Understanding LSTM, a tutorial into Long Short-Term Memory Recurrent Neural Networks

Alireza Yahyanejad

June, 2024

Contents

1 Introduction

This article introduces Artificial Intelligence, focusing on Long Short-Term Memory Recurrent Neural Networks (LSTM-RNN), initially as supplementary lecture material. Readers can deepen their understanding of LSTM-RNN, noting its evolution since the 1990s. Modern LSTM-RNN research uses different notations and more concise derivations. The authors believe this approach will be useful to readers.

Machine learning develops algorithms that improve with practice. The goal is to create a classifier function from training data and measure its performance on new data. Artificial Neural Networks (ANNs) are inspired by biological systems and model interconnected neurons. Neurons process inputs to produce an output, with feed-forward neural networks organized into layers: input, hidden, and output layers. These networks are suited for static classification tasks but need dynamic classifiers for time prediction tasks.

Recurrent Neural Networks (RNNs) extend feed-forward networks by feeding signals from previous timesteps back into the network. RNNs can look back approximately ten timesteps, but their signals can vanish or explode. LSTM-RNNs address this issue and can learn more than 1,000 timesteps, depending on network complexity.

2 Perceptron and Delta Learning Rule

2.1 The Perceptron

A perceptron is the simplest type of artificial neuron. It has multiple external input links, a threshold, a single external output link, and an internal bias input, b. Each input value is weighted, and during training, the perceptron learns these weights from the training data. It sums the weighted inputs and 'fires' if the sum exceeds a predefined threshold. The perceptron's output is Boolean, firing if the output is '1' and deactivated at '-1', with a threshold value typically set to '0'. The perceptron outputs y; which is computed by the formula

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} W_i x_i + b > 0 \\ -1 & \text{if } \textit{otherwise} \end{cases}$$

The input vector is defined as:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

The weighted input is given by:

$$\sum_{i=1}^{n} W_i x_i + b$$

The state of the perceptron is:

$$s = z + b$$

2.2 Linear Separability

Linear separability refers to the ability to separate a set of points in a space using a single linear boundary, such as a line in two dimensions, a plane in three dimensions, or a hyperplane in higher dimensions. In the context of machine learning and neural networks, especially perceptrons, linear separability is a key concept for understanding which problems can be solved by these models.

2.3 Delta Learning Rule

Perceptron training, a type of supervised learning, adjusts weights based on output comparison with training data. If misclassification occurs, weights are modified. Perceptrons converge to correct behavior if training data is linearly separable; otherwise, convergence is not guaranteed.

Two common training algorithms are the perceptron learning rule and the delta learning rule. Both start with random weights and ensure convergence with linearly separable data.

Perceptron Learning Rule: Updates weights as

$$W_i' = W_i + \Delta W_i \quad \text{with} \quad \Delta W_i = \eta(d-y)x_i$$

where η is the learning rate, d is the desired output, and y is the calculated output. Convergence requires a sufficiently small learning rate.

Delta Learning Rule: Handles both linearly separable and non-separable data. It modifies weights using gradient descent to minimize error, forming the basis of the error backpropagation algorithm.

2.4 The Sigmoid Threshold Unit

The sigmoid threshold unit is a type of artificial neuron that, unlike the perceptron, calculates its output using a sigmoid function. y is computed by the formula

$$y = \frac{1}{1 - e^{-l \times s}};$$

with

$$s = \sum_{i=1}^{n} W_i x_i + b;$$

where b is the bias and l is a positive constant that determines the steepness of the sigmoid function. The output is "squashed" by a continuous function ranging between 0 and 1. The function $\frac{1}{1-e^{-l \times s}}$ is called the 'squashing' function because it maps a large input domain to a small output range. For low input values, the sigmoid output is near zero; for high input values, it is near one. The threshold value adjusts the slope of the sigmoid function. Neural networks using sigmoid units can represent non-linear functions, unlike perceptrons, which are limited to linear functions.

3 Feed-Forward Neural Networks and Backpropagation

Feed-forward neural Networks (FFNNs), where neurons are organized in layers and compute weighted sums of inputs. Input neurons receive signals from the environment, output neurons present signals to the environment, and hidden neurons are not directly connected to the environment. FFNNs are loop-free and fully connected. Single-layer perceptron networks are the simplest FFNNs, with input and output layers directly connected. Each perceptron in a single-layer network calculates the sum of weighted inputs and fires if it exceeds a threshold. Multilayer FFNNs have at least one hidden layer between the input and output layers. They can express non-linear decision surfaces using sigmoid threshold functions and can approximate any function with enough hidden units. The error backpropagation algorithm, prevalent in neural network learning, utilizes gradient descent to adjust weights in multilayer networks. It iteratively updates weights from the output layer to the input layer. Initialization involves setting weights to small, normalized random numbers with biases. Units in the network are categorized as input (i), hidden (h), and output (o). The output (y_u) of a non-input unit is determined using the sigmoid activation function:

$$y_u = \frac{1}{1 + e^{-s_u}}$$

where s_u is the state of unit u, calculated as:

$$s_u = z_u + b_u$$

and z_u, the weighted input of unit u, is computed as:

$$z_u = \sum_{v \in Pre(u)} W[v,u] y_v$$

where $Pre(u)$ represents the set of units preceding u, including input units and hidden units that provide their outputs (y_v) multiplied by corresponding weights ($W[v,u]$) to unit u.
Inputs propagate forward through the network to produce observable output (y). The backpropagation algorithm updates weights and biases to minimize error. For an output unit o, error e_o is calculated as $e_o = (d_o - y_o)$, and overall error E is defined as $E = \frac{1}{2}\sum_{o \in O} e_o^2$.

To update the weight $W[u,v]$, we use the formula:

$$\Delta W[u,v] = -\eta \frac{\partial E}{\partial W[u,v]}$$

where η is the learning rate. We utilize the partial derivatives $\frac{\partial y_u}{\partial y_u}$ and $\frac{\partial s_u}{\partial s_u}$ to calculate the weight update by deriving the error with respect to the activation, and the activation in terms of the state, and in turn the derivative of the state with respect to the weight:

$$\Delta W[u,v] = -\eta \frac{\partial E}{\partial y_u} \frac{\partial y_u}{\partial s_u} \frac{\partial s_u}{\partial W[u,v]}$$

The derivative of the error concerning the activation for output units is:

$$\frac{\partial E}{\partial y_o} = -(d_o - y_o)$$

Now, the derivative of the activation concerning the state for output units is:

$$\frac{\partial y_o}{\partial s_o} = y_o(1 - y_o)$$

The derivative of the state for a weight that connects the hidden unit h to the output unit o is:

$$\frac{\partial s_u}{\partial W[u,v]} = y_h$$

Let us define, for the output unit o, the error signal by:

$$\mathcal{V}_o = -\frac{\partial E}{\partial y_o} \frac{\partial y_o}{\partial s_o}$$

For output units, the error signal is calculated as:

$$\mathcal{V}_o = (d_o - y_o) y_o (1 - y_o)$$

We update the weight between the hidden unit h and the output unit o by:

$$\Delta W[h,o] = \eta \mathcal{V}_o y_h$$

Now, for a hidden unit h, if we consider its error about its contribution to faulty output, we can backpropagate the error from the output units that h sends signals to. Specifically, for an input unit i, we expand the equation $\Delta W[i,h] = -\eta \frac{\partial E}{\partial W[i,h]}$ to:

$$\Delta W[i,h] = -\eta \sum_o \frac{\partial E}{\partial y_o} \frac{\partial y_o}{\partial s_o} \frac{\partial s_o}{\partial y_h} \frac{\partial y_h}{\partial s_h} \frac{\partial s_h}{\partial W[i,h]}$$

where o belongs to the set $Suc(h)$, which includes units that receive input from h. Solving the partial derivatives, we obtain:

$$\Delta W[i,h] = -\eta \sum_o \mathcal{V}_o W[h,o] \frac{\partial y_h}{\partial s_h} \frac{\partial s_h}{\partial W[i,h]}$$

$$= \eta \sum_{o} \mathcal{V}_o W[h,o] y_h (1-y_h) y_i$$

If we define the error signal of the hidden unit h by:

$$\mathcal{V}_h = \sum_{o} \mathcal{V}_o W[h,o] y_h (1-y_h)$$

with o belonging to the set $Suc(h)$, then we have a uniform expression for weight change:

$$\Delta W[v,u] = \eta \mathcal{V}_u y_v$$

We calculate $\Delta W[v;u]$ repeatedly until all network outputs are within an acceptable range, or some other terminating condition is reached.

4 Recurrent Neural Networks

Recurrent neural networks (RNNs) are dynamic systems with an internal state at each time step of the classification process. They feature circular connections between neurons in different layers and may include self-feedback connections. These feedback connections allow RNNs to carry information from past events to current processing steps, enabling them to build a memory of time series events.

4.1 Basic Architecture

Recurrent neural networks (RNNs) can range from partly to fully connected. Two simple RNNs are the Elman and Jordan networks. The Elman network has three layers, with hidden layer outputs saved in 'context cells' that provide feedback to the hidden neurons. It is trained with standard backpropagation. The Jordan network is similar, but its context cells are fed by the output layer. RNNs require different training methods than feed-forward neural networks (FFNNs) due to their recurrent connections. The most common training algorithms are backpropagation through time (BPTT), which is an offline method, and real-time recurrent learning (RTRL), which is an online method.

5 Training Recurrent Neural Networks

The most common methods to train recurrent neural networks are Backpropagation Through Time (BPTT) and Real-Time Recurrent Learning (RTRL), with BPTT being the most widely used. The main difference between them lies in how they calculate weight changes. LSTM-RNNs originally used a combination of both BPTT and RTRL, so both algorithms are briefly covered.

5.1 Backpropagation Through Time

The Backpropagation Through Time (BPTT) algorithm trains recurrent neural networks (RNNs) by unfolding them in time to create an equivalent feed-forward neural network (FFNN). This unfolded network can be trained using standard backpropagation. The steps for BPTT are as follows:

1. **Unfold the RNN:** Create a separate layer for each time step, maintaining identical weights across layers.

2. **Calculate the error:** At the end of a training sequence, compute the error for the output units using a chosen error measure.

3. **Backpropagate the error:** Inject the error backward through the network to calculate weight updates for all time steps.

4. **Update weights:** Sum the weight updates (deltas) over all time steps and apply them to the recurrent version of the network.

Key Formulas:

1. Output of unit u at time τ:

$$y_u(\tau) = f_u(z_u(\tau))$$

where f_u is the differentiable, non-linear squashing function of unit u.

2. Weighted input $z_u(\tau+1)$:

$$z_u(\tau+1) = \sum_l W[u,l]X[l,u](\tau+1)$$

where $l \in \mathrm{Pre}(u)$.

3. Total error $E_{\text{total}}(t_0, t)$:

$$E_{\text{total}}(t_0, t) = \sum_{\tau=t_0}^{t} E(\tau)$$

with the error $E(\tau)$ at time τ defined as:

$$E(\tau) = \frac{1}{2}\sum_{u\in U}(e_u(\tau))^2$$

where $e_u(\tau)$ is the error for unit u at time τ:

$$e_u(\tau) = \begin{cases} d_u(\tau) - y_u(\tau) & \text{if } u \in T(\tau) \\ 0 & \text{otherwise} \end{cases}$$

4. Error signal $\delta_u(\tau)$:

$$\delta_u(\tau) = \frac{\partial E(\tau)}{\partial z_u(\tau)}$$

with:

$$\delta_u(\tau) = \begin{cases} f'_u(z_u(\tau))e_u(\tau) & \text{if } \tau = t \\ f'_u(z_u(\tau))\sum_{k\in U} W[k,u]\delta_k(\tau+1) & \text{if } t_0 \le \tau < t \end{cases}$$

5. Weight update $\Delta W[u,v]$:

$$\Delta W[u,v] = -\eta \frac{\partial E_{\text{total}}(t_0,t)}{\partial W[u,v]}$$

where:

$$\frac{\partial E_{\text{total}}(t_0,t)}{\partial W[u,v]} = \sum_{\tau=t_0}^{t} \delta_u(\tau)\frac{\partial z_u(\tau)}{\partial W[u,v]}$$

and:

$$\frac{\partial z_u(\tau)}{\partial W[u,v]} = X[u,v](\tau)$$

These steps and formulas collectively describe how BPTT trains RNNs by handling the temporal dependencies inherent in their structure.

5.2 Real-Time Recurrent Learning

The Real-Time Recurrent Learning (RTRL) algorithm does not require error propagation. Instead, it collects all necessary information to compute the gradient as the input stream is presented to the network, eliminating the need for a dedicated training interval. However, it comes with significant computational costs per update cycle and requires an additional notion called sensitivity of the output. The memory required depends only on the network size, not the input size.

For network units $v \in I \cup U$ and $u, k \in U$, and time steps $t_0 \le \tau \le t$, the training objective in RTRL is to minimize the overall network error given at time step τ by:

$$E(\tau) = \frac{1}{2}\sum_{k\in U}(d_k(\tau) - y_k(\tau))^2$$

The gradient of the total error is the sum of the gradient for all previous time steps and the current time step:

$$\nabla_W E_{\text{total}}(t_0, t+1) = \nabla_W E_{\text{total}}(t_0, t) + \nabla_W E(t+1)$$

During the presentation of the time series to the network, we accumulate the gradient values at each time step. The overall weight change for $W[u,v]$ is given by:

$$\Delta W[u,v] = \sum_{\tau=t_0+1}^{t} \Delta W[u,v](\tau)$$

The weight changes are calculated as:

$$\Delta W[u,v](\tau) = -\eta \frac{\partial E(\tau)}{\partial W[u,v]}$$

Expanding this via gradient descent and using the error definition, we get:

$$\Delta W[u,v](\tau) = -\eta \sum_{k\in U} (d_k(\tau) - y_k(\tau)) \left(\frac{\partial y_k(\tau)}{\partial W[u,v]} \right)$$

We define the quantity $p_{uv}^k(\tau)$ to measure the sensitivity of the output of unit k at time τ to a small change in the weight $W[u,v]$:

$$p_{uv}^k(\tau) = \frac{\partial y_k(\tau)}{\partial W[u,v]}$$

The gradient information is forward-propagated. Using the output and weighted input formulas:

$$y_k(t+1) = f_k(z_k(t+1))$$

$$z_k(t+1) = \sum_l W[k,l] X[k,l](t+1) = \sum_{v\in U} W[k,v] y_v(t) + \sum_{i\in I} W[k,i] y_i(t+1)$$

By differentiating these equations, we calculate $p_{uv}^k(t+1)$:

$$p_{uv}^k(t+1) = \frac{\partial y_k(t+1)}{\partial W[u,v]} = f_k'(z_k(t+1)) \left(\delta_{uk} X[u,v](t+1) + \sum_{l\in U} W[k,l] p_{uv}^l(t) \right)$$

where δ_{uk} is the Kronecker delta:

$$\delta_{uk} = \begin{cases} 1 & \text{if } u = k \\ 0 & \text{otherwise} \end{cases}$$

Assuming the initial state of the network has no functional dependency on the weights, the derivative for the first time step is:

$$p_{uv}^k(t_0) = \frac{\partial y_k(t_0)}{\partial W[u,v]} = 0$$

Using these equations, we recursively calculate p_{uv}^k for subsequent time steps.

6 Solving the Vanishing Error Problem

Standard Recurrent Neural Networks (RNNs) struggle to bridge more than 5-10 time steps due to back-propagated error signals either growing or shrinking at each step. This leads to either blown-up or vanishing errors, causing oscillating weights or excessively slow learning, respectively.

Key Formulas

1. Weight Update:

$$\Delta W[u,v] = -\eta \frac{\partial E_{\text{total}}(t_0,t)}{\partial W[u,v]}$$

where:

$$\frac{\partial E_{\text{total}}(t_0,t)}{\partial W[u,v]} = \sum_{\tau=t_0}^{t} \delta_u(\tau) X[u,v](\tau)$$

2. Backpropagated Error Signal:

$$\delta_u(\tau) = f'_u(z_u(\tau)) \sum_{v\in U} W_{v,u}\delta_v(\tau+1)$$

3. Error Propagation Over Time:

$$\frac{\partial \delta_v(t_0)}{\partial \delta_o(t)} = \begin{cases} f'_v(z_v(t_0))W[o,v] & \text{if } t-t_0=1 \\ f'_v(z_v(t_0)) \sum_{u\in U} \frac{\partial \delta_u(t_0+1)}{\partial \delta_o(t)} W[u,v] & \text{if } t-t_0>1 \end{cases}$$

4. Unrolling Over Time: For $t_0 \le \tau \le t$, let u_τ be a non-input-layer neuron in one of the replicas in the unrolled network at time τ. Setting $u_t = v$ and $u_{t_0} = o$:

$$\frac{\partial \delta_v(t_0)}{\partial \delta_o(t)} = \sum_{u_{t_0}\in U} \cdots \sum_{u_{t-1}\in U} \prod_{\tau=t_0+1}^{t} f'_{u_\tau}(z_{u_\tau}(t-\tau+t_0))W[u_\tau, u_{\tau-1}]$$

5. Error Signal Behavior:

$$\left| f'_{u_\tau}(z_{u_\tau}(t-\tau+t_0))W[u_\tau, u_{\tau-1}] \right| > 1$$

If this condition holds for all τ, the product grows exponentially, causing the error to blow up and potentially leading to oscillating weights and unstable learning.

Conversely:

$$\left| f'_{u_\tau}(z_{u_\tau}(t-\tau+t_0))W[u_\tau, u_{\tau-1}] \right| < 1$$

If this condition holds for all τ, the product decreases exponentially, causing the error to vanish and preventing the network from learning within an acceptable time period.

6. Global Error:

$$\sum_{o\in O} \frac{\partial \delta_v(t_0)}{\partial \delta_o(t)}$$

If the local error vanishes, then the global error also vanishes.

These steps and formulas describe how standard RNNs compute gradients and the issues related to vanishing and exploding gradients during backpropagation through time.

7 Long Short-Term Neural Networks

A gradient-based method called long short-term memory (LSTM) addresses the vanishing error problem. LSTM can handle time lags of over 1,000 discrete time steps using constant error carousels (CECs) to maintain a steady error flow within special cells. Multiplicative gate units manage access to these cells by learning when to allow it.

7.1 Constant Error Carousel

In a system with one unit u that has a single connection to itself, the local error backflow at a time-step τ is given by:

$$\delta_u(\tau) = f'_u(z_u(\tau))W[u,u]\delta_u(\tau+1)$$

To ensure a constant error flow through u, it must hold that:

$$f'_u(z_u(\tau))W[u,u] = 1.0$$

Integrating this, we get:

$$f_u(z_u(\tau)) = \frac{z_u(\tau)}{W[u,u]}$$

This implies that f_u must be linear, and u's activation must remain constant over time:

$$y_u(\tau+1) = f_u(z_u(\tau+1)) = f_u(y_u(\tau)W[u,u]) = y_u(\tau)$$

This is ensured by using the identity function $f_u = \text{id}$ and setting $W[u,u] = 1.0$. This preservation of error, known as the constant error carousel (CEC), is the central feature of LSTM, allowing for short-term memory storage over extended periods. The various components of LSTM networks manage the connections from other units to u.

7.2 Memory Blocks

In the absence of new inputs, the CEC's backflow remains constant. However, in a neural network, the CEC connects to other units, and these connections can cause conflicting weight update signals. To address this, LSTM adds input and output gates to manage these connections. Input gates, which use a sigmoid activation function, control signals from the network to the memory cell by scaling them, protecting the cell from irrelevant signals. Output gates control access to the memory cell contents, shielding other cells from disturbances. These gates allow or deny access to the constant error flow through the CEC, resulting in a more complex LSTM unit called a memory block.

8 Training LSTM-RNNs - the Hybrid Learning Approach

To preserve the constant error carousel (CEC) in LSTM memory cells, the original LSTM used Back-propagation Through Time (BPTT) for components after the cells and Real-Time Recurrent Learning (RTRL) for components before and including the cells. RTRL computes partial derivatives related to the cell state at each step. Currently, only the cell's gradient is propagated through time, with other gradients truncated. Discrete-time steps $\tau = 1, 2, 3, \ldots$ involve a forward pass for unit activations and a backward pass for error signal calculations.

8.1 The Forward Pass

Let M be the set of memory blocks, mc the c-th memory cell in block m, and $W[u,v]$ the weight connecting unit u to v.

Each memory block m has one input gate in_m and one output gate out_m. The state of mc at time $\tau+1$ is updated based on its state $s_{mc}(\tau)$ and the weighted input $z_{mc}(\tau+1)$ multiplied by the input gate activation $y_{in_m}(\tau+1)$. The cell activation $y_{mc}(\tau+1)$ is computed using the output gate activation $z_{out_m}(\tau+1)$.

The input gate activation y_{in_m} is:

$$y_{in_m}(\tau+1) = f_{in_m}(z_{in_m}(\tau+1))$$

$$z_{in_m}(\tau+1) = \sum_{u \in \text{Pre}(in_m)} W[in_m, u]X[u, in_m](\tau+1) = \sum_{v \in U} W[in_m, v]y_v(\tau) + \sum_{i \in I} W[in_m, i]y_i(\tau+1)$$

The output gate activation y_{out_m} is:

$$y_{out_m}(\tau+1) = f_{out_m}(z_{out_m}(\tau+1))$$

$$z_{out_m}(\tau+1) = \sum_{u\in \mathrm{Pre}(out_m)} W[out_m, u]X[u, out_m](\tau+1) = \sum_{v\in U} W[out_m, v]y_v(\tau) + \sum_{i\in I} W[out_m, i]y_i(\tau+1)$$

The non-linear squashing function f for the gates is:

$$f(s) = \frac{1}{1+e^{-s}}$$

The weighted input for memory cell m_c is:

$$z_{m_c}(\tau+1) = \sum_{u\in \mathrm{Pre}(m_c)} W[m_c, u]X[u, m_c](\tau+1) = \sum_{v\in U} W[m_c, v]y_v(\tau) + \sum_{i\in I} W[m_c, i]y_i(\tau+1)$$

The state $s_{m_c}(\tau+1)$ is updated as:

$$s_{m_c}(\tau+1) = s_{m_c}(\tau) + y_{in_m}(\tau+1)g(z_{m_c}(\tau+1))$$

with $s_{m_c}(0) = 0$ and the non-linear squashing function g:

$$g(z) = \frac{4}{1+e^{-z}} - 2$$

The output y_{m_c} is:

$$y_{m_c}(\tau+1) = y_{out_m}(\tau+1)h(s_{m_c}(\tau+1))$$

with the squashing function h:

$$h(z) = \frac{2}{1+e^{-z}} - 1$$

For a recurrent neural network, the output unit activation y_o is:

$$y_o(\tau+1) = f_o(z_o(\tau+1))$$

$$z_o(\tau+1) = \sum_{u\in U-G} W[o, u]y_u(\tau+1)$$

where G is the set of gate units, and the logistic sigmoid function f_o is used.

8.2 Forget Gates

In a standard LSTM network, the self-connection has a fixed weight of 1 to preserve the cell state over time. However, this can cause the cell states to grow linearly with continuous input, leading to the memory cell losing its memorizing capability and functioning like a regular RNN neuron. Manually resetting the cell state at the start of each sequence is impractical for continuous input.

To address this, an adaptive forget gate can be added to the self-connection. This gate learns to reset the memory cell state when the stored information is no longer needed. The fixed weight of 1 in the self-connection is replaced with a multiplicative forget gate activation y_φ, computed similarly to other gates:

$$y_{\varphi m}(\tau+1) = f_{\varphi m}(z_{\varphi m}(\tau+1) + b_{\varphi m})$$

where f is a squashing function with a range [0, 1], $b_{\varphi m}$ is the forget gate bias (initially set to 0 and then fixed at 1 for improved performance), and

$$z_{\varphi m}(\tau+1) = \sum_{u\in \mathrm{Pre}(\varphi_m)} W[\varphi_m, u]X[u, \varphi_m](\tau+1) = \sum_{v\in U} W[\varphi_m, v]y_v(\tau) + \sum_{i\in I} W[\varphi_m, i]y_i(\tau+1)$$

The internal cell state s_{m_c} is updated as:

$$s_{m_c}(\tau+1) = s_{m_c}(\tau) \cdot y_{\varphi m}(\tau+1) + y_{in_m}(\tau+1) \cdot g(z_{m_c}(\tau+1))$$

with $s_{m_c}(0) = 0$ and using the squashing function, which has a range $[-2, 2]$.

During training, the bias weights of input and output gates are initialized with negative values, and those of the forget gate with positive values. This ensures that the forget gate activation starts close to 1.0, making the memory cell behave like a standard LSTM cell initially, preventing it from forgetting before learning anything.

8.3 Backward Pass

LSTM uses BPTT for output units, hidden units, and output gates, and RTRL for input gates, forget gates, and cells. The network error at time step τ is:

$$E(\tau) = \frac{1}{2} \sum_{o \in O} (d_o(\tau) - y_o(\tau))^2$$

For units using BPTT, the individual error is:

$$\upsilon_u(\tau) = -\frac{\partial E(\tau)}{\partial z_u(\tau)}$$

The weight update for these units is:

$$\Delta W[u, v](\tau) = \eta \upsilon_u(\tau) X[v, u](\tau)$$

For units using RTRL, the cell error is:

$$\upsilon_{m_c}(\tau) = y_{out_m}(\tau) h'(s_{m_c}(\tau)) \sum_{o \in O} W[o, m_c] \upsilon_o(\tau) + \upsilon_{m_c}(\tau+1) y_{\varphi m}(\tau+1)$$

The weight update for cells is:

$$\Delta W[m_c, v](\tau) = \eta \upsilon_{m_c}(\tau) \frac{\partial s_{m_c}(\tau)}{\partial W[m_c, v]}$$

with $\frac{\partial s_{m_c}(\tau+1)}{\partial W[u,v]}$ depending on the nature of unit u.

8.4 Complexity

We present a complexity measure for LSTM based on Gers' principles [22]. Assuming each memory block has one cell and output units receive signals only from cells, let B, C, In, and Out be the numbers of memory blocks, cells per block, input units, and output units. The complexity is approximately $O(B^2 \cdot C^2)$. Input and output connections complexity is $O(In \cdot B \cdot S)$ and $O(Out \cdot B \cdot S)$ respectively. With the number of weight updates bounded by connections, LSTM's computational complexity per step and weight is $O(1)$.

8.5 Strengths and limitations of LSTM-RNNs

LSTM is adept at tasks requiring long-term memory retention with limited data. Memory blocks, featuring input and output gates, control data flow and maintain relevance. Forget gates manage information retention, enabling continuous prediction and preventing bias. However, LSTM's network topology remains fixed, limiting memory capacity. Increasing network size uniformly is unlikely to overcome this, suggesting modularization for effective learning, although the process is not clearly defined.

9 Problem specific topologies

9.1 Bidirectional LSTM

bidirectional LSTM (BLSTM), analyzing both past and future points in a sequence. BLSTM presents architectural advantages over unidirectional LSTM for phoneme classification. It removes one-step truncation and implements full error gradient calculation, easing training with standard BPTT. Introduced in 2006, the Connectionist Temporal Classification (CTC) objective function allows LSTM-RNN to handle unsegmented input data, leading to the common BLSTM-CTC variant.

9.2 Grid LSTM

Grid LSTM, introduced in the referenced work, extends the benefits of LSTM, such as selective input processing, to deep networks with a unified architecture. An N-dimensional Grid LSTM (N-LSTM) arranges LSTM cells in a grid across N dimensions, facilitating communication between layers. Unlike stacked LSTM, which only adds cells along the depth dimension, Grid LSTM incorporates cells both along and in between layers. N-LSTM networks with $N > 2$ resemble multi-dimensional LSTM but are distinguished by their depth dimension cells and improved stability in layer interactions. Consider a trained LSTM network with weights W, whose hidden cells emit a collection of signals represented by the vector $\vec{y}_h$ and whose memory units emit a collection of signals represented by the vector $\vec{s}_m$. Whenever this LSTM network is provided an input vector $\vec{x}$, there is a change in the signals emitted by both hidden units and memory cells; let $\vec{y}'_h$ and $\vec{s}'_m$ represent the new values of signals. Let P be a projection matrix, the concatenation of the new input signals and the recurrent signals is given by $X = [P\vec{x}, \vec{y}_h]$. An LSTM transform, which changes the values of hidden and memory signals as previously mentioned, can be formulated as follows:

$$(X, \vec{s}_m) \rightarrow (\vec{y}'_h, \vec{s}'_m)$$

9.2.1 Stacked LSTM

A stacked LSTM increases capacity by layering multiple LSTM networks. The first network uses $\mathbf{X}_1$ For the i-th network, the input $\mathbf{X}_i$ is defined as:

$$\mathbf{X}_i = [\mathbf{y}_{h_{i-1}}\, \mathbf{y}_{h_i}]$$

This replaces the input signals $\mathbf{x}$ with the hidden signals from the previous LSTM, effectively "stacking" them.

9.2.2 Multidimensional LSTM

Multidimensional LSTM networks handle inputs structured as N-dimensional grids, such as 3D arrays of voxels. These networks extend the number of recurrent connections from 1 to N, receiving N hidden vectors $\vec{y}_{h1}, \ldots, \vec{y}_{hN}$ and N memory vectors $\vec{s}_{m1}, \ldots, \vec{s}_{mN}$ as input. They then output a single hidden vector $\vec{y}_h$ and a single memory vector $\vec{s}_m$.

The memory signal vector $\vec{s}_m$ is calculated as:

$$\vec{s}_m = \sum_{i=1}^{N} \vec{\varphi}_i \odot \vec{s}_{mi} + \vec{in}_m \odot \vec{z}_m$$

where $\odot$ is the Hadamard product, $\vec{\varphi}_i$ is a vector of N forget signals (one for each $\vec{y}_{hi}$), $\vec{in}_m$ is the input gate signal, and $\vec{z}_m$ is the weighted input of the memory cell.

9.2.3 Grid LSTM Blocks

Large multidimensional LSTM networks tend to be unstable due to their high number of connections. An alternative approach, Grid LSTM, computes new memory vectors differently. Unlike multidimensional LSTM, a Grid LSTM block outputs distinct sets of N hidden vectors and N memory vectors.

9.3 Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) architecture as an alternative to LSTM for recurrent neural networks (RNNs). GRU has been observed to perform better than LSTM in most tasks, except for language modeling with naive initialization.

GRU units do not have a memory cell like LSTM but incorporate gating units: a reset gate and an update gate. Let H be the set of GRU units. The activation $y_{\text{res}_u}(\tau+1)$ of the reset gate res_u at time $\tau+1$ is defined by:

$$y_{\text{res}_u}(\tau+1) = f_{\text{res}_u}(s_{\text{res}_u}(\tau+1))$$

where:
f_{res_u} is the squashing function of the reset gate, typically a sigmoid function.
$s_{\text{res}_u}(\tau+1) = z_{\text{res}_u}(\tau+1) + b_{\text{res}_u}$ represents the state of the reset gate res_u at time $\tau+1$.
$z_{\text{res}_u}(\tau+1) = \sum_{h\in H} W[\text{res}_u, h]y_h(\tau) + \sum_{i\in I} W[\text{res}_u, i]y_i(\tau+1)$ denotes the weighted input of the reset gate at time $\tau+1$.

Similarly, the activation $y_{\text{upd}_u}(\tau+1)$ of the update gate upd_u at time $\tau+1$ is defined as:

$$y_{\text{upd}_u}(\tau+1) = f_{\text{upd}_u}(s_{\text{upd}_u}(\tau+1))$$

where:
f_{upd_u} represents the squashing function of the update gate, typically a sigmoid function.
$s_{\text{upd}_u}(\tau+1) = z_{\text{upd}_u}(\tau+1) + b_{\text{upd}_u}$ indicates the state of the update gate upd_u at time $\tau+1$.
$z_{\text{upd}_u}(\tau+1) = \sum_{h\in H} W[\text{upd}_u, h]y_h(\tau) + \sum_{i\in I} W[\text{upd}_u, i]y_i(\tau+1)$ defines the weighted input of the update gate at time $\tau+1$.

GRU reset and input gates operate similarly to standard units in a recurrent network. The distinctive feature of GRU lies in the method by which the activation of its units is determined. A GRU unit $u \in H$ has an associated candidate activation $\tilde{y}_u(\tau+1)$ at time $\tau+1$, formally defined by:
$\tilde{y}_u(\tau+1) = f_u\left(\sum_{i\in I} W[u,i]y_i(\tau+1) + y_{\text{res}_u}(\tau+1)\sum_{h\in H} W[u,h]y_h(\tau) + b_u\right)$

10 Applications of LSTM-RNN

In this concluding segment, we delve into a range of renowned publications that have remained significant over time.

10.1 Early learning tasks

Early experiments with LSTM demonstrated its versatility in tackling learning tasks that were previously deemed challenging. These tasks included recalling high-precision real numbers amidst noisy sequences, learning context-free languages, and handling tasks involving precise timing and counting. Additionally, LSTM was effectively applied in meta-learning, as evidenced by its success in program search tasks aimed at approximating learning algorithms for quadratic functions. Furthermore, LSTM showcased its capability in reinforcement learning, particularly in solving non-Markovian learning tasks with long-term dependencies.

10.2 Cognitive learning tasks

STM-RNNs, or Short-Term Memory Recurrent Neural Networks, have demonstrated significant prowess in tackling diverse cognitive learning tasks. Predominantly highlighted in literature are their effectiveness in speech and handwriting recognition, as well as more recent applications in machine translation. Additionally, STM-RNNs have shown promise in tasks such as emotion recognition from speech, text generation, handwriting generation, constituency parsing, and conversational modeling.

10.2.1 Speech recognition

In this section an overview of the evolution and advancements of neural networks in natural language-related tasks, particularly focusing on speech recognition discussed. It begins with the early successes of LSTM-RNN networks in language modeling tasks and their application to speech recognition in 2003. Bidirectional training with BLSTM improved results, eventually outperforming HMM-based systems. Further advancements include stacked BLSTM-CTC models with modified objective functions, achieving outstanding performance. LSTM/HMM hybrid architectures also yielded comparable results. Recent developments include LSTM utilization in sequence-to-sequence frameworks and attention-based learning. Specialized architectures like the 'listener' and 'attend and spell' functions, incorporating BLSTM with pyramid structure and attention-based LSTM transducer, respectively, have demonstrated significant improvements in speech recognition tasks.

10.2.2 Handwriting recognition

its introduction in 2007 and subsequent advancements, including combining it with probabilistic language models for direct transcription of raw online handwriting data. Additionally, it highlights the system's high automation rate and comparable error rate to humans in real-world scenarios. Another application involved combining BLSTM-CTC with multidimensional LSTM for offline handwriting recognition, surpassing classifiers based on Hidden Markov models. Finally, in 2013, the successful regularization method dropout was applied to further improve the performance of these systems.

10.2.3 Machine translation

In 2014, the authors utilized this architecture to enhance the performance of a statistical machine translation system. This approach, based on prior research, was further validated by subsequent studies. Additionally, researchers tackled challenges such as rare word translation and the translation of long sentences, with improvements achieved through techniques like sequence-to-sequence learning and attention mechanisms.

10.2.4 Image processing

BSLTM, a machine learning technique, showed superior performance in tasks like keyword spotting and mode detection in handwritten documents, outperforming other methods like HMMs and SVMs. Additionally, in 2012, there were advancements in the classification of high-resolution images from the ImageNet database. In 2015, LSTM variants were successfully used to generate English sentences describing images and combined with deep hierarchical visual feature extractors for tasks such as activity recognition and image/video description.

10.2.5 Other learning tasks

Early papers explored the application of LSTM-RNN to various real-world problems, such as protein secondary structure prediction and music generation. Additionally, researchers applied LSTM-RNN to network security, specifically the DARPA intrusion detection dataset. Computational tasks were also tackled using LSTM-RNN, with one study evaluating short computer programs using the Sequence to Sequence framework in 2014, and another study using a modified version of the framework to learn solutions for combinatorial optimization problems in 2015.

11 Conclusions

The article delves into the derivation of Long Short-Term Memory (LSTM) networks, focusing on addressing the vanishing error problem inherent in traditional Recurrent Neural Networks (RNNs). LSTM tackles this issue by maintaining a constant error flow through special memory cells, enabling it to handle long time-lag problems spanning over 1,000 time steps. Additionally, two LSTM extensions are introduced: self-resets, which allow the network to clear irrelevant memory, and precise timing learning.

12 Appendix

Figure in The Perceptron 2.1:

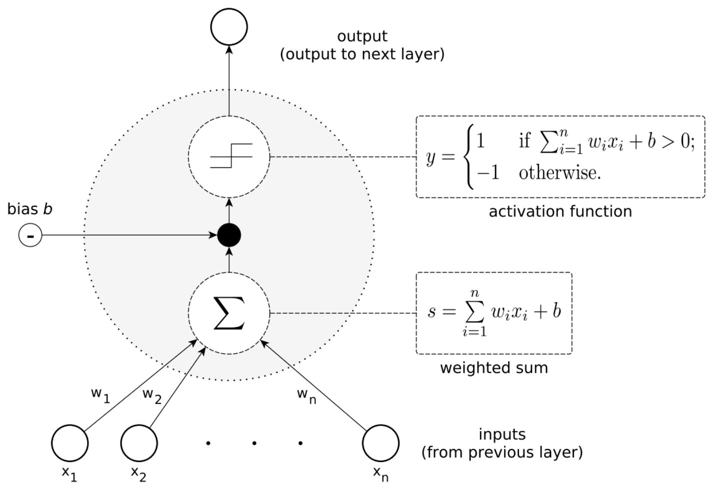
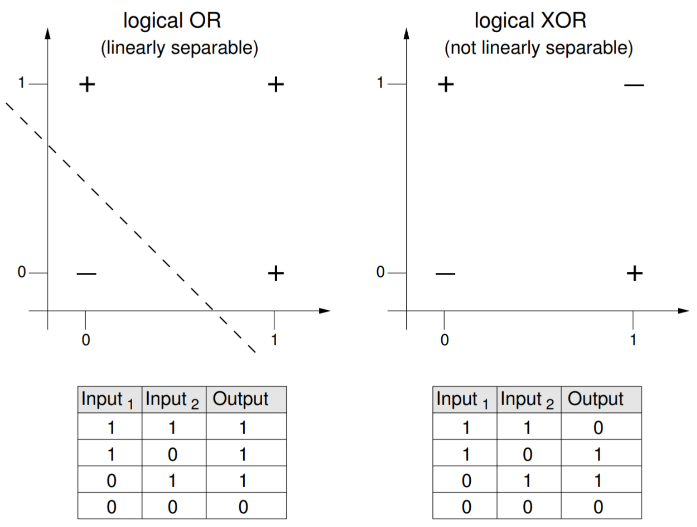

output
(output to next layer)
$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i + b > 0; \\ -1 & \text{otherwise.} \end{cases}$
activation function
bias b
-
$s = \sum_{i=1}^{n} w_i x_i + b$
weighted sum
w_1
w_2
w_n
inputs
(from previous layer)
x_1
x_2
x_n

Figure 1: The general structure of the most basic type of artificial neuron, called a perceptron. Single perceptrons are limited to learning linearly separable functions.

Figure in Linear Separability 2.2:



Input $_1$	Input $_2$	Output
1	1	1
1	0	1
0	1	1
0	0	0

Input $_1$	Input $_2$	Output
1	1	0
1	0	1
0	1	1
0	0	0

Figure 2: Representations of the Boolean functions OR and XOR. The figures show that the OR function is linearly separable, whereas the XOR function is not.

Figure in The Sigmoid Threshold Unit 2.4:

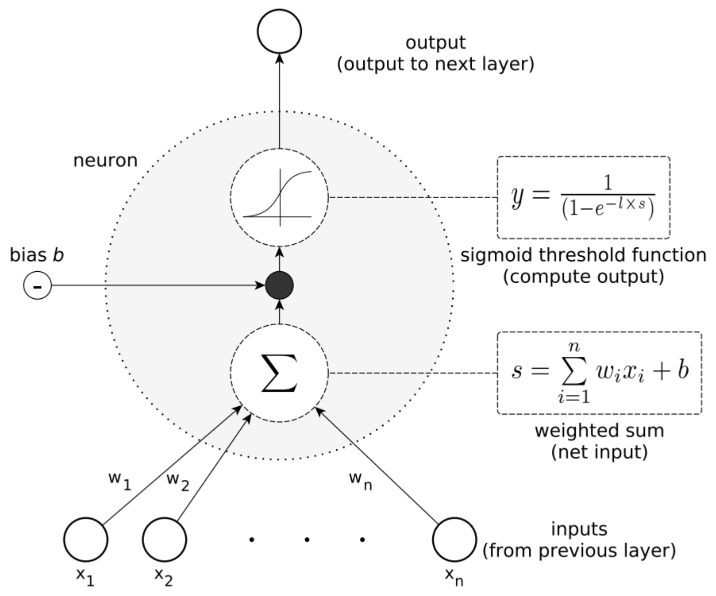


Figure 3: The sigmoid threshold unit is capable of representing non-linear functions. Its output is a continuous function of its input, which ranges between 0 and 1.

Figures in Feed-Forward Neural Networks and Backpropagation 3:

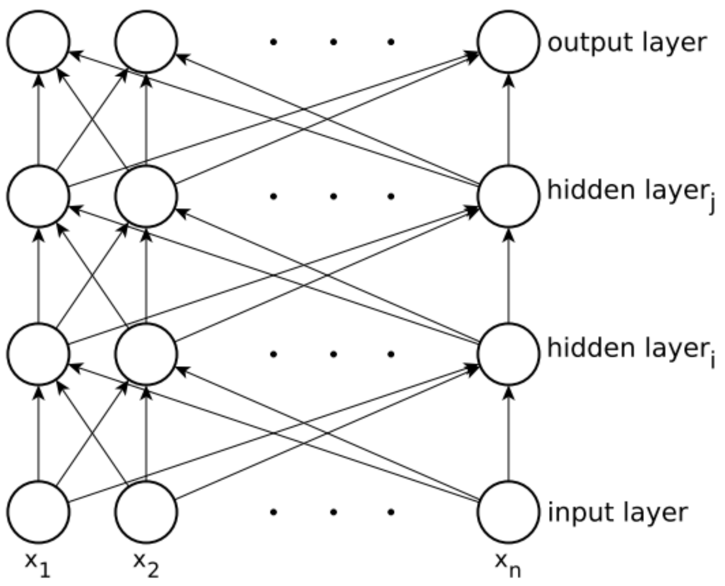


Figure 4: A multilayer feed-forward neural network with one input layer, two hidden layers, and an output layer. Using neurons with sigmoid threshold functions, these neural networks are able to express non-linear decision surfaces.

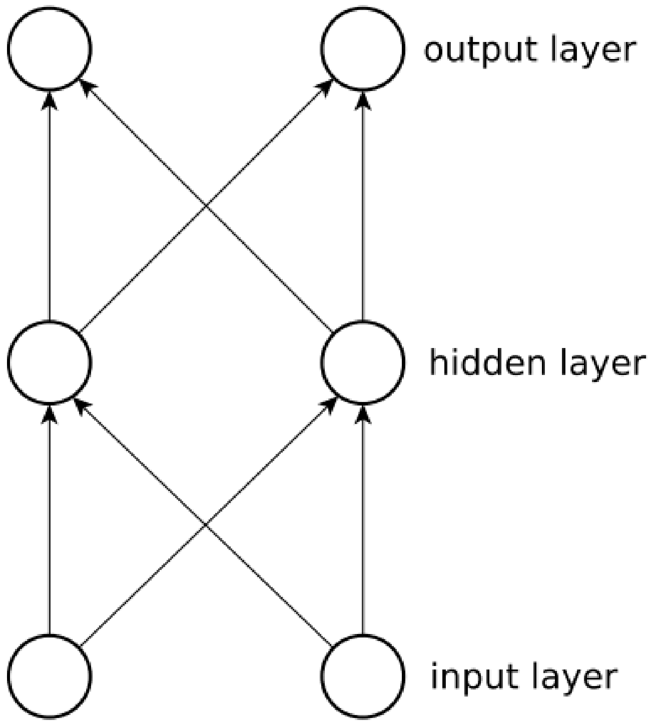


Figure 5: This figure shows a feed-forward neural network.

Figures in the Basic Architecture 4.1:

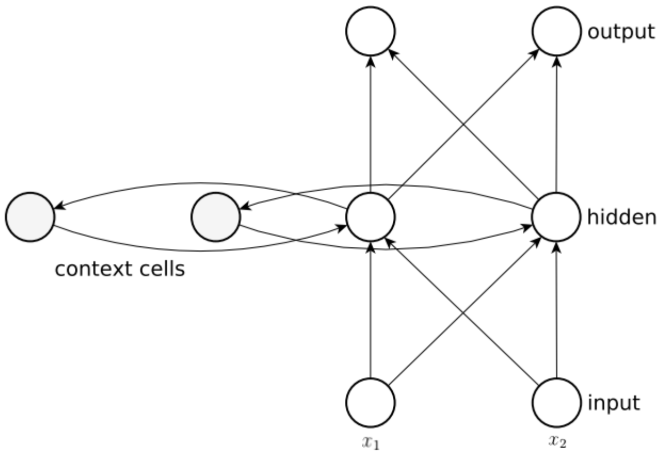


Figure 6: This figure shows an Elman neural network.

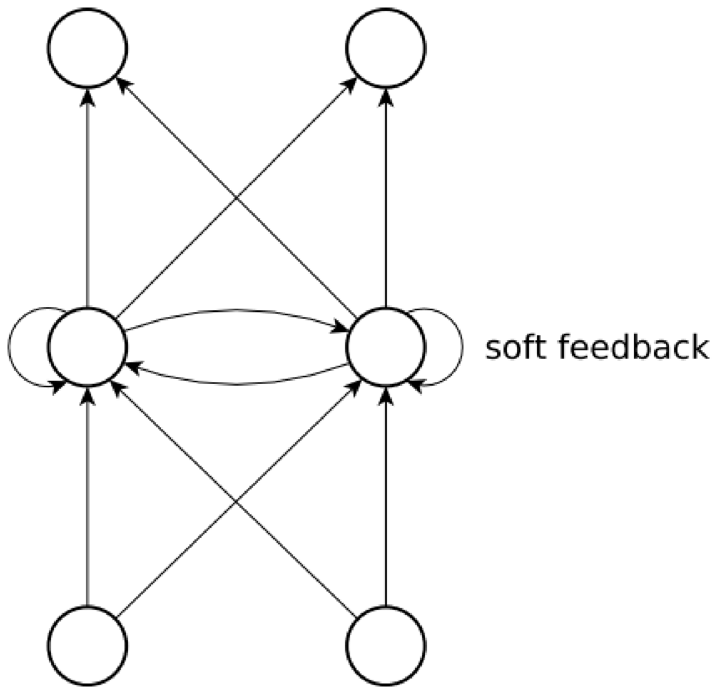


Figure 7: This figure shows a partially recurrent neural network with selffeedback in the hidden layer.

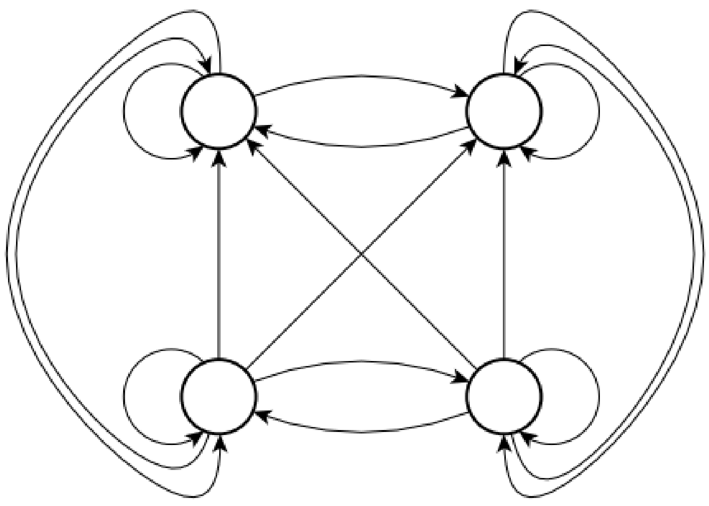


Figure 8: This figure shows a fully recurrent neural network (RNN) with selffeedback connections.

Figure in Backpropagation Through Time 5.1:

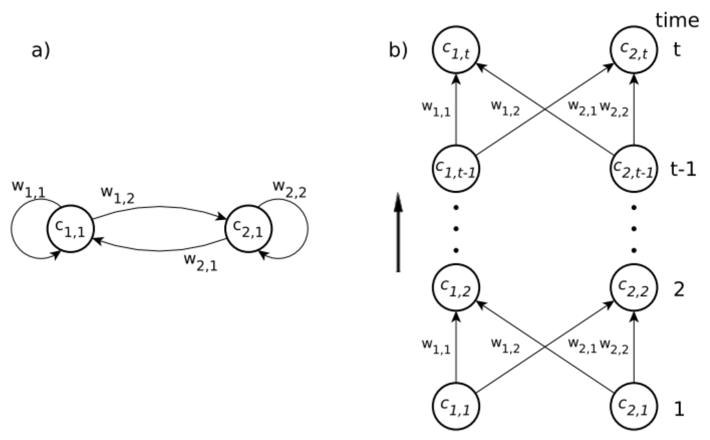


Figure 9: Figure a shows a simple fully recurrent neural network with a twoneuron layer. The same network unfolded over time with a separate layer for each time step is shown in Figure b. The latter representation is a feed-forward neural network.

Figures in The Forward Pass 8.1:

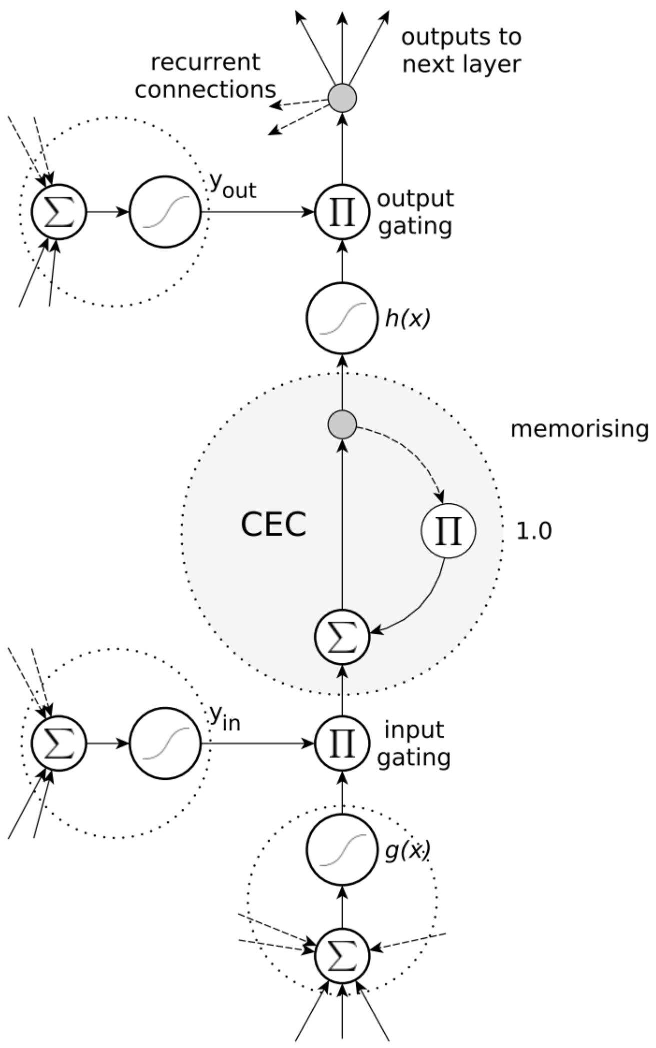


Figure 10: A standard LSTM memory block. The block contains (at least) one cell with a recurrent self-connection (CEC) and weight of '1'. The state of the cell is denoted as s_c. Read and write access is regulated by the input gate, y_{in}, and the output gate, y_{out}. The internal cell state is calculated by multiplying the result of the squashed input, g, by the result of the input gate, y_{in}, and then adding the state of the last time step, $s_c(t-1)$. Finally, the cell output is calculated by multiplying the cell state, s_c, by the activation of the output gate, y_{out}.

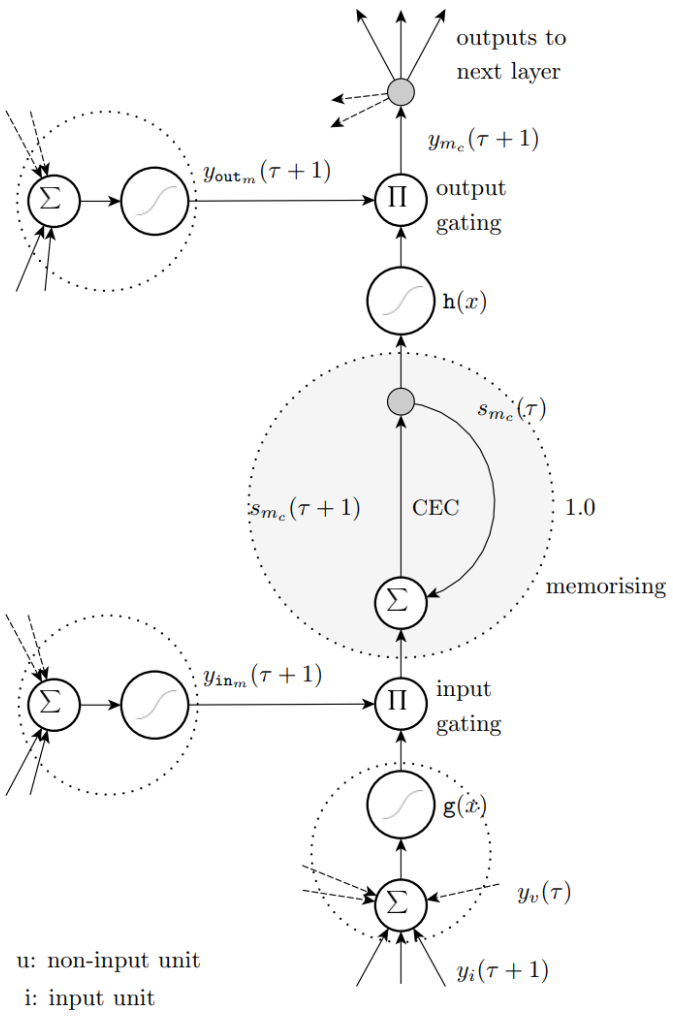


Figure 11: A standard LSTM memory block. The block contains (at least) one cell with a recurrent self-connection (CEC) and weight of '1'. The state of the cell is denoted as s_c. Read and write access is regulated by the input gate, y_{in}, and the output gate, y_{out}. The internal cell state is calculated by multiplying the result of the squashed input, $g(x)$, by the result of the input gate and then adding the state of the current time step, $s_{m_c}(\tau)$, to the next, $s_{m_c}(\tau+1)$. Finally, the cell output is calculated by multiplying the cell state by the activation of the output gate.

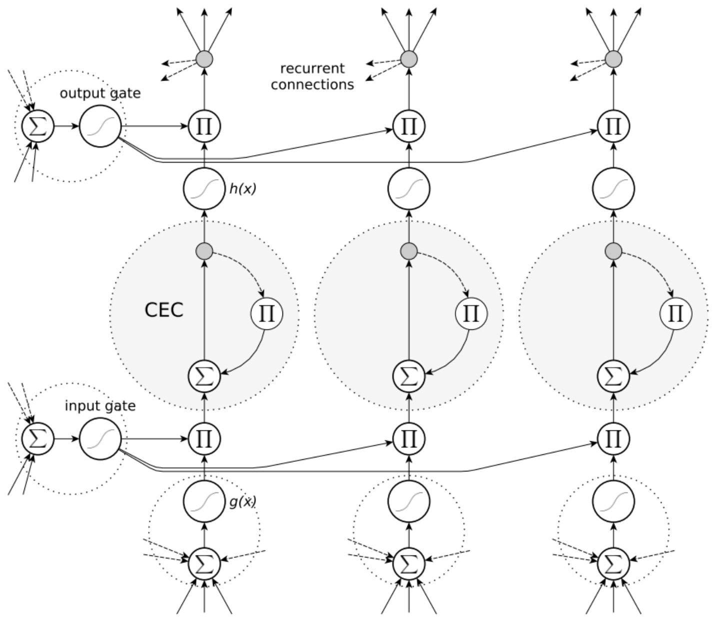


Figure 12: A three cell LSTM memory block with recurrent self-connections.