Chapter 2

Taxonomy for Memory in RNNs

2.1 Memory in Brain

Memory is a crucial part of any cognitive model studying the human mind. This section briefly reviews memory types studied throughout the cognitive and neuroscience literature. Fig. 2.1 shows a taxonomy of cognitive memory (Kotseruba and Tsotsos, 2018).

2.1.1 Short-term Memory

Sensory memory Sensory memory caches impressions of sensory information after the original stimuli have ended. It can also preprocess the information before transmitting it to other cognitive processes. For example, echoic memory keeps acoustic stimulus long enough for perceptual binding and feature extraction processes. Sensory memory is known to associate with temporal lope in the brain. In the neural network literature, sensory memory can be designed as neural networks without synaptic learning (Johnson et al., 2013).

Working memory Working memory holds temporary storage of information related to the current task such as language comprehension, learning, and reasoning (Baddeley, 1992). Just like computer that uses RAM for its computations, the brain needs working memory as a mechanism to store and update information to perform cognitive tasks such as attention, reasoning and learning. Human neuroimaging studies show that when people perform tasks requiring them to hold short-term memory, such as the location of a flash of light, the prefrontal cortex becomes active (Curtis and D'Esposito, 2003). As we shall see later, recurrent neural networks must construct some form of working memory to help the networks learn the task at hand. As working memory is short-term (Goldman-Rakic, 1995), the working memory in RNNs also tends to vanish quickly and needs the support from other memory mechanisms to learn complex tasks that require long-term dependencies.

2.1.2 Long-term Memory

Motor/procedural memory The procedural memory, which is known to link to basal ganglia in the brain, contains knowledge about how to get things done in motor task domain. The knowledge may involve co-coordinating sequences of motor activity, as would be needed when dancing, playing sports or musical instruments. This procedural knowledge can be implemented by a set of if-then rules learnt for a particular domain or a neural network representing perceptual-motor associations (Salgado et al., 2012).

Semantic memory Semantic memory contains knowledge about facts, concepts, and ideas. It allows us to identify objects and relationships between them. Semantic memory is a highly structured system of information learnt gradually from the world. The brain's neocortex is responsible for semantic memory and its processing is seen as the propagation of activation amongst neurons via weighted connections that slowly change (Kumaran et al., 2016).

Episodic memory Episodic memory stores specific instances of past experience. Different from semantic memory, which does not require temporal and spatial information, episodic remembering restores past experiences indexed by event time or context (Tulving et al., 1972). Episodic memory is widely acknowledged to depend on the hippocampus, acting like an autoassociate memory that binds diverse inputs from different brain areas that represent the constituents of an event (Kumaran et al., 2016). It is conjectured that the experiences stored in hippocampus transfer to neocortex to form semantic knowledge as we sleep via consolidation process. Recently, many attempts have been made to integrate episodic memory into deep

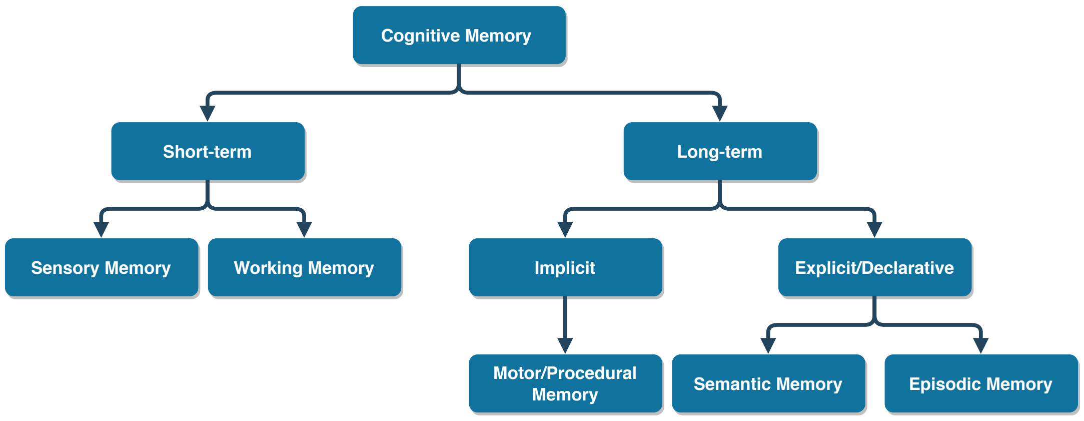

Figure 2.1: Types of memory in cognitive models

learning models and achieved promising results in reinforcement (Mnih et al., 2015; Blundell et al., 2016; Pritzel et al., 2017) and supervised learning (Graves et al., 2016; Lopez-Paz et al., 2017; Le et al., 2018b).

2.2 Neural Networks and Memory

2.2.1 Introduction to Neural Networks

Feed-forward neural networks

A feed-forward neural network arranges neurons in layers with connections going forward from one layer to another, creating a directed acyclic graph. That is, connections going backwards or between nodes within a layer are prohibited. Each neuron in the network is a computation unit, which takes inputs from outputs of other neurons, then applies a weighted sum followed by a nonlinear transform, and produces an output. The multilayer perceptron (MLP) is a commonly used feed-forward neural network for classifying data or approximating an unknown function. An example MLP is shown in Fig. 2.2, with three layers: input, output and a single "hidden" layer. In order to distinguish linearly inseparable data points, the activation function must be nonlinear. The weight of a connection, which resembles synapse of the neocortex, is simply a coefficient by which the output of a neuron is multiplied before being taken as the input to another neuron. Hence, the total input to a neuron j is

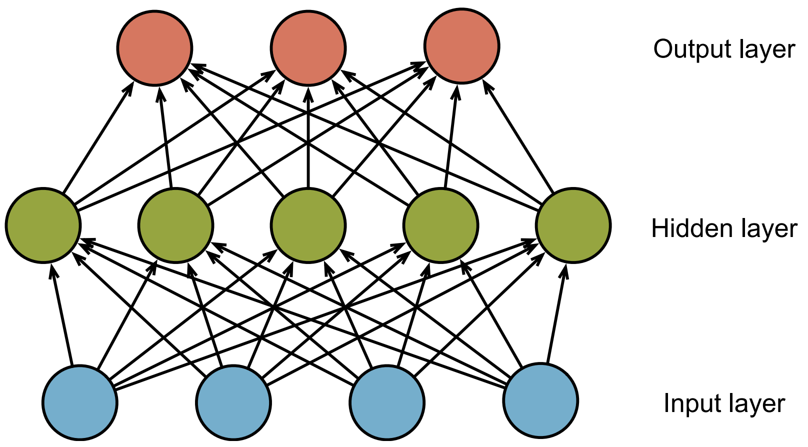

Figure 2.2: A multilayer perceptron with a single hidden-layer.

$$y_j = \sum_i w_{ij} x_i + b_j \tag{2.1}$$

where x_i is the output of a neuron i, w_{ij} is the weight of the connection from neuron i to neuron j, and b_j is a constant offset or bias. The output of neuron j, or x_j, is the result of applying an activation function to y_j. The following lists common activation functions used in modern neural networks,

$$\text{sigmoid}\left(z\right) = \frac{1}{1+e^{-z}} \tag{2.2}$$

$$\tanh\left(z\right) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \tag{2.3}$$

$$\text{relu}\left(z\right) = \max(z, 0) \tag{2.4}$$

Given a set of training data with ground truth label for each data points, the network is typically trained with gradient-based optimisation algorithms, which estimate the parameters by minimising a loss function. A popular loss function is the average negative log likelihood

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} \log P\left(\hat{y}_i = y_i | x_i\right) \tag{2.5}$$

where N is the number of training samples, x_i and y_i is the i-th data sample and its label, respectively, and $\hat{y}_i$ is the predicted label. During training, forward propagation outputs $\hat{y}_i$ and calculates the loss function. An algorithm called back-propagation, which was first introduced in (Rumelhart et al., 1988), computes the gradients of the loss function $\mathcal{L}$ with respect to (w.r.t) the parameters $\theta = \{w_{ij}, b_j\}$. Then, an optimisation algorithm such as stochastic gradient descent updates the parameters based on their gradients $\left\{\frac{\partial \mathcal{L}}{\partial w_{ij}}, \frac{\partial \mathcal{L}}{\partial b_j}\right\}$ as follows,

$$w_{ij} := w_{ij} - \lambda \frac{\partial \mathcal{L}}{\partial w_{ij}} \tag{2.6}$$

$$b_j := b_j - \lambda \frac{\partial \mathcal{L}}{\partial b_j} \tag{2.7}$$

where λ is a small learning rate.

