Lecture notes on recurrent neural networks (RNNs)

1.1 Simple RNNs

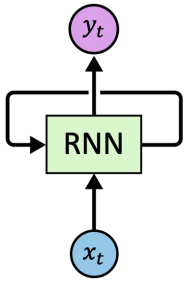
Mathematical form

Assume we want to build a neural network to process a data sequence $\boldsymbol{x}_{1:T} = (\boldsymbol{x}_1, ..., \boldsymbol{x}_T)$. Recurrent neural networks (RNNs) are neural networks suited for processing sequential data, which, if well trained, can model dependencies within a sequence of arbitrary length.

We also assume a supervised learning task which aims to learn the mapping from inputs $\boldsymbol{x}_{1:T}$ to outputs $\mathbf{y}_{1:T} = (\mathbf{y}_1, ..., \mathbf{y}_T)$. Then a simple RNN computes the following mapping for $t = 1, ..., T$:

$$\boldsymbol{h}_t = \phi_h(W_h \boldsymbol{h}_{t-1} + W_x \boldsymbol{x}_t + \boldsymbol{b}_h), \quad (1)$$

$$\mathbf{y}_t = \phi_y(W_y \boldsymbol{h}_t + \boldsymbol{b}_y). \quad (2)$$



Here the network parameters are $\boldsymbol{\theta} = \{W_h, W_x, W_y, \boldsymbol{b}_h, \boldsymbol{b}_y\}$, ϕ_h and ϕ_y are the non-linear activation functions for the hidden state $\boldsymbol{h}_t$ and the output y_t, respectively. For $t = 1$ the convention is to set $\boldsymbol{h}_0 = \mathbf{0}$ so that $\boldsymbol{h}_1 = \phi_h(W_x \boldsymbol{x}_1 + \boldsymbol{b}_h)$; alternatively $\boldsymbol{h}_0$ can also be added to $\boldsymbol{\theta}$ as a learnable parameter.

Back-propagation through time (BPTT)

A loss function is required for training an RNN. Assuming the following loss function to minimise:

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{t=1}^{T} \mathcal{L}(\mathbf{y}_t). \quad (3)$$

This is a common form for the loss function in many sequential modelling tasks such as video/audio sequence reconstruction. The derivative of the loss function w.r.t. $\boldsymbol{\theta}$ is $\frac{d}{d\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) = \sum_{t=1}^{T} \frac{d}{d\boldsymbol{\theta}}\mathcal{L}(\mathbf{y}_t)$, therefore it remains to compute $\frac{d}{d\boldsymbol{\theta}}\mathcal{L}(\mathbf{y}_t)$ for $\boldsymbol{\theta} = \{W_h, W_x, W_y, \boldsymbol{b}_h, \boldsymbol{b}_y\}$.

- Derivative of $\mathcal{L}(\mathbf{y}_t)$ w.r.t. W_y and $\boldsymbol{b}_y$:

$$\frac{d\mathcal{L}(\mathbf{y}_t)}{dW_y} = \frac{d\mathcal{L}(\mathbf{y}_t)}{d\mathbf{y}_t}\frac{d\mathbf{y}_t}{dW_y}, \quad \frac{d\mathcal{L}(\mathbf{y}_t)}{d\boldsymbol{b}_y} = \frac{d\mathcal{L}(\mathbf{y}_t)}{d\mathbf{y}_t}\frac{d\mathbf{y}_t}{d\boldsymbol{b}_y}.$$

- Derivative of $\mathcal{L}(\mathbf{y}_t)$ w.r.t. W_x and $\boldsymbol{b}_h$:

$$\frac{d\mathcal{L}(\mathbf{y}_t)}{dW_x} = \frac{d\mathcal{L}(\mathbf{y}_t)}{d\mathbf{y}_t}\frac{dy_t}{d\boldsymbol{h}_t}\frac{d\boldsymbol{h}_t}{dW_x}, \quad \frac{d\mathcal{L}(\mathbf{y}_t)}{d\boldsymbol{b}_h} = \frac{d\mathcal{L}(\mathbf{y}_t)}{d\mathbf{y}_t}\frac{dy_t}{d\boldsymbol{h}_t}\frac{d\boldsymbol{h}_t}{d\boldsymbol{b}_h}.$$

  Here W_x and $\boldsymbol{b}_h$ contributes to $\boldsymbol{h}_t$ in two ways: both direct and indirect contributions, where the latter is through $\boldsymbol{h}_{t-1}$. This means:

$$\frac{d\boldsymbol{h}_t}{dW_x} = \frac{\partial \boldsymbol{h}_t}{\partial W_x} + \frac{d\boldsymbol{h}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{h}_{t-1}}{dW_x}, \quad \frac{d\boldsymbol{h}_t}{d\boldsymbol{b}_h} = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{b}_h} + \frac{d\boldsymbol{h}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{h}_{t-1}}{d\boldsymbol{b}_h}.$$

  Derivations of these derivatives requires the usage of chain rule which will be explained next.

- Derivative of $\mathcal{L}(\mathbf{y}_t)$ w.r.t. W_h: by chain rule, we have

$$\frac{d\mathcal{L}(\mathbf{y}_t)}{dW_h} = \frac{d\mathcal{L}(\mathbf{y}_t)}{d\boldsymbol{h}_t}\frac{d\boldsymbol{h}_t}{dW_h}$$

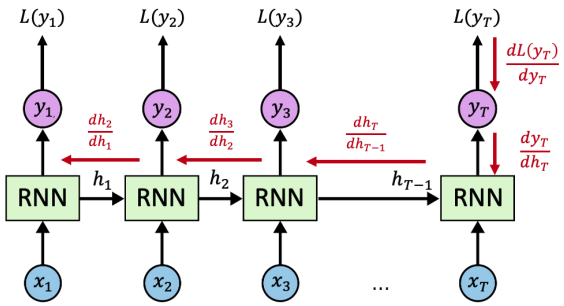


Figure 1: Visualising Back-propagation through time (BPTT) without truncation. The black arrows show forward pass computations, while the red arrows show the gradient back-propagation in order to compute $\nabla_{W_h}\mathcal{L}(y_t)$.

where $\frac{d\mathcal{L}(\mathbf{y}_t)}{d\boldsymbol{h}_t} = \frac{d\mathcal{L}(\mathbf{y}_t)}{d\mathbf{y}_t}\frac{d\mathbf{y}_t}{d\boldsymbol{h}_t}$. Importantly, here the entries in the Jacobian $\frac{d\boldsymbol{h}_t}{dW_h}$ contains the *total gradient* of $\boldsymbol{h}_t[i]$ w.r.t. $W_h[m,n]$. It remains to compute $\frac{d\boldsymbol{h}_t}{dW_h}$ and notice that $\boldsymbol{h}_t$ depends on $\boldsymbol{h}_{t-1}$ which also depends on W_h:

$$\frac{d\boldsymbol{h}_t}{dW_h} = \frac{\partial \boldsymbol{h}_t}{\partial W_h} + \frac{d\boldsymbol{h}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{h}_{t-1}}{dW_h}. \tag{4}$$

Here the entries in $\frac{\partial \boldsymbol{h}_t}{\partial W_h}$ contains *partial gradient* only (by treating $\boldsymbol{h}_{t-1}$ as a constant w.r.t. W_h, note that $\boldsymbol{h}_t$ depends on $\boldsymbol{h}_{t-1}$). By expanding the $\frac{d\boldsymbol{h}_{t-1}}{dW_h}$ term further, we have:

$$\begin{aligned}\frac{d\boldsymbol{h}_t}{dW_h} &= \frac{\partial \boldsymbol{h}_t}{\partial W_h} + \frac{d\boldsymbol{h}_t}{d\boldsymbol{h}_{t-1}}\frac{\partial \boldsymbol{h}_{t-1}}{\partial W_h} + \frac{d\boldsymbol{h}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{h}_{t-1}}{d\boldsymbol{h}_{t-2}}\frac{d\boldsymbol{h}_{t-2}}{dW_h} = \dots \\ &= \sum_{\tau=1}^{t}\left(\prod_{l=\tau}^{t-1}\frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l}\right)\frac{\partial \boldsymbol{h}_\tau}{\partial W_h},\end{aligned} \tag{5}$$

with the convention that when $\tau = t$, $\prod_{l=t}^{t-1}\frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l} = 1$. This means the chain rule of the gradients needs to be computed in an reversed order from time $t = T$ to time $t = 1$, hence the name *Back-propagation through time (BPTT)*. A visualisation of BPTT is provided in Figure 1. Truncation with length L might be applied to this back-propagation procedure, and with *truncated BPTT* the gradient is computed as

$$\text{truncate}[\frac{d\boldsymbol{h}_t}{dW_h}] = \sum_{\tau=\max(1,t-L)}^{t}\left(\prod_{l=\tau}^{t-1}\frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l}\right)\frac{\partial \boldsymbol{h}_\tau}{\partial W_h}.$$

Gradient vanishing/explosion issues

Simple RNNs are often said to suffer from gradient vanishing or gradient explosion issues. To understand this, notice that

$$\frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l}^\top = \phi_h'(W_h\boldsymbol{h}_l + W_x\boldsymbol{x}_{l+1} + \boldsymbol{b}_h)\odot W_h, \tag{6}$$

Here $\phi_h'(W_h\boldsymbol{h}_l + W_x\boldsymbol{x}_{l+1} + \boldsymbol{b}_h)$ denotes a vector containing element-wise derivatives, and we reload the element-wise product operator for vector $\boldsymbol{a}\in\mathbb{R}^{d\times 1}$ and $\boldsymbol{B}\in\mathbb{R}^{d\times d'}$ as the "broadcasting element-wise product" $\boldsymbol{a}\odot\boldsymbol{B} := [\underbrace{\boldsymbol{a},...,\boldsymbol{a}}_{\text{repeat } d' \text{ times}}]\odot\boldsymbol{B}$. This means $\prod_{l=\tau}^{t-1}\frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l}$ contains products of $t-\tau$ copies of W_h and the derivative $\phi_h'(\cdot)$ at time steps $l = \tau, ..., t-1$.

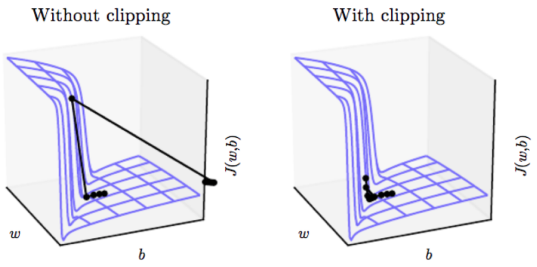


Figure 2: Visualising the gradient step with/out gradient clipping. Source: Goodfellow et al. [2016].

Now consider a simple case where $\phi_h(\cdot)$ is an identity mapping so that $\phi_h'(\cdot) = 1$. We further assume the hidden states have scalar values, i.e. $\dim(\boldsymbol{h}_t) = 1$. Then we have $\prod_{l=\tau}^{t-1} \frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l} = (W_h^{t-\tau})^\top$ which can vanish or explode when $t - \tau$ is large, depending on whether $W_h < 1$ or not. In the general case when W_h is a matrix, depending on whether the largest singular value (i.e. maximum of the absolute values of the largest and smallest eigenvalues) of W_h is smaller or larger than 1, the spectral norm of $\prod_{l=\tau}^{t-1} \frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l} = (W_h^{t-\tau})^\top$ will vanish or explode when $t - \tau$ increases. When $\phi_h(\cdot)$ is selected as the sigmoid function or the hyperbolic tangent function, gradient vanishing problem can still happen. Take hyperbolic tangent function as an example. When entries in $\boldsymbol{h}_t$ is close to ± 1, then $\phi_h'(\cdot) \approx 0$, i.e. the derivative is saturated. Multiplying several of such saturated derivatives together also leads to the gradient vanishing problem.

Some tricks to fix the gradient vanishing/explosion issues

There are a handful of empirical tricks to fix the gradient vanishing/explosion issues discussed above.

- Gradient clipping:
  This trick is often used to prevent the gradient from explosion. With a fixed hyper-parameter γ, a gradient $\boldsymbol{g}$ is clipped when $||\boldsymbol{g}|| > \eta$:

  $$\boldsymbol{g} \leftarrow \frac{\gamma}{||\boldsymbol{g}||}\boldsymbol{g}.$$

  This trick ensures the gradients used in optimisation has their maximum norm bounded by a pre-defined hyper-parameter. It introduces biases in the gradient-based optimisation procedure, but in certain cases it can be beneficial. Figure 2 visualises such an example, where with gradient clipping, the updates can stay in the valley of the loss function.

- Good initialisation of the recurrent weight matrix W_h:
  The IRNN approach [Le et al., 2015] uses ReLU activation's for ϕ_h and initialise $W_h = \mathbf{I}$, $\boldsymbol{b}_h = \mathbf{0}$. This makes $\phi_h'(t) = \delta(t > 0)$ and $\frac{d\boldsymbol{h}_{l+1}}{d\boldsymbol{h}_l} = \delta(W_x \boldsymbol{x}_{l+1} > 0)$ at initialisation. While there is no guarantee of eliminating the gradient vanishing/explosion problem during the whole course of training, empirically RNNs with this trick have achieve competitive performance to LSTMs in a variety of tasks.

- Alternatively, one can construct the recurrent weight matrix W_h to be orthogonal or unitary matrix. See e.g. Saxe et al. [2014]; Arjovsky et al. [2016] for examples.

1.2 Long Short-Term Memory (LSTM)

Mathematical form

The Long Short-term Memory [Hochreiter and Schmidhuber, 1997] was proposed with the motivation of addressing the gradient vanishing/explosion problem. It introduces *memory cell states* and *gates*

to control the error flows, in detail the computation goes as follows (with $\sigma(\cdot)$ as sigmoid function):

$$\begin{aligned}
\boldsymbol{f}_t &: \text{forget gate} & \boldsymbol{f}_t &= \sigma(W_f \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f) && (7)\\
\boldsymbol{i}_t &: \text{input gate} & \boldsymbol{i}_t &= \sigma(W_i \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i) && (8)\\
\boldsymbol{o}_t &: \text{output gate} & \boldsymbol{o}_t &= \sigma(W_o \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o) && (9)\\
\boldsymbol{x}_t &: \text{input} & \tilde{\boldsymbol{c}}_t &= \tanh(W_c \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_c) && (10)\\
\boldsymbol{c}_t &: \text{memory cell state} & \boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t && (11)\\
\boldsymbol{h}_t &: \text{hidden state} & \boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) && (12)
\end{aligned}$$

The parameters of an LSTM are therefore $\boldsymbol{\theta} = \{W_f, W_i, W_o, W_c, \boldsymbol{b}_f, \boldsymbol{b}_i, \boldsymbol{b}_o, \boldsymbol{b}_c\}$. Again by convention, $\boldsymbol{h}_0$ and $\boldsymbol{c}_0$ are either set to zero vectors or added to the learnable parameters. We note that if the initial cell state $\boldsymbol{c}_0$ is initialised to zero, then we have the elements in $\boldsymbol{c}_t$ and $\boldsymbol{h}_t$ bounded within $(-1, 1)$. The output y_t can be produced by a 1-layer neural network similar to the simple RNN case: $\mathbf{y}_t = \phi_y(W_y \boldsymbol{h}_t + \boldsymbol{b}_y)$.

*Gradient computation

Readers are encouraged to derive themselves the gradient of $\mathcal{L}(\mathbf{y}_t)$ with respect to $\boldsymbol{\theta}$. Specifically for the recurrent weight matrix W_c, computing the derivative $\frac{d\mathcal{L}(\mathbf{y}_t)}{dW_c}$ requires the following terms:

$$\frac{d\mathcal{L}(\mathbf{y}_t)}{dW_c} = \frac{d\mathcal{L}(\mathbf{y}_t)}{d\boldsymbol{h}_t}\frac{d\boldsymbol{h}_t}{dW_c} \tag{13}$$

$$\frac{d\boldsymbol{h}_t}{dW_c} = \boldsymbol{o}_t \odot \frac{d\tanh(\boldsymbol{c}_t)}{dW_c} + \tanh(\boldsymbol{c}_t) \odot \frac{d\boldsymbol{o}_t}{dW_c} \tag{14}$$

$$\frac{d\boldsymbol{o}_t}{dW_c} = \frac{d\boldsymbol{o}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{h}_{t-1}}{dW_c} \tag{15}$$

$$\frac{d\boldsymbol{c}_t}{dW_c} = \boldsymbol{f}_t \odot \frac{d\boldsymbol{c}_{t-1}}{dW_c} + \boldsymbol{c}_{t-1} \odot \frac{d\boldsymbol{f}_t}{dW_c} + \boldsymbol{i}_t \odot \frac{d\tilde{\boldsymbol{c}}_t}{dW_c} + \tilde{\boldsymbol{c}}_t \odot \frac{d\boldsymbol{i}_t}{dW_c}. \tag{16}$$

As $\boldsymbol{h}_{t-1}$ also depends on $\boldsymbol{c}_{t-1}$ and $\boldsymbol{o}_{t-1}$, it means that

$$\frac{d\boldsymbol{f}_t}{dW_c} = \frac{d\boldsymbol{f}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{h}_{t-1}}{dW_c} = \frac{d\boldsymbol{f}_t}{d\boldsymbol{h}_{t-1}}\left(\boldsymbol{o}_{t-1} \odot \frac{d\tanh(\boldsymbol{c}_{t-1})}{dW_c} + \tanh(\boldsymbol{c}_{t-1}) \odot \frac{d\boldsymbol{o}_{t-1}}{dW_c}\right) \tag{17}$$

$$\frac{d\boldsymbol{i}_t}{dW_c} = \frac{d\boldsymbol{i}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{h}_{t-1}}{dW_c} = \frac{d\boldsymbol{i}_t}{d\boldsymbol{h}_{t-1}}\left(\boldsymbol{o}_{t-1} \odot \frac{d\tanh(\boldsymbol{c}_{t-1})}{dW_c} + \tanh(\boldsymbol{c}_{t-1}) \odot \frac{d\boldsymbol{o}_{t-1}}{dW_c}\right) \tag{18}$$

$$\frac{d\tilde{\boldsymbol{c}}_t}{dW_c} = \frac{\partial\tilde{\boldsymbol{c}}_t}{\partial W_c} + \frac{d\tilde{\boldsymbol{c}}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{h}_{t-1}}{dW_c} = \frac{\partial\tilde{\boldsymbol{c}}_t}{\partial W_c} + \frac{d\tilde{\boldsymbol{c}}_t}{d\boldsymbol{h}_{t-1}}\left(\boldsymbol{o}_{t-1} \odot \frac{d\tanh(\boldsymbol{c}_{t-1})}{dW_c} + \tanh(\boldsymbol{c}_{t-1}) \odot \frac{d\boldsymbol{o}_{t-1}}{dW_c}\right). \tag{19}$$

Note again the difference between $\frac{d\tilde{\boldsymbol{c}}_t}{dW_c}$ and $\frac{\partial\tilde{\boldsymbol{c}}_t}{\partial W_c}$. The former Jacobian $\frac{d\tilde{\boldsymbol{c}}_t}{dW_c}$ has its entries as the *total gradient* of $\tilde{\boldsymbol{c}}_t[i]$ w.r.t. $W_c[m, n]$, while the latter partial gradient $\frac{\partial\tilde{\boldsymbol{c}}_t}{\partial W_c}$ has its entries as the *partial gradient* of $\tilde{\boldsymbol{c}}_t[i]$ w.r.t. $W_c[m, n]$ (by treating $\boldsymbol{h}_{t-1}$ as a constant w.r.t. W_c, note that $\tilde{\boldsymbol{c}}_t$ depends on $\boldsymbol{h}_{t-1}$ as well). Combining the derivations, we have (notice that $\frac{d\tanh(\boldsymbol{c}_{t-1})}{dW_c} = \frac{d\tanh(\boldsymbol{c}_{t-1})}{d\boldsymbol{c}_{t-1}}\frac{d\boldsymbol{c}_{t-1}}{dW_c}$):

$$\frac{d\boldsymbol{o}_t}{dW_c} = \frac{d\boldsymbol{o}_t}{d\boldsymbol{h}_{t-1}}\left(\boldsymbol{o}_{t-1} \odot \frac{d\tanh(\boldsymbol{c}_{t-1})}{dW_c} + \tanh(\boldsymbol{c}_{t-1}) \odot \frac{d\boldsymbol{o}_{t-1}}{dW_c}\right) \tag{20}$$

$$\frac{d\boldsymbol{c}_t}{dW_c} = \underbrace{\left(\boldsymbol{f}_t + \boldsymbol{o}_{t-1} \odot \frac{d\tanh(\boldsymbol{c}_{t-1})}{d\boldsymbol{c}_{t-1}} \odot \frac{d\boldsymbol{c}_t}{d\boldsymbol{h}_{t-1}}\right)}_{=\frac{d\boldsymbol{c}_t}{d\boldsymbol{c}_{t-1}}}\frac{d\boldsymbol{c}_{t-1}}{dW_c} + \tanh(\boldsymbol{c}_{t-1}) \odot \frac{d\boldsymbol{c}_t}{d\boldsymbol{h}_{t-1}}\frac{d\boldsymbol{o}_{t-1}}{dW_c} + \boldsymbol{i}_t \odot \frac{\partial\tilde{\boldsymbol{c}}_t}{\partial W_c} \tag{21}$$

$$\frac{d\boldsymbol{c}_t}{d\boldsymbol{h}_{t-1}} = \boldsymbol{c}_{t-1} \odot \frac{d\boldsymbol{f}_t}{d\boldsymbol{h}_{t-1}} + \tilde{\boldsymbol{c}}_t \odot \frac{d\boldsymbol{i}_t}{d\boldsymbol{h}_{t-1}} + \boldsymbol{i}_t \odot \frac{d\tilde{\boldsymbol{c}}_t}{d\boldsymbol{h}_{t-1}}. \tag{22}$$

This means for computing $\frac{d\boldsymbol{c}_t}{dW_c}$ it requires computing

$$\prod_{l=\tau}^{t-1} \frac{d\boldsymbol{c}_{l+1}}{d\boldsymbol{c}_l} = \prod_{l=\tau}^{t-1} [\boldsymbol{f}_{l+1} + \boldsymbol{o}_l \odot \frac{d\tanh(\boldsymbol{c}_l)}{d\boldsymbol{c}_l} \odot \frac{d\boldsymbol{c}_{l+1}}{d\boldsymbol{h}_l}]$$

for all $\tau = 1, ..., t$. There is no guarantee that this term will not vanish or explode, however the usage of forget gates makes the issue less severe. To see this, notice that by expanding the product term above, we have that it contains terms proportional to $\boldsymbol{f}_{i+1} \odot \prod_{l=\tau}^{i} \boldsymbol{o}_l \odot \frac{d\boldsymbol{c}_{l+1}}{d\boldsymbol{h}_l}$ for $i = \tau+1, ..., t-1$. So if in the forward pass the network sets $\boldsymbol{f}_{i+1} \to 0$ (i.e. forgetting the previous cell state), then this will also likely to bring $\boldsymbol{f}_{i+1} \odot \prod_{l=\tau}^{i} \boldsymbol{o}_l \odot \frac{d\boldsymbol{c}_{l+1}}{d\boldsymbol{h}_l} \approx 0$, which is helpful to cope with the gradient explosion problem. On the other hand, $\frac{d\boldsymbol{c}_t}{dW_c}$ also contains terms proportional to $\prod_{l=\tau+1}^{i} \boldsymbol{f}_l \odot \boldsymbol{o}_\tau \odot \frac{d\boldsymbol{c}_{\tau+1}}{d\boldsymbol{h}_\tau}$ for $i = \tau+1, ..., t-1$. This means if the network sets $\boldsymbol{f}_l \to 1$ for $l = \tau+1, ..., i$ (i.e. maintaining the cell state until at least time $t = i$), then it will be likely that $\prod_{l=\tau+1}^{i} \boldsymbol{f}_l \odot \boldsymbol{o}_\tau \odot \frac{d\boldsymbol{c}_{\tau+1}}{d\boldsymbol{h}_\tau} \approx \boldsymbol{o}_\tau \odot \frac{d\boldsymbol{c}_{\tau+1}}{d\boldsymbol{h}_\tau}$, which helps prevent the gradient info at time τ from vanishing when $\boldsymbol{o}_\tau \to 1$, and thus be helpful to learn longer term dependencies. The gradients $\frac{d\boldsymbol{c}_t}{dW_c}$ and $\frac{d\boldsymbol{o}_t}{dW_c}$ also require computing products of $\frac{d\boldsymbol{o}_{i+1}}{d\boldsymbol{h}_i}$ and $\frac{d\boldsymbol{c}_{i+1}}{d\boldsymbol{h}_i}$ terms, and analogous analysis can be done for those product terms. It is worth emphasising again that LSTM does NOT solve the gradient vanishing/explosion problem completely, however empirical evidences have shown that it is easier for LSTMs to learn longer term dependencies when compared with the simple RNN.

Gated Recurrent Unit (GRU): a simplified gated RNN

The Gated Recurrent Unit (GRU) [Cho et al., 2014] improves the simple RNN with the gating mechanism as well. Compared with LSTM, GRU removes the input/output gates and the cell state, but still maintains the forgetting mechanism in some form:

$$\boldsymbol{z}_t = \sigma(W_z \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_z) \tag{23}$$

$$\boldsymbol{r}_t = \sigma(W_r \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_r) \tag{24}$$

$$\tilde{\boldsymbol{h}}_t = \tanh(W_h \cdot [\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_h) \tag{25}$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t \tag{26}$$

The network parameters are then $\boldsymbol{\theta} = \{W_z, W_r, W_h, \boldsymbol{b}_z, \boldsymbol{b}_r, \boldsymbol{b}_h\}$. We see here $\boldsymbol{z}_t$ acts as the the update gate which determines the incorporation of the current info to the hidden states, and $\boldsymbol{r}_t$ is named the reset gate which also impact on the maintenance of the historical information.

1.3 Sequence-to-sequence models

Given dataset of input-output sequence pairs $(\boldsymbol{x}_{1:T}, \mathbf{y}_{1:L})$, the goal of sequence prediction is to build a model $p_{\boldsymbol{\theta}}(\mathbf{y}_{1:L}|\boldsymbol{x}_{1:T})$ to fit the data. Note here that the $\boldsymbol{x}$, $\mathbf{y}$ sequences might have different lengths, and the input/output length T and L can vary across input-output pairs. So to handle sequence outputs of arbitrary length, we define an *auto-regressive model*

$$p_{\boldsymbol{\theta}}(\mathbf{y}_{1:L}|\boldsymbol{x}_{1:T}) = \prod_{l=1}^{L} p_{\boldsymbol{\theta}}(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v}), \quad \boldsymbol{v} = enc(\boldsymbol{x}_{1:T}). \tag{27}$$

Here $p_{\boldsymbol{\theta}}(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v})$ is defined by a sequence decoder, e.g. an LSTM:

$$p_{\boldsymbol{\theta}}(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v}) = p_{\boldsymbol{\theta}}(\mathbf{y}_l|\boldsymbol{h}_l^d, \boldsymbol{c}_l^d), \quad \boldsymbol{h}_l^d, \boldsymbol{c}_l^d = LSTM_{\boldsymbol{\theta}}^{dec}(\mathbf{y}_{<l}), \tag{28}$$

and the decoder LSTM has its internal recurrent states $\boldsymbol{h}_0^d$, $\boldsymbol{c}_0^d$ initialised using the input sequence representation $\boldsymbol{v} = enc(\boldsymbol{x}_{1:T})$. The encoder also uses an LSTM, meaning that

$$\boldsymbol{v} = enc(\boldsymbol{x}_{1:T}) = NN_{\boldsymbol{\theta}}(\boldsymbol{h}_T^e, \boldsymbol{c}_T^e), \quad \boldsymbol{h}_T^e, \boldsymbol{c}_T^e = LSTM_{\boldsymbol{\theta}}^{enc}(\boldsymbol{x}_{1:T}). \tag{29}$$

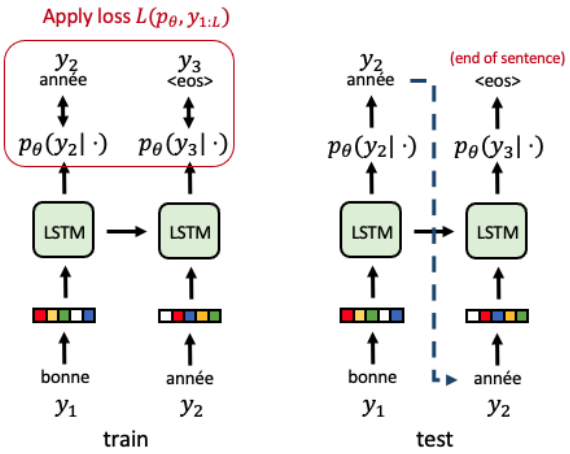


Figure 3: Visualising the forward pass of Seq2Seq model in train (left) and test (right) time.

Regarding the prediction for the first output $\mathbf{y}_1$, either $p_{\boldsymbol{\theta}}(\mathbf{y}_1|\boldsymbol{x}_{1:T})$ can be produced using the last recurrent states of the encoder LSTM (i.e. $p_{\boldsymbol{\theta}}(\mathbf{y}_1|\boldsymbol{x}_{1:T}) = p_{\boldsymbol{\theta}}(\mathbf{y}_1|\boldsymbol{h}^e_T, \boldsymbol{c}^e_T)$), or we can add in a "start of sentence" token as $\mathbf{y}_0$ and compute the probability vector for $\mathbf{y}_1$ using the LSTM decoder: $p_{\boldsymbol{\theta}}(\mathbf{y}_1|\boldsymbol{x}_{1:T}) = p_{\boldsymbol{\theta}}(\mathbf{y}_1|\mathbf{y}_0, \boldsymbol{v})$. This model is named *Sequence-to-sequence* model or *Seq2Seq* model in short, which is proposed by Sutskever et al. [2014].

The decoder LSTM forward pass in training and test times are different, which is visualised in Figure 3. In training, since maximum likelihood training requires evaluating $p_{\boldsymbol{\theta}}(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v})$ for the output sequences $\mathbf{y}_{1:L}$ from the dataset, this means the inputs to the decoder LSTM are words in the data label sequence, and the output of the LSTM is the probability vector for the current word, which is then used to compute the MLE objective (i.e. negative cross-entropy). In test time, however, there is no ground truth output sequence provided, therefore the input to the decoder LSTM at step l is the *predicted* word $\mathbf{y}_{l-1} \sim p_{\boldsymbol{\theta}}(\mathbf{y}_{l-1}|\mathbf{y}_{<l-1}, \boldsymbol{v})$. In practice prediction is done by e.g. beam search rather than naive sequential sampling [Sutskever et al., 2014].

In NLP applications such as machine translation, both $\boldsymbol{x}_{1:T}$ and $\mathbf{y}_{1:L}$ are sequence of words which cannot be directly processed by neural networks. Instead each $\boldsymbol{x}_t$ ($\mathbf{y}_l$) needs to be mapped to a real-value vector before feeding it to the encoder (decoder) LSTM. A naive approach is to use one-hot encoding: assume that the input sentence is in English and we have a sorted English vocabulary of size V, then

$$\boldsymbol{x}_t \to (\underbrace{0, 0, ..., }_{k-1} 1, ..., 0), \quad \text{if } \boldsymbol{x}_t \text{ is the } k\text{th word in the vocabulary.}$$

This is clearly inefficient since English vocabulary has tens of thousands of words. Instead, it is recommended to map the words to their *word embeddings* using word2vec [Mikolov et al., 2013a] or GloVe [Pennington et al., 2014], which has much smaller dimensions. But more importantly, semantics are preserved to some extent in these word embeddings, e.g. it has been shown that vector calculus results like "emb(king) - emb(male) + emb(female) = emb(queen)" hold for word2vec embeddings.

In many NLP applications the decoder output is a probability vector $p_{\boldsymbol{\theta}}(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v})$ which specifies the predictive probability of each of the words in the vocabulary. When the vocabulary is large (which is often so), applying softmax to obtain the probability vector can be very challenging. Interested readers can check e.g. hierarchical softmax [Mikolov et al., 2013a] and negative sampling [Mikolov et al., 2013b] for solutions to mitigate this issue.

1.4 *Generative models for sequences

To generate sequential data such as video, text and audio, one needs to build a generative model $p_{\boldsymbol{\theta}}(x_{1:T})$ and train it with e.g. (approximate) maximum likelihood. In the following we discuss two types of latent variable models that are often used in sequence generation tasks.

Sequence VAE with global latent variables

Similar to VAEs for image generation, one can define a latent variable model with a global latent variable for sequence generation [Fabius and van Amersfoort, 2014; Bowman et al., 2016]:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{1:T}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x}_{1:T}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z}. \tag{30}$$

If variational lower-bound is used for training, then this also requires an approximate posterior $q_{\boldsymbol{\phi}}(z|\boldsymbol{x}_{1:T})$ to be optimised:

$$\boldsymbol{\phi}^*, \boldsymbol{\theta}^* = \arg\max \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta}), \quad \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta}) = \mathbb{E}_{p_{\text{data}}(\boldsymbol{x}_{1:T})}[\underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_{1:T})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_{1:T}|\boldsymbol{z})] - \text{KL}[q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_{1:T})||p(\boldsymbol{z})]}_{:=\mathcal{L}(\boldsymbol{x}_{1:T}, \boldsymbol{\phi}, \boldsymbol{\theta})}]. \tag{31}$$

Now it remains to define the encoder and decoder distributions, such that they can process sequence of any length. This can be achieved using e.g. LSTMs to define an auto-regressive decoder:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{1:T}|z) = \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{x}_{<t}, \boldsymbol{z}), \quad p_{\boldsymbol{\theta}}(\boldsymbol{x}_1|\boldsymbol{x}_{<1}, \boldsymbol{z}) = p_{\boldsymbol{\theta}}(\boldsymbol{x}_1|\boldsymbol{z}). \tag{32}$$

with the distributional parameters of $p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{x}_{<t}, \boldsymbol{z})$ defined by $LSTM_{\boldsymbol{\theta}}(\boldsymbol{x}_{<t})$ which has its recurrent states $\boldsymbol{h}_0, \boldsymbol{c}_0$ initialised using $\boldsymbol{z}$. For the encoder, LSTMs can also be used to process the input:

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_{1:T}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}_{1:T}), diag(\boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\boldsymbol{x}_{1:T}))), \quad \boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}_{1:T}), \log \boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x}_{1:T}) = LSTM_{\boldsymbol{\phi}}(\boldsymbol{x}_{1:T}). \tag{33}$$

State-space models

State-space models assume that for every observation $\boldsymbol{x}_t$ at time t, there is a latent variable $\boldsymbol{z}_t$ that generates it, and the sequence dynamic model is defined in the latent space rather than in the observation space. In detail, a *prior dynamic model* is assumed on the transitions of the latent states $\boldsymbol{z}_t$, often in an auto-regressive way:

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}_{1:T}) = \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{z}_t|\boldsymbol{z}_{<t}), \quad p_{\boldsymbol{\theta}}(\boldsymbol{z}_1|\boldsymbol{z}_{<1}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}_1). \tag{34}$$

The observation $\boldsymbol{x}_t$ at time t is assumed to be conditionally dependent on $\boldsymbol{z}_t$ only, and this conditional distribution is also called the *emission model*:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{z}_{1:T}) = p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{z}_t). \tag{35}$$

Combining both definitions, we have the sequence generative model defined as

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{1:T}) = \int \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{z}_t)p_{\boldsymbol{\theta}}(\boldsymbol{z}_t|\boldsymbol{z}_{<t})d\boldsymbol{z}_{1:T}. \tag{36}$$

A variational lower-bound objective for training this state-space model require an approximate posterior distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T})$:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta}) &= \mathbb{E}_{p_{\text{data}}(\boldsymbol{x}_{1:T})}[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_{1:T}|\boldsymbol{z}_{1:T})] - \text{KL}[q_{\boldsymbol{\phi}}(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T})||p_{\boldsymbol{\theta}}(\boldsymbol{z}_{1:T})]] \\ &= \mathbb{E}_{p_{\text{data}}(\boldsymbol{x}_{1:T})}\left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T})}[\sum_{t=1}^{T} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{z}_t)] - \text{KL}[q_{\boldsymbol{\phi}}(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T})||p_{\boldsymbol{\theta}}(\boldsymbol{z}_{1:T})]\right].\end{aligned} \tag{37}$$

Different from the image generation case, now the prior distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z}_{1:T})$ also has learnable parameters in $\boldsymbol{\theta}$, so in this case it is less appropriate to view this KL term as a "regulariser".

The expanded expression for the variational lower-bound depends on the definition of the encoder distribution $q_{\boldsymbol{\theta}}(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T})$. The simplest solution is to use a factorised approximate posterior

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}_{1:T}|\boldsymbol{x}_{1:T}) = \prod_{t=1}^{T} q(\boldsymbol{z}_t|\boldsymbol{x}_{\leq t}), \tag{38}$$

and the variational lower-bound becomes

$$\mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta}) = \mathbb{E}_{p_{\text{data}}(\boldsymbol{x}_{1:T})}\left[\sum_{t=1}^{T} \mathbb{E}_{q(\boldsymbol{z}_{<t}|\boldsymbol{x}_{<t})}\left[\underbrace{\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}_t|\boldsymbol{x}_{\leq t})}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{z}_t)] - \mathrm{KL}[q_{\boldsymbol{\phi}}(\boldsymbol{z}_t|\boldsymbol{x}_{\leq t})||p_{\boldsymbol{\theta}}(\boldsymbol{z}_t|\boldsymbol{z}_{<t})]}_{:=\mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{z}_{<t})}\right]\right]. \tag{39}$$

We see that the term $\mathcal{L}(\boldsymbol{x}_t, \boldsymbol{\phi}, \boldsymbol{\theta}, \boldsymbol{z}_{<t})$ resembles the VAE objective in the image generation case, except that the prior distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z}_t|\boldsymbol{z}_{<t})$ is conditioned on the previous latent states $\boldsymbol{z}_{<t}$ rather than a standard Gaussian, and the q distribution takes $\boldsymbol{x}_{\leq t}$ as the input rather than a single frame $\boldsymbol{x}_t$.

Neural networks can be used to construct the conditional distributions in the following way. The distributional parameters (e.g. mean and variance) of the emission model $p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{z}_t)$ can be defined by a neural network transformation of $\boldsymbol{z}_t$, similar to deep generative models for images. The prior dynamic model $p_{\boldsymbol{\theta}}(\boldsymbol{z}_t|\boldsymbol{z}_{<t})$ can be defined as (e.g. with an LSTM)

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}_t|\boldsymbol{z}_{<t}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}_t|\boldsymbol{h}_t^p, \boldsymbol{c}_t^p), \quad \boldsymbol{h}_t^p, \boldsymbol{c}_t^p = LSTM_{\boldsymbol{\theta}}(\boldsymbol{z}_{<t}). \tag{40}$$

This means the previous latent states $\boldsymbol{z}_{<t}$ are summarised by the LSTM internal recurrent states $\boldsymbol{h}_t^p$ and $\boldsymbol{c}_t^p$, which are then transformed into the distributional parameters of $p_{\boldsymbol{\theta}}(\boldsymbol{z}_t|\boldsymbol{z}_{<t})$. For the factorised encoder distribution, it can also be defined using an LSTM:

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}_t|\boldsymbol{x}_{\leq t}) = q_{\boldsymbol{\phi}}(\boldsymbol{z}_t|\boldsymbol{h}_t^q, \boldsymbol{c}_t^q), \quad \boldsymbol{h}_t^q, \boldsymbol{c}_t^q = LSTM_{\boldsymbol{\phi}}(\boldsymbol{x}_{\leq t}). \tag{41}$$

References

Arjovsky, M., Shah, A., and Bengio, Y. (2016). Unitary evolution recurrent neural networks. In *International Conference on Machine Learning*, pages 1120–1128.

Bowman, S. R., Vilnis, L., Vinyals, O., Dai, A., Jozefowicz, R., and Bengio, S. (2016). Generating sentences from a continuous space. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pages 10–21. Association for Computational Linguistics.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734. Association for Computational Linguistics.

Fabius, O. and van Amersfoort, J. R. (2014). Variational recurrent auto-encoders. *arXiv preprint arXiv:1412.6581*.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Le, Q. V., Jaitly, N., and Hinton, G. E. (2015). A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26:3111–3119.

Pennington, J., Socher, R., and Manning, C. D. (2014). GloVe: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Saxe, A. M., McClelland, J. L., and Ganguli, S. (2014). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *International Conference on Learning Representations*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27:3104–3112.