temporal tasks, as long as the network outputs are only needed at the end of some input segment.
- Furthermore, even for online temporal tasks, such as automatic dictation, bidirectional algorithms can be used as long as it is acceptable to wait for some natural break in the input, such as a pause in speech, before processing a section of the data.

Truncated BPTT (TBPTT)

- It processes a sequence of length n one timestep at a time, and every k_1 timesteps, it runs BPTT for k_2 timesteps, so a parameter update can be cheap if k_2 is small.
- Consequently, its hidden states have been exposed to many timesteps and so may contain useful information about the far past, which would be opportunistically exploited.

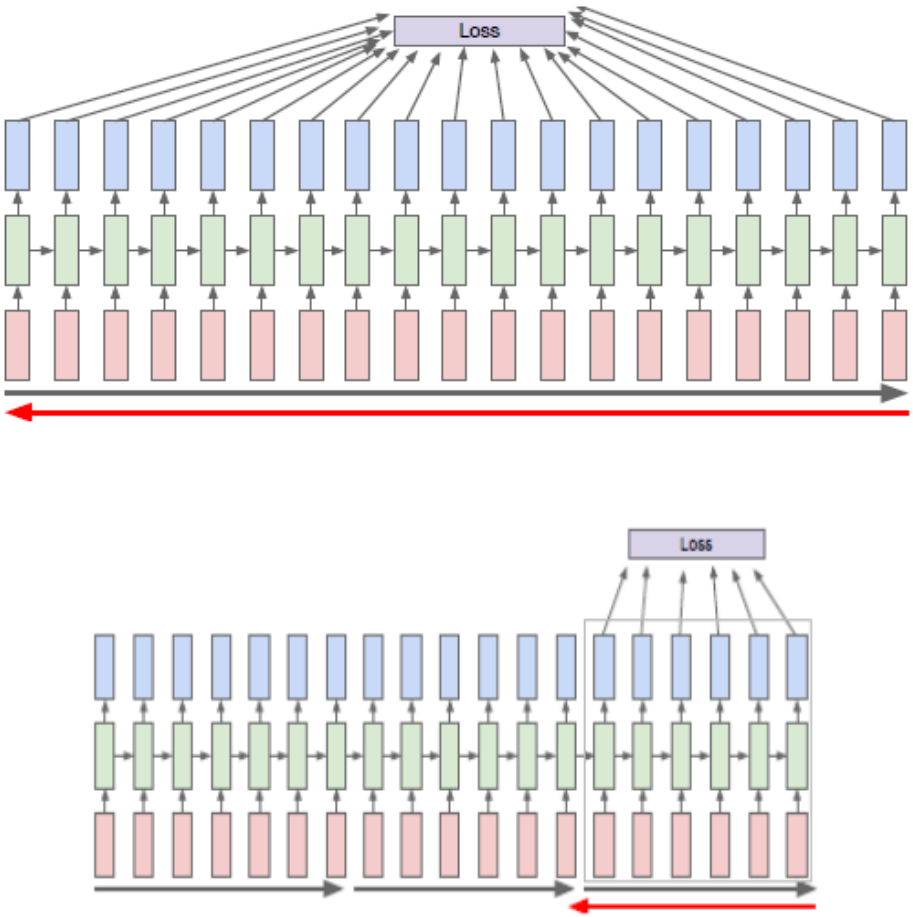
Algorithm 3 TBPTT algorithm

1: Present a sequence of k_1 timesteps of input and output pairs to the network.
2: Unroll the network then calculate and accumulate errors across k_2 timesteps.
3: Roll-up the network and update weights.
4: Repeat.

Truncated BPTT (TBPTT) (2)

- Generally, k_2 should be large enough to capture the temporal structure in the problem for the network to learn. Too large a value results in vanishing gradients.
- $k_1 = k_2 < n$ is called *epochwise BPTT algorithm*. A common configuration used in TensorFlow.

Truncated BPTT (TBPTT) (3)



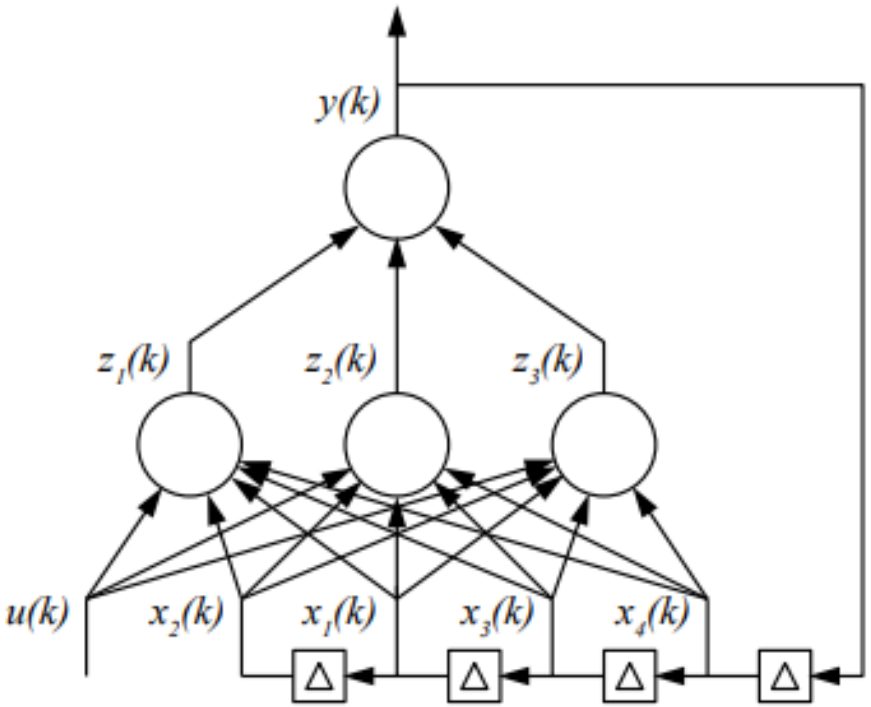
- Problem: Does not allow to learn long term dependencies.

Skip connections

- Skip connections going from layer i to layer $i+2$ or higher.
- These skip connections make it easier for the gradient to flow from output layers to layers nearer the input.
- Thus, it reduces the length of the shortest path from the lower layers parameters to the output, and thus mitigate the vanishing gradient problem.
- As we have seen, gradients may vanish or explode exponentially with respect to the number of time steps.

Skip connections (2)

- Lin et al. (1996). Learning long-term dependencies is not as difficult with NARX recurrent neural networks: introduced recurrent connections with a time-delay of d to mitigate this problem.
- This allows the learning algorithm to capture longer dependencies although not all long-term dependencies may be represented well in this way.



Leaky units

- Rather than an integer skip of d time steps, the effect can be obtained smoothly by using a fixed parameter α (non-trainable) to compute a running average $\mu(t)$ of some $v(t)$:

$$\mu(t) = \alpha\mu(t-1) + (1-\alpha)v(t)$$

- This is called a linear self connection.
- When α is close to 1, running average remembers information from the past for a long time and when it is close to 0, information is rapidly discarded.
- Hidden units with linear self connections are called *leaky units*.

Teaching forcing

- Teacher forcing during training means:
- Instead of summing activations from incoming units (possibly erroneous), each unit sums correct teacher activations as input for the next iteration.
- During training time model receives ground truth output $y(t)$ as input at time $t + 1$.
- It emerges from the maximum likelihood criterion.

Teaching forcing (2)

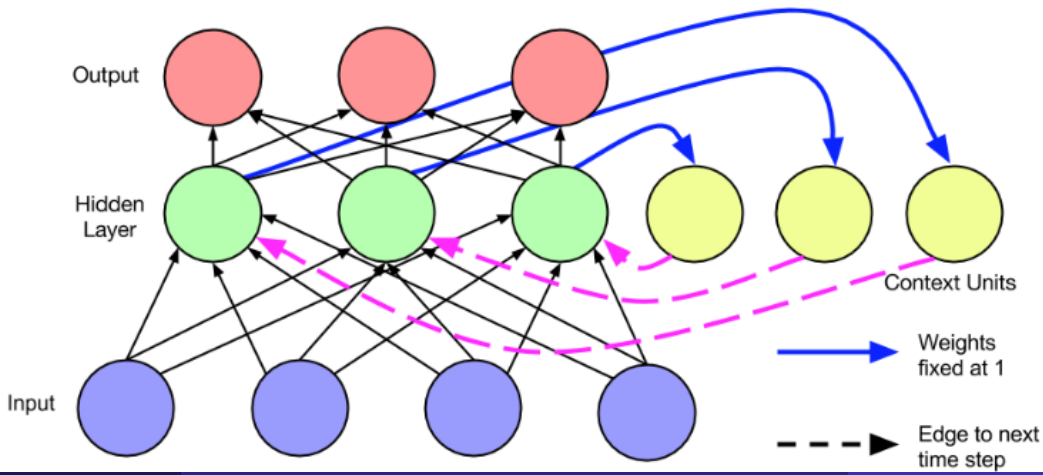
- Consider a sequence with two time steps:
- The conditional maximum likelihood criterion is

$$\log(p(y(1), y(2)|x(1), x(2)) = \log p(y(2)|y(1), x(1), x(2)) + \log p(y(1)|x(1), x(2))$$

- At time t=2, model is trained to maximize conditional probability of $y(2)$ given both the $\mathbf{x}$ sequence so far and the previous y value from the training set.
- We are using $y(1)$ as teacher forcing, rather than only $x(i)$.
- Maximum likelihood specifies that during training, rather than feeding the models own output back to itself, target values should specify what the correct output should be.

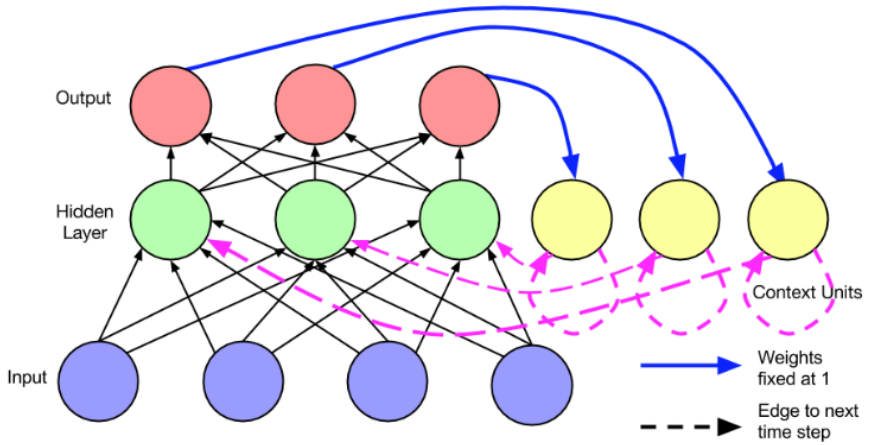
Elman Networks

- In a Elman network, each context unit is associated with a unit in the hidden layer.
- Each such unit j' takes as input the state of the corresponding hidden node j at the previous time step, along an edge of fixed weight $w_{jj} = 1$.
- This architecture is equivalent to a simple RNN in which each hidden node has a single self-connected recurrent edge.
- It generates an output at each time t. Problem for sequence to sequence tasks.



Jordan Networks

- Jordan proposes a feedforward network with a single hidden layer that is extended with special units which are called *state units*.
- State units receive the output signals (teaching forcing), which then feed these values to the hidden nodes at the following time step.
- If the output values are actions, the state units can remember actions taken at previous time steps.
- The state units are self-connected. Intuitively, these edges allow sending information across multiple time steps without perturbing the output at each intermediate time step.



