Published in final edited form as:
Neural Netw. 2012 January ; 25(1): 70–83. doi:10.1016/j.neunet.2011.07.003.

A generalized LSTM-like training algorithm for second-order recurrent neural networks

Derek Monner* and **James A. Reggia**
Department of Computer Science, University of Maryland, College Park, MD 20742, USA

Abstract

The Long Short Term Memory (LSTM) is a second-order recurrent neural network architecture that excels at storing sequential short-term memories and retrieving them many time-steps later. LSTM's original training algorithm provides the important properties of spatial and temporal locality, which are missing from other training approaches, at the cost of limiting it's applicability to a small set of network architectures. Here we introduce the Generalized Long Short-Term Memory (LSTM-g) training algorithm, which provides LSTM-like locality while being applicable without modification to a much wider range of second-order network architectures. With LSTM-g, all units have an identical set of operating instructions for both activation and learning, subject only to the configuration of their local environment in the network; this is in contrast to the original LSTM training algorithm, where each type of unit has its own activation and training instructions. When applied to LSTM architectures with peephole connections, LSTM-g takes advantage of an additional source of back-propagated error which can enable better performance than the original algorithm. Enabled by the broad architectural applicability of LSTM-g, we demonstrate that training recurrent networks engineered for specific tasks can produce better results than single-layer networks. We conclude that LSTM-g has the potential to both improve the performance and broaden the applicability of spatially and temporally local gradient-based training algorithms for recurrent neural networks.

Keywords

recurrent neural network; gradient-based training; long short term memory (LSTM); temporal sequence processing; sequential retrieval

1. Introduction

The Long Short Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) is a recurrent neural network architecture that combines fast training with efficient learning on tasks that require sequential short-term memory storage for many time-steps during a trial. Since its inception, LSTM has been augmented and improved with forget gates (Gers & Cummins, 2000) and peephole connections (Gers & Schmidhuber, 2000); despite this, the usefulness of the LSTM training algorithm is limited by the fact that it can only train a small set of

*Corresponding author. Telephone: +1 (301) 586-2495. Fax: (301) 405-6707. dmonner@cs.umd.edu (Derek Monner), reggia@cs.umd.edu (James A. Reggia) .

second-order recurrent network architectures. By *second-order neural network* we mean one that not only allows normal weighted connections which propagate activation from one sending unit to one receiving unit, but also allows second-order connections: weighted connections from *two* sending units to one receiving unit, where the signal received is dependent upon the product of the activities of the sending units with each other and the connection weight (Miller & Giles, 1993). When LSTM is described in terms of connection gating, as we discuss below, we see that the gate units serve as the additional sending units for the second-order connections.

The original LSTM architecture has an input layer, a hidden layer consisting of *memory block* cell assemblies, and an output layer. Each memory block is composed of *memory cell* units that retain state across time-steps, as well as three types of specialized gate units that learn to protect, utilize, or destroy this state as appropriate. The LSTM training algorithm back-propagates errors from the output units through the memory blocks, adjusting incoming connections of all units in the blocks, but then truncates the back-propagated errors. As a consequence, LSTM's training algorithm cannot be used to effectively train second-order networks with units placed between the memory blocks and the input layer. More generally, LSTM's training algorithm cannot be used to train arbitrary second-order recurrent neural architectures, as the error propagation and weight updates it prescribes are dependent upon the specific network architecture described in the original paper.

While there exist other methods capable of training arbitrary second-order networks, we are aware of none that share a principal advantage of LSTM's training algorithm: locality in time and space. By *spatial locality* we mean that the weight changes that happen at some point in the network should be directly computable from information available within the spatial neighborhood of the connection in question. Similarly *temporal locality* means that weight changes cannot rely upon records of information from arbitrarily far in the past. A training algorithm that possesses these properties can, if it is general enough, be applied without modification to any network architecture, and in fact even to architectures that change during training. The human brain is an obvious example of an architecture that changes during learning, and our concerns about spatial and temporal locality might be summarized as a desire to maintain the locality constraints that brains appear to have.

Error back-propagation for feed-forward networks (Rumelhart et al., 1986) and Simple Recurrent Networks (SRNs; Elman, 1990) are examples of training algorithms that exhibit locality in time and space while generalizing to a wide variety of architectures. To our knowledge no such algorithm exists for arbitrary second-order networks. Algorithms like Back-Propagation Through Time (BPTT; Werbos, 1990) that have been used to train second order architectures (e.g., Graves & Schmidhuber, 2008) violate our desire for temporal locality by basing weight updates on perfect records of network activations extending back in time to the beginning of arbitrarily long input sequences; the same goes for the evolutionary training method known as Evolino (Schmidhuber et al., 2007). Real-Time Recurrent Learning (RTRL; Williams & Zipser, 1989) is not local spatially since the gradient term for a given weight depends directly on every other weight in the network. Decoupled Extended Kalman Filters (DEKF; Gers et al., 2003; Puskorius & Feldkamp, 1994) utilize a host of external matrix operations to control training, and thus are not spatially local. While this list is not exhaustive, every such training algorithm of which we are aware, other than LSTM, is either spatially or temporally non-local.

Motivated by a desire for a spatially and temporally local, architecture-independent training algorithm, we developed what we call Generalized Long Short-Term Memory (LSTM-g), a training algorithm that retains the spatial an temporal locality of the original LSTM training algorithm and can be applied, without modification, to a much wider range of second-order

recurrent neural networks. Each unit in a network trained by LSTM-g has the same set of operating instructions, relying only on its local network environment to determine whether it will fulfill the role of memory cell, gate unit, both, or neither. In addition, every unit is trained by LSTM-g in the same way—a sharp contrast from the original LSTM training algorithm, where each of several different types of units has a unique training regimen. LSTM-g reinterprets the gating of unit activations seen in LSTM; instead of gate units modulating other unit activations directly, they are viewed as modulating the weights on connections between units. This change in perspective, while mathematically equivalent to the original design, offers increased flexibility to network designers that wish to explore arbitrary architectures where gates can temporarily isolate one part of the network from another. While previous work (Bayer et al., 2009) has examined alternative architectures for LSTM-style second-order networks—as derived by network evolution to suit a particular task—that work relies on the non-local BPTT algorithm to train the evolved networks. While our work also focuses on alternative second-order architectures, our primary aim is to provide a local algorithm to train alternative LSTM-style architectures.

In addition to the expanded architectural applicability that it a ords, LSTM-g provides all of the benefits of the LSTM training algorithm when applied to the right type of architecture. LSTM-g was designed to perform exactly the same weight updates as the original algorithm when applied to identical network architectures. However, on LSTM architectures with peephole connections, LSTM-g often performs better than the original algorithm by utilizing a source of back-propagated error that appears to have heretofore gone unnoticed.

In what follows, we present LSTM and its training algorithm as previously described, followed by the generalized version. We then present an analysis of the additional error signals which LSTM-g uses to its advantage, followed by experimental evidence that LSTM-g often performs better than the LSTM algorithm when using the original architecture. Further experiments show that LSTM-g performs well on two architectures specifically adapted to two computational problems. The appendices give the mathematical derivation of the LSTM-g learning rules and prove that LSTM-g is a generalization of LSTM training.

2. LSTM

The LSTM architecture was developed as a neural network architecture for processing long temporal sequences of data, and is trained using a hybrid descendant of truncated BPTT and RTRL. Other recurrent neural networks trained with various gradient methods proved to be ineffective when the input sequences were too long (Hochreiter & Schmidhuber, 1997). Analysis showed that for neural networks trained with back-propagation or other gradient-based methods, the error signal is likely to vanish or diverge as error travels backward through network space or through time. This is because, with every pass backwards through a unit, the error signal is scaled by the derivative of the unit's activation function times the weight that the forward signal traveled along. The further error travels back in space or time, the more times this scaling factor is multiplied into the error term. If the factor is consistently less than 1, the error will vanish, leading to small, ineffective weight updates; if it is greater than 1, the error term will diverge, potentially leading to weight oscillations or other types of instability. One way to preserve the value of the error is requiring the scaling factor to be equal to 1, which can only be consistently enforced with a linear activation function (whose derivative is 1) and a fixed weight of $w_{jj} = 1$. LSTM adopts this requirement for its *memory cell* units, which have linear activation functions and self-connections with a fixed weight of 1 (Fig. 1a). This allows them to maintain unscaled activations and error derivatives across arbitrary time lags if they are not otherwise disturbed. Since back-propagation networks require nonlinear hidden unit activation functions to be effective, each

memory cell's state is passed through a squashing function—such as the standard logistic function—before being passed on to the rest of the network.

The processing of long temporal sequences is complicated by the issue of interference. If a memory cell is currently storing information that is not useful now but will be invaluable later, this currently irrelevant information may interfere with other processing in the interim. This in turn may cause the information to be discarded, improving performance in the near term but harming it in the long term. Similarly, a memory cell may be perturbed by an irrelevant input, and the information that would have been useful later in the sequence can be lost or obscured. To help mitigate these issues, each memory cell has its net input modulated by the activity of another unit, termed an *input gate*, and has its output modulated by a unit called an *output gate* (Fig. 1a). Each input gate and output gate unit modulates one or a small number of memory cells; the collection of memory cells together with the gates that modulate them is termed a *memory block*. The gates provide a context-sensitive way to update the contents of a memory cell and protect those contents from interference, as well as protecting downstream units from perturbation by stored information that has not become relevant yet. A later innovation was a third gate, termed the *forget gate*, which modulates amount of activation a memory cell keeps from the previous time-step, providing a method to quickly discard the contents of memory cells after they have served their purpose (Gers & Cummins, 2000).

In the original formulation of LSTM, the gate units responsible for isolating the contents of a given memory cell face a problem. These gates may receive input connections from the memory cell itself, but the memory cell's value is gated by its output gate. The result is that, when the output gate is closed (i.e., has activity near zero), the memory cell's visible activity is near zero, hiding its contents even from those cells—the associated gates—that are supposed to be controlling its information flow. Recognition of this fact resulted in the inclusion of *peephole connections*—direct weighted connections originating from an intermediate stage of processing in the memory cell and projecting to each of the memory cell's gates (Gers & Schmidhuber, 2000). Unlike all other connections originating at the memory cell, the peephole connections see the memory cell's state *before* modulation by the output gate, and thus are able to convey the true contents of the memory cell to the associated gates at all times. By all accounts, peephole connections improve LSTM performance significantly (Gers & Schmidhuber, 2001), leading to their adoption as a standard technique employed in applications (Graves et al., 2004; Gers et al., 2003).

The LSTM network ostensibly has only three layers: an input layer, a layer of memory block cell assemblies, and an output layer. For expository purposes, however, it will be useful to think of the memory block assemblies as composed of multiple separate layers (see Fig. 2): the input gate layer (ι), the forget gate layer (φ), the memory cell layer (c), and the output gate layer (ω). For notational simplicity, we will stipulate that each of these layers has the same number of elements, implying that a single memory cell c_j is associated with the set of gates ι_j, φ_j, and ω_j; it is trivial to generalize to the case where a set of gates can control more than one memory cell. The input layer projects a full set of connections to each of these layers; the memory cell layer projects a full set of connections to the output layer (θ). In addition, each memory cell c_j projects a single ungated peephole connection to each of its associated gates (see Fig. 1a). The architecture can be augmented with direct input-to-output connections and/or delayed recurrent connections among the memory cell and gate layers. As will become evident in the following sections, the operation of the LSTM training algorithm is very much dependent upon the specific architecture that we have just described.

