Department of Computing

Imperial College of Science, Technology and Medicine

example

example description

Author: Anton Zhitomirsky

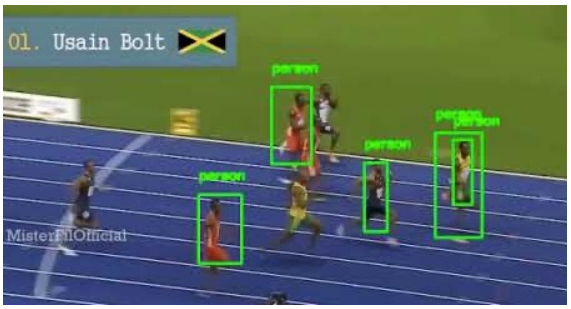
Contents

1 RNN Basics

1.1 Sequential Data

Object tracking in videos:

- Data sequence: $(x_1, \dots, x_T)$
  - x_t: video frame at time t
- Label sequence: $(y_1, \dots, y_T)$
  - y_t: object identifier, bounding box coordinates, … at time t
- Goal: learn a mapping
  $(x_1, \dots, x_T) \rightarrow (y_1, \dots, y_T)$

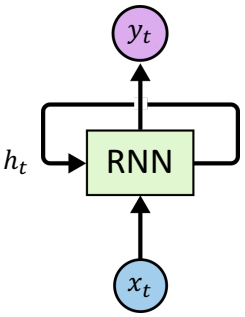


This kind of applciation works better than a supervised learning context (With labels x and y) becuase this binding box information has high sequential time dependencies

1.2 Advantages

- Model dependencies within the sequence
- Can handle inputs/outputs of different lengths

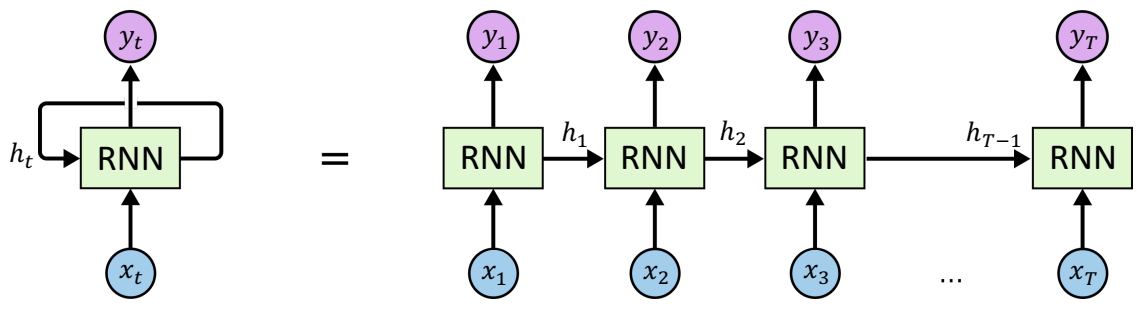
1.3 Simple Architecture



$$h_t = \phi_h(W_h h_{t-1} + W_x x_t + b_h)$$
$$y_t = \phi_y(W_y h_t + b_y)$$

ϕ_h: activation function for recurrent state
ϕ_y: activation function for output

A rucurrent state is maintained thorugh-out the network to maintain information that the network has seen in the past.



These architectures can be unrolled through time

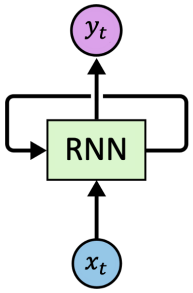
1.3.1 Mathematical Form

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.

1.4 Training RNNs

1.4.1 Back-propagation through time (BPTT)

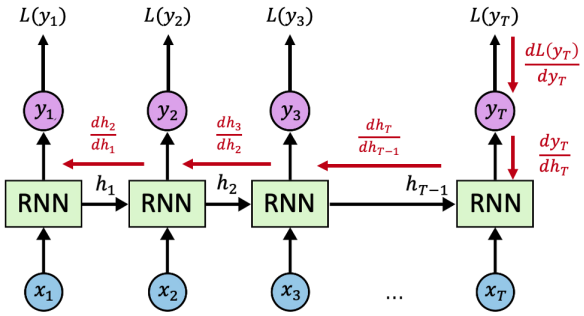


Figure 1: Visualising Back-propagation through time (BPTT) without truncation. The black arrows show forward pass computations, while the red arrows show the gradient back-propagation in order to compute $\nabla_{W_h} \mathcal{L}(y_t)$.

We want to minimise the loss of $\mathcal{L}(\boldsymbol{\theta}) = L_{total}(\boldsymbol{\theta}) = \sum_{t=1}^{T} L(y_t)$ using gradient descent. This means we need to compute the gradient of the loss with respect to the parameters of the model.

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

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

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

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

1.4.2 Gradient vanishing/explosion issues

Depending on the weight matrix and the activation function the product of the partial graident can vanish or explode as the number of time steps increases.

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 $l = \tau, ..., t-1$.

Here, the circled dot operator is typically used to represent element-wise multiplication

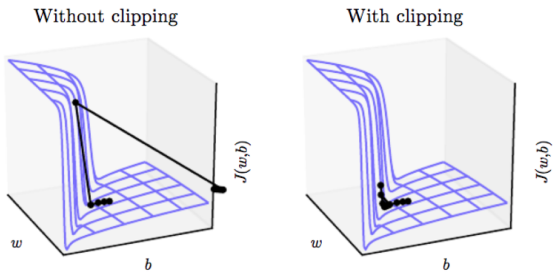
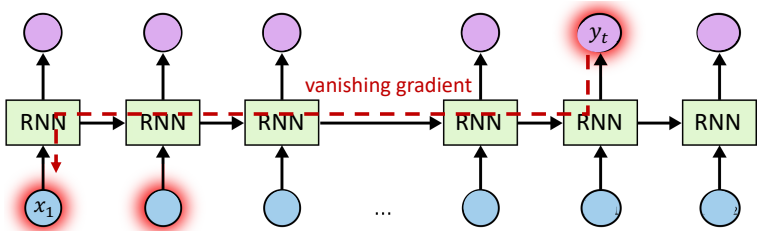


Figure 2: Visualising the gradient step with/out gradient clipping. Source: Goodfellow et al. [2016].

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.

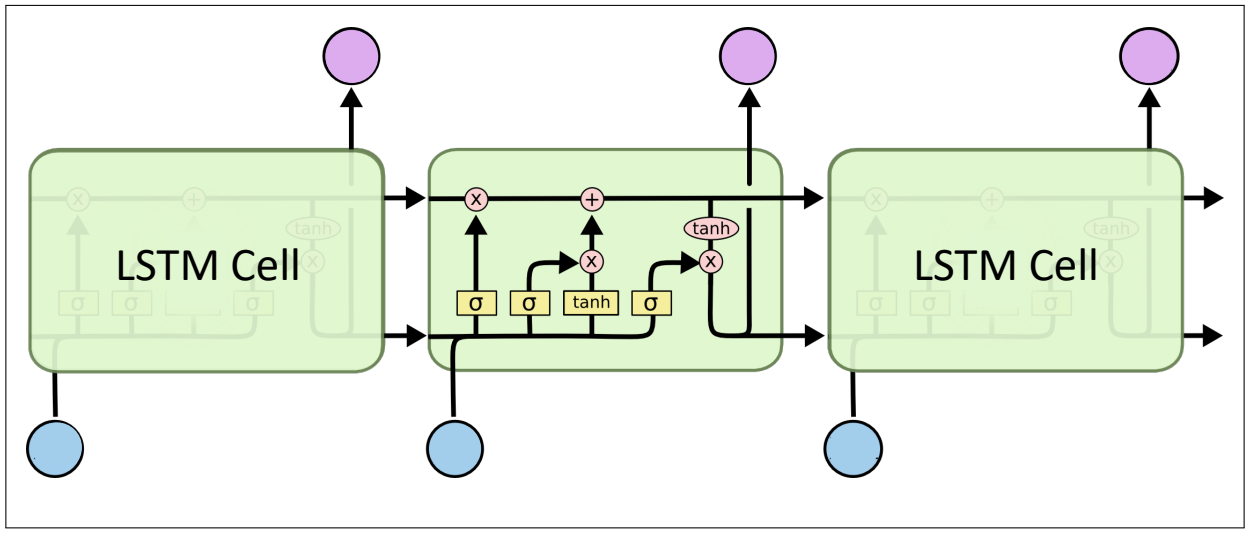
Therefore, long term dependencies are getting harder to learn. With vanishing gradient, Since the gradient for W_h is the sum of back-prop gradient with different window length, this gradient will be dominated especially when $t >> \tau$; the learning signal will be dominated by the short term dependencies.



Tricks for how to fix the gradient vanishing or explosion issue are detailed in the Appendix A.1.

2 Long Short-Term Memory (LSTM)

2.1 Forward Pass



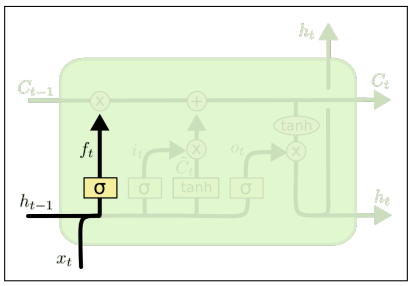
Addresses the problem of gradient descent problem. They perform better at learning long-term dependencies than simple RNNs

The Long Short-term Memory was proposed with the motivation of adderssing the graident 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 funciton):

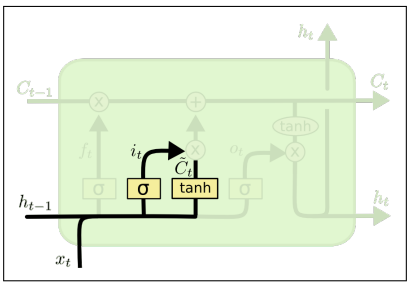
$\boldsymbol{f}_t$:forget gate	$\boldsymbol{f}_t = \sigma(W_f \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f)$	(7)
$\boldsymbol{i}_t$:input gate	$\boldsymbol{i}_t = \sigma(W_i \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i)$	(8)
$\boldsymbol{o}_t$:output gate	$\boldsymbol{o}_t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

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

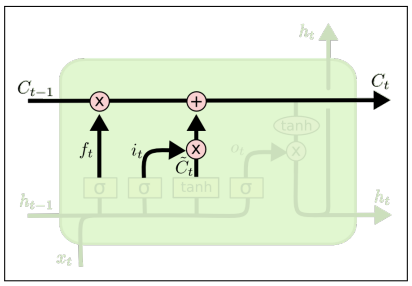
Here, commas (e.g.) $[h_{t-1}, x_t]$ mean concatenations



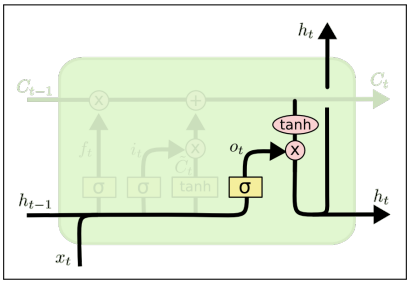
(a) Forget gate: used to control the cell state update. It depends on the previous hidden state and current input. This is a one layer network with sigmoid activation function. Equation (7) above.



(b) The second step is to propose a candidate update value $\tilde{c}_t$ to the cell state. In this case an input gate is also computed to form the cell state value as well. Equations (8) and (10) above.



(c) Cell state update. f_t controls the maintenance of the previous cell states. Similarly, the input gate controlls whether the candidate update will be accepted or not. Equation (11) above.



(d) Finally, compute the updates for the hidden states h_t. Here h_t is used to output a cell state c_t. But here, an output gate is also being used. THis means sometimes the hidden state will be 0 given the output gate values. Equations (9) and (12) above.

The Prediction of y_t can proceed in a simlar way as in simple RNNS: $y_t = \Phi_y(W_y h_t + b_y)$ by transforming the hidden state h_t with one neural network layer.

2.2 Backwards Pass

In this case, the recurrent state we care about is the cell state c_t. Back-prop thorugh time applies to computing the graident with respect to W_c the weight matrix for computing the candidate cell state.

2.2.1 *Gradient computation

How to derive $\frac{dc_t}{dc_{t-1}}$: Notice c_t depends on c_{t-1} in 4 paths:

direct dependence

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

Depends on $h_{t-1} = o_{t-1} \odot c_{t-1}$ (indirect dependence)

"To give an idea of how this gradient is derrived, we can look at the recurrent update equation for the cell state c_t. Here, c_t depends on c_{t-1} as shown in the red box. However, there are also indirect paths via $f_t \wedge i_t \wedge c_t$ because they all depend of h_{t-1} which depends on c_{t-1}. This gives us an idea of how to compute the total gradient."

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.

2.3 How LSTMs alleviate training difficulties of RNNs

BPTT in LSTMs:

Requires computing $\prod_{l=1}^{t-1} \frac{dc_{l+1}}{dc_l} = \prod_{l=1}^{t-1}(f_{l+1} + o_l \odot \frac{dtanh(c_l)}{dc_l} \odot \frac{dc_{l+1}}{dh_l})$

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \odot \tanh(c_t)$$
$$y_t = \phi_y(W_y h_t + b_y)$$

Alleviating gradient explosion:

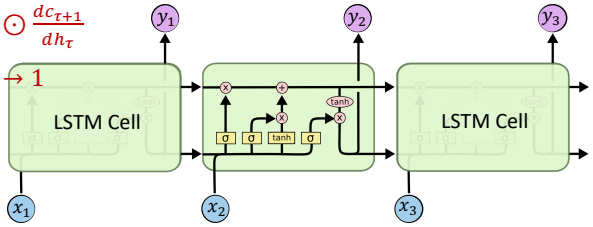
The gradient contains terms proportional to $f_{i+1} \prod_{l=1}^{i-1} o_l \odot \frac{dc_{l+1}}{dh_l}$

≈ 0 when $f_{i+1} \approx 0$

Alleviating gradient vanishing:

The gradient contains terms proportional to $\prod_{l=\tau+1}^{i} f_l \odot o_\tau \odot \frac{dc_{\tau+1}}{dh_\tau}$

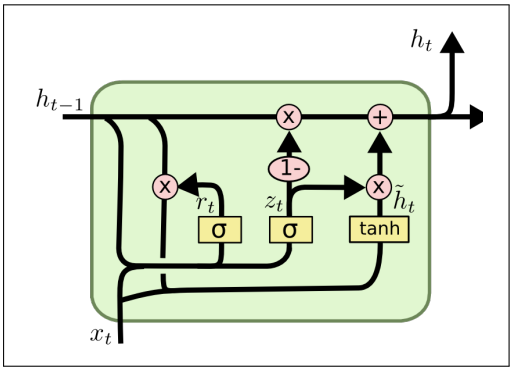
$\approx o_\tau \odot \frac{dc_{\tau+1}}{dh_\tau}$ when $f_l \to 1$ for $l = \tau + 1, \ldots, i$



Hochreiter and Schmidhuber (1997). Long Short-Term Memory. Neuro Computation.
Figure adapted from https://colah.github.io/posts/2015-08-Understanding-LSTMs/

The usage of forget gates is useful for gradient explosion problems, becuase the network can learn a forget gate value of 0. Similarly, the product of gradients after expansion contains terms like product of f_t terms times other terms. If the forget gates are open, then this term becomes the gradient of $c_{\tau+1}$ with resepect ot h_τ. If τ is small and i is big, then errors can also be back-propogated and doesn't vanish. This is useful for learning long-term dependencies.

2.4 Alternatives — Gated Recurrent Unit (GRU)



Simplified versions of the LSTM have been proposed. Compared to LSTMs, GRU only uses switching gates z_t and reset gates r_t. you can still see that GRU still implements gating mechanisms to control the updates fo rthe recurrent states.

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.

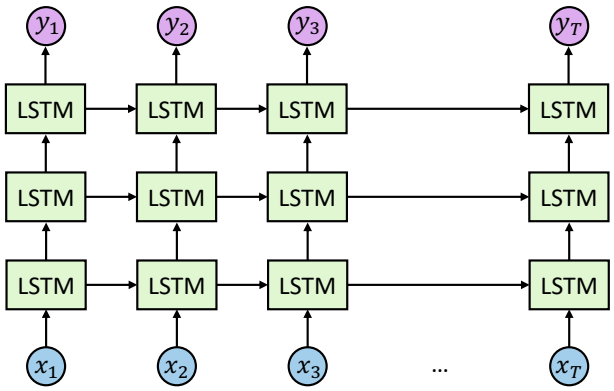
2.4.1 LSTM vs GRU

- Other gated RNN variants exists, but LSTM and GRU are the most widely-used
- GRU is quicker to compute and has fewer parameters
- No conclusive evidence for LSTM ¿ GRU or vice versa
- LSTM is a good default choice (especially if your data has particularly long dependencies, or you have lots of training data)
- Switch to GRU if you want more efficient compute & less overfitting

2.5 Stacking LSTMs

Stacking multiple LSTM layers:

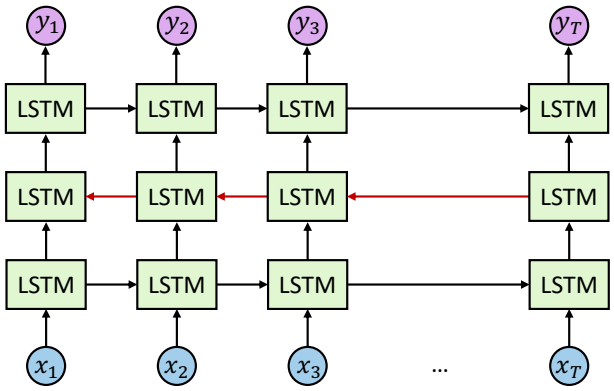
- Hidden states of the previous LSTM later as inputs to the next LSTM layer;
- No need to wait for previous LSTM layer to finish forward pass;



2.6 Bidirectional LSTMs

Bidirectional LSTM:

- Stacking some LSTM layers;
- For some LSTM layers, the forward pass is reversed from time t $= T$ to t $= 1$;
- If two consecutive LSTM layers are of reversed time ordering, then the top layer needs to wait for the bottom one to finish forward pass.



Has shown to improve performance

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

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

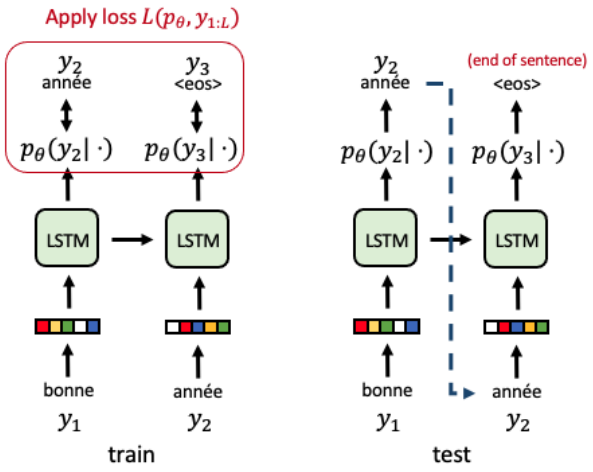


Figure 3: Visualising the forward pass of Seq2Seq model in train (left) and test (right) time.

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.

A RNN notes

A.1 Tricks to fix the gradient vanishing/explosion issue

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.

A.2 Long Short-Term Memory (LSTM)

A.3 *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.

A.3.1 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:

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

A.3.2 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:

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

B Attention Notes

In sequence-to-sequence models, the goal is to learn the following model

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

with $p_{\boldsymbol{\theta}}(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v})$ and $enc(\boldsymbol{x}_{1:T})$ defined by LSTMs. Although LSTMs suffers less from the gradient vanishing/explosion problems, it can still be challenging for LSTMs to learn long-term dependencies. A practical trick to boost the performance is to reverse the order of the input, i.e. using $\boldsymbol{x}_{T:1}$ instead of $\boldsymbol{x}_{1:T}$ as the input to the encoder LSTM [Sutskever et al., 2014]. Still the difficulty of learning long-term dependencies remains unsolved even with this trick. Furthermore, it is possible that different $\mathbf{y}_l$ words require different information extracted from $\boldsymbol{x}_{1:T}$, so a shared global representation $\boldsymbol{v}$ for the input sequence might be sub-optimal.

B.1 *Attention in Bahdanau et al.

Bahdanau et al. [2015] proposed an attention-based approach (the authors called it as "alignment") to address the above issues. Recall that at time t the encoder LSTM updates its internal recurrent states $\boldsymbol{c}_t^e$ and $\boldsymbol{h}_t^e$ using the current input $\boldsymbol{x}_t$:

$$\boldsymbol{h}_t^e, \boldsymbol{c}_t^e = LSTM_{\boldsymbol{\theta}}^{enc}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}^e, \boldsymbol{c}_{t-1}^e). \tag{2}$$

Similarly for the decoder LSTM, we need to maintain its internal recurrent states $\boldsymbol{c}_l^d$ and $\boldsymbol{h}_l^d$:

$$p_{\boldsymbol{\theta}}(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v}) = p_{\boldsymbol{\theta}}(\mathbf{y}_l|\boldsymbol{h}_l^d, \boldsymbol{c}_l^d), \quad \boldsymbol{h}_l^d, \boldsymbol{c}_l^d = LSTM_{\boldsymbol{\theta}}^{dec}(\mathbf{y}_{l-1}, \boldsymbol{h}_{l-1}^d, \boldsymbol{c}_{l-1}^d). \tag{3}$$

In the original sequence-to-sequence model, the global representation $\boldsymbol{v}$ is obtained by transforming the last hidden state $\boldsymbol{h}_T^e$ of the encoder LSTM, and it is used to initialise $\boldsymbol{h}_0^d$ and $\boldsymbol{c}_0^d$. Instead, Bahdanau et al. [2015] proposes using different representations of the input sequence $\boldsymbol{x}_{1:T}$ at different steps for predicting $\mathbf{y}_l$, i.e.

$$\begin{aligned} p_{\boldsymbol{\theta}}(\mathbf{y}_{1:L}|\boldsymbol{x}_{1:T}) &= \prod_{l=1}^{L} p_{\boldsymbol{\theta}}(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v}_l), \\ p_{\boldsymbol{\theta}}(\mathbf{y}_l|\mathbf{y}_{<l}, \boldsymbol{v}_l) = p_{\boldsymbol{\theta}}(\mathbf{y}_l|\boldsymbol{h}_l^d, \boldsymbol{c}_l^d), &\quad \boldsymbol{h}_l^d, \boldsymbol{c}_l^d = LSTM_{\boldsymbol{\theta}}^{dec}([\mathbf{y}_{l-1}, \boldsymbol{v}_l], \boldsymbol{h}_{l-1}^d, \boldsymbol{c}_{l-1}^d). \end{aligned} \tag{4}$$

where the representation $\boldsymbol{v}_l$ is obtained as follows:

$$\boldsymbol{f}_t = T_{\boldsymbol{\theta}}(\boldsymbol{h}_t^e), \quad \text{(feature output of the encoder at time } t\text{)} \tag{5}$$

$$e_{lt} = a(\boldsymbol{h}_{l-1}^d, \boldsymbol{f}_t), \quad \text{(compute similarity/alignment score)} \tag{6}$$

$$\alpha_l = softmax(\boldsymbol{e}_l), \quad \boldsymbol{e}_l = (e_{l1}, ..., e_{lT}), \tag{7}$$

$$\boldsymbol{v}_l = \sum_{t=1}^{T} \alpha_{lt} \boldsymbol{f}_t. \quad \text{(weighted aggregation of input features)} \tag{8}$$

The key idea of using $\boldsymbol{v}_l$ as a weighted average of individual features $\boldsymbol{f}_t$ is to allow the decoder LSTM to directly access the representation for each input $\boldsymbol{x}_t$, therefore the issue of lacking long-term dependencies between $\mathbf{y}_l$ and $\boldsymbol{x}_t$ is addressed. This is in contrast with the global representation vector $\boldsymbol{v}$ in the original Seq2Seq model: since $\boldsymbol{v}$ is computed using the last recurrent state of the encoder LSTM, it is questionable whether $\boldsymbol{v}$ can capture long-term dependencies within $\boldsymbol{x}_{1:T}$. Another notable difference is that in Bahdanau et al. [2015] the features $\boldsymbol{v}_l$ are used as the input to the decoder LSTM (together with $\mathbf{y}_{l-1}$), while in the original Seq2Seq model the global representation $\boldsymbol{v}$ is used to initialise the decoder LSTM's recurrent states.

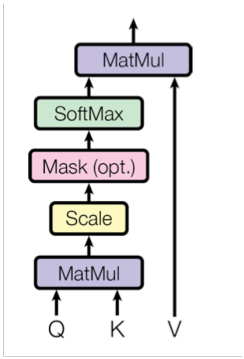
The alignment score $e_{lt} = a(\boldsymbol{h}_{l-1}^d, \boldsymbol{f}_t)$ is computed between $\boldsymbol{h}_{l-1}^d$ (which summarises $\mathbf{y}_{<l}$, required for the auto-regressive model to predict $\mathbf{y}_l$) and $\boldsymbol{f}_t$ (a feature representation for $\boldsymbol{x}_t$ but is also dependant on $\boldsymbol{x}_{<t}$). These scores are then passed through a softmax layer to obtain the *attention weight* α_l, with larger attention weight value α_{lt} the final representation $\boldsymbol{v}_l$ will focus more on the input feature $\mathbf{f}_t$ for $\boldsymbol{x}_t$.

B.2 Attention in transformers

B.2.1 Single-head attention

The *scaled dot product attention* method is introduced by Vaswani et al. [2017]. Intuitively it can be understood from an information retrieval point of view. Queries are submitted to the system which are represented by the *query vectors* $\boldsymbol{q}_i \in \mathbb{R}^{d_q}$, and the system will first check the matching/alignment/"similarity" between the query and the keys (which are represented by *key vectors* $\boldsymbol{k}_j \in \mathbb{R}^{d_q}$), then return the retrieved values to the user. Each key vector $\boldsymbol{k}_j$ is associated with a *value vector* $\boldsymbol{v}_j \in \mathbb{R}^{d_v}$, so that if $\boldsymbol{q}_i$ and $\boldsymbol{k}_j$ are aligned, the value vector $\boldsymbol{v}_j$ will be returned in some form.

In detail, the mathematical form for single-head attention is the following:



$$Attention(Q, K, V; a) = a\left(\frac{QK^\top}{\sqrt{d_q}}\right)V \tag{9}$$

$$Q = (\boldsymbol{q}_1, ..., \boldsymbol{q}_N)^\top \in \mathbb{R}^{N \times d_q} \tag{10}$$

$$K = (\boldsymbol{k}_1, ..., \boldsymbol{k}_M)^\top \in \mathbb{R}^{M \times d_q} \tag{11}$$

$$V = (\boldsymbol{v}_1, ..., \boldsymbol{v}_M)^\top \in \mathbb{R}^{M \times d_v} \tag{12}$$

There are two important ingredients in the scaled dot product attention process. First, an *attention matrix* $A = a\left(\frac{QK^\top}{\sqrt{d_q}}\right)$ is computed to indicate the alignment of each query vector $\boldsymbol{q}_i$ to the key vectors $\boldsymbol{k}_j$. Then given the attention matrix A, the attention output for each query vector $\boldsymbol{q}_i$ is the weighted sum of the value vectors $\sum_{j=1}^{M} A_{ij}\boldsymbol{v}_j$, which is similar to the final output of the attention method by Bahdanau et al. Here $a(\cdot)$ is an activation function applied row-wise, and in *soft attention*, $a(\cdot)$ is the softmax function, i.e. $A_{ij} = softmax((\langle \boldsymbol{q}_i, \boldsymbol{k}_1\rangle, ..., \langle \boldsymbol{q}_i, \boldsymbol{k}_M\rangle)/\sqrt{d_q})$. This is in contrast with *hard attention* where $a(\cdot)$ returns a one-hot vector for each row with the $j^* = \arg\max_j \langle \boldsymbol{q}_i, \boldsymbol{k}_j\rangle$ element equals to 1. Hard attention can be interpreted exactly as an information retrieval system since it returns the corresponding value for the best key match to the query.

When the dimensionality of the query/key vector d_q is large, the dot product $\langle \boldsymbol{q}_i, \boldsymbol{k}_j\rangle$ can be large as well. This is likely to increase the gap between the dot product values so that the softmax output could be dominated by a single entry (i.e. close to hard attention). Normalisation of the logits $\langle \boldsymbol{q}_i, \boldsymbol{k}_j\rangle$ could be useful to address this issue. For the specific choice of $\sqrt{d_q}$, this comes from the assumption that the elements in $\boldsymbol{q}_i$ and $\boldsymbol{k}_j$ are independently distributed with variance 1. If so then the variance of $\langle \boldsymbol{q}_i, \boldsymbol{k}_j\rangle$ is d_q, so by normalising the dot product with $\sqrt{d_q}$, the logit will have its variance equal to 1.

In some cases, *masking* is applied to the attention procedure. For example, a typical masking strategy will define a mask matrix $M \in \{0, 1\}^{N \times M}$, so that if $M_{ij} = 0$, then the corresponding attention weight $A_{ij} = 0$, so that $\boldsymbol{v}_j$ does not contribute to the final output for query $\boldsymbol{q}_i$.

Self-attention is also in wide usage which sets $K = Q$.

B.2.2 Complexity figures

The time complexity for scaled dot-product attention is $\mathcal{O}(MNd_q + MNd_v)$. These include the dot product $QK^\top$ which has $\mathcal{O}(MNd_q)$ run-time cost, the computation of A matrix given the dot product which has $\mathcal{O}(MN)$ cost, and the dot product AV which has $\mathcal{O}(MNd_v)$ cost.

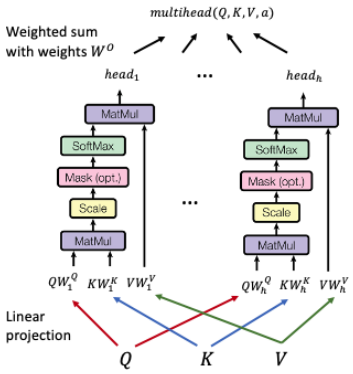
The space complexity for the scaled dot-product is $\mathcal{O}(MN+Nd_v)$ since back-propagation requires storing some intermediate results. These include the dot product $QK^\top$ which has $\mathcal{O}(MN)$ memory cost and the final output AV which has $\mathcal{O}(Nd_v)$ cost.

For self-attention, since $Q = K$, then the time complexity and space complexity figures are $\mathcal{O}(N^2 d_q + N^2 d_v)$ and $\mathcal{O}(N^2 + Nd_v)$, respectively.

B.2.3 Multi-head attnetion

Multi-head attention repeats the single-head attention process with different "views". Intuitively, consider information retrieval with the input query representing "cat". Then depending on the definition of similarity, it can match to key "tiger" or key "pet" and so on.

Multi-head attention allows the definition of multiple alignment processes, by projecting the inputs into different sub-spaces then performing dot product attention in such sub-spaces. The outputs of each attention head are concatenated and projected to produce the final output.



$$
\begin{aligned}
Multihead(Q, K, V; a) &= concat(head_1, ..., head_h)W^O, \\
head_i(Q, K, V; a) &= Attention(QW_i^Q, KW_i^K, VW_i^V; a).
\end{aligned} \tag{13}
$$

It is clear that the time and space complexity figures of multi-head attention are h times of those for a single head plus the extra costs for linear projections. Assume the projection matrices have sizes $W_i^Q \in \mathbb{R}^{d_q \times \tilde{d}_q}$, $W_i^K \in \mathbb{R}^{d_q \times \tilde{d}_q}$, $W_i^V \in \mathbb{R}^{d_v \times \tilde{d}_v}$ and $W^O \in \mathbb{R}^{h\tilde{d}_v \times d_{out}}$. This means

$$
\text{time complexity:} \quad \mathcal{O}(\underbrace{h(MN\tilde{d}_q + MN\tilde{d}_v)}_{\text{attention heads}} + \underbrace{h(\tilde{d}_q d_q(M+N) + \tilde{d}_v d_v M)}_{\text{input projections}} + \underbrace{Nh\tilde{d}_v d_{out}}_{\text{combined outputs}}),
$$

$$
\text{space complexity:} \quad \mathcal{O}(\underbrace{h(MN + N\tilde{d}_v)}_{\text{attention heads}} + \underbrace{h(\tilde{d}_q(M+N) + \tilde{d}_v M)}_{\text{input projections}} + \underbrace{Nd_{out}}_{\text{combined outputs}}).
$$

To keep the costs close to performing single-head attention in the original space, often the $\tilde{d}_q$, $\tilde{d}_v$ dimensions are set to be $\tilde{d}_q = \lfloor \frac{d_q}{h} \rfloor$ and $\tilde{d}_v = \lfloor \frac{d_v}{h} \rfloor$, respectively.

B.3 Ingredients in transformers

Transformer (proposed in Vaswani et al. [2017]) is an encoder-decoder type of architecture, which is visualised in Figure 1. We discuss some of the key ingredients as follows.

B.3.1 Position encoding

From the equations of scaled dot-product attention, we see that attention is equivariant to row permutations in the query matrix Q. To see this, notice that permuting rows in a matrix is equivalent to left multiplying a permutation matrix P to Q. Since the non-linearity $a(\cdot)$ is applied row-wise, this leads to the permutation equivariance result:

$$Attention(PQ, K, V; a) = a\left(\frac{PQK^\top}{\sqrt{d_q}}\right)V = Pa\left(\frac{QK^\top}{\sqrt{d_q}}\right)V = PAttention(Q, K, V; a). \quad (14)$$

Therefore, for an input sequence of queries $\boldsymbol{q}_1, ..., \boldsymbol{q}_N$ in which the ordering information (i.e. the subscript n) is useful for the task (e.g. time series regression), these ordering information needs to

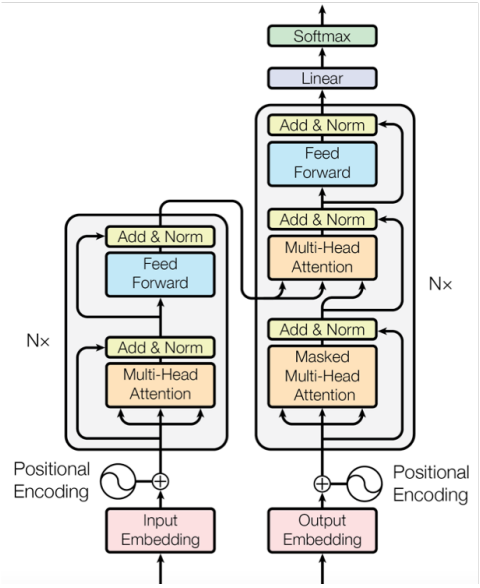


Figure 1: The Transformer architecture in Vaswani et al. [2017].

be added in some form to the attention inputs. This issue is addressed by position encoding, which constructs the query input $\tilde{Q}$ to the attention module as:

$$\tilde{Q} = (\tilde{\boldsymbol{q}}_1, ..., \tilde{\boldsymbol{q}}_N)^\top, \quad \tilde{q}_n = f(\boldsymbol{q}_n, PE(n)), \tag{15}$$

where the f transformation is often set to be simple operations such as summation and concatenation, etc. $PE(n)$ is called the *position encoding* of input index n, which can either be learned (e.g. having a set of learnable parameters $\{\boldsymbol{p}_1, ..., \boldsymbol{p}_n\}$ or using an NN-parameterised function), or computed using a pre-defined function. Learned embedding of the form $\{\boldsymbol{p}_1, ..., \boldsymbol{p}_n\}$ is well suited if the maximum value of the index is known. Otherwise, in the case where unseen index can occur in test time, a pre-defined function might be preferred.

A popular choice of such pre-defined functions is the *sinusoid embedding* [Vaswani et al., 2017]:

$$\begin{aligned} PE(n) &= (PE(n,0), ..., PE(n,2I)), \\ PE(n,2i) &= sin(n/10000^{2i/d_q}), \\ PE(n,2i+1) &= cos(n/10000^{2i/d_q}). \end{aligned} \tag{16}$$

We see that sinusoid embeddings use multiple periodic functions to embed the input index n, where the frequency of each sine/cosine wave is determined by i. The authors of Vaswani et al. [2017] did not give a formal justification of this approach and instead stated that "we hypothesized it would allow the model to easily learn to attend by relative positions". My personal hypothesis is that, in the case of using concatenation to construct the new query (i.e. $\tilde{q}_n = f(\boldsymbol{q}_n, PE(n)) = [\boldsymbol{q}_n, PE(n)]$) which is then multiplied by learnable weight matrices, the sinusoid embedding allows the network to learn a very flexible position encoding function as a combination of many sine/cosine waves of different frequencies (think about the Fourier series of a continuous function).

B.3.2 Layer normalization

Layer normalisation [Ba et al., 2016] is a normalisation technique used to stabilise neural network training. Assuming a feed-forward layer with the pre-activation computed as $\boldsymbol{a} = (a_1, ..., a_H) = W\boldsymbol{x} + \boldsymbol{b}$, layer normalisation can be applied to the pre-activation vector before feeding to the non-

linearity:

$$\boldsymbol{a} \leftarrow \frac{\boldsymbol{a}-\mu}{\sigma}, \quad \mu = \frac{1}{H}\sum_{h=1}^{H} a_h, \quad \sigma = \sqrt{\frac{1}{H}\sum_{h=1}^{H}(a_h-\mu)^2}. \tag{17}$$

In transformers, layer normalisation is applied together with a residual link: $Add\&Norm(x) = LayerNorm(x + Sublayer(x))$, where $Sublayer(\cdot)$ can either be a multi-head attention block or a point-wise feed-forward network.

B.3.3 Point-wise feed-forward network

The point-wise feed-forward network is used as the "feed forward" layer in Transformer's architecture (see Figure 1). The multi-head attention (after "$Add\&Norm$") returns a matrix of size $N \times d_{out}$, representing the attention results for N queries. These output can be processed "point-wise" (i.e. row-wise) with a feed-forward neural network, which effectively treat the rows in the attention outputs as "datapoint" inputs for the next layer.