Recurrent neural networks

A recurrent neural network (RNN) is an artificial neural network where connections between nodes form a directed graph with self-looped feedback. This allows the network to capture the hidden states calculated so far when activation functions of neurons in the hidden layer are fed back to the input layer at every time step in conjunction with other input features. The ability to maintain the state of the system makes RNN especially useful for processing sequential data such as sound, natural language or time series signals. So far, many varieties of RNN have been proposed such as Hopfield Network (Hopfield, 1982), Echo State Network (Jaeger and Haas, 2004) and Jordan Network (Jordan, 1997). Here, for the ease of analysis, we only discuss Elman's RNN model (Elman, 1990) with single hidden layer as shown in Fig. 2.3.

An Elman RNN consists of three layers, which are input ($x \in \mathbb{R}^{\mathrm{N}}$), hidden ($h \in \mathbb{R}^{\mathrm{D}}$) and output ($o \in \mathbb{R}^{\mathrm{M}}$) layer. At each timestep, the feedback connection forwards the previous hidden state h_{t-1} to the current hidden unit, together with the values from input layer x_t, to compute the current state h_t and output value o_t. The forward pass begins with a specification of the initial state h_0, then we apply the following

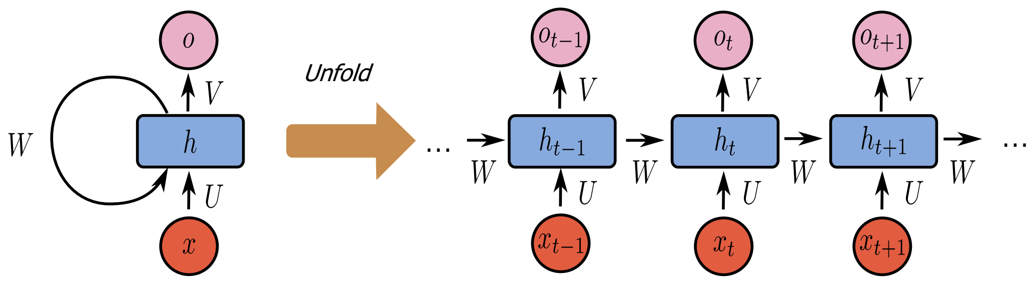

Figure 2.3: A typical Recurrent Neural Network (Left) and its unfolded representation (Right). Each neuron at timestep t takes into consideration the current input x_t and previous hidden state h_{t-1} to generate the t-th output o_t. W, U and V are learnable weight matrices of the model.

update equations

$$h_t = f\left(h_{t-1}W + x_t U + b\right) \tag{2.8}$$

$$o_t = g\left(h_t V + c\right) \tag{2.9}$$

where $b \in \mathbb{R}^{\mathrm{D}}$ and $c \in \mathbb{R}^{\mathrm{M}}$ are the bias parameters. $U \in \mathbb{R}^{\mathrm{N\times D}}$, $V \in \mathbb{R}^{\mathrm{D\times M}}$ and $W \in \mathbb{R}^{\mathrm{D\times D}}$ are weight matrices for input-to-hidden, hidden-to-output and hidden-to-hidden connections, respectively. f and g are functions that help to add non-linearity to the transformation between layers. For classification problems, g is often chosen as the softmax function and the output o_t represents the conditional distribution of t-th output given previous inputs. The final output $\hat{y}_t$ is the label whose probability score is the highest. By repeating the updates, one can map the input sequence $x = \{x_1, x_2, ..., x_T\}$ to an output sequence $\hat{y} = \{\hat{y}_1, \hat{y}_2, ..., \hat{y}_T\}$. The total loss for a given sequence x paired with a ground-truth sequence $y = \{y_1, y_2, ..., y_T\}$ would then be the sum of the losses over all the timesteps

$$\mathcal{L}\left(y|x\right) = \sum_{t=1}^{T} \mathcal{L}_t\left(y_t|x_1, x_2, ..., x_t\right) = -\sum_{t=1}^{T} \log P\left(\hat{y}_t = y_t|x_1, x_2, ..., x_t\right)$$

The loss function can be minimised by using gradient descent approach. The derivatives w.r.t the parameters can be determined by the Back-Propagation Through Time algorithm (Werbos, 1990). RNNs are widely used in sequential tasks such as language modeling (Mikolov et al., 2010), handwriting generation (Graves, 2013) and speech recognition (Graves et al., 2013). RNNs demonstrate better performance than other classical approaches using Hidden Markov Model (HMM) or Conditional Random Fields (CRFs).

2.2.2 Semantic Memory in Neural Networks

Neural networks learn structured knowledge representation from the data by adjusting connection weights amongst the units in the network under supervised training paradigms (Hinton et al., 1986; Rumelhart et al., 1988; Plunkett and Sinha, 1992). The connection weights capture the semantic structure of the domain under modeling (McClelland et al., 1995; Rogers and McClelland, 2004). The trained model generalises to novel examples rather than just naively memorising training items. However, modern deep learning models are often massively over-parameterised and thus prone to overfitting, even to noise (Zhang et al., 2016b). Further investigations indicate that although deep networks may employ brute-force memorising strategy, they should operate in a fashion that can perform inductive generalisation (Arpit et al., 2017; Krueger et al., 2017). Unfortunately, since all of these arguments are validated empirically or via simulations, no theoretical principles governing semantic knowledge extraction were given.

The lack of theoretical guarantee remained until recently when Saxe et al. (2019) confirmed the existence of semantic memory in neural network by theoretically describing the trajectory of knowledge acquisition and organisation of neural semantic representations. The paper is restricted to a simple linear neural network with one hidden layer. The network is trained to correctly output the associated properties or features of the input items (e.g., dog →bark, horse →big). Each time a training sample i is presented as $\{x_i, y_i\}$, the weights of the network W_1 and W_2 are adjusted by a small amount to gradually minimise the squared error loss $\mathcal{L} = \|y_i - \hat{y}_i\|^2$. The parameter update rule is derived via standard back propagation as follows,

$$\Delta W_1 = \lambda W_2^\top (y_i - \hat{y}_i) x_i^\top \tag{2.10}$$

$$\Delta W_2 = \lambda (y_i - \hat{y}_i)(W_1 x_i)^\top \tag{2.11}$$

where λ is the learning rate. We are interested in estimating the total weight change after epoch t, which can be approximated, when $\lambda \ll 1$, as the following,

$$\Delta W_1(t) \approx \lambda P W_2(t)^\top (\varSigma^{yx} - W_2(t) W_1(t) \varSigma^x) \tag{2.12}$$

$$\Delta W_2(t) \approx \lambda P (\varSigma^{yx} - W_2(t) W_1(t) \varSigma^x) W_1(t)^\top \tag{2.13}$$

where P is the number of training samples; $\Sigma^{x} = E\left[xx^{\top}\right]$ and $\Sigma^{yx} = E\left[yx^{\top}\right]$ are input and input-output correlation matrices, respectively. We can take the continuum limit of this difference equation to obtain the following system of differential equations

$$\tau \frac{d}{dt} W_1 = W_2^{\top} \left(\Sigma^{yx} - W_2 W_1 \Sigma^{x}\right) \tag{2.14}$$

$$\tau \frac{d}{dt} W_2 = \left(\Sigma^{yx} - W_2 W_1 \Sigma^{x}\right) W_1^{\top} \tag{2.15}$$

where $\tau = \frac{1}{P\lambda}$. To simplify the equations, we assume $\Sigma^{x} = I$ and apply reparametrisation trick to obtain

$$\tau \frac{d}{dt} \overline{W}_1 = \overline{W}_2^{\top} \left(S - \overline{W}_2 \overline{W}_1\right) \tag{2.16}$$