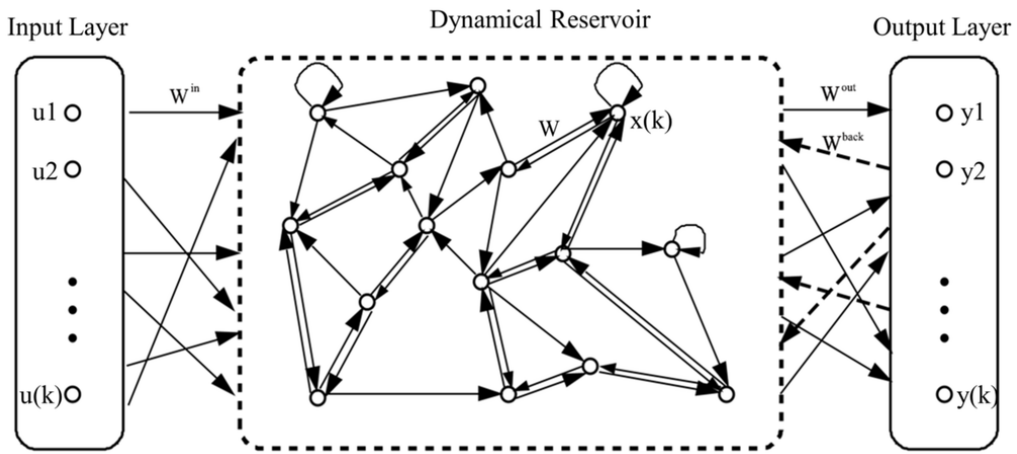
Reservoir computing

- As we saw, classical recurrent neural networks may exhibit vanishing/exploding gradient descent.
- The reservoir computing framework is a family of methods that fed an input signal into a fixed (random) dynamical system called a *reservoir* and the dynamics of the reservoir map the input to a higher dimension.
- Then only the outputs weights are trained to model the outputs according to the reservoir.

Echo State Networks (ESN)

- ESNs are applied to supervised time series where for a given training input signal $u_t \in R^I$ a desired target output signal $y_t \in R^Q$ is known.
- The error measure E is the Root-Mean-Square Error (RMSE) or other MSE based loss function.

$$E(\hat{y}, y) = \frac{1}{Q} \sum_{i=1}^{Q} \frac{1}{T} \sum_{t=1}^{T} (\hat{y}_{it} - y_{it})^2,$$



Echo State Networks (ESN)

- W_{in} is a $Z \times (1 + I)$ with the input weights.
- W is a $Z \times Z$ matrix with the recurrent weights.
- W_{out} is a $Q \times (1 + I + Z)$ matrix with the output weight matrix,
- where Z is the number of hidden nodes.
- Let $\mathbf{x}_t$ be the vector of reservoir neuron activations and $\tilde{\mathbf{x}}_t$ is its update at time t, then
- $\tilde{\mathbf{x}}_t = \tanh(W^{in}[1; u_t] + W x_{t-1}))$,
- $\mathbf{x}_t = (1 - \alpha)\mathbf{x}_{t-1} + \alpha\tilde{\mathbf{x}}_t$,
- As we seen $\mathbf{x}_t$ are leaky units. However some version get rid of it $(\alpha = 1)$.
- The value of the kth output node is given by

$$y_t(k) = W_{out}[1; u_t; \mathbf{x}_t],$$

ESN algorithm

Algorithm 4 ESN algorithm

1: Generate a large random reservoir (W^{in}, W, α)
2: Using the input u_t compute the reservoir response $\mathbf{x}_t$
3: Compute the readout weight W^{out} using linear regression minimizing the MSE between $\hat{y}_t$ and y_t
4: Use the networks weights to compute $\hat{y}_t$ from an input vector u_t.

Spectral radius

- For the supervised learning algorithms, it is crucial that the ESN network state $\mathbf{x}_t$ is uniquely determined by any left-infinite input sequence $\ldots, u_{t-1}, u_t$.
- This is so-called the *echo state property* (ESP).
- As we seen before, the spectral radius $\rho(W)$ of an RNN needs to be close to 1.
- W is divided by $\rho(W)$ to obtain a matrix with a unit spectral radius.
- Recent works have shown that echo state property often holds for $\rho(W) \geq 1$ for nonzero inputs u_t.
- This can be explained by the derivative of the tanh() nonlinearities will approach zero on many time steps, and help to prevent the explosion resulting from a large spectral radius.

Training readouts

- Since readouts from an ESN are typically linear and feed-forward, the model predictions can be written in a matrix notation as

$$Y = W_{out} X,$$

- where $Y \in Q \times T$ are all y_t, and $X \in (1 + I + Z) \times T$ are all $[1; u_t; x_t]$.
- To simplify the notation, we use here a single X instead of $[1; U; X]$.
- The goal now is to find the optimal weights W_{out} that minimize the squared error between $\hat{y}$ and y.

Training readouts (2)

- As we previously known this is equivalent to solve a typically overdetermined system of linear equations

$$\hat{Y} = W_{out} X,$$

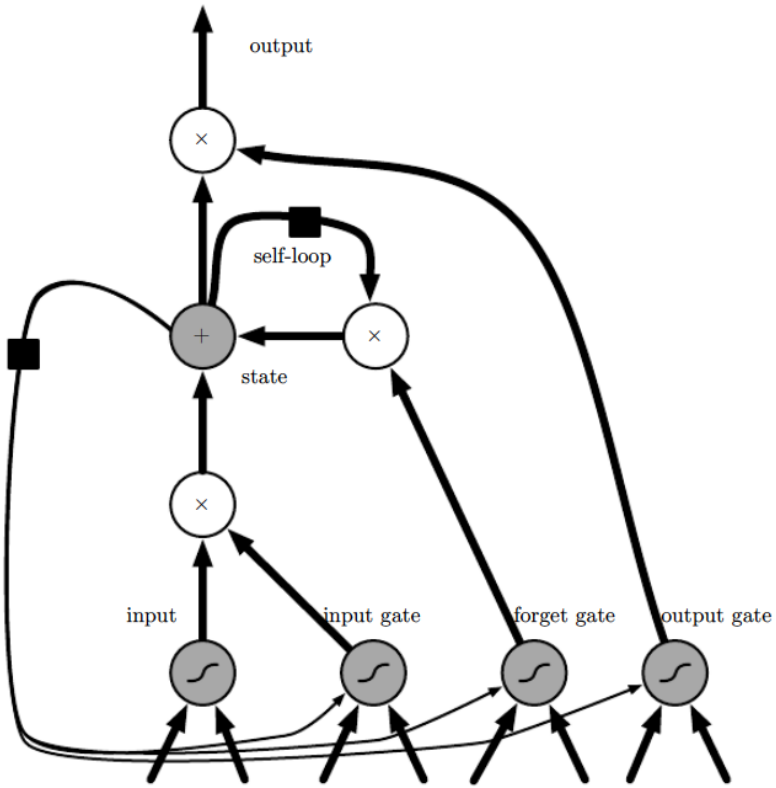
- where $\hat{Y}$ is a $Q \times T$ matrix.
- Using ridge regression we have $W_{out} = \left(XX^T + \beta I\right)^{-1} X^T \hat{Y}$

Long Short-term Memory (LSTM)

- Like leaky units, LSTMs are based on the idea of creating paths without the above-mention gradient descent problem.
- Leaky unit allow the network accumulate information from the past.
- However, once that information has been used, it might be useful for the network to forget the old state.
- The main goal of a LSTM is to automatically decide when remember and when forget the information.
- A LSTM network contains a set of LSTM blocks instead of, regular network units.
- A LSTM block contains gates that determine when the input is significant enough to remember, when it should continue to remember or forget the value, and when it should output the value.

Long Short-term Memory (LSTM) (2)

- Input gate can admit to remember more instances.
- The self-loop of the hidden state is controlled by the forget gate.
- The output of the cell can be shut off by the output gate.



Long Short-term Memory (LSTM) (3)

- The state unit $s_i^{(t)}$ has a linear loop similar to the leaky units of the ESN.
- However, the self-loop weight is controlled by the forget gate $f_i^{(t)}$.

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{ij}^f x_j^{(t)} + \sum_j W_{ij}^f h_j^{(t-1)} + \sum_j V_{ij}^f s_j^{(t-1)}\right)$$

- where σ is typically a sigmoid function. Additionally, b^f, U^f and W^f are the biases, input weights and recurrent weights respectively.
- The state unit update is given by

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + \sigma\left(b_i + \sum_j U_{ij} x_j^{(t)} + \sum_j W_{ij} h_j^{(t-1)}\right)$$

Long Short-term Memory (LSTM) (4)

- Similarly to the forget gate, the external input and output gates $g_i^{(t)}$ and $q_i^{(t)}$ respectively are computed by

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{ij}^g x_j^{(t)} + \sum_j W_{ij}^g h_j^{(t-1)} + \sum_j V_{ij}^g s_j^{(t-1)}\right)$$

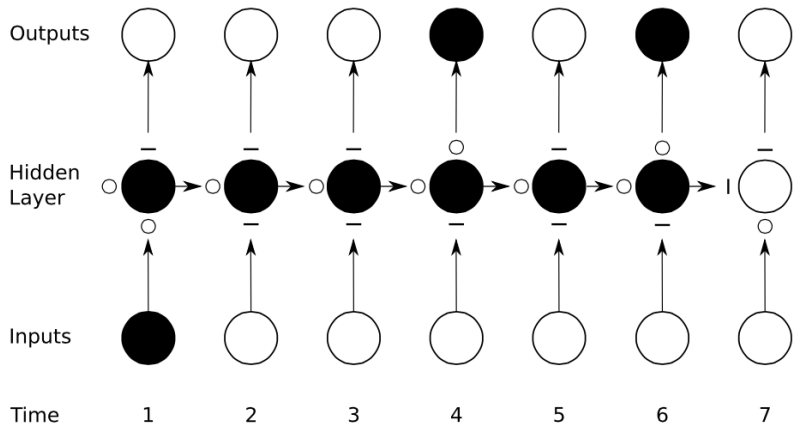
$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{ij}^o x_j^{(t)} + \sum_j W_{ij}^o h_j^{(t-1)} + \sum_j V_{ij}^o s_j^{(t-1)}\right)$$

- The output $h_i^{(t)}$ can be shut off by the output gate

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

LSTM preserves the flow of information

- The input, forget, and output gate activations are displayed to the left and above the memory block (respectively). The gates are either entirely open (O) or entirely closed (-).
- The memory cell remembers the first input as long as the forget gate is open and the input gate is closed.
- The sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.



GRU

- Gated recurrent units (GRU) can be viewed as a simplified version of LSTM.
- The main difference with the former approach is a single gating simultaneously controls the forgetting factor and the decision to update the state unit.
- A GRU has two gates, a reset gate, and an update gate.

GRU (2)

- The network is updated as follows

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + \left(1 - u_i^{(t-1)}\right) \sigma \left(b_i + \sum_j U_{ij} x_j^{(t)} + \sum_j W_{ij} r_j^{(t-1)} h_j^{(t-1)} \right)$$

- Update and reset gates are computed as

$$u_i^{(t)} = \sigma \left(b_i^u + \sum_j U_{ij}^u x_j^{(t)} + \sum_j W_{ij}^u h_j^{(t)} \right)$$

$$r_i^{(t)} = \sigma \left(b_i^r + \sum_j U_{ij}^r x_j^{(t)} + \sum_j W_{ij}^r h_j^{(t)} \right)$$

GRU (3)

- Intuitively, the reset gate determines how to combine the new input with the previous memory.
- And the update gate defines how much of the previous memory to keep around.
- If we set the reset to all 1s and update gate to all 0s we again arrive at our plain RNN model.
- The input and forget gates are coupled by an update gate and the reset gate is applied directly to the previous hidden state.
- Thus, the responsibility of the reset gate in a LSTM is really split up into both u and r.

Any questions?