CS-E4890: Deep Learning
Recurrent neural networks

Alexander Ilin

Sequence modeling

- Previously: inputs and outputs are vectors of fixed sizes
  - MNIST: inputs: 28x28 images, outputs: 10 classes
- In some tasks, inputs can be sequences, each sequence can have a different number of elements:

$$\left(x_1^{(1)}, x_2^{(1)}, x_3^{(1)}\right) \rightarrow y^{(1)}$$
$$\left(x_1^{(2)}, x_2^{(2)}, x_3^{(2)}, x_4^{(2)}\right) \rightarrow y^{(2)}$$

- Example: sentiment analysis

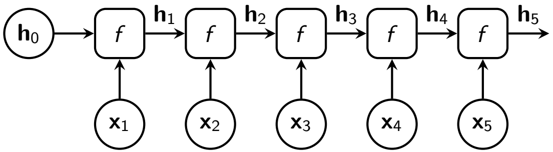
Dear #XYZ there is no network in my area and internet service is pathetic from past one week. Kindly help me out.	negative review
Although the value added services being provided are great but the prices are high.	mixed review
Great work done #XYZ Problem resolved by customer care in just one day.	postive review

How can we process a sequence?

- Example: count the number of zeros in an input sequence $(x_1, x_2, x_3, \dots x_T)$

```
h = 0
for x in input_sequence:
    if x == 0:
        h = h + 1
```

- How to implement this in a computational graph:



```
def f(x, h):
    return h + (x == 0)
```

Generic processor

- How can we learn to process sequences from training examples?
- Example: sentiment analysis

Dear #XYZ there is no network in my area and internet service is pathetic from past one week. Kindly help me out.	negative review
Although the value added services being provided are great but the prices are high.	mixed review
Great work done #XYZ Problem resolved by customer care in just one day.	postive review

- To build a generic processor, we can use the same computational graph with a learnable **f**:

$$\begin{tikzcd}
\mathbf{h}_0 \arrow[r] & f \arrow[r, "\mathbf{h}_1"] & f \arrow[r, "\mathbf{h}_2"] & f \arrow[r, "\mathbf{h}_3"] & f \arrow[r, "\mathbf{h}_4"] & f \arrow[r, "\mathbf{h}_5"] & {} \\
 & \mathbf{x}_1 \arrow[u] & \mathbf{x}_2 \arrow[u] & \mathbf{x}_3 \arrow[u] & \mathbf{x}_4 \arrow[u] & \mathbf{x}_5 \arrow[u] &
\end{tikzcd}$$

Vanilla recurrent neural network (RNN)

- We can use the same building block as in the standard multilayer perceptron (MLP):

$$f(\mathbf{x}, \mathbf{h}) = \tanh(\mathbf{W}\mathbf{h} + \mathbf{U}\mathbf{x} + \mathbf{b})$$

$$\begin{tikzcd}
\mathbf{h}_0 \arrow[r] & f \arrow[r, "\mathbf{h}_1"] & f \arrow[r, "\mathbf{h}_2"] & f \arrow[r, "\mathbf{h}_3"] & f \arrow[r, "\mathbf{h}_4"] & f \arrow[r, "\mathbf{h}_5"] & \text{softmax} \\
 & \mathbf{x}_1 \arrow[u] & \mathbf{x}_2 \arrow[u] & \mathbf{x}_3 \arrow[u] & \mathbf{x}_4 \arrow[u] & \mathbf{x}_5 \arrow[u] &
\end{tikzcd}$$

- Recurrence, thus recurrent neural network (RNN).
- $\mathbf{h}$ is often called *hidden state*.