$$\tau \frac{d}{dt} \overline{W}_2 = \left(S - \overline{W}_2 \overline{W}_1\right) \overline{W}_1^{\top} \tag{2.17}$$

where S is the diagonal matrix in the singular value decomposition of $\Sigma^{yx} = USV^{\top}$; $\overline{W}_1$ and $\overline{W}_2$ are new variables such that $W_1 = R\overline{W}_1 V^{\top}$ and $W_2 = U\overline{W}_2 R$ with an arbitrary orthogonal matrix R. When $\overline{W}_1(0)$ and $\overline{W}_2(0)$ are initialised with small random weights, we can approximate them with diagonal matrices of equal modes. A closed form solution of the scalar dynamic corresponding to each mode of Eqs. (2.16) and (2.17) can be derived as follows,

$$a_{\alpha}(t) = \frac{s_{\alpha} e^{2 s_{\alpha} t/\tau}}{e^{2 s_{\alpha} t/\tau} - 1 + s_{\alpha}/a_{\alpha}(0)} \tag{2.18}$$

where a_{α} is a diagonal element of the time-dependent diagonal matrix $A(t)$ such that $A(t) = \overline{W}_2(t)\,\overline{W}_1(t)$. Inverting the change of variables yields

$$W_1(t) = Q\sqrt{A(t)}V^{\top} \tag{2.19}$$

$$W_2(t) = U\sqrt{A(t)}Q^{-1} \tag{2.20}$$

where Q is an arbitrary invertible matrix. If the initial weights are small, then the matrix Q will be close to a rotation matrix. Factoring out the rotation, the hidden representation of item i is

$$h_i^\alpha(t) = \sqrt{a_\alpha(t)}v_i^\alpha \tag{2.21}$$

where $v_i^\alpha = V^\top[\alpha, i]$. Hence, we obtain a temporal evolution of internal representations h of the deep network. By using multi-dimensional scaling (MDS) visualisation of the evolution of internal representations over developmental time, Saxe et al. (2019) demonstrated a progressive differentiation of hierarchy in the evolution, which matched the data's underlying hierarchical structure. When we have the explicit form of the evolution (Eq. (2.21)), this matching can be proved as an inevitable consequence of deep learning dynamics when exposed to hierarchically structured data (Saxe et al., 2019).

2.2.3 Associative Neural Networks

Associative memory is used to store associations between items. It is a general concept of memory that spans across episodic, semantic and motor memory in the brain. We can use neural networks (either feed-forward or recurrent) to implement associative memory. There are three kinds of associative networks:

- Heteroassociative networks store Q pair of vectors $\{x^1 \in \mathcal{X}, y^1 \in \mathcal{Y}\}$, ..., $\left\{x^Q \in \mathcal{X},\ y^Q \in \mathcal{Y}\right\}$ such that given some key x^k, they return value y^k.
- Autoassociative networks are a special type of the heteroassociative networks, in which $y^k = x^k$ (each item is associated with itself).
- Pattern recognition networks are also a special case where x^k is associated with a scalar k representing the item's category.

Basically, these networks are used to represent associations between two vectors. After two vectors are associated, one can be used as a cue to retrieve the other. In principle, there are three functions governing an associative memory:

- Encoding function $\otimes : \mathcal{X} \times \mathcal{Y} \to \mathcal{M}$ associates input items into some form of memory trace $\mathcal{M}$.
- Trace composition function $\oplus : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ combines memory traces to form the final representation for the whole dataset.

- Decoding function $\bullet : \mathcal{X} \times \mathcal{M} \to \mathcal{Y}$ produces a (noisy) version of the item given its associated.

Different models employ different kinds of functions (linear, non-linear, dot product, outer product, tensor product, convolution, etc.). Associative memory concept is potential to model memory in the brain (Marr and Thach, 1991). We will come across some embodiment of associative memory in the form of neural networks in the next sections.

2.3 The Constructions of Memory in RNNs

2.3.1 Attractor dynamics

Attractor dynamics denotes neuronal network dynamics which is dominated by groups of persistently active neurons. In general, such a persistent activation associates with an attractor state of the dynamics, which for simplicity, can take the form of fixed-point (Amit, 1992). This kind of network can be used to implement associative memory by allowing the network's attractors to be exactly those vectors we would like to store (Rojas, 2013). The approach supports memory for the items per se, and thus differs from semantic memory in the sense that the items are often stored quickly and what being stored cannot represent the semantic structure of the data. Rather, attractor dynamics resembles working and episodic memory. Like episodic memory, it acts as an associative memory, returning stored value when triggered with the right clues. The capacity of attractor dynamics is low, which reflects the short-term property of working memory. In the next part of the sub-section, we will study these characteristics through one embodiment of attractor dynamics.

Hopfield network

The Hopfield network, originally proposed in 1982 (Hopfield, 1982), is a recurrent neural network that implements associative memory using fix-points as attractors. The function of the associative memory is to recognise previously learnt input vectors, even in the case where some noise has been added. To achieve this function, every neuron in the network is connected to all of the others (see Fig. 2.4 (a)).

Each neuron outputs discrete values, normally 1 or -1, according to the following equation

$$x_i(t+1) = \text{sign}\left(\sum_{j=1}^{N} w_{ij} x_j(t)\right) \tag{2.22}$$

where $x_i(t)$ is the state of i-th neuron at time t and N is the number of neurons. Hopfield network has a scalar value associated with the state of all neurons x, referred to as the "energy" or Lyapunov function,

$$E(x) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij} x_i x_j \tag{2.23}$$

If we want to store Q patterns x^p, $p = 1, 2, ..., Q$, we can use the Hebbian learning rule (Hebb, 1962) to assign the values of the weights as follows,

$$w_{ij} = \sum_{p=1}^{Q} x_i^p x_j^p \tag{2.24}$$

which is equivalent to setting the weights to the elements of the correlation matrix of the patterns[1].

Upon presentation of an input to the network, the activity of the neurons can be updated (asynchronously) according to Eq. (2.22) until the energy function has been minimised (Hopfield, 1982). Hence, repeated updates would eventually lead to convergence to one of the stored patterns. However, the network will possibly converge to spurious patterns (different from the stored patterns) as the energy in these spurious patterns is also a local minimum.

The capacity problem

The memorisation of some pattern can be retrieved when the network produces the desired vector x^p such that $x(t+1) = x(t) = x^p$. This happens when the crosstalk computed by

[1] As an associative memory, Hopfield network implements $\otimes$, $\oplus$, $\bullet$ by outer product, addition and nonlinear recurrent function, respectively.

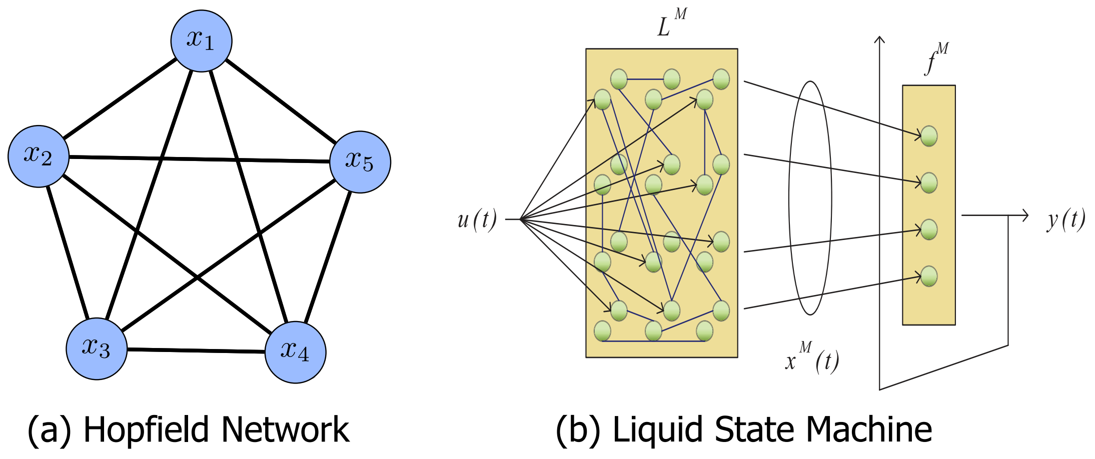

Figure 2.4: (a) Hopfield network with five neurons. (b) Structure of a Liquid State Machine M. The machine wants to transform input stream $u(\cdot)$ into output stream $y(\cdot)$ using some dynamical system L^M (the liquid).

$$\sum_{q=1,q\neq p}^{Q} x^q\left(x^p \cdot x^q\right) \tag{2.25}$$

is less than N. If the crosstalk term becomes too large, it is likely that previously stored patterns are lost because when they are presented to the network, one or more of their bits are flipped by the associative computation. We would like to keep the probability that this could happen low, so that stored patterns can always be recalled. If we set the upper bound for one bit failure at 0.01, the maximum capacity of the network is $Q \approx 0.18N$ (Rojas, 2013). With this low capacity, RNNs designed as attractor dynamics have difficulty handling big problems with massive amount of data.

2.3.2 Transient Dynamics

One major limitation of memorising by attractor mechanisms is the incapability of remembering sequences of past inputs. This demands a new paradigm to explain the working memory mechanism that enable RNNs to capture sequential dependencies and memorise information between distance external stimuli. Within this new paradigm, the trajectories of network states should become the main carriers of information about external sensory stimuli. Recent proposals (Maass et al., 2002; Maass, 2011; Jaeger and Haas, 2004) have suggested that an arbitrary recurrent network could store information about recent input sequences in its transient dynamics despite the presence of attractors (the pattern might or might not converge to the attractors). A useful analogy is the surface of a liquid. Transient ripples on the surface can encode information about past objects that were thrown in even though the water surface has no attractors (Ganguli et al., 2008). In the light of transient dynamics, RNNs carry past information to serve a given task as a working memory.

Liquid State Machines

Liquid State Machines (LSMs) (Maass et al., 2002) use a dynamic reservoir/liquid (L^M), which consists of nodes randomly connected to each other, to handle time-series data. The purpose is to map an input function of time $u\left(t\right)$–a continuous sequence of disturbances, to an output function $y\left(t\right)$ that provides a real-time analysis of the input sequence. In order to achieve that, we assume that at every time t, L^M generates an internal "liquid state" $x^M\left(t\right)$, which constitutes its current response to preceding perturbations $u(s)$ for $s \leq t$. After a certain time-period, the state of the liquid $x^M\left(t\right)$ is read as input for a readout network f^M, which by assumption, has no temporal integration capability of its own. This readout network learns to map the states of the liquid to the target outputs as illustrated in Fig. 2.4 (b).

All information about the input $u(s)$ from preceding time points $s \leq t$ that is needed to produce a target output $y(t)$ at time t has to be contained in the current liquid state $x^M\left(t\right)$. LSMs allow realisation of large computational power on functions of time even if all memory traces are continuously decaying. Instead of worrying about the code and location where information about past inputs is stored, the approach focuses on addressing the separation question: for which later time point t will any two significantly different input functions of time $u\left(t\right)$ and $v\left(t\right)$ cause significantly different liquid states $x_u^M(t)$ and $x_v^M(t)$ (Maass, 2011).

Most implementations of LSMs use the reservoir of untrained neurons. In other words, there is no need to train the weights of the RNN. The recurrent nature of the connections fuses the input sequence into a spatio-temporal pattern of neuronal activation in the liquid and computes a large variety of nonlinear functions on the input. This mechanism is theoretically possible to perform universal continuous computations. However, separation and approximation properties must be fulfilled for the system to work well. Similar neural network design can be found in Echo state networks (Jaeger and Haas, 2004). A Liquid State Machine is a particular kind of spiking neural networks that more closely mimics biological neural networks (Maass, 1997).

Memory trace of recurrent networks

When viewing recurrent networks as transient dynamics, one may want to measure the lifetimes of transient memory traces in the networks. Ganguli et al. (2018) studied a discrete time network whose dynamics is given by

$$x_i\left(n\right) = f\left(\left[Wx\left(n-1\right)\right]_i + v_i s\left(n\right) + z_i\left(n\right)\right), \, i = 1, ..., N \tag{2.26}$$

Here, a scalar time-varying signal $s(n)$ drives an RNN of N neurons. $x(n)$ is the network state at n-th timestep, $f(\cdot)$ is a general sigmoidal function, W is an $N \times N$ recurrent connectivity matrix, and v is a vector of feed-forward connections encoding the signal into the network. $z(n)$ denotes a zero mean Gaussian white noise with covariance $\langle z_i(k_1), z_j(k_2)\rangle = \varepsilon\delta_{k_1,k_2}\delta_{i,j}$.

The authors built upon Fisher information to construct useful measures of the efficiency with which the network state $x(n)$ encodes the history of the signal $s\left(n\right)$, which can be derived as

$$J\left(k\right) = v^\top W^{k\top}\left(\varepsilon\sum_{k=0}^{\infty} W^k W^{k\top}\right)^{-1} W^k v \tag{2.27}$$

where $J\left(k\right)$ measures the Fisher information that $x(n)$ retains about a signal entering the network at k time steps in the past. For a special case of normal networks having a normal connectivity matrix W, Eq. (2.27) simplifies to

$$J\left(k\right) = \sum_{i=1}^{N} v_i^2 \left|\lambda_i\right|^{2k}\left(1 - \left|\lambda_i\right|^2\right) \tag{2.28}$$

where λ_i is the i-th eigenvalue of W. For large k, the decay of the Fisher information is determined by the magnitudes of the largest eigenvalues and it decays exponentially. Similar findings with different measurements on the memory trace in modern recurrent networks are also found in a more recent work (Le et al., 2019).

2.4 External Memory for RNNs

Recurrent networks can in principle use their feedback connections to store representations of recent input events in the form of implicit memory (either attractor or transient dynamics). Unfortunately, from transient dynamics perspective, the implicit memory tends to decay quickly (Ganguli et al., 2008; Le et al., 2019). This phenomenon is closely related to gradient vanishing/exploding problems (Bengio et al., 1994; Hochreiter and Schmidhuber, 1997; Pascanu et al., 2013) which often occur when training RNNs with gradient-based algorithms such as Back-Propagation Through Time (Williams and Zipser, 1989; Werbos, 1990). A solution is to equip RNNs with external memory to cope with exponential decay of the implicit short-term memory. The external memory enhances RNNs with stronger working (Hochreiter and Schmidhuber, 1997; Cho et al., 2014b; Graves et al., 2014, 2016) or even episodic-like memory (Graves et al., 2014; Santoro et al., 2016). We will spend the next sections to analyse different types of external memory and their memory operation mechanisms. Examples of modern recurrent neural networks that utilise external memory are also discussed.

2.4.1 Cell Memory

Despite the fact that RNNs offer working memory mechanisms to handle sequential inputs, learning what to put in and how to utilise the memory is challenging. Back-Propagation Through Time (Williams and Zipser, 1989; Werbos, 1990) is the most common learning algorithm for RNNs, yet it is inefficient in training long sequences mainly due to insufficient or decaying backward error signals. This section will review the analysis of this problem and study a group of methodologies that overcome the problem through the use of cell memory and gated operation.

Hochreiter's analysis on gradient vanishing/exploding problems

Let us assume that the hidden layer of an RNN has n neurons. With differentiable activation function f_i, the activation of a neuron i at step t of the recurrent computation is as follow,

$$\begin{aligned} y^{i}\left(t\right) &= f_{i}\left(\sum_{j} w_{ij} y^{j}\left(t-1\right)\right) && (2.29)\\ &= f_{i}\left(net_{i}\left(t\right)\right) && (2.30)\end{aligned}$$

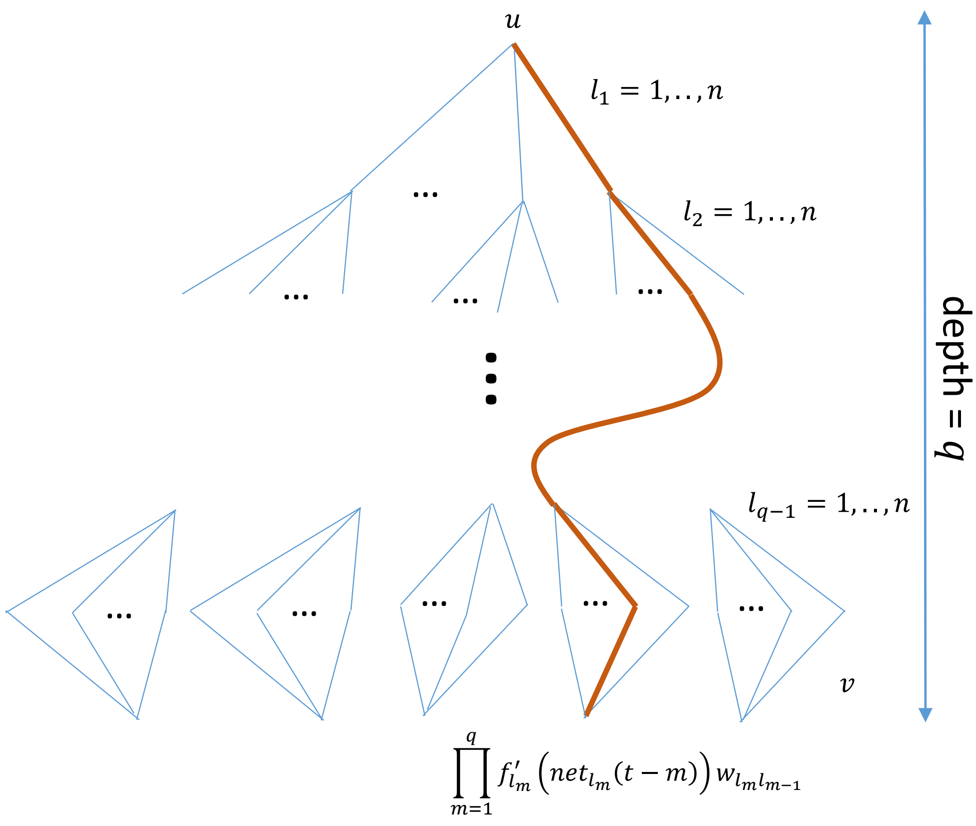
u
$l_1 = 1,..,n$
$l_2 = 1,..,n$
$l_{q-1} = 1,..,n$
depth = q
v
$\prod_{m=1}^{q} f'_{l_m}\left(net_{l_m}(t-m)\right) w_{l_m l_{m-1}}$

Figure 2.5: Error back flow from $\vartheta_u(t)$ to $\vartheta_v(t-q)$ in the computation graph. Each computation node has n children. Each product term corresponds to a computation path of depth q from node u to v. The sum of n^{q-1} products is the total error.

The backpropagated error signal for neuron j at step t is

$$\vartheta_j(t) = f'_j(net_j(t)) \sum_i w_{ij} \vartheta_i(t+1) \tag{2.31}$$

The error occurring at an arbitrary neuron u at time step t ($\vartheta_u(t)$) is backpropagated through time for q timesteps to an arbitrary neuron v ($\vartheta_v(t-q)$). We can measure the contribution of the former to the latter as the following,

$$\frac{\partial \vartheta_v(t-q)}{\partial \vartheta_u(t)} = \begin{cases} f'_v(net_v(t-1))\, w_{uv} & ; q = 1 \\ f'_v(net_v(t-q)) \sum_{l=1}^{n} \frac{\partial \vartheta_l(t-q+1)}{\partial \vartheta_u(t)} w_{lv} & ; q > 1 \end{cases} \tag{2.32}$$

By induction, we can obtain expressive form of the recursive Eq. (2.32) as

$$\frac{\partial \vartheta_v(t-q)}{\partial \vartheta_u(t)} = \sum_{l_1=1}^{n} \dots \sum_{l_{q-1}=1}^{n} \prod_{m=1}^{q} f'_{l_m}(net_{l_m}(t-m))\, w_{l_m l_{m-1}} \tag{2.33}$$

where $l_q = v$ and $l_0 = u$. The computation can be visually explained through a drawing of the computation graph as in Fig. 2.5.

It is obvious to realise that if $\left| f'_{l_m}(net_{l_m}(t-m))\, w_{l_m l_{m-1}} \right|$ is greater (smaller) than 1 for all m, then the largest product increases (decreases) exponentially with q, which represents the exploding and vanishing gradient problems in training neural networks. These problems are critical since they prevent proper update on the weights of the model, and thus freeze or disturb the learning process. With nonlinear activation functions such as sigmoid, the term $\left| f'_{l_m}(net_{l_m})\, w_{l_m l_{m-1}} \right|$ goes to zero when $w_{l_m l_{m-1}} \to \infty$ and is less than 1 when $\left| w_{l_m l_{m-1}} \right| < 4$, which implies vanishing gradient tends to occur with nonlinear activation function.

We can also rewrite Eq. (2.32) in matrix form for $q > 1$ as follows,

$$\frac{\partial \vartheta_v(t-q)}{\partial \vartheta_u(t)} = W_u^{\top} F'(t-1) \prod_{m=2}^{q-1} (W F'(t-m))\, W_v f'_v(net_v(t-q)) \tag{2.34}$$

where the weight matrix W have its elements $W_{ij} = w_{ij}$. W_u and W_v are u's incoming weight vector and v's outgoing weight vector, respectively, such that $[W_u]_i = w_{ui}$ and $[W_v]_i = w_{vi}$. $F'(t-m)$ is a diagonal matrix whose diagonal elements $F'(t-m)_{ii} = f'_i(net_i(t-m))$. Using a matrix norm $\|\cdot\|_A$ compatible with vector norm $\|\cdot\|_p$, we define

$$f'_{max} \coloneqq \max_{m=1,\ldots,q} \left\{ \left\| F'(t-m) \right\|_A \right\} \tag{2.35}$$

By applying norm sub-multiplicativity and using the inequality

$$\left| x^T y \right| \leq n \left\| x \right\|_\infty \left\| y \right\|_\infty \leq n \left\| x \right\|_p \left\| y \right\|_p,$$

we obtain a weak upper bound for the contribution

$$\left| \frac{\partial \vartheta_v (t-q)}{\partial \vartheta_u (t)} \right| \leq n \left(f'_{max} \left\| W \right\|_A \right)^q \tag{2.36}$$

This result confirms the exploding and vanishing gradient problems since the error backprob contribution decays (when $f'_{max} \left\| W \right\|_A < 1$) or grows (when $f'_{max} \left\| W \right\|_A$ >1) exponentially with q. More recent analyses on the problems are presented by Bengio et al., (1994) and Pascanu et al., (2013).

Problem with naive solution

When analysing a single neuron j with a single connection to itself, avoiding the exploding and vanishing gradient problems requires

$$f'_j \left(net_j (t) \right) w_{jj} = 1 \tag{2.37}$$

In this case, the constant error flow is enforced by using linear function f_j and constant activation (e.g., $f_j (x) = x$ with $\forall x$ and setting $w_{jj} = 1$). These properties are known as the *constant error carousel* (CEC). The strict constraint makes this solution unattractive because it limits computation capacity of RNNs with linear activation. Even worse, neuron j is connected to other neurons as well, which makes thing complicated. Let us consider an additional input weight w_{ji} connecting neuron i to j. w_{ji} is learnt to keep relevant external input from i such that $w_{ji} y_i > 0$ when the input signal y_i is relevant. Assume that the loss function is reduced by keeping neuron j active (> 0) for a while between two occurrences of two relevant inputs. During that period, activation of neuron j is possibly disturbed since with a fixed w_{ji}, $w_{ji} y_i < 0$ with irrelevant inputs. Since $y^j (t) = f_j \left(w_{jj} y^j (t-1) + w_{ji} y^i (t-1) \right)$

where f_j is linear, $y^j(t-1)$ is kept constant and $y^i(t-1)$ scales with the external input, it is likely to deactivate neuron j. Hence, if naively following CEC, learning a w_{ji} to capture relevant inputs while protecting neuron j from disturbances of irrelevant inputs is challenging (input weight conflict (Hochreiter and Schmidhuber, 1997)). Similar problem happens with the output weight (output weight conflict). These conflicts make the learning hard, and require a more flexible mechanism for controlling input/output weight impact conditioned on the input signal.

The original Long Short-Term Memory (LSTM)

Hochreiter and Schmidhuber (1997) originally proposed LSTM using multiplicative gate units and a memory cell unit to overcome the weight conflicts while following CEC. The idea is to apply CEC to neurons specialised for memorisation, each of which has an internal state independent from the activation function. This separation between memorisation and computation is essential for external memory concept. Besides, to control input/output weight impact, gate units conditioned on the inputs are multiplied with the incoming/outgoing connections, modifying the connection value through time. In particular, if a neuron c_j becomes cell memory, its output is computed as

$$y^{c_j}(t) = y^{out_j}(t)\, h\left(s_{c_j}(t)\right) \tag{2.38}$$

where $y^{out_j}(t)$ is the output gate, h is a differentiable function for scaling down the neuron's output, and s_{c_j} captures past information by using the dynamics

$$s_{c_j}(0) = 0 \tag{2.39}$$

$$s_{c_j}(t) = y^{fg_j}(t)\, s_{c_j}(t-1) + y^{in_j}(t)\, f\left(net_{c_j}(t)\right) \text{ for } t > 0 \tag{2.40}$$

where $y^{in_j}(t)$ is the input gate, $y^{fg_j}(t)$ is the (optional) forget gate and f is the activation function, which can be nonlinear. Without forget gate, c_j can be viewed as a neuron with an additional fixed self-connection. The computation paths that mainly pass through this special neuron preserve the backward error. The remaining problem is to protect this error from disturbance from other paths. The gates are calculated as

$$y^{g_j}(t) = f_{g_j}\left(\sum_u w_{g_j u} y^u(t-1)\right) \tag{2.41}$$

where g can represent input, output and forget gate. The gates are adaptive according to the input from other neurons, hence, it is possible to learn $\left\{w_{g_j u}\right\}$ to resolve the input/output weight conflict problem.

Although the cell memory provides a potential solution to cope with training RNN over long time lag, unfortunately, in practice, the multiplicative gates are not good enough to overcome a fundamental challenge of LSTM: the gates are not coordinated at the start of training, which can cause s_{c_j} to explode quickly (internal state drift). Various variants of LSTM have been proposed to tackle the problem (Greff et al., 2016). We will review some of them in Chapter 3.

Cell memory as external memory

From Eq. (2.40), we can see the internal state of the cell memory holds two types of information: (i) the previous cell state and (ii) the normal state of RNN, which is the activation of current computation. Therefore, the cell state contains a new form of external memory for RNNs. The size of the memory is often equal the number of hidden neurons in RNNs and thus, cell memory is also known as vector memory. The memory supports writing and reading mechanisms implemented as gated operations in $y^{in_j}(t)$ and y^{out_j}, respectively. They control how much to write to and read from the cell state. With the cell state, which is designed to keep information across timesteps, the working memory capacity of LSTM should be greater than that of RNNs. The memory reading and writing are also important to determine the memory capacity. For instance, if writing irrelevant information too often, the content in the cell state will saturate and the memory fails to hold much information. Later works make use of the gating mechanism to build skip-connections between inputs (a source of raw memory) and neurons in higher layers (Srivastava et al., 2015; He et al., 2016), opening chance to ease the training of very deep networks.

2.4.2 Holographic Associative Memory

The holographic associative memory (HAM) roots its operation on the principle of optical holography, where two beams of light are associated with one another in a holograms such that reconstruction of one original beam can be made by presenting another beam. Recall that the capacity of associative memory using attractor dynamics is low. To maintain Q pairs of key-value (in Hopfield network, value is also key), it requires N^2 weight storage where $Q \approx 0.18N$. HAM presents a solution to compress the key-values into a fixed size vector via Holographic Reduced Representation (HRR) without substantial loss of information (Plate, 1995). This can be done in real or complex domain using circular convolution or element-wise complex multiplication for the encoding function ($\otimes$), respectively. The compressed vector ($\mathcal{M}$), as we shall see, can be used as external memory for RNNs.

Holographic Reduced Representation

Consider a complex-valued vector key $x \in \mathbb{C}^N$,

$$x = \left[x_a\left[1\right]e^{ix_\phi[1]}, ..., x_a\left[N\right]e^{ix_\phi[N]}\right] \tag{2.42}$$

The association encoding is computed by

$$\begin{aligned} m &= x \circledast y & (2.43)\\ &= \left[x_a\left[1\right]y_a\left[1\right]e^{i\left(x_\phi[1]+y_\phi[1]\right)}, ..., x_a\left[N\right]y_a\left[N\right]e^{i\left(x_\phi[N]+y_\phi[N]\right)}\right] & (2.44)\end{aligned}$$

where $\circledast$ is element-wise complex multiplication, which multiplies the moduli and adds the phases of the elements. Trace composition function is simply addition

$$m = x^1 \circledast y^1 + x^2 \circledast y^2 + ... + x^Q \circledast y^Q \tag{2.45}$$

Although the memory m is a vector with the same dimension as that of stored items, it can store many pairs of items since we only need to store the information that discriminates them. The decoding function is multiplying an inverse key $x^{-1} = \left[x_a\left[1\right]^{-1}e^{-ix_\phi[1]}, ..., x_a\left[N\right]^{-1}e^{-ix_\phi[N]}\right]$ with the memory as follows,

$$\tilde{y} = x^{-1} \circledast m \tag{2.46}$$

$$= x^{-1} \circledast \left(\sum_{\forall k} x^k \circledast y^k \right) \tag{2.47}$$

$$= y + x^{-1} \circledast \left(\sum_{\forall k: x^k \neq x} x^k \circledast y^k \right) \tag{2.48}$$

The second term in Eq. (2.48) is noise and should be minimised. Under certain conditions, the noise term has zero mean (Plate, 1995). One way to reconstruct better is to pass the retrieved vector through an auto-associative memory to correct any errors.

Redundant Associative Long Short-Term Memory

One recent attempt to apply HRR to LSTM is the work by Danihelka et al. (2016). The authors first propose Redundant Associative Memory, an extension of HRR with multiple memory traces for multiple transformed copies of each key vector. In particular, each key vector will be transformed S times using S constant random permutation matrix P_s. Hence, we obtain the memory trace c_s for the s-th copy

$$c_s = \sum_{\forall k} \left(P_s x^k \right) \circledast y^k \tag{2.49}$$

The k-th value is retrieved as follows,

$$\tilde{y}^k = \frac{1}{S} \sum_{s=1}^{S} \left(\overline{P_s x^k} \right) \circledast c_s \tag{2.50}$$

$$= y^k + \sum_{k' \neq k} y^{k'} \circledast \frac{1}{S} \sum_{s=1}^{S} P_s \left[\overline{x^k} \circledast x^{k'} \right] \tag{2.51}$$

where $\overline{P_s x^k}$ and $\overline{x^k}$ are the complex conjugates of $P_s x^k$ and x^k, respectively, which are equal to the inverses if the modulus $x^k_a = 1$. Since permuting the key decorrelates the retrieval noise, the noise term has variance $O\left(\frac{Q}{S}\right)$ and increase the number of copies will enhance retrieval quality.

Applying the idea to LSTM, we can turn the cell memory to a holographic memory by encoding the term containing input activation in Eq. (2.40) before added up to the cell memory. The network learns to generate the key x^k and the inverse key $(x^{-1})^k$ for k-th timestep. It should be noted that the inverse key at k-th timestep can associate to some preceding key. Following Redundant Associative Memory extension, multiple copies of cell memory are employed. The cell state will be decoded to retrieve some past input activation necessary for current output (Danihelka et al., 2016). Then the decoded value will be multiplied with the output gate as in Eq. (2.38).