The following equations detail the operation of the LSTM network and its original training algorithm through a single time-step. We consider a time-step to consist of the presentation

of a single input, followed by the activation of all subsequent layers of the network in order. We think that this notion of a time-step, most often seen when discussing feed-forward networks, enables a more intuitive description of the activation and learning phases that occur due to each input. Following Graves et al. (2004), all variables in the following refer to the most recently calculated value of that variable (whether during this time-step or the last), with the exception that variables with a hat (^) always refer to the value calculated one time-step earlier; this only happens in cases where the new value of a variable is being defined in terms of its immediately & preceding value. Following Gers & Schmidhuber (2000), we deviate from the original description of LSTM by reducing the number of squashing functions for the memory cells; here, however, we omit the input-squashing function g (equivalent to defining $g(x) = x$) and retain the output-squashing function, naming it f_{c_j} for memory cell j. In general, we will use the subscript index j to refer to individual units within the layer in question, with i running over all units which project connections to unit j, and k running over all units that receive connections from j.

2.1. Activation dynamics

When an input is presented, we proceed through an entire time-step, activating each layer in order: ι, φ, c, ω, and finally the output layer θ. In general, when some layer λ is activated, each unit λ_j in that layer calculates its *net input* x_{λ_j} as the weighted sum over all its input connections from units i (Eq. 1). The units i vary for each layer and potentially include recurrent connections; the most recent activation value of the sending unit is always used, even if it is from the previous time-step as for recurrent connections. For units that are not in the memory cell layer, the *activation* y_{λ_j} of the unit is the result of applying the unit's *squashing function* f_{λ_j} (generally taken to be the logistic function) to its net input (Eq. 2).

Each memory cell unit, on the other hand, retains its previous state $\widehat{s_{c_j}}$ in proportion to the activation of the associated forget gate; current *state* s_{c_j} is updated by the net input modulated by the activation of the associated input gate (Eq. 3). A memory cell's state is passed through its squashing function and modulated by the activation of its output gate to produce the cell's activation (Eq. 4).

$$x_{\lambda_j} \sum_i w_{\lambda_{ji}} \quad y_i \quad \text{for} \quad \lambda \in \{\iota, \varphi, c, \omega, \theta\} \tag{1}$$

$$y_{\lambda_j} = f_{\lambda_j}\left(x_{\lambda_j}\right) \quad \text{for} \quad \lambda \in \{\iota, \varphi, \omega, \theta\} \tag{2}$$

$$s_{c_j} = y_{\varphi_j} \quad \widehat{s_{c_j}} + y_{\iota_j} x_{c_j} \tag{3}$$

$$y_{c_j} = y_{\omega_j} \quad f_{c_j}\left(s_{c_j}\right) \tag{4}$$

When considering peephole connections in the context of the equations in this section, one should replace the sending unit activation y_i with the memory cell state s_{c_i} since peephole connections come directly from the internal state of the memory cells rather than their activations.

2.2. Learning rules

In order to learn effectively, each unit needs to keep track of the activity flow over time through each of its connections. To this end, each unit maintains an *eligibility trace* for each of its input connections, and updates this trace immediately after calculating its activation. The eligibility trace for a given connection is a record of activations that have crossed the connection which may still have influence over the state of the network, and is similar to those used in temporal-difference learning (Sutton & Barto, 1998), except that here eligibility traces do not decay. When a target vector is presented, the eligibility traces are used to help assign error responsibilities to individual connections. For the output gates and output units, the eligibility traces are instantaneous—they are simply the most recent activation value that crossed the connection (Eq. 5). For the memory cells (Eq. 6), forget gates (Eq. 7), and input gates (Eq. 8), the eligibility traces are partial derivatives of the state of the memory cell with respect to the connection in question; simplifying these partial derivatives results in Eqs. 6-8. Previous eligibility traces are retained in proportion to the amount of state that the memory cell retains (i.e., the forget gate activation $y_{\varphi j}$), and each is incremented according to the effect it has on the memory cell state.

$$\varepsilon_{\lambda_{ji}} = y_i \quad \text{for} \quad \lambda \in \{\omega, \theta\} \tag{5}$$

$$\varepsilon_{c_{ji}} = y_{\varphi_j} \quad \widehat{\varepsilon_{c_{ji}}} + y_{\iota_j} \quad y_i \tag{6}$$

$$\varepsilon_{\varphi_{ji}} = y_{\varphi_j} \quad \widehat{\varepsilon_{\varphi_{ji}}} + \widehat{s_{c_j}} \quad f'_{\varphi_j}\left(x_{\varphi_j}\right) \quad y_i \tag{7}$$

$$\varepsilon_{\iota_{ji}} = y_{\varphi_j} \quad \widehat{\varepsilon_{\iota_{ji}}} + x_{c_j} \quad f'_{\iota_j} \quad \left(x_{\iota_j}\right) \quad y_i \tag{8}$$

Between time-steps of the activation dynamics (i.e., after the network has generated an output for a given input), the network may be given a target vector t to compare against, where all values in t are in the range [0, 1]. The difference between the network output and the target is calculated using the cross-entropy function (Eq. 9) (Hinton, 1989). Since $E \leq 0$ when the t and y values fall in the range [0, 1] as we require, one trains the network by driving this function towards zero using gradient ascent. Deriving Eq. 9 with respect to the output unit activations reveals the *error responsibility* δ_{θ_j} for the output units (Eq. 10). One obtains the deltas for the output gates (Eq. 11) and the remaining units (Eq. 12) by propagating the error backwards through the network.

$$E = \sum_{j \in \theta} \left(t_j \quad \log\left(y_{\theta_j}\right) + \left(1 - t_j\right) \quad \log\left(1 - y_{\theta_j}\right)\right) \tag{9}$$

$$\delta_{\theta_j} = t_j - y_{\theta_j} \tag{10}$$

$$\delta_{\omega_j} = f'_{\omega_j}\left(x_{\omega_j}\right) \quad f_{c_j}\left(s_{c_j}\right) \sum_{k \in \theta} \delta_{\theta_k} \quad w_{\theta_k c_j} \tag{11}$$

$$\delta_{\lambda_j} = f'_{c_j}\left(s_{c_j}\right) \quad y_{\omega_j} \sum_{k \in \theta} \delta_{\theta_k} \quad w_{\theta_k c_j} \quad \text{for} \quad \lambda \in \{\iota, \varphi, c\} \tag{12}$$

Finally, the connection weights into all units in each layer λ are updated according to the product of the learning rate α, the unit's error responsibility δ_{λ_j}, and the connection's eligibility trace $\varepsilon_{\lambda_{ji}}$ (Eq. 13).

$$\Delta w_{\lambda_{ji}} = \alpha \quad \delta_{\lambda_j} \quad \varepsilon_{\lambda_{ji}} \quad \text{for} \quad \lambda \in \{\iota, \varphi, c, \omega, \theta\} \tag{13}$$

3. Generalized LSTM

The Long Short Term Memory algorithm, as presented in Section 2, is an e cient and powerful recurrent neural network training method, but is limited in applicability to the architecture shown in Fig. 2 and sub-architectures thereof[1]. In particular, any architectures with multiple hidden layers (where another hidden layer projects to the memory block layer) cannot be e ciently trained because error responsibilities are truncated at the memory blocks instead of being passed to upstream layers. This section details our generalized version of LSTM training, which confers all the benefits of the original algorithm, yet can be applied without modification to arbitrary second-order neural network architectures.

With the Generalized Long Short-Term Memory (LSTM-g) approach, the gating mechanism employed by LSTM is reinterpreted. In LSTM, gate units directly act on the states of individual units—a memory cell's net input in the case of the input gate, the memory cell state for the forget gate, and the memory cell output for the output gate (Eqs. 3-4). By contrast, units in LSTM-g can gate at the level of individual connections. The effect is that, when passing activity to unit j from unit i across a connection gated by k, the result is not simply $w_{ji}\, y_i$, but instead $w_{ji}\, y_i\, y_k$. In this sense, LSTM-g is similar in form to traditional second-order networks (e.g., Giles & Maxwell, 1987; Psaltis et al., 1988; Shin & Ghosh, 1991; Miller & Giles, 1993), but with an asymmetry: Our notation considers the connection in this example to be primarily defined by j and i (note that the weight is denoted w_{ji} and not w_{jki}), where k provides a temporary *gain* on the connection by modulating its weight multiplicatively. This notation is convenient when considering connections which require an output and an input, but may or may not be gated; in other words, we can refer to a connection without knowing whether it is a first- or second-order connection.

In LSTM-g, every unit has the potential to be like LSTM's memory cells, gate units, both, or neither. That is to say, all units contain the same set of operating instructions for both activation and learning. Self-connected units can retain state like a memory cell, and any unit can directly gate any connection. The role each unit takes is completely determined by its placement in the overall network architecture, leaving the choice of responsibilities for each unit entirely up to the architecture designer.

[1]LSTM can also train architectures with additional layers that operate in *parallel* with the memory block layer, but the important point here is that LSTM cannot effectively train architectures containing layers that operate in *series* with the memory block layer.

Equations 14-24 describe the operation of LSTM-g networks through a single time-step. Just as in our description of the original LSTM training algorithm, a time-step consists of the presentation of a single input pattern, followed by the ordered activation of all non-input layers of the network. The order in which layers are activated is pre-determined and remains fixed throughout training. If a layer to be activated receives recurrent connections from a layer which has not yet been activated this time-step, the sending layer's activations from the previous time-step are used. The full derivation of the LSTM-g training algorithm can be found in Appendix A.

3.1. Activation dynamics

LSTM-g performs LSTM-like gating by having units modulate the effectiveness of individual connections. As such, we begin by specifying the *gain* g_{ji} on the connection from unit i to unit j (Eq. 14).

$$g_{ji} = \begin{cases} 1 & \text{if connection from } i \text{ to } j \text{ is not gated} \\ y_k & \text{if unit } k \text{ gates the connection from } i \text{ to } j \end{cases} \tag{14}$$

Much as with memory cells in LSTM, any unit in an LSTM-g network is capable of retaining state from one time-step to the next, based only on whether or not it is self-connected. The *state* s_j of a unit j (Eq. 15) is the sum of the weighted, gated activations of all the units that project connections to it. If the unit is self-connected it retains its state in proportion to the gain on the self-connection. As in LSTM, self-connections in LSTM-g, where they exist, have a fixed weight of 1; otherwise $w_{jj} = 0$. Given the state s_j, the *activation* y_j is calculated via the application of the unit's *squashing function* f_j (Eq. 16).

$$s_j = g_{jj} \; w_{jj} \; \widehat{s_j} + \sum_{i \neq j} g_{ji} \; w_{ji} \; y_i \tag{15}$$

$$y_j = f_j\left(s_j\right) \tag{16}$$

When considering these equations as applied to the LSTM architecture, for a unit $j \notin c$ we can see that Eq. 15 is a generalization of Eq. 1. This is because the first term of Eq. 15 evaluates to zero on this architecture (since there is no self-connection w_{jj}), and all the $g_{ji} = 1$ since no connections into the unit are gated. The equivalence of Eq. 16 and Eq. 2 for these units follows immediately. For a memory cell $j \in c$, on the other hand, Eq. 15 reduces to Eq. 3 when one notes that the self-connection gain g_{jj} is just $y_{\varphi j}$, the self-connection weight w_{jj} is 1, and the g_{ji} are all equal to $y_{\iota j}$ and can thus be pulled outside the sum. However, for the memory cell units, Eq. 16 is not equivalent to Eq. 4, since the latter already multiplies in the activation $y_{\omega j}$ of the output gate, whereas this modulation is performed at the connection level in LSTM-g.