RNN vs feedforward network

Computational graph of a feedforward network:

$$\begin{tikzcd}
& \mathbf{W}_1 \arrow[d] & \mathbf{W}_2 \arrow[d] & \\
\boxed{\mathbf{x}} \arrow[r] & \boxed{f_1} \arrow[r, "\mathbf{h}_1"] & \boxed{f_2} \arrow[r, "\mathbf{h}_2"] & \boxed{\mathcal{L}} \\
& & & \boxed{\mathbf{y}} \arrow[u]
\end{tikzcd}$$

Computational graph of an RNN:

$$\begin{tikzcd}
& & \boldsymbol{\theta} \arrow[dl] \arrow[dr] & & \\
\boxed{\mathbf{h}_0} \arrow[r] & \boxed{f} \arrow[rr, "\mathbf{h}_1"] & & \boxed{f} \arrow[r, "\mathbf{h}_2"] & \boxed{\mathcal{L}} \\
& \boxed{\mathbf{x}_1} \arrow[u] & & \boxed{\mathbf{x}_2} \arrow[u] & \boxed{\mathbf{y}} \arrow[u]
\end{tikzcd}$$

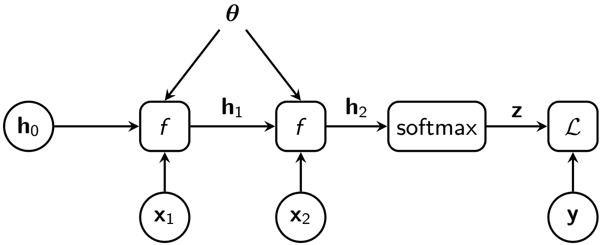
- External inputs are added at every step.
- Same parameters are used in every layer.

- Where did we previously use parameters in multiple places of a computational graph?

Training recurrent neural networks

Training an RNN

- Just like for a feedforward network, the parameters of an RNN can be found by (stochastic) gradient descent.



- For example, we can tune parameters $\boldsymbol{\theta}$ by minimizing the cost function

$$\boldsymbol{\theta}_* = \arg\min_{\boldsymbol{\theta}} -\frac{1}{N}\sum_{n=1}^{N}\sum_{j=1}^{K} y_j^{(n)} \log z_j^{(n)}$$

- We need to compute gradients wrt parameters $\boldsymbol{\theta}$.

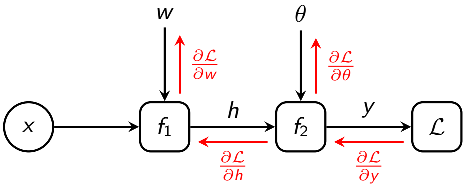
Backpropagation

- Recall backpropagation in a multi-layer model that operates with scalars:

$$\mathcal{L} = \mathcal{L}(y), \quad y = f_2(h, \theta), \quad h = f_1(x, w)$$

- We can compute the derivatives wrt the model parameters θ and w using the chain rule.

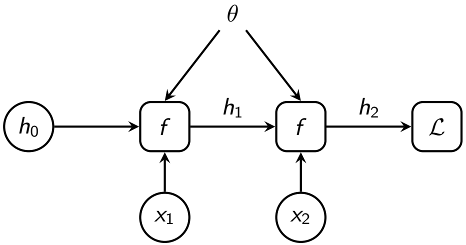
$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \theta}$$
$$\frac{\partial \mathcal{L}}{\partial w} = \underbrace{\frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial h}}_{\frac{\partial \mathcal{L}}{\partial h}}\frac{\partial h}{\partial w}$$


$$\begin{tikzcd} & w \arrow[d, shift right] & & \theta \arrow[d, shift right] & \\ x \arrow[r] & f_1 \arrow[r, "h"] \arrow[u, shift right, red, "\frac{\partial \mathcal{L}}{\partial w}"'] & f_2 \arrow[r, "y"] \arrow[l, shift left, red, "\frac{\partial \mathcal{L}}{\partial h}"] \arrow[u, shift right, red, "\frac{\partial \mathcal{L}}{\partial \theta}"'] & \mathcal{L} \arrow[l, shift left, red, "\frac{\partial \mathcal{L}}{\partial y}"] \end{tikzcd}$$

Backpropagation in RNN

- The difference in the RNN is that each layer implements the same function f with the same (shared) parameters θ:

$$\mathcal{L} = \mathcal{L}(h_2), \quad h_2 = f(x_1, h_1, \theta)$$
$$h_1 = f(x_1, h_0, \theta)$$