2.4.3 Matrix Memory

Correlation matrix memory

Correlation Matrix Memory (CMM) stores associations between pairs of vectors using outer product as the encoding function. Although the purpose looks identical to that of attractor dynamics, CMM is arranged differently using feed-forward neural network without self-loop connections. The memory construction $(\otimes + \oplus)$ follows Hebbian learning

$$M = \sum_{i=1}^{Q} y_i x_i^\top \tag{2.52}$$

where Q is the number of stored patterns, x_i and y_i are the i-th key-value pair. The memory retrieval $(\bullet)$ is simply dot product

$$\begin{aligned}
\tilde{y_j} &= M x_j && (2.53)\\
&= \left(\sum_{i=1}^{Q} y_i x_i^\top\right) x_j && (2.54)\\
&= \sum_{i=1, i\neq j}^{Q} y_i x_i^\top x_j + y_j \left\|x_j\right\|^2 && (2.55)
\end{aligned}$$

If the keys are orthonormal, then the retrieval is exact. Actually, linear independence is enough for exact retrieval. In this case, WidrowHoff learning rule should be used.

When the stored values are binary vectors, a threshold function is applied. The capacity for binary CMM is heavily dependent on the sparsity of the patterns (the sparser the better). In general, CMM offers a capacity that is at least comparable to that of the Hopfield model (Baum et al., 1988).

Fast-weight

Fast-weights refer to synapses that change slower than neuronal activities but much faster than the standard slow weights. These fast weights form temporary memories of the recent past that support the working memory of RNNs (Hinton and Plaut, 1987; Schmidhuber, 1992; Ba et al., 2016). In a recent fast-weight proposal (Ba et al., 2016), the memory is similar to a correlation matrix memory with decaying factor to put more weight on the recent past. In particular, the fast memory weight matrix A is computed as follows,

$$A\left(t\right)=\lambda A\left(t-1\right)+\eta h\left(t\right)h\left(t\right)^{\top} \tag{2.56}$$

where λ and η are the decay and learning rate, respectively. $h\left(t\right)$ is the hidden state of the RNN and also the pattern being stored in the associative memory. The memory is used to iteratively refine the next hidden state of RNN as the following,

$$h_{s+1}\left(t+1\right)=f\left(\left[Wh\left(t\right)+Cx\left(t\right)\right]+A\left(t\right)h_{s}\left(t+1\right)\right) \tag{2.57}$$

where $h_0\left(t+1\right)=f\left(Wh\left(t\right)+Cx\left(t\right)\right)$, following the ordinary dynamics of RNNs and $h_s\left(t+1\right)$ is the hidden state at s-th step of refinement.

Tensor product representation

Tensor product representation (TPR) is a mechanism to store symbolic structures. It shares common properties with CMM when the tensor is of order 2, in which tensor product is equivalent to outer product. In TPR, relations between concepts are described by the set of filler-role bindings. The vector space of filler and role are denoted as $V_{\mathcal{F}}$ and $V_{\mathcal{R}}$, respectively. The TPR is defined as a tensor T in a vector space $V_{\mathcal{F}}\otimes V_{\mathcal{R}}$, where $\otimes$ is the tensor product operator, which is computed as

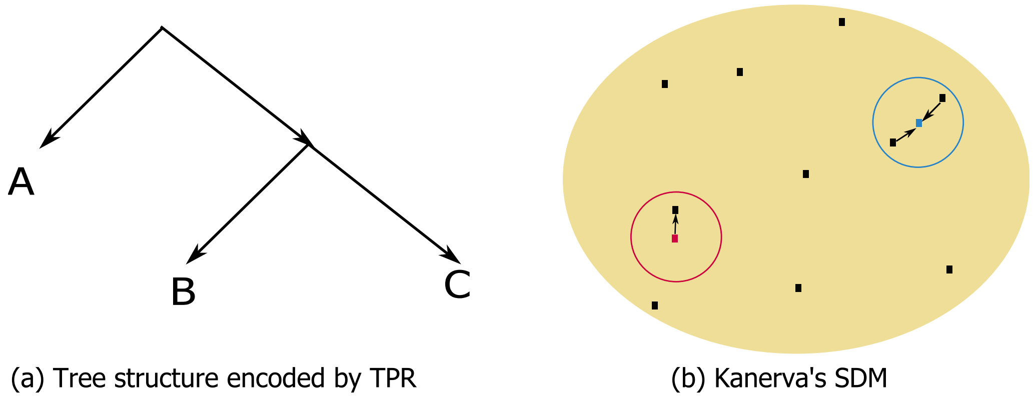

Figure 2.6: (a) Example of a tree encoded by TPR. (b) SDM's memory write (red) and read (blue) access. The read and write involve all memory locations around the queried points.

$$T = \sum_i f_i \otimes r_i \tag{2.58}$$

where f_i and r_i are vectors representing some filler and role, respectively. The tensor dot product $\bullet$ is used to decode the memory as follows,

$$f_j = T \bullet r_j \tag{2.59}$$

For example, the following 4 concepts have relations: $dog(bark)$ and $horse(big)$ in which the set of filler is $\mathcal{F} = \{bark, horse\}$ and the set of role is $\mathcal{R} = \{bark, big\}$. The TPR of these concepts is

$$T = f_{dog} \otimes r_{bark} + f_{horse} \otimes r_{big} \tag{2.60}$$

Or we can encode a tree structure as in Fig. 2.6 (a) by the following operations:

$$\begin{aligned} T &= A \otimes r_0 \otimes + (B \otimes r_0 + C \otimes r_1) \otimes r_1 && (2.61)\\ &= A \otimes r_0 \otimes + B \otimes r_0 \otimes r_1 + C \otimes r_1 \otimes r_1 && (2.62)\\ &= A \otimes r_0 \otimes + B \otimes r_{01} + C \otimes r_{11} && (2.63) \end{aligned}$$

This mechanism allows storing symbolic structures and grammars and thus supports reasoning. For further details, we refer readers to the original work (Smolensky, 1990) and recent application to deep learning (Schlag and Schmidhuber, 2018).

2.4.4 Sparse Distributed Memory

Matrix memory is a direct extension to vector memory for RNNs. There are two ways to build a matrix memory: correlation matrix memory (or tensor memory) and sparse distributed memory. While the former focuses on storing the associations amongst items (e.g., Hopfield network, Holographic memory and CMM), the latter aims to store each item as a high-dimensional vector, which is closer to Random Access Memory in computer architecture. Because each vector is physically stored in a memory slot, we also refer to this model as slot-based memory. Sparse distributed memory (SDM) can represent correlation matrix memory, computer memory, feed-forward artificial neural networks and associative-memory models of the cerebellum. Such a versatility naturally results in SDM's applications to RNN as one form of external memory.

Kanerva memory model

In 1988, Pentti Kanerva introduced the SDM as a new approach to model human long-term memory (Kanerva, 1988). The model revolves around a simple idea that the distances between concepts in our minds correspond to the distances between points of a high-dimensional space. As we, when hinted by key signals, tend to remember specific things such as individual, object, scene and place, the brain must make the identification nearly automatic, and high-dimensional vectors as internal representations of things do that. Another important property of high dimensional spaces is that distance between two random points should be far, which allows inexact representation of the point of interest. In other words, using long vectors to store items enables a fault-tolerant and robust memory.

The SDM stores items (binary vectors) in a large number of hard locations or memory slots whose addresses (m_a) are given by binary strings of length D, randomly distributed throughout the address space $\{0,1\}^D$. Input to the memory consists of two binary patterns, an address pattern (location to be accessed) and a content pattern (item to be stored). The pattern is called self-addressing when its content is also its address. Furthermore, in SDM, each memory slot m is armed with a vector of counters m_c initialised to 0 with the same length of the content. The memory operations are based on similarity between the addresses.

Algorithm 2.1 Memory writing in SDM

```
Require: input $x$ and SDM
 1: Find a set of chosen locations $M(x)$ using Eq. (2.64)
 2: for each $m$ in $M(x)$ do
 3:     for $i = 1, D$ do
 4:         if $x_c[i] == 1$ then
 5:             $m_c[i] += 1$
 6:         else
 7:             $m_c[i] -= 1$
 8:         end if
 9:     end for
10: end for
```

Memory writing When storing input item $x = (x_a, x_c)$ to the SDM, the address pattern x_a is compared against all memory location addresses. Relevant physical locations to consider are those which lie within a hypersphere of radius r centered on the address pattern point

$$M\left(x\right) = \left\{m : d\left(m_a, x_a\right) < r\right\} \tag{2.64}$$

where d is some similarity measure between 2 vectors. In the original model, Kanerva used Hamming distance. The content is distributed in the set of locations $M\left(x\right)$ as in Algo. 2.1.

Memory reading Basically, reading from any point in the memory space pools the data of all nearby locations. Given a cue address x'_a, contents of the counters at locations near x'_a are summed and thresholded at zero to return the binary content. The proximity criteria still follows Eq. (2.64). The reading mechanism allows SDM to retrieve data from imprecise or noisy cues. Fig. 2.6 (b) visualises the memory access behaviors.

The assumption underlying the original SDM are: (i) the location addresses are fixed, and only the contents of the locations are modifiable, (ii) the locations are sparse and distributed across the address space $\{0, 1\}^D$ (e.g., randomly sample 10^6 addresses from an address space of 1000 dimensions). These assumptions make the model perform well on storing random input data.

SDM as an associative matrix memory We can implement SDM by using three operations of associative memory. The minimum setting for this implementation includes:

- A hard-address matrix $A \in \mathbb{B}^{N \times D}$ where N and D are the numbers of memory locations and the dimension of the address space, respectively.
- A counter (content) matrix $C \in \mathbb{B}^{N \times D}$.
- Cosine similarity is used to measure proximity.
- Threshold function $\boldsymbol{y}$ that maps distances to binary values:$\boldsymbol{y}(d) = 1$ if $d \geq r$ and vice versa.
- Threshold function $\boldsymbol{z}$ that converts a vector to binary vector: $\boldsymbol{z}(x) = 1$ if $x \geq 0$ and vice versa.

Then, the memory writing ($\otimes + \oplus$) and reading ($\bullet$) become

$$C := C + \boldsymbol{y}(Ax_a)\, x_c^\top \tag{2.65}$$

$$x_c' = \boldsymbol{z}\left(C^\top \boldsymbol{y}(Ax_a')\right) \tag{2.66}$$

These expressions are closely related to attention mechanisms commonly used nowadays (Sec. 3.2.2).

In general, SDM overcomes limitations of correlation matrix memory such as Hopfield network since the number of stored items in SDM is not limited by the number of processing elements. Moreover, one can design SDM to store a sequence of patterns. Readers are referred to Keeler (1988) for a detailed comparison between SDM and Hopfield network (Keeler, 1988).

Memory-augmented neural networks and attention mechanisms

The current wave of deep learning has leveraged the concept of SDM to external neural memory capable of supporting the working memory of RNNs (Weston et al., 2014; Graves et al., 2014, 2016; Miller et al., 2016). These models enhance the SDM with real-valued vectors and learnable parameters. For example, the matrices A and C can be automatically generated by a learnable neural network. To make whole architecture learnable, differentiable functions and flexible memory operations must be used. Attention mechanisms are the most common operations used in MANNs to facilitate the similarity-based memory access of SDM. Through various ways of employing attentions, RNNs can access the external memory in the same manner as one accesses SDM. Details on neural distributed (slot-based) memory and attention mechanisms will be provided in Chapter 3.

2.5 Relation to Computational Models

Automatons are abstract models of machines that perform computations on an input by moving through a series of states (Sipser et al., 2006). Once the computation reaches a finish state, it accepts and possibly produces the output of that input. In terms of computational capacity, there are three major classes of automaton:

- Finite-state machine
- Pushdown automata
- Turing machine

Pushdown automata and Turing machine can be thought of as extensions of finite-state machines (FSMs) when equipped with an external storage in the form of stack and memory tape, respectively. With stored-program memory, an even more powerful class of machines, which simulates any other Turing machines, can be built as universal Turing machine (Turing, 1936). As some Turing machines are also universal, they are usually regarded as one of the most general and powerful automata besides universal Turing machines.

One major objective of automata theory is to understand how machines compute functions and measure computation power of models. For example, RNNs, if properly wired, are Turing-complete (Siegelmann and Sontag, 1995), which means they can compute arbitrary sequences if they have unlimited memory. Nevertheless, in practice, RNNs struggle to learn from the data to predict output correctly given simple input sequence (Bengio et al., 1994). This poses a question on the effective computation power of RNNs.

Another way to measure the capacity of RNNs is via simulations of operations that they are capable of doing. The relationship between RNNs and FSMs has been discovered by many (Giles et al., 1992; Casey, 1996; Omlin and Giles, 1996; Tiňo et al., 1998), which suggest that RNNs can mimic FSMs by training with data. The states of an RNN must be grouped into partitions representing the states of the generating automation. Following this line of thinking, we can come up with neural architectures that can simulate pushdown automata, Turing machine and universal Turing machine. Neural stack is an example which arms RNN with a stack as its external memory (Mozer and Das, 1993; Joulin and Mikolov, 2015; Grefenstette et al., 2015). By simulating push and pop operations, which are controlled by the RNN, neural stack mimics the working mechanism of pushdown automata. Neural Turing Machine and its extension Differentiable Neural Computer (Graves et al., 2014, 2016) are prominent neural realisations of Turing machine. They use an RNN controller to read from and write to an external memory in a manner resembling Turing machine's operations on its memory tape. Since the memory access is not limited to the top element as in neural stack, these models have more computational flexibility. Until recently, Le et al. (2020) extended the simulation to the level of universal Turing machine by employing the stored-program principle (Turing, 1936; von Neumann, 1993). We save a thorough analysis on the correspondence between these MANNs and Turing machines for Chapter 7. Here, we briefly draw a correlation between models of recurrent neural networks and automata (see Fig. 2.7).

It should be noted that the illustration is found on the organisation of memory in the models rather than the computational capacity. For example, some Turing machines are equivalent to universal Turing machine in terms of capacity; RNNs are on par with other MANNs because they are all Turing-complete. Having said that, when neural networks are organised in a way that simulates powerful automata, their effective capacity is often greater and thus, they tend to perform better in complicated sequential learning tasks (Graves et al., 2014, 2016; Le et al., 2020). A similar taxonomy with proof of inclusion relation amongst models can be found in the literature (Ma and Principe, 2018).

2.6 Closing Remarks

We have briefly reviewed different kinds of memory organisations in the neural network literature. In particular, we described basic neural networks such as Feed-forward and Recurrent Neural Networks and their primary forms of memory con-

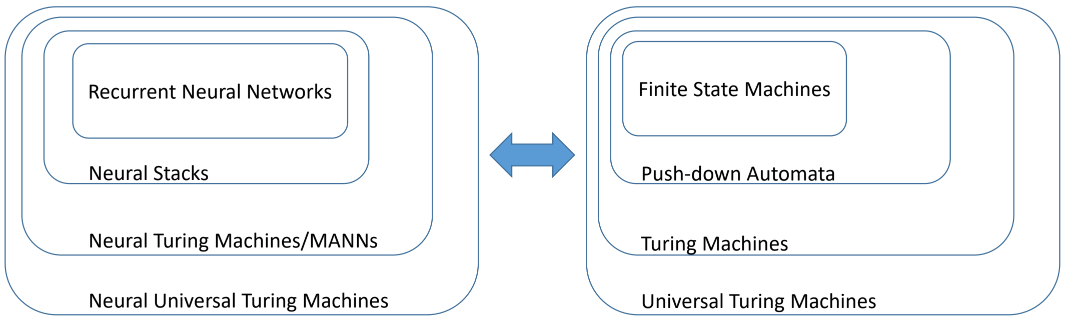
Recurrent Neural Networks
Neural Stacks
Neural Turing Machines/MANNs
Neural Universal Turing Machines
Finite State Machines
Push-down Automata
Turing Machines
Universal Turing Machines

Figure 2.7: Relation between external memory and computational models

structions, followed by a taxonomy on mathematical models of well-known external memory designs based on memory operational mechanisms and relations to automation theory. In the next chapter, we narrow the scope of literature review to the main content of this thesis: Memory-augment Neural Networks and their extensions.