3.2. Learning rules

As in LSTM, each unit keeps an *eligibility trace* ε_{ji} for each of its input connections (Eq. 17). This quantity keeps track of how activity that has crossed this connection has influenced the current state of the unit, and is equal to the partial derivative of the state with respect to the connection weight in question (see Appendix A). For units that do not have a self-connection, the eligibility trace ε_{ji} reduces to the most recent input activation modulated by the gating signal.

$$\varepsilon_{ji} = g_{jj} \quad w_{jj} \quad \widehat{\varepsilon_{ji}} + g_{ji} \quad y_i \tag{17}$$

In the context of the LSTM architecture, Eq. 17 reduces to Eq. 5 for the output gates and output units; in both cases, the lack of self-connections forces the first term to zero, and the remaining $g_{ji}\, y_i$ term is equivalent to LSTM's y_i.

If unit j gates connections into other units k, it must maintain a set of *extended eligibility traces* ε_{ji}^{k} for each such k (Eq. 18). A trace of this type captures the effect that the connection from i potentially has on the state of k through its influence on j. Eq. 18 is simpler than it appears, as the remaining partial derivative term is 1 if and only if j gates k's self-connection, and 0 otherwise. Further, the index a, by definition, runs over only those units whose connections to k are gated by j; this set of units may be empty.

$$\varepsilon_{ji}^{k} = g_{kk} \quad w_{kk} \quad \widehat{\varepsilon_{ji}^{k}} + f_k'\left(s_j\right) \quad \varepsilon_{ji} \left(\frac{\partial_{g_{kk}}}{\partial_{y_j}} w_{kk} \quad \widehat{s_k} + \sum_{a \neq k} w_{ka} \quad y_a \right) \tag{18}$$

It is worth noting that LSTM uses traces of exactly this type for the forget gates and input gates (Eq. 7-8); it just so happens that each such unit gates connections into exactly one other unit, thus requiring each unit to keep only a single, unified eligibility trace for each input connection. This will be the case for the alternative architectures we explore in this paper as well, but is not required. A complete explanation of the correspondence between LSTM's eligibility traces and the extended eligibility traces utilized by LSTM-g can be found in Appendix B.

When a network is given a target vector, each unit must calculate its *error responsibility* δ_j and adjust the weights of its incoming connections accordingly. Output units, of course, receive their δ values directly from the environment based on the global error function (Eq. 9), just as in LSTM (Eq. 10). The error responsibility δ_j for any other unit j in the network can be calculated by back-propagating errors. Since each unit keeps separate eligibility traces corresponding to projected activity (ε_{ji}) and gating activity $\left(\varepsilon_{ji}^{k}\right)$, we divide the error responsibility accordingly. First, we define P_j to be the set of units k which are downstream from j—that is, activated after j during a time-step—and to which j projects weighted connections (Eq. 19), and G_j to be the set of units k which are downstream from j that receive connections gated by j (Eq. 20). We restrict both of these sets to downstream units because an upstream unit k has its error responsibility updated after j during backward error propagation, meaning that the error responsibility information provided by k is not available when j would need to use it.

$$P_j = \{k \mid j \text{ projects a connection to } k \text{ and } k \text{ is downstream of } j\} \tag{19}$$

$$G_j = \{k \mid j \text{ gates a connection into } k \text{ and } k \text{ is downstream of } j\} \tag{20}$$

We first find the error responsibility of unit j with respect to the projected connections in P_j (Eq. 21), which is calculated as the sum of the error responsibilities of the receiving units weighted by the gated connection strengths that activity from j passed over to reach k.

$$\delta_{P_j} = f_j'(s_j) \sum_{k \in P_j} \delta_k \quad g_{kj} \quad w_{kj} \tag{21}$$

Since the memory cells in LSTM only project connections and perform no gating themselves, δ_{P_j} of Eq. 21 translates directly into Eq. 12 for these units, which can be seen by noting that the gain terms g_{kj} are all equal to the output gate activation y_{ω_j} and can be pulled out of the sum.

The error responsibility of j with respect to gating activity is the sum of the error responsibilities of each unit k receiving gated connections times a quantity representing the gated, weighted input that the connections provided to k (Eq. 22). This quantity, as with the same quantity in Eq. 18, is simpler than it appears, with the partial derivative evaluating to 1 only when j is gating k's self-connection and zero otherwise, and the index a running over only those units projecting a connection to k on which j is the gate.

$$\delta_{G_j} = f_j'(s_j) \sum_{k \in G_j} \delta_k \left(\frac{\partial g_{kk}}{\partial y_j} w_{kk} \quad \widehat{s_k} + \sum_{a \neq k} w_{ka} \quad y_a \right) \tag{22}$$

To find j's total error responsibility (Eq. 23), we add the error responsibilities due to projections and gating.

$$\delta_j = \delta_{P_j} + \delta_{G_j} \tag{23}$$

In order to obtain weight-changes similar to LSTM training, δ_j is not used directly in weight adjustments; its purpose is to provide a unified δ value that can be used by upstream units to calculate their error responsibilities due to unit j. Instead, the weights are adjusted by combining the error responsibilities and eligibility traces for projected activity and adding the products of extended eligibility traces and error responsibilities of each unit receiving gated connections. The result is multiplied by the learning rate α (Eq. 24).

$$\Delta w_{ji} = \alpha \quad \delta_{P_j} \quad \varepsilon_{ji} + \alpha \sum_{k \in G_j} \delta_k \quad \varepsilon_{ji}^k \tag{24}$$

Appendix B provides a detailed derivation that shows that the weight changes made by both the LSTM (Eq. 13) and LSTM-g (Eq. 24) algorithms are identical when used on an LSTM architecture without peephole connections. This establishes that LSTM-g is indeed a generalization of LSTM's training algorithm.

4. Comparison of LSTM and LSTM-g

As stated in Section 3 and proved in Appendix B, an LSTM-g network with the same architecture as an LSTM network will produce the same weight changes as LSTM training would, provided that peephole connections are not present. When peephole connections are added to the LSTM architecture, however, LSTM-g utilizes a source of error that LSTM training neglects: error responsibilities back-propagated from the output gates across the peephole connections to the associated memory cells, and beyond. To see this, we will calculate the error responsibility for a memory cell j in an LSTM-g network, and compare the answer to the error responsibility for that same unit as prescribed by LSTM training.

We begin with the generic error responsibility equation from LSTM-g (Eq. 23). Since the cell in question is the architectural equivalent of a memory cell, it performs no gating functions; thus the set of gated cells G_j is empty and $_{G_j}$ is zero, leaving δ_{P_j} alone as the error responsibility. Substituting Eq. 21 for δ_{P_j} we obtain Eq. 25. At this point we ask: Which units are in P_j? The memory cell in question projects connections to all the output units and sends peephole connections to its controlling input gate, forget gate, and output gate. From this set of receiving units, only the output units and the output gate are downstream from the memory cell, so they comprise P_j. Taking each type of unit in P_j individually, we expand the sum and obtain Eq. 26. Finally, we recognize that the peephole connection to the output gate is not gated, so the $g_{\omega_j c_j}$ term goes to 1 (by Eq. 14); in addition, all the output connections are gated by the output gate, so every $g_{\theta_k c_j}$ term becomes y_{ω_j}, and we can pull the term outside the sum.

$$\delta_j = \delta_{P_j} = f'_{c_j}\left(s_{c_j}\right) \sum_{k \in P_j} \delta_k \quad g_{kj} \quad w_{kj} \tag{25}$$

$$= f'_{c_j}\left(s_{c_j}\right)\left(\sum_{k \in \theta} \delta_{\theta_k} \quad g_{\theta_k c_j} \quad w_{\theta_k c_j} + \delta_{\omega_j} \quad g_{\omega_j c_j} \quad w_{\omega_j c_j}\right) \tag{26}$$

$$= f'_{c_j}\left(s_{c_j}\right)\left(y_{\omega_j} \sum_{k \in \theta} \delta_{\theta_k} \quad w_{\theta_k c_j} + \delta_{\omega_j} \quad w_{\omega_j c_j}\right) \tag{27}$$

The resulting Eq. 27 should be equal to $_{c_j}$ as shown in Eq. 28 (derived from Eq. 12) to make LSTM-g equivalent to LSTM training in this case.

$$\delta_{c_j} = f'_{c_j}\left(s_{c_j}\right) \quad y_{\omega_j} \sum_{k \in \theta} \delta_{\theta_k} \quad w_{\theta_k c_j} \tag{28}$$

Upon inspection, we see that that LSTM-g includes a bit of extra back-propagated error ($\delta_{\omega_j} w_{\omega_j c_j}$) originating from the output gate. Besides giving a more accurate weight update for connections into memory cell j, this change in error will be captured in δ_j and passed upstream to the forget gates and input gates. As demonstrated in Section 5, this extra information helps LSTM-g perform a bit better than the original algorithm on an LSTM architecture with peephole connections.

5. Experiments

To examine the effectiveness of LSTM-g, we implemented the algorithm described in Section 3 and performed a number of experimental comparisons using various architectures, with the original LSTM algorithm and architecture as a control comparison.

5.1. Distracted Sequence Recall on the standard architecture

In the first set of experiments, we trained different neural networks on a task we call the Distracted Sequence Recall task. This task is our variation of the “temporal order” task, which is arguably the most challenging task demonstrated by Hochreiter & Schmidhuber (1997). The Distracted Sequence Recall task involves 10 symbols, each represented locally

by a single active unit in an input layer of 10 units: 4 *target* symbols, which must be recognized and remembered by the network, 4 *distractor* symbols, which never need to be remembered, and 2 *prompt* symbols which direct the network to give an answer. A single trial consists of a presentation of a temporal sequence of 24 input symbols. The first 22 consist of 2 randomly chosen target symbols and 20 randomly chosen distractor symbols, all in random order; the remaining two symbols are the two prompts, which direct the network to produce the first and second target in the sequence, in order, regardless of when they occurred. Note that the targets may appear at any point in the sequence, so the network cannot rely on their temporal position as a cue; rather, the network must recognize the symbols as targets and preferentially save them, along with temporal order information, in order to produce the correct output sequence. The network is trained to produce no output for all symbols except the prompts, and for each prompt symbol the network must produce the output symbol which corresponds to the appropriate target from the sequence.

The major difference between the “temporal order” task and our Distracted Sequence Recall task is as follows. In the former, the network is required to activate one of 16 output units, each of which represents a possible ordered sequence of both target symbols. In contrast, the latter task requires the network to activate one of only 4 output units, each representing a single target symbol; the network must activate the correct output unit for each of the targets, in the same order they were observed. Requiring the network to produce outputs in sequence adds a layer of difficulty; however, extra generalization power may be imparted by the fact that the network is now using the same output weights to indicate the presence of a target, regardless of its position in the sequence. Because the “temporal order” task was found to be unsolvable by known architectures other than the LSTM architecture when trained by gradient descent-based methods (Hochreiter & Schmidhuber, 1997), we do not include methods other than LSTM and LSTM-g in the comparisons. Further, because we wish to restrict our comparisons to temporally and spatially local training algorithms, we do not include comparisons to second-order networks trained with BPTT, RTRL, or other such methods.

Our first experiment was designed to examine the impact of the extra error information utilized by LSTM-g. We trained two networks on the Distracted Sequence Recall task. The first network serves as our control and is a typical LSTM network with forget gates, peephole connections, and direct input-to-output connections (see Fig. 2), and trained by the LSTM algorithm. The second network has the same architecture as the first, but is trained by the LSTM-g algorithm, allowing it to take advantage of back-propagated peephole connection error terms.

All runs of each network used the same basic approach and parameters: one unit in the input layer for each of the 10 input symbols, 8 units in the memory cell layer and associated gate layers, 4 units in the output layer for the target symbols, and a learning rate $\alpha = 0.1$. Both networks are augmented with peephole connections and direct input-to-output connections. Thus, both algorithms are training networks with 416 weights. Networks were allowed to train on random instances of the Distracted Sequence Recall task until they achieved the performance criterion of 95% accuracy on a test set of 1000 randomly selected sequences which the network had never encountered during training. To get credit for processing a sequence correctly, the network was required to keep all output units below an activation level of 0.5 during all of the non-prompt symbols in the sequence and activate only the correct target symbol—meaning that all units must have activations on the same side of 0.5 as the target—for both prompts. This correctness criterion was used both for networks trained by the LSTM algorithm and for those trained by the LSTM-g algorithm to ensure that the two types were compared on an equal footing. Each network type was run 50 times using randomized initial weights in the range $[-0.1, 0.1)$.

Fig. 3 compares the number of presentations required to train the two networks. All runs of both networks reached the performance criterion as expected, but there were differences in how quickly they achieved this. In particular, LSTM-g is able to train the LSTM network architecture significantly faster than the original algorithm (as evaluated by a Welch two-sample t-test after removing outliers greater than 2 standard deviations from the sample mean, with $t \approx 5.1$, $df \approx 80.9$, $p < 10^{-5}$). This demonstrates that LSTM-g can provide a clear advantage over LSTM in terms of the amount of training required, even on an LSTM-compatible architecture.

5.2. Distracted Sequence Recall on a customized architecture

Our second experiment was designed to investigate the relative performance of the standard LSTM architecture compared to other network architectures which would require modifications to the LSTM training paradigm. We trained three additional networks on the same Distracted Sequence Recall task. The first serves as the control and utilizes the LSTM algorithm to train a standard LSTM architecture that is the same as in the previous experiment except for the addition of recurrent connections from all (output-gated) memory cells to all the gate units. We call this architecture the *gated recurrence architecture* (Fig. 4a). The second network also uses the gated recurrence architecture, but is trained by LSTM-g. The third network is a new *ungated recurrence architecture* (Fig. 4b), which starts with the standard LSTM architecture and adds direct, ungated connections from each memory cell to all gate units. These connections come from the ungated memory cell output like peephole connections would, but unlike peephole connections these are projected to gate units both inside and outside of the local memory block. The intuition behind this architecture comes from the idea that a memory cell should be able to communicate its contents not only to its controlling gates but also to the other memory blocks in the hidden layer, while still hiding these contents from downstream units. Such communication would intuitively be a major boon for sequential storage and retrieval tasks because it allows a memory block to choose what to store based on what is already stored in other blocks, even if the contents of those blocks are not yet ready to be considered in calculating the network output. These ungated cell-to-gate connections are a direct generalization of peephole connections, but the new architecture that results can only be trained by the LSTM algorithm if it were to be modified to suit the architecture. As such, we present only the results of training the ungated recurrence architecture with LSTM-g, which requires no special treatment of these ungated cell-to-gate connections

Each of the three networks uses the same approach and parameters as in the previous experiment. This means the both types of network using the gated recurrence architecture have 608 trainable connections, and the ungated recurrence architecture has only 584 because the 24 peephole connections used in the gated recurrence architecture would be redundant[2].

Though these networks are more complex than those in the first experiment, they are able to learn the task more quickly. Fig. 5 shows, for each run, the number of input presentations necessary before each of three networks reached the performance criterion. Again we see a slight speed advantage for LSTM-g over LSTM when applied to the LSTM-compatible gated recurrence architecture, though this difference misses statistical significance ($t \approx 1.8$, $df \approx 87.9$, $p < 0.08$). More interesting is the improvement that LSTM-g achieves on the novel ungated recurrence architecture, which reaches significance easily compared to both

[2]The experiments reported here were also run with a variant of the gated recurrence architecture without the 24 peephole connections, leaving it with the same 584 weights as the ungated recurrence architecture; however, the lack of peephole connections caused a severe learning slowdown. In the interest of comparing LSTM-g against the strongest possible control, we report only the results from the gated recurrence architecture with peephole connections as described above.

the gated recurrence architecture trained with LSTM ($t \approx 15.3$, $df \approx 79.6$, $p < 10^{-15}$) and with LSTM-g (t 10.2, $df \approx 69.6$ $p < 10^{-14}$). LSTM-g is able to train the ungated recurrence architectur faster than either it or LSTM train the gated recurrence architecture. This difference illustrates the potential benefits of the wide range of customized architectures that LSTM-g can train.

5.3. Language Understanding on a two-stage architecture

For our final experiment we adopt a more involved and substantially different Language Understanding task, similar to those studied recently using other neural network models (Monner & Reggia, 2009). In this task, a network is given an English sentence as input and is expected to produce a set of predicates that signifies the meaning of that sentence as output. The input sentence is represented as a temporal sequence of phonemes, each of which is a vector of binary auditory features, borrowed directly from Weems & Reggia (2006). The network should produce as output a temporal sequence of predicates which bind key concepts in the sentence with their referents. For example, for the input sentence “the red pyramid is on the blue block” we would expect the network to produce the predicates red(X), pyramid(X), blue(Y), block(Y), and on(X, Y). The variables are simply identifiers used to associate the various predicates with each other; in this example, the three predicates in which variable X participates come together to signify that a single object in the world is a red pyramid which is on top of something else. In the actual output representation used by the network, each predicate type is represented as a single one in a vector of zeroes, and the variables required by each prediate are also represented in this way. In other words, a network performing this task requires a set of output neurons to represent the types of predicates, with each unit standing for a single predicate type, and two additional, independent sets of neurons which each represent a variable, since there can be at most two variables involved in any predicate. We chose to require the network to produce the required predicates in a temporal fashion to avoid imposing architectural limits on the number of predicates that a given input sentence could entail.

We chose to examine this task in part because it is hierarchically decomposable. To come up with a readily generalizable solution, common wisdom suggests that the best strategy for the network to use is to aggregate the incoming phonemes into words, words into phrases, and phrases into a unified sentence meaning. Our intuition was that architectures capable of directly supporting this type of hierarchical decomposition would be superior to those that do not. To test this notion, we developed an architecture that we call the *two-stage architecture*, shown in Fig. 6. At first it may appear complex, but it is essentially the standard LSTM architecture with peephole connections, except with a second hidden layer of memory block assemblies in series with the first. LSTM cannot e ciently train such an architecture, because the back-propagated error signals would be truncated and never reach the earlier layer of memory blocks. LSTM-g, on the other hand, trains the two-stage architecture without difficulty. As a control to our LSTM-g-trained two-stage architecture, we train a standard LSTM network with peephole connections (see Fig. 2) appropriately adjusted to match the resources of the two-stage network as closely as possible.

Both networks in this experiment use essentially the same parameters as in the previous two experiments; the only difference is in the size of the networks. The two-stage architecture has 34 input units (corresponding to the size of the phoneme feature vectors used as input), 40 memory blocks in the first stage, 40 additional memory blocks in the second stage, and 14 output units, giving a total of 13,676 trainable connections. The standard LSTM control has the same 34 input units and 14 output units, with a single hidden layer of 87 memory blocks, giving it a slight edge with 7 more total memory blocks and 13,787 trainable connections. These numbers were selected to give the two networks as near to parity in terms of computational resources as the architecture designs and problem constraints allow.

During a single trial in the Language Understanding task, the network being tested is given each individual phoneme from a sentence as input in consecutive time-steps, and after the entire input sequence has been processed, the network must output one complete predicate on each subsequent time-step, until the network produces a special “done” predicate to signify that it is finished producing relevant predicates. The networks are trained to produce the predicates for a given sentence in a specified order, but are scored in such a way that correct predicates produced in any order count as correct answers. A predicate is deemed to be correct if all units have activations on the correct side of 0.5. We also tracked the number of unrelated predicates that the networks generated; however, we found this number to closely track the inverse of the fraction of correct predicates produced, and as such, we only report the latter measure.

A small, mildly context-sensitive grammar was used to generate the sentences and corresponding predicates for this simple version of the Language Understanding task. The sentences contained combinations of 10 different words suggesting meanings involving 8 different types of predicates with up to 3 distinct objects referenced per sentence. The simplest sentences required only 3 output predicates to express their meanings, while the most complex required the networks to produce as many as 9 predicates in sequence as output. Each network was run 32 times on this task. On each run, the network in question began with random weights and was allowed to train through 1 million sentence presentations. The performance of the network was gauged periodically on a battery of 100 test sentences on which the network never received training. The duration of training was more than sufficient to ensure that all networks had reached their peak performance levels.

Fig. 7 shows a comparison of the peak performance rates of the two types of network, based on the fraction of correct predicates produced on the novel test sentences. The two-stage network trained with LSTM-g was able to achieve significantly better generalization performance than the standard LSTM network on average ($t \approx 9.4$, $df \approx 57.9$, $p < 10^{-12}$). In addition, the two-stage network was able to achieve this performance much more quickly than the control. Fig. 8 plots the number of sentence presentations required for each network to produce 80% of predicates correctly; this number was chosen because every run of every network was able to achieve this performance level. The two-stage network required approximately 4 times fewer sentence presentations to reach the 80% performance criterion, which is a significant difference ($t \approx 10.9$, $df \approx 31.4$, $p < 10^{-11}$). These results underscore the value of using LSTM-g to train customized architectures that traditional LSTM cannot.

6. Discussion

The original LSTM architecture and the associated training algorithm was an important advance in gradient training methods for recurrent neural networks that allows networks to learn to handle temporal input and output series, even across long time lags. While the LSTM architecture and extensions thereof have proven useful in a variety of contexts, the LSTM training algorithm has often been replaced in these studies by the non-local BPTT, which was necessary to train more complex network architectures. Thus, the original LSTM training algorithm was limited in scope to a small family of second-order recurrent neural architectures. This paper has introduced LSTM-g, a generalized algorithm that provides the power, speed, and spatial and temporal locality of the LSTM algorithm, but unlike said algorithm is applicable to arbitrary second-order recurrent neural networks.

In addition to the increased architectural applicability it provides, LSTM-g makes use of extra back-propagated error when applied to the canonical LSTM network architecture with peephole connections. We found that this error can be put to good use, with LSTM-g converging after less training than the LSTM training algorithm required in experiments

which utilize the standard LSTM network architecture. Further, we found that customized network architecture strained with LSTM-g can produce better performance than either LSTM or LSTM-g can produce when restricted to the standard architecture. In light of previous research that shows LSTM to outperform other recurrent network architectures when using training methods of comparable computational complexity (Hochreiter & Schmidhuber, 1997), the results contained herein suggest that LSTM-g may find fruitful application in many areas where customizable or dynamically changing network architectures are desirable.

LSTM-g is already being applied to learning problems such as natural language grounding where maintaining brain-like spatial and temporal locality is essential (Monner & Reggia, 2011). In the future, however, it will likely be worth investigating whether LSTM-g may also be of use in situations where spatial and temporal locality are not hard requirements—situations in which BPTT, RTRL, DEKF, or other methods are generally used to train second-order network architectures. While we think it unlikely that LSTM-g would outperform these algorithms in terms of final error rates, it seems plausible that the locality properties of LSTM-g would lead to a better performance-to-computation ratio, resulting in faster convergence in terms of required computation time. Regardless of the outcome of such tests, LSTM-g has the potential to broaden the applicability of local training algorithms for second-order recurrent neural networks.

Appendix A. Learning rule derivation

Here we derive the LSTM-g learning algorithm by calculating the gradient of the cross-entropy function (Eq. 9) for a general unit in an LSTM-g network. Such a unit may both project normal weighted connections to other units and multiplicatively modulate the weights of other connections. In order to know the error responsibility for unit j, we must approximate the gradient of the error function with respect to this unit's state s_j (Eq. A.1). To do this, we begin with the approximation that the error responsibility of unit j depends only upon units k which are immediately downstream from j (Eq. A.2). The remaining error gradients for k are δ_k by definition. We break up the second partial derivative into a product which includes j's activation directly so as to account for the effects of j's squashing function separately (Eq. A.3). The dependence of j's activation on its state is simply the derivative of the squashing function, which is constant across the sum and thus can be moved outside (Eq. A.4). At this point, we can separate our set of units k into two (possibly overlapping) sets (Eq. A.5)—those units to which j projects weighted connections (Eq. 19), and those units that receive connections gated by j (Eq. 20). We will thus handle error responsibilities for projection and gating separately, even in cases where j both projects and gates connections into the same unit k, thereby defining δ_{P_j} and δ_{G_j} (Eq. A.6, cf. Eq. 23).

$$\delta_j = \frac{\partial E}{\partial s_j} \tag{A.1}$$

$$\approx \sum_k \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial s_j} \tag{A.2}$$

$$= \sum_k \delta_k \frac{\partial s_k}{\partial y_j} \frac{\partial y_j}{\partial s_j} \tag{A.3}$$

$$f_j'(s_j)\sum_k \delta_k \frac{\partial s_k}{\partial y_j} \tag{A.4}$$

$$f_j'(s_j)\sum_{k\in P_j}\delta_k \quad \frac{\partial s_k}{\partial y_j} + f_j'(s_j)\sum_{k\in G_j}\delta_k \frac{\partial s_k}{\partial y_j} \tag{A.5}$$

$$=\delta_{P_j}+\delta_{G_j} \tag{A.6}$$

In calculating δ_{P_j} (Eq. A.7), we expand s_k using its definition from Eq. 15 to obtain Eq. A.8. Recall that we are only concerned with cases where j projects connections; as such, g_{kk} and $g_{kj'}$ do not depend on j and are treated as constants. Since the previous state of k does not depend on the current activation of j, the first term in the parentheses vanishes completely. Individual terms in the sum vanish as well, except for the one case when $j' = j$, leaving us with $g_{kj}\, w_{kj}$ for the entire derivative (Eq. A.9, cf. Eq. 21).

$$\delta_{P_j} = f_j'(s_j)\sum_{k\in P_j}\delta_k \frac{\partial s_k}{\partial y_j} \tag{A.7}$$

$$=f_j'(s_j)\sum_{k\in P_j}\delta_k \frac{\partial}{\partial y_j}\left(g_{kk} \quad w_{kk} \quad \widehat{s_k} + \sum_{j'\neq k} g_{kj'} \quad w_{kj'} \quad y_{j'} \right) \tag{A.8}$$

$$f_j'(s_j)\sum_{k\in P_j}\delta_k \quad g_{kj} \quad w_{kj} \tag{A.9}$$

As above, to find δ_{G_j} (Eq. A.10) we first expand s_k (Eq. A.11). In this case, we are considering only connections which j gates. Now the g_{kk} term is in play, since it is equal to y_j if j gates k's self-connection (see Eq. 14); this leads to the first term inside the parentheses in Eq. A.12, where the derivative can takes the values 1 or 0. For individual terms in the sum, we know $y_{j'} \neq y_j$ since we are not dealing with connections j projects to k. However, $g_{kj'}$ may be equal to y_j in some cases; where it is not, j does not gate the connection and the term goes to zero. Thus, in Eq. A.12 (cf. Eq. 22), the sum is reindexed to include only those units a whose connections to k are gated by j.

$$\delta_{G_j} = f_j'(s_j)\sum_{k\in G_j}\delta_k \frac{\partial s_k}{\partial y_j} \tag{A.10}$$

$$= f_j'(s_j) \sum_{k \in G_j} \delta_k \frac{\partial}{\partial y_j} \left(g_{kk} \quad w_{kk} \quad \widehat{s_k} + \sum_{j' \neq k} g_{kj'} \quad w_{kj'} \quad y_{j'} \right) \tag{A.11}$$

$$f_j'(s_j) \sum_{k \in G_j} \delta_k \left(\frac{\partial g_{kk}}{\partial y_j} w_{kk} \quad \widehat{s_k} + \sum_{a \neq k} w_{ka} \quad y_a \right) \tag{A.12}$$

We can now calculate the error responsibility δ_j of any unit j by back-propagation. We start at the level of units k that are immediately downstream from j (Eq. A.13). We can separate the k units by function again (Eq. A.14), and break up the remaining partial derivative in the first sum (Eq. A.15). Rearranging terms in the first sum (Eq. A.16), we see a grouping which reduces to δ_{P_j} (see Eq. A.7). The remaining derivative in the first term is, by definition, the familiar eligibility trace ε_{ji}; in the second term we find the definition of the extended eligibility trace ε_{ji}^k, leaving us with Eq. A.17 (cf. Eq. 24).

$$\Delta w_{ji} \quad \approx \quad \alpha \sum_k \frac{\partial E}{\partial s_k} \frac{\partial s_k}{\partial w_{ji}} \tag{A.13}$$

$$= \alpha \sum_{k \in P_j} \delta_k \frac{\partial s_k}{\partial w_{ji}} + \alpha \sum_{k \in G_j} \delta_k \frac{\partial s_k}{\partial w_{ji}} \tag{A.14}$$

$$= \alpha \sum_{k \in P_j} \delta_k \frac{\partial s_k}{\partial y_j} \frac{\partial y_j}{\partial s_j} \frac{\partial s_j}{\partial w_{ji}} + \alpha \sum_{k \in G_j} \delta_k \frac{\partial s_k}{\partial w_{ji}} \tag{A.15}$$

$$\alpha \left(f_j'(s_j) \sum_{k \in P_j} \delta_k \frac{\partial s_k}{\partial y_j} \right) \frac{\partial s_j}{\partial w_{ji}} + \alpha \sum_{k \in G_j} \delta_k \frac{\partial s_k}{\partial w_{ji}} \tag{A.16}$$

$$= \alpha \quad \delta_{P_j} \quad \varepsilon_{ji} + \alpha \sum_{k \in G_j} \delta_k \quad \varepsilon_{ji}^k \tag{A.17}$$

To calculate the eligibility trace ε_{ji} (Eq. A.18) we simply substitute s_j from Eq. 15 to obtain Eq. A.19. We assume unit j does not gate its own self-connection, so g_{jj} is a constant. The previous state $\widehat{s_j}$ depends on w_{ji} producing a partial derivative that simplifies to the previous value of the eligibility trace, $\widehat{\varepsilon_{ji}}$. The only term in the sum with a non-zero derivative is the case where $i' = i$, leaving us with Eq. A.20 (cf. Eq. 17).

$$\varepsilon_{ji} = \frac{\partial s_j}{\partial w_{ji}} \tag{A.18}$$

$$\frac{\partial}{\partial w_{ji}} \left(g_{jj} \quad w_{jj} \quad \widehat{s_j} + \sum_{i' \neq j} g_{ji'} w_{ji'} y_{i'} \right) \tag{A.19}$$

$$g_{jj} \quad w_{jj} \quad \widehat{\varepsilon_{ji}} + g_{ji} \quad y_i \tag{A.20}$$

Working on the extended eligibility trace ε^k_{ji} (Eq. A.21), we again expand s_k to form Eq. A. 22. Performing the partial derivative on the first term, note that g_{kk} may be equal to y_j, which depends on w_{ji}; also, $\widehat{s_k}$ depends on w_{ji} via its effect on s_j, so we require the product rule to derive the first term. For terms in the sum, unit j can gate the connection from j' to k, but $k \neq j$ so $w_{kj'}$ can be treated as constant, as can $y_{j'}$ since we are not concerned here with connections that j projects forward. In Eq. A.23, we are left with the result of the product rule and the remaining terms of the sum, where the a index runs over only those connections into k that j does in fact gate. Moving to Eq. A.24, we note that the first partial derivative is simply the previous value of the extended eligibility trace. We pull out partial derivatives common to the latter two terms, finally replacing them in Eq. A.25 (cf. Eq. 18) with the names of their stored variable forms. The partial derivatives in the sum are all 1 by the definition of the a index. The remaining partial derivative inside the parentheses reduces to 1 when j gates k's self-connection and 0 otherwise.

$$\varepsilon^k_{ji} = \frac{\partial s_k}{\partial w_{ji}} \tag{A.21}$$

$$= \frac{\partial}{\partial w_{ji}} \left(g_{kk} \quad w_{kk} \quad \widehat{s_k} + \sum_{j' \neq k} g_{kj'} w_{kj'} y_{j'} \right) \tag{A.22}$$

$$g_{kk} w_{kk} \frac{\partial \widehat{s_k}}{\partial w_{ji}} + \frac{\partial g_{kk}}{\partial w_{ji}} w_{kk} \quad \widehat{s_k} + \sum_{a \neq k} \frac{\partial g_{ka}}{\partial w_{ji}} w_{ka} y_a \tag{A.23}$$

$$= g_{kk} \quad w_{kk} \quad \widehat{\varepsilon^k_{ji}} + \frac{\partial y_j}{\partial s_j} \frac{\partial s_j}{\partial w_{ji}} \left(\frac{\partial g_{kk}}{\partial y_j} w_{kk} \quad \widehat{s_k} + \sum_{a \neq k} \frac{\partial g_{ka}}{\partial y_j} w_{ka} y_a \right) \tag{A.24}$$

$$= g_{kk} \quad w_{kk} \quad \widehat{\varepsilon_{ji}^{k}} + f_j'\left(s_j\right)\varepsilon_{ji}\left(\frac{\partial g_{kk}}{\partial y_j} w_{kk} \quad \widehat{s_k} + \sum_{a \neq k} w_{ka} \quad y_a\right) \tag{A.25}$$

Appendix B. LSTM as a special case of LSTM-g

Here we show that LSTM is a special case of LSTM-g in the sense that, when LSTM-g is applied to a particular class of network architectures, the two algorithms produce identical weight changes. To prove that LSTM-g provides the same weight updates as the original algorithm on the LSTM architecture, we first need to precisely articulate which architecture we are considering. For notational simplicity, we will assume that the desired LSTM architecture has only one memory cell per block; this explanation can be trivially extended to the case where we have multiple memory cells per block. The input layer projects weighted connections forward to four distinct layers of units (see Fig. 2), which are activated in the following order during a time-step, just as in LSTM: the input gate layer ι, the forget gate layer φ, the memory cell layer c, and the output gate layer ω. Each of these layers has the same number of units; a group of parallel units are associated via the pattern of network connectivity and collectively function like an LSTM memory block. Inputs to each cell in the memory cell layer are gated by the associated input gate unit. The memory cell layer is the only layer in which each unit has a direct self-connection; these each have a fixed weight of 1 and are gated by the appropriate forget gate. A final output layer receives weighted connections from the memory cell layer which are gated by the output gates.

With this simple LSTM network, LSTM-g produces the same weight changes as LSTM. If we add peephole connections to the network, the error responsibilities would differ, as LSTM-g is able to use error back-propagated from the output gates across these connections, whereas LSTM does not; this is discussed further in Section 4.

Appendix B.1. State and activation equivalence

We begin by demonstrating the equivalence fo the activation dynamics of LSTM and LSTM-g, when the latter is applied to the standard LSTM architecture.

Appendix B.1.1. Gate units

First we show that the activation for each gate unit (i.e., for a general unit λ_j where $\lambda \in \{\iota, \varphi, \omega\}$) is the same here as it is in LSTM. Starting from the LSTM-g state definition (Eq. B.1, cf. Eq. 15), we note that the activations of the gate units are stateless because they have no self connections, meaning $w_{\lambda_j \lambda_j} = 0$ and we can drop the first term (Eq. B.2). On this architecture, the only connections into any of the gate units come from the input layer and are ungated, so all the $g_{\lambda_j i}$ terms are 1 (Eq. B.3). We are left with the definition of the net input to a normal LSTM unit (Eq. B.4, cf. Eq. 1).

$$s_{\lambda_j} = g_{\lambda_j \lambda_j} \quad w_{\lambda_j \lambda_j} \widehat{s_{\lambda_j}} + \sum_{i \neq j} g_{\lambda_j i} \quad w_{\lambda_j i} \quad y_i \tag{B.1}$$

$$= \sum_{i \neq j} g_{\lambda_j i} \quad w_{\lambda_j i} \quad y_i \tag{B.2}$$

$$= \sum_{i \neq j} w_{\lambda_j i} \quad y_i \tag{B.3}$$

$$= x_{\lambda_j} \text{for} \quad \lambda \in \{\iota, \varphi, \omega\} \tag{B.4}$$

Appendix B.1.2. Output units

The situation is similar for the output units λ_j. Starting from the same LSTM-g state equation (Eq. B.5, cf. Eq. 15), we drop the first term due to lack of output unit self-connections (Eq. B.6). Since the connections into the output units come from the memory cells, each connection is gated by its corresponding output gate (Eq. B.7). We regroup the terms (Eq. B. 8) and note that, by Eq. 4, the output gate activation times the LSTM-g memory cell activation gives us the LSTM memory cell activation (Eq. B.9). The result is a equal to the net input of an LSTM output unit (Eq. B.10, cf. Eq. 1).

$$s_{\theta_j} = g_{\theta_j \theta_j} \quad w_{\theta_j \theta_j} \quad \widehat{s_{\theta_j}} + \sum_{i \in c} g_{\theta_j i} \quad w_{\theta_j i} \quad y_i \tag{B.5}$$

$$= \sum_{i \in c} g_{\theta_j i} \quad w_{\theta_j i} \quad y_i \tag{B.6}$$

$$= \sum_{i \in c} y_{\omega_i} \quad w_{\theta_j i} \quad y_i \tag{B.7}$$

$$= \sum_{i \in c} w_{\theta_j i} \left(y_{\omega_i} \quad y_i \right) \tag{B.8}$$

$$= \sum_{i \in c} w_{\theta_j i} \quad y_{c_i} \tag{B.9}$$

$$= x_{\theta_j} \tag{B.10}$$

We have now proved the equivalence of x_{λ_j} in LSTM and s_{λ_j} in LSTM-g for non-memory cells. It follows directly for these units that the activation y_{λ_j} in LSTM is equivalent to the quantity of the same name in LSTM-g (Eq. B.11, cf. Eq. 16), assuming equivalent squashing functions f_{λ_j} (Eq. B.12, cf. Eq. 2).

$$y_{\lambda_j} = f_{\lambda_j} \left(s_{\lambda_j} \right) \tag{B.11}$$

$$= f_{\lambda_j} \left(x_{\lambda_j} \right) \quad \text{for} \quad \lambda \in \{\iota, \varphi, \omega, \theta\} \tag{B.12}$$

Appendix B.1.3. Memory cells

Next we demonstrate the equivalence of memory cell states in LSTM and LSTM-g. Starting again from the unit state equation for LSTM-g (Eq. B.13, cf. Eq. 15), we note first that the self-connection weight is fixed at 1 and the self-connection gate is the associated forget gate (Eq. B.14). Next we note that all of the connections coming into the memory cell in question are gated by the memory cell's associated input gate, so $g_{cji} = y_{\iota j} \forall i$ and we can bring the term outside the sum (Eq. B.15). Finally, we note that the remaining sum is equal to the net (ungated) input to the memory cell, leaving us with the equation for LSTM memory cell states (Eq. B.16, cf. Eq. 3).

$$s_{c_j} = g_{c_j c_j} \quad w_{c_j c_j} \quad \widehat{s_{c_j}} + \sum_{i \neq c_j} g_{c_j i} \quad w_{c_j i} \quad y_i \tag{B.13}$$

$$= y_{\varphi_j} \quad \widehat{s_{c_j}} + \sum_{i \neq c_j} g_{c_j i} \quad w_{c_j i} \quad y_i \tag{B.14}$$

$$= y_{\varphi_j} \quad \widehat{s_{c_j}} + y_{\iota_j} \sum_{i \neq c_j} w_{c_j i} \quad y_i \tag{B.15}$$

$$= y_{\varphi_j} \quad \widehat{s_{c_j}} + y_{\iota_j} \quad x_{c_j} \tag{B.16}$$

The activation variable y_{c_j} does not line up directly in LSTM-g and LSTM, since LSTM requires the output gate to modulate the activation of the memory cell directly, while LSTM-g defers the modulation until the activation is passed on through a gated connection. The distinction has already been noted and appropriately dealt with in the discussion of Eq. B.8, and is not problematic for the proof at hand.

Appendix B.2. Weight change equivalence

We have shown the equivalence of activation dynamics when LSTM-g is used on the LSTM architecture. We now must show the equivalence of the weight changes performed by each algorithm.

Appendix B.2.1. Output units

Since the output units in LSTM-g get the same error responsibility from the environment as in LSTM, we need only consider whether each connection in question has the same eligibility trace in the two schemes; proving this will trivially show weight change equivalence. We need only consider the general LSTM-g eligibility trace equation, as the output units perform no gating (Eq. B.17, cf. Eq. 17). The first term drops out since the output units have no self-connections (Eq. B.18). Since the connection in question is from the memory cell layer, the gating term becomes the output gate activation (Eq. B.19). The two remaining factors are equal to LSTM's definition of the memory cell activation (Eq. B. 20), which is consistent with LSTM using only the sending unit's activation as an output unit eligibility trace (Eq. B.21, cf. Eq. 5).

$$\varepsilon_{ji} = g_{jj} \quad w_{jj} \quad \widehat{\varepsilon_{ji}} + g_{ji} \quad y_i \tag{B.17}$$

$$= g_{ji} \quad y_i \tag{B.18}$$

$$= y_{\omega_i} \quad y_i \tag{B.19}$$

$$= y_{c_i} \tag{B.20}$$

$$= \varepsilon_{\theta_{ji}} \tag{B.21}$$

Appendix B.2.2. Output gates

To prove equivalent changes to weights into the output gate units, we begin with the generalized eligibility trace equation from LSTM-g (Eq. B.22, cf. Eq. 17). Output gate units have no self-connections, so the first term drops out (Eq. B.23). The incoming connections to the output gate are not gated, so the gating term is 1 (Eq. B.24). The result is the most recent activity of the sending unit on this connection, which is the same eligibility trace as in LSTM (Eq. B.25, cf. Eq. 5).

$$\varepsilon_{ji} = g_{jj} \quad w_{jj} \quad \widehat{\varepsilon_{ji}} + g_{ji} \quad y_i \tag{B.22}$$

$$= g_{ji} \quad y_i \tag{B.23}$$

$$= y_i \tag{B.24}$$

$$= \varepsilon_{\omega_{ji}} \tag{B.25}$$

Next, we can simplify the extended eligibility traces for output gates, starting from LSTM-g (Eq. B.26, cf. Eq. 18). Output gates only modulate connections to the output layer, so we know $k \in \theta$, and none of these units have self-connections, so we drop the first term outside the parentheses as well as the first term inside the parentheses (Eq. B.27). Each output gate will modulate only a single connection into each output unit—the connection from its associated memory cell to the output unit in question—so the sum reduces to a single term (Eq. B.28).

$$\varepsilon^k_{ji} = g_{kk} \quad w_{kk} \quad \widehat{\varepsilon^k_{ji}} + f'_j\left(s_j\right) \quad \varepsilon_{ji}\left(\frac{\partial g_{kk}}{\partial y_j} w_{kk} \quad \widehat{s_k} + \sum_{a \neq k} w_{ka} \quad y_a\right) \tag{B.26}$$

$$= f'_j\left(s_j\right) \varepsilon_{ji} \sum_{a \neq k} w_{ka} \quad y_a \tag{B.27}$$

$$= f_j'(s_j) \quad \varepsilon_{ji} \quad w_{\theta_k c_j} \quad y_{c_j} \tag{B.28}$$

Finally, starting from the LSTM-g weight change equation (Eq. B.29, cf. Eq. 24) we can simplify to find the LSTM weight change. First we note that for output gates, which project no weighted connections, the set P_j is empty, making δ_{P_j} zero and eliminating the first term (Eq. B.30). Noting that, for output gates, the set G_j is precisely the set of output units θ, we replace the eligibility trace term with the simplified version from Eq. B.28, moving common terms in the sum to the outside (Eq. B.31). Rearranging the equation, we find a term (in parentheses, Eq. B.32) that is equal to LSTM's formulation of the error responsibility for an output gate (Eq. 11). Making that replacement and the replacement of the eligibility trace shown above (Eq. B.33), we now have the exact weight change prescribed by LSTM (Eq. B. 34, cf. Eq. 13).

$$\Delta w_{ji} = \alpha \quad \delta_{P_j} \quad \varepsilon_{ji} + \alpha \sum_{k \in G_j} \delta_k \quad \varepsilon_{ji}^k \tag{B.29}$$

$$= \alpha \sum_{k \in G_j} \delta_k \quad \varepsilon_{ji}^k \tag{B.30}$$

$$= \alpha \quad f_{\omega_j}'(s_{\omega_j}) \quad \varepsilon_{ji} \quad y_{c_j} \sum_{\theta_k} \delta_{\theta_k} \quad w_{\theta_k c_j} \tag{B.31}$$

$$= \alpha \left(f_{\omega_j}'(s_{\omega_j}) \quad y_{c_j} \sum_{\theta_k} \delta_{\theta_k} \quad w_{\theta_k c_j} \right) \varepsilon_{ji} \tag{B.32}$$

$$= \alpha \quad \delta_{\omega_j} \quad \varepsilon_{\omega_{ji}} \tag{B.33}$$

$$= \Delta w_{\omega_{ji}} \tag{B.34}$$

Appendix B.2.3. Memory cells

As with the output units, with the memory cells we need only consider the basic eligibility trace (Eq. B.35, cf. Eq. 17), since the memory cells perform no gating functions. We note that the self-connection weight is 1, and the self-connection gate is the associated forget gate (Eq. B.36). We also note that the input connection must be gated by the appropriate input gate (Eq. B.37), leaving us with precisely the form of the eligibility trace from LSTM (Eq. B.38, cf. Eq. 6).

$$\varepsilon_{ji} = g_{jj} \quad w_{jj} \quad \widehat{\varepsilon_{ji}} + g_{ji} \quad y_i \tag{B.35}$$

$$= y_{\varphi_j} \quad \widehat{\varepsilon_{ji}} + g_{ji} \quad y_i \tag{B.36}$$

$$= y_{\varphi_j} \quad \widehat{\varepsilon_{ji}} + y_{\iota_j} \quad y_i \tag{B.37}$$

$$= \varepsilon_{c_{ji}} \tag{B.38}$$

Again due to the lack of gating, we need only consider $\delta_j = \delta_{P_j}$ (Eq. B.39, cf. Eq. 21). Recognizing that all connections forward are gated by the associated output gate, we can replace all the g_{kj} terms in the sum with y_{ω_j}, making this a factor outside the sum (Eq. B.40). Now we simply rename derivative and weight terms based on the architecture (Eq. B.41), and we have recovered δ_{c_j} from LSTM (Eq. B.42, cf. Eq. 12).

$$\delta_{P_j} = f_j'\left(s_j\right) \sum_{k \in P_j} \delta_k \quad g_{kj} \quad w_{kj} \tag{B.39}$$

$$= f_j'\left(s_j\right) \quad y_{\omega_j} \sum_{k \in P_j} \delta_k \quad w_{kj} \tag{B.40}$$

$$= f_{cj}'\left(s_{c_j}\right) \quad y_{\omega_j} \sum_{k \in \theta} \delta_{\theta_k} \quad w_{\theta_k c_j} \tag{B.41}$$

$$= \delta_{c_j} \tag{B.42}$$

Starting with the weight change equation from LSTM-g (Eq. B.43, cf. Eq. 24), we remove the second term since G_j is empty (Eq. B.44) and substitute the equal quantities derived above (Eq. B.45) to reproduce LSTM's weight change for memory cells (Eq. B.46, cf. Eq. 13).

$$\Delta w_{ji} = \alpha \quad \delta_{P_j} \quad \varepsilon_{ji} + \alpha \sum_{k \in G_j} \delta_k \quad \varepsilon_{ji}^k \tag{B.43}$$

$$= \alpha \quad \delta_{P_j} \quad \varepsilon_{ji} \tag{B.44}$$

$$= \alpha \quad \delta_{c_j} \quad \varepsilon_{c_{ji}} \tag{B.45}$$

$$= \Delta w_{c_{ji}} \tag{B.46}$$

Appendix B.2.4. Forget gates

Similarly for forget gates, we first simplify the general LSTM-g eligibility trace (Eq. B.47, cf. Eq. 17), noting that the forget gates themselves do not self-connect (Eq. B.48), and the inputs are not gated (Eq. B.49).

$$\varepsilon_{ji} = g_{jj} \quad w_{jj} \quad \widehat{\varepsilon_{ji}} + g_{ji} \quad y_i \tag{B.47}$$

$$= g_{ji} \quad y_i \tag{B.48}$$

$$= y_i \tag{B.49}$$

Next we expand the extended eligibility trace (Eq. B.50, cf. Eq. 18), noting first that the only k we need to worry about is the single memory cell whose self-connection this unit gates. This leads us to rewrite the first gate term as the forget gate activation. In the parentheses, the derivative term and w_{kk} both go to 1, and since there are no non-self-connections that this unit gates, the sum is empty (Eq. B.51). Reorganizing and renaming some terms—as well as replacing the eligibility trace as calculated in Eq. B.49—we end up with the form in Eq. B.52. If we make the inductive assumption that the eligibility trace from the previous time-step, $\widehat{\varepsilon^{c_j}_{ji}}$, is equal to the previous LSTM eligibility trace $\widehat{\varepsilon_{\varphi_{ji}}}$ (Eq. B.53), we see that we in fact have the same eligibility traces for the current step as well (Eq. B.54, cf. Eq. 7).

$$\varepsilon^k_{ji} = g_{kk} \quad w_{kk} \quad \widehat{\varepsilon^k_{ji}} + f_j'(s_j) \quad \varepsilon_{ji} \left(\frac{\partial g_{kk}}{\partial y_j} w_{kk} \quad \widehat{s_k} + \sum_{a \neq k} w_{ka} \quad y_a \right) \tag{B.50}$$

$$\varepsilon^{c_j}_{ji} = y_{\varphi_j} \quad \widehat{\varepsilon^{c_j}_{ji}} + f_j'(s_j) \quad \varepsilon_{ji} \quad \widehat{s_{c_j}} \tag{B.51}$$

$$= y_{\varphi_j} \quad \widehat{\varepsilon^{c_j}_{ji}} + \widehat{s_{c_j}} \quad f_{\varphi_j}'(x_{\varphi_j}) \quad y_i \tag{B.52}$$

$$= y_{\varphi_j} \quad \widehat{\varepsilon_{\varphi_{ji}}} + \widehat{s_{c_j}} \quad f_{\varphi_j}'(x_{\varphi_j}) \quad y_i \tag{B.53}$$

$$= \varepsilon_{\varphi_{ji}} \tag{B.54}$$

As it was for the output gates, δ_{P_j} is zero in Eq. B.55 (cf. Eq. 24) since P_j is empty for the forget gates, leaving us with Eq. B.56. For forget gates, G_j has a single element—the memory cell whose self-connection this unit gates (Eq. B.57). As show above, the eligibility traces are equal (Eq. B.58), and $\delta_{c_j} = \delta_{\varphi_j}$ as shown in Eq. 12, leaving us with the LSTM-prescribed weight change for forget gate connections (Eq. B.59, cf. Eq. 13).

$$\Delta w_{ji} = \alpha \quad \delta_{p_j} \quad \varepsilon_{ji} + \alpha \sum_{k \in G_j} \delta_k \quad \varepsilon_{ji}^k \tag{B.55}$$

$$= \alpha \sum_{k \in G_j} \delta_k \quad \varepsilon_{ji}^k \tag{B.56}$$

$$= \alpha \quad \delta_{c_j} \quad \varepsilon_{ji}^{c_j} \tag{B.57}$$

$$= \alpha \quad \delta_{\varphi_j} \quad \varepsilon_{\varphi_{ji}} \tag{B.58}$$

$$= \Delta w_{\varphi_{ji}} \tag{B.59}$$

Appendix B.2.5. Input gates

As for forget gates, for input gates we start by simplifying the LSTM-g eligibility trace (Eq. B.60, cf. Eq. 17), dropping the first term since input gates do not self-connect (Eq. B.61) and letting g_{ji} go to 1 since the inputs are not gated (Eq. B.62).

$$\varepsilon_{ji} = g_{jj} \quad w_{jj} \quad \widehat{\varepsilon_{ji}} + g_{ji} \quad y_i \tag{B.60}$$

$$= g_{ji} \quad y_i \tag{B.61}$$

$$= y_i \tag{B.62}$$

The extended eligibility trace again begins as Eq. B.63 (cf. Eq. 18), where the only k in play is the single memory cell whose input connections are modulated by this input gate. Thus, the first gain term can be replaced by the forget gate activation, and the derivative term inside the parentheses goes to zero (Eq. B.64). Next we rename the f_j' term appropriately, replace the eligibility trace with its simplification from Eq. B.62, and recognize that the sum we have left is in fact the ungated net input to the memory cell in question (Eq. B.65). If we make the inductive assumption that the previous step's eligibility traces are equal (Eq. B. 66), we see immediately that those of the current step are equal as well (Eq. B.67, cf. Eq. 8).

$$\varepsilon_{ji}^k = g_{kk} \quad w_{kk} \quad \widehat{\varepsilon_{ji}^k} + f_j'\left(s_j\right) \quad \varepsilon_{ji} \quad \left(\frac{\partial g_{kk}}{\partial y_j} w_{kk} \widehat{s_k} + \sum_{a \neq k} w_{ka} \quad y_a\right) \tag{B.63}$$

$$\varepsilon_{ji}^{c_j} = y_{\varphi_j} \quad \widehat{\varepsilon_{ji}^{c_j}} + f_j'\left(s_j\right) \quad \varepsilon_{ji} \sum_{a \neq c_j} w_{c_j a} y_a \tag{B.64}$$

$$= y_{\varphi_j} \quad \widehat{\varepsilon_{ji}^{c_j}} + f_{\iota_j}'\left(x_{\iota_j}\right) \quad y_i \quad x_{c_j} \tag{B.65}$$

$$= y_{\varphi_j} \quad \widehat{\varepsilon_{\iota_{ji}}} + x_{c_j} \quad f_{\iota_j}'\left(x_{\iota_j}\right) \quad y_i \tag{B.66}$$

$$= \varepsilon_{\iota_{ji}} \tag{B.67}$$

Starting again with the unified LSTM-g weight update equation (Eq. B.68, cf. Eq. 24), we drop the first term since δ_{P_j} is zero (Eq. B.69). Next we recognize that the only cell in the set G_j is the associated memory cell c_j of this input gate, simplifying the sum to a single term (Eq. B.70). Using Eq. 12, we note that $\delta_{c_j} = \delta_{\iota_j}$ for this network, so we make that replacement as well as replacing $\varepsilon_{ji}^{c_j}$ according to Eq. B.67 to obtain Eq. B.71. We end up with precisely the weight changes specified by LSTM for input gate connections (Eq. B.72, cf. Eq. 13).

$$\Delta w_{ji} = \alpha \quad \delta_{P_j} \quad \varepsilon_{ji} + \alpha \sum_{k \in G_j} \delta_k \quad \varepsilon_{ji}^k \tag{B.68}$$

$$= \alpha \sum_{k \in G_j} \delta_k \quad \varepsilon_{ji}^k \tag{B.69}$$

$$= \alpha \quad \delta_{c_j} \quad \varepsilon_{ji}^{c_j} \tag{B.70}$$

$$= \alpha \quad \delta_{\iota_j} \quad \varepsilon_{\iota_{ji}} \tag{B.71}$$

$$= \Delta w_{\iota_{ji}} \tag{B.72}$$

This section has shown that every unit in a canonical LSTM network architecture calculates identical activations and weight updates under LSTM and LSTM-g, thus concluding the proof that LSTM is a special case of LSTM-g.

References

Bayer, J.; Wierstra, D.; Togelius, J.; Schmidhuber, J. Evolving memory cell structures for sequence learning; Proceedings of the International Conference on Artificial Neural Networks; 2009;

Elman JL. Finding Structure in Time. Cognitive Science. 1990; 14:179–211.

Gers FA, Cummins F. Learning to Forget: Continual Prediction with LSTM. Neural Computation. 2000; 12:2451–2471. [PubMed: 11032042]

Gers FA, Pérez-Ortiz JA, Eck D, Schmidhuber J. Kalman filters improve LSTM network performance in problems unsolvable by traditional recurrent nets. Neural Networks. 2003; 16:241–250. [PubMed: 12628609]

Gers, FA.; Schmidhuber, J. Recurrent Nets that Time and Count; Proceedings of the International Joint Conference on Neural Networks; 2000; p. 189-194.

Gers FA, Schmidhuber J. LSTM Recurrent Networks Learn Simple Context-Free and Context-Sensitive Languages. IEEE Transactions on Neural Networks. 2001; 12:1333–1340. [PubMed: 18249962]

Giles CL, Maxwell T. Learning, invariance, and generalization in high-order neural networks. Applied Optics. 1987; 26:4972–4978. [PubMed: 20523475]

Graves, A.; Eck, D.; Beringer, N.; Schmidhuber, J. Biologically Plausible Speech Recognition with LSTM Neural Nets; Proceedings of the International Workshop on Biologically Inspired Approaches to Advanced Information Technology; 2004; p. 127-136.

Graves, A.; Schmidhuber, J. Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks. In: Koller, D.; Schuurmans, D.; Bengio, Y.; Bottou, L., editors. Neural Information Processing Systems. Vol. 21. 2008.

Hinton GE. Connectionist learning procedures. Artificial Intelligence. 1989; 40:185–234.

Hochreiter S, Schmidhuber J. Long Short-Term Memory. Neural Computation. 1997; 9:1735–1780. [PubMed: 9377276]

Miller CB, Giles CL. Experimental comparison of the effect of order in recurrent neural networks. Pattern Recognition. 1993; 7:849–872.

Monner, D.; Reggia, JA. An unsupervised learning method for representing simple sentences; 2009 International Joint Conference on Neural Networks; 2009; p. 2133-2140.

Monner, D.; Reggia, JA. Systematically Grounding Language Through Vision in a Deep, Recurrent Neural Network; Proceedings of the Conference on Artificial General Intelligence; 2011;

Psaltis D, Park C, Hong J. Higher order associative memories and their optical implementations. Neural Networks. 1988; 1:149–163.

Puskorius GV, Feldkamp L. a. Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks. IEEE Transactions on Neural Networksetworks. 1994; 5:279–97.

Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. Nature. 1986; 323:533–536.

Schmidhuber J, Wierstra D, Gagliolo M, Gomez F. Training recurrent networks by Evolino. Neural Computation. 2007; 19:757–79. [PubMed: 17298232]

Shin, Y.; Ghosh, J. The pi-sigma network: An e cient higher-order neural network for pattern classification and function approximation; Proceedings of the International Joint Conference on Neural Networks; 1991; p. 13-18.

Sutton, RS.; Barto, AG. Reinforcement Learning: An Introduction. MIT Press; 1998. Temporal-difference learning; p. 167-200.

Weems SA, Reggia JA. Simulating Single Word Processing in the Classic Aphasia Syndromes Based on the Wernicke-Lichtheim-Geschwind Theory. Brain and Language. 2006; 98:291–309. [PubMed: 16828860]

Werbos PJ. Backpropagation through time: What it does and how to do it. Proceedings of the IEEE. 1990; 78:1550–1560.

Williams RJ, Zipser D. A learning algorithm for continually running fully recurrent neural networks. Neural Computation. 1989; 1:270–280.

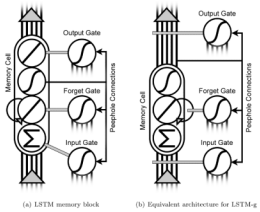


Figure 1.

Architectural comparison of LSTM's memory block to the equivalent in an LSTM-g network. Weighted connections are shown as black lines, and gating relationships are shown as thicker gray lines. The elongated enclosure represents the extent of the memory cell. In (a), connections into the LSTM memory cell are first summed (the input-squashing function is taken to be the identity); the result is gated by the input gate. The gated net input progresses to the self-recurrent linear unit, whose activity is gated by the forget gate. The state of the recurrent unit is passed through the output-squashing function, which is then modulated by the output gate. This modulated value is passed to all receiving units via weighted connections. Note that the peephole connections project from an internal stage in the memory cell to the controlling gate units; this is an exception to the rule that only the final value of the memory cell is visible to other units. In (b), the weights on the input connections to the LSTM-g memory cell are modulated directly by the input gate before being summed by the linear unit. Unmodulated output leaves the memory cell via weighted connections. Connections to downstream units can have their weights modulated directly by the output gate, but this is not required, as can be seen with the equivalent of LSTM's peephole connections proceeding normally from the output of the memory cell. This scheme is capable of producing the same results as the LSTM memory block, but allows greater architectural flexibility.

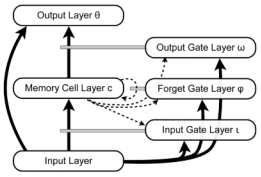


Figure 2.

The standard LSTM architecture in terms of layers. The memory block assemblies are broken up into separate layers of memory cells, input gates, forget gates, and output gates, in addition to the input and output layers. Solid arrows indicate full all-to-all connectivity between units in a layer, and dashed arrows indicate connectivity only between the units in the two layers that have the same index (i.e., the first unit of the sending layer only projects to the first unit of the receiving layer, the second unit only projects to the second, and so forth). The gray bars denote gating relationships, and are displayed as they are conceived in LSTM-g, with the modulation occurring at the connection level. Units in each gate layer modulate only those connections into or out of their corresponding memory cell. The circular dashed connection on the memory cell layer indicates the self-connectivity of the units therein. This diagram shows the optional peephole connections—the dashed arrows originating at the memory cells and ending at the gate layers—as well as the optional direct input-to-output layer connections on the left.

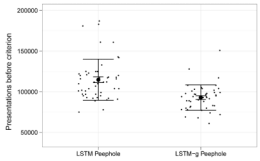
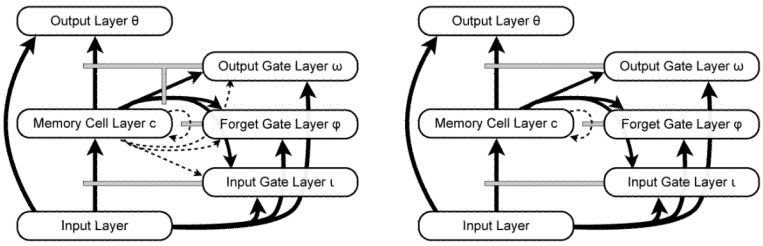


Figure 3.

Plot of the results of an experiment that pitted LSTM-g against LSTM, each training an identical standard peephole LSTM architecture to perform the Distracted Sequence Recall task. Small points are individual network runs, jittered to highlight their density. The large black point for each network type is the mean over all 50 runs, with standard error (small bars) and standard deviation (large bars). The results show clearly the beneficial impact of the way LSTM-g utilizes extra error information.



(a) Gated recurrence architecture (b) Ungated recurrence architecture

Figure 4.

The network architectures used in the second experiment (c.f. Fig. 2). In (a), the previous LSTM architecture is augmented with a full complement of recurrent connections from each memory cell to each gate, regardless of memory block associations; all these connections are gated by the appropriate output gate. In (b), we strip the (now redundant) peephole connections from the original architecture and in their place put a full complement of ungated connections from each memory cell to every gate. This second architectural variant is incompatible with the LSTM training algorithm, as it requires all connections out of the memory cell layer to be gated by the output gate. The network can still be trained by LSTM-g, however.

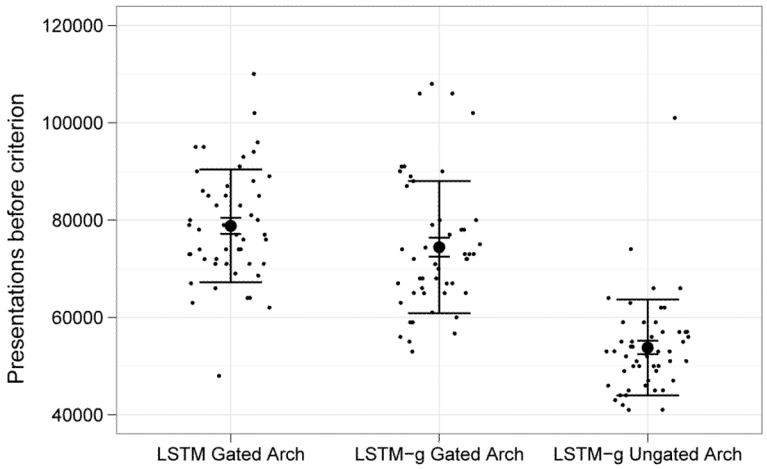


Figure 5.

Plot of the results on the Distracted Sequenced Recall task for three networks: an LSTM network augmented with peephole connections and gated recurrent connections from all memory cells to all gates; and an LSTM-g network with ungated recurrent connections from all memory cells to all gates. The graph clearly shows that the ungated recurrence network, trainable only with LSTM-g, reaches the performance criterion after less training than the comparable gated recurrence network as trained by either LSTM or LSTM-g.

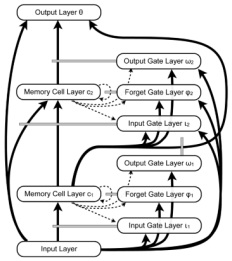


Figure 6.

The two-stage network architecture, used in the third experiment. This architecture is a variant of the standard LSTM architecture with peephole connections (Fig. 2) that has a second layer of memory block assemblies in series with the first. The traditional LSTM training algorithm cannot effectively train this architecture due to the truncation of error signals, which would never reach the earlier layer of memory blocks. Intuition suggests that the two self-recurrent layers allow this network to excel at hierarchically decomposable tasks such as the Language Understanding task.

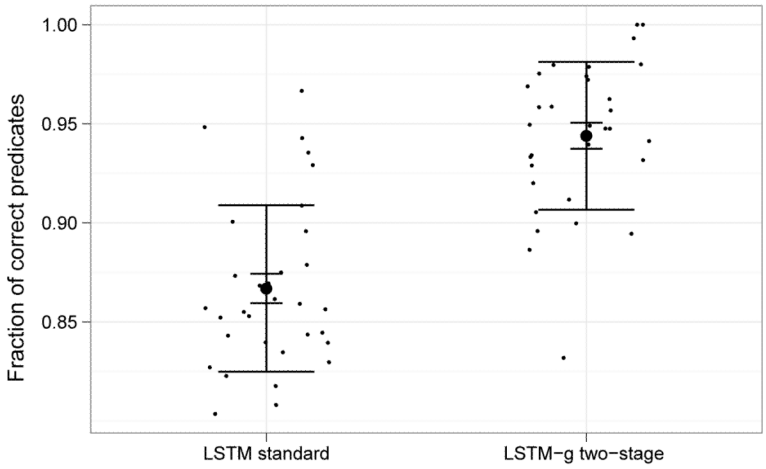


Figure 7.
Plot of the performance results on the Language Understanding task produced by a standard LSTM network and the two-stage architecture trained by LSTM-g. We see that the standard LSTM networks were able to produce approximately 87% of predicates correctly at peak performance, while the two-stage LSTM-g networks garnered 94% on average.

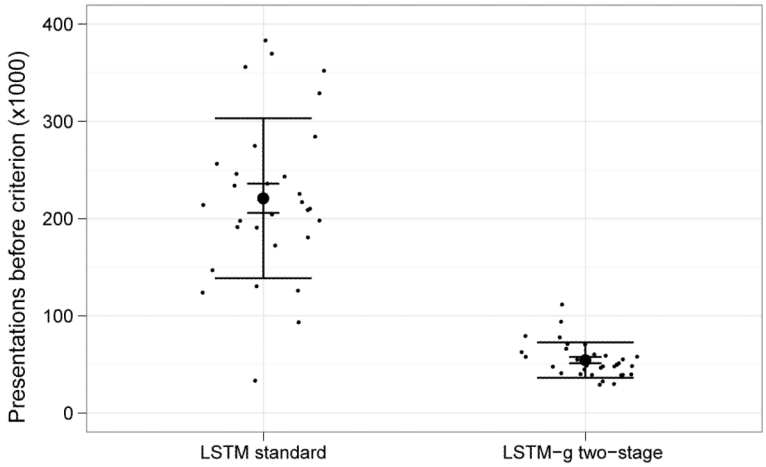


Figure 8.
Plot of the training duration required for each type of network to reach the criterion of producing 80% of the required predicates for input sentences. The standard LSTM network required an average of about 220,000 sentence presentations to reach this performance criterion, while the two-stage network trained by LSTM-g required fewer than 60,000.