Backpropagation in RNN

- The difference in the RNN is that each layer implements the same function f with the same (shared) parameters θ:

$$\mathcal{L} = \mathcal{L}(h_2), \quad h_2 = f(x_1, h_1, \theta)$$
$$h_1 = f(x_1, h_0, \theta)$$

$$\begin{tikzcd}
 & & \theta \arrow[dl] \arrow[dr] & & \\
h_0 \arrow[r] & f \arrow[rr, "h_1"] & & f \arrow[r, "h_2"] & \mathcal{L} \\
 & x_1 \arrow[u] & & x_2 \arrow[u] &
\end{tikzcd}$$

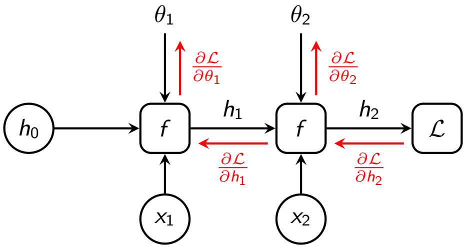
- Let us assume for now that the parameters of the layers are not shared.

$$\begin{tikzcd}
 & \theta_1 \arrow[d] & \theta_2 \arrow[d] & \\
h_0 \arrow[r] & f \arrow[r, "h_1"] & f \arrow[r, "h_2"] & \mathcal{L} \\
 & x_1 \arrow[u] & x_2 \arrow[u] &
\end{tikzcd}$$

Backpropagation in RNN

- We can compute the derivatives wrt parameters θ_1 and θ_2 using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial h_2} \frac{\partial h_2}{\partial \theta_2}$$
$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \underbrace{\frac{\partial \mathcal{L}}{\partial h_2} \frac{\partial h_2}{\partial h_1}}_{\frac{\partial \mathcal{L}}{\partial h_1}} \frac{\partial h_1}{\partial \theta_1}$$

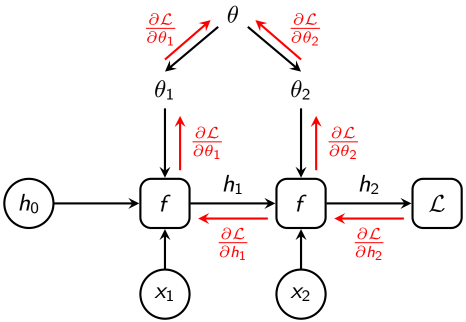

$$\begin{tikzcd}
 & \theta_1 \arrow[d] & & \theta_2 \arrow[d] & \\
h_0 \arrow[r] & f \arrow[u, bend right, "\frac{\partial \mathcal{L}}{\partial \theta_1}"'] \arrow[r, "h_1"] & & f \arrow[u, bend right, "\frac{\partial \mathcal{L}}{\partial \theta_2}"'] \arrow[r, "h_2"] \arrow[ll, bend left, "\frac{\partial \mathcal{L}}{\partial h_1}"] & \mathcal{L} \arrow[l, bend left, "\frac{\partial \mathcal{L}}{\partial h_2}"] \\
 & x_1 \arrow[u] & & x_2 \arrow[u] &
\end{tikzcd}$$

- We can compute the derivatives efficiently using backpropagation.

Backpropagation in RNN

- Finally, we can combine the gradients wrt shared parameters:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \theta_1}\frac{\partial \theta_1}{\partial \theta} + \frac{\partial \mathcal{L}}{\partial \theta_2}\frac{\partial \theta_2}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \theta_1} + \frac{\partial \mathcal{L}}{\partial \theta_2}$$



θ
$\frac{\partial \mathcal{L}}{\partial \theta_1}$
$\frac{\partial \mathcal{L}}{\partial \theta_2}$
θ_1
θ_2
h_0
f
h_1
h_2
$\mathcal{L}$
$\frac{\partial \mathcal{L}}{\partial h_1}$
$\frac{\partial \mathcal{L}}{\partial h_2}$
x_1
x_2

- We need to compute gradients through all possible paths and aggregate them.
- The backpropagation algorithm applied to RNN is called *backpropagation through time*.

Problems with RNN training

Does recurrence cause problems for training?

- Consider a vanilla RNN:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{W}, \mathbf{U}, \mathbf{b}) = \phi(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$

- Assume that we are not careful about selecting ϕ and we select it to be an identity mapping $\phi(a) = a$, $\mathbf{h}_0 = 0$ and $\mathbf{b} = 0$.
- Let us write the hidden state at time t:

$$\mathbf{h}_t = \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t = \mathbf{W}\left(\mathbf{W}\mathbf{h}_{t-2} + \mathbf{U}\mathbf{x}_{t-1}\right) + \mathbf{U}\mathbf{x}_t$$

$$= \mathbf{W}\mathbf{W}\mathbf{h}_{t-2} + \mathbf{W}\mathbf{U}\mathbf{x}_{t-1} + \mathbf{U}\mathbf{x}_t = \sum_{\tau=1}^{t} \mathbf{W}^{t-\tau}\mathbf{U}\mathbf{x}_\tau$$

$$\begin{tikzcd}
\mathbf{h}_0 \arrow[r] & f \arrow[r, "\mathbf{h}_1"] & f \arrow[r, "\mathbf{h}_2"] & f \arrow[r, "\mathbf{h}_3"] & f \arrow[r, "\mathbf{h}_4"] & f \arrow[r, "\mathbf{h}_5"] & {} \\
& \mathbf{x}_1 \arrow[u] & \mathbf{x}_2 \arrow[u] & \mathbf{x}_3 \arrow[u] & \mathbf{x}_4 \arrow[u] & \mathbf{x}_5 \arrow[u] &
\end{tikzcd}$$

Analysis for diagonalizable W

- For simplicity, let us assume that matrix $\mathbf{W}$ is diagonalizable and its eigenvalue decomposition $\mathbf{W} = \mathbf{Q\Lambda Q}^\top$ exists, where $\mathbf{Q}$ is orthogonal and $\mathbf{\Lambda}$ is diagonal. We can then re-write:

$$\mathbf{W}^{t-\tau} = \underbrace{\mathbf{Q\Lambda Q}^\top \mathbf{Q\Lambda Q}^\top \dots \mathbf{Q\Lambda Q}^\top}_{t-\tau \text{ times}} = \mathbf{Q\Lambda}^{t-\tau}\mathbf{Q}^\top$$

- Let us look at the (squared) norm of one term in $\mathbf{h}_t = \sum_{\tau=1}^{t} \mathbf{W}^{t-\tau}\mathbf{U}\mathbf{x}_\tau$:

$$\left\|\mathbf{W}^{t-\tau}\mathbf{U}\mathbf{x}_\tau\right\|^2 = \left\|\mathbf{Q\Lambda}^{t-\tau}\underbrace{\mathbf{Q}^\top\mathbf{U}\mathbf{x}_\tau}_{\mathbf{z}}\right\|^2 = \left\|\mathbf{Q\Lambda}^{t-\tau}\mathbf{z}\right\|^2 = \left\|\mathbf{\Lambda}^{t-\tau}\mathbf{z}\right\|^2 = \sum_i (\lambda_i^{t-\tau} z_i)^2$$

  where λ_i is the i-th diagonal element of $\mathbf{\Lambda}$ and z_i is the i-th element of $\mathbf{z}$.

- If there is an eigenvalue λ_i such that $|\lambda_i| > 1$ (and the corresponding z_i is non-zero), then the norm will grow exponentially causing explosions in the forward computations.

Analysis for a more general case (home reading)

- Let $\mathbf{Q}_m$ be an $n \times m$ matrix containing the m linear independent unit-norm eigenvectors of $\mathbf{W}$ in its columns and $\mathbf{\Lambda}$ be a diagonal matrix made of the corresponding eigenvectors λ_i:

$$\mathbf{W}\mathbf{Q}_m = \mathbf{Q}_m\mathbf{\Lambda}$$

- We can write $\mathbf{U}\mathbf{x}_\tau = \mathbf{Q}_m\mathbf{z} + \mathbf{y}$ where $\mathbf{y}$ belongs to the null space of $\mathbf{Q}_m$.
- Then (ignoring terms that contain $\mathbf{y}$):

$$\begin{aligned}\mathbf{W}^{t-\tau}\mathbf{U}\mathbf{x}_\tau &= \mathbf{W}^{t-\tau-1}\mathbf{W}\mathbf{Q}_m\mathbf{z} + \ldots = \mathbf{W}^{t-\tau-1}\mathbf{Q}_m\mathbf{\Lambda}\mathbf{z} + \ldots \\ &= \mathbf{W}^{t-\tau-2}\mathbf{Q}_m\mathbf{\Lambda}^2\mathbf{z} + \ldots = \mathbf{Q}_m\mathbf{\Lambda}^{t-\tau}\mathbf{z} + \ldots\end{aligned}$$

- The (squared) norm of this vector is $\left\|\mathbf{Q}_m\mathbf{\Lambda}^{t-\tau}\mathbf{z}\right\|^2 = \left\|\mathbf{\Lambda}^{t-\tau}\mathbf{z}\right\|^2 = \sum_{i=1}^m (\lambda_i^{t-\tau} z_i)^2$
- Again, if one of the eigenvalues of $\mathbf{W}$ is such that $|\lambda_i| > 1$ (and the corresponding z_i is non-zero), then this norm will grow exponentially causing explosions in the forward computations.

Explosions in forward computations

- The largest absolute value of the eigenvalues is called *spectral radius*:

$$\text{spectral radius}(\mathbf{W}) = \max_i |\lambda_i|$$

- *Forward explosions happen if the spectral radius of* **W** *is greater than 1.*
- Will explosions happen if we use tanh nonlinearity at each time step?

$$\mathbf{h}_t = \phi(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}) = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$

Explosions in forward computations

- The largest absolute value of the eigenvalues is called *spectral radius*:

$$\text{spectral radius}(\mathbf{W}) = \max_i |\lambda_i|$$

- *Forward explosions happen if the spectral radius of* **W** *is greater than 1.*
- Will explosions happen if we use tanh nonlinearity at each time step?

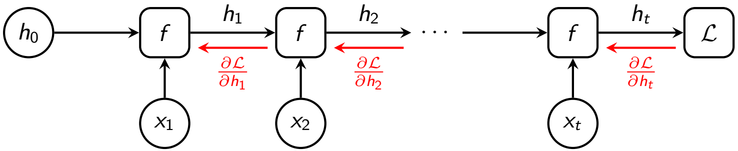
$$\mathbf{h}_t = \phi(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}) = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$

- Since tanh is bounded in $(-1, 1)$, the explosions cannot happen.
- This is the reason why tanh is most commonly used in RNNs.

Are there similar problems in backward computations?

- Lets us look at the longest path of derivative computations (red) for an RNN

$$\mathbf{h}_t = \phi(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$$



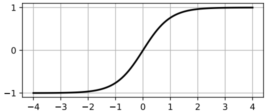
h_0
f
h_1
f
h_2
$\cdots$
f
h_t
$\mathcal{L}$
$\frac{\partial \mathcal{L}}{\partial h_1}$
$\frac{\partial \mathcal{L}}{\partial h_2}$
$\frac{\partial \mathcal{L}}{\partial h_t}$
x_1
x_2
x_t

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}^\top = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\dots,2} \frac{\partial \mathbf{h}_\tau}{\partial \mathbf{h}_{\tau-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\dots,2} \mathrm{diag}(\phi'_\tau)\mathbf{W}$$

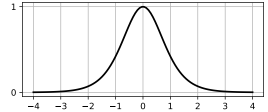
- $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}$ is a column vector of partial derivatives $\frac{\partial \mathcal{L}}{\partial h_{1i}}$
- $\phi'_\tau = \phi'(\mathbf{W}\mathbf{h}_{\tau-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b})$

Gradient explosions (Pascanu et al., 2013)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}^\top = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\dots,2} \frac{\partial \mathbf{h}_\tau}{\partial \mathbf{h}_{\tau-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\dots,2} \text{diag}(\phi'_\tau)\mathbf{W}$$



$\phi(h)$



$\phi'(h)$

- Suppose $\phi(h) = \tanh(h)$ and all our neurons in an RNN are not saturated, which means that

$$|\phi'_\tau| \geq \gamma$$

- If the spectral radius of **W** is greater than $1/\gamma$, then the gradient explodes.
- The gradient may explode even for a bounded activation function ϕ!
- To avoid explosions, it is good to keep neurons in the saturated regime where derivatives ϕ' are small.

How to cope with gradient explosions?

- Gradient explosions (caused by recurrence) is one problem with training RNNs.
- One workaround: clip the gradient if it is larger than some pre-defined value:
  - can be done element-wise (Mikolov, 2012) or by clipping the norm (Pascanu et al., 2013):

$$\text{if } \|\mathbf{g}\| \geq \Delta, \text{ then } \mathbf{g} \leftarrow \Delta \frac{\mathbf{g}}{\|\mathbf{g}\|}$$

- In PyTorch, clipping of gradients can be done by re-writing `parameter.grad.data` after calling `loss.backward()`.

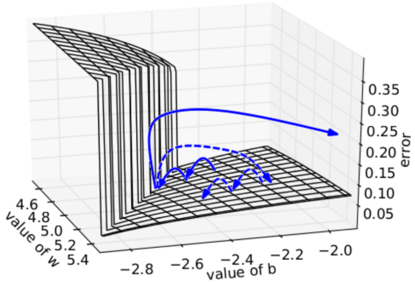


image from (Pascanu et al., 2013)

Vanishing gradients

- Let us look at the gradients again:

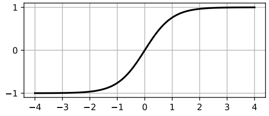
$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}^{\top} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^{\top} \prod_{\tau=t,\dots,2} \frac{\partial \mathbf{h}_\tau}{\partial \mathbf{h}_{\tau-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^{\top} \prod_{\tau=t,\dots,2} \mathrm{diag}(\phi'_\tau)\mathbf{W}$$

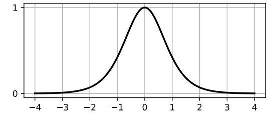
- The absolute values of $|\phi'_\tau|$ are bounded:

$$0 < |\phi'_\tau| \leq 1$$

- If the spectral radius of **W** is smaller than 1, the gradient will vanish (its norm will decay exponentially with increase of t).

- To avoid vanishing gradients, it is good to keep neurons in the non-saturated regime where derivatives ϕ' are close to 1.


$\phi(h)$


$\phi'(h)$

Vanishing gradients

- The vanishing gradients problem makes it difficult to learn long-range dependencies in the data:
    - In sentiment analysis, it is difficult to capture the effect of the first words in a paragraph on the predicted class.
    - In time-series modeling, it is difficult to capture slowly changing phenomena.
- Vanilla RNNs $\mathbf{h}_t = \phi(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x} + \mathbf{b})$ are rarely used in practice.
- Recurrent units with gating mechanisms work better.
    - Gated recurrent unit (GRU) (Cho et al., 2014)
    - Long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997)

Historical note on RNNs

- Recurrent neural networks for sequential data processing were proposed in the 80s (Rumelhart et al., 1986; Elman, 1990; Werbos, 1988).
- RNNs did not gain much popularity because they were particularly difficult to train with backpropagation:
    - Unstable training because of gradient explosions
    - Difficulty to learn long-term dependencies due to vanishing gradients (Bengio et al., 1994)
- The breakthrough came with the invention of Long Short-Term Memory (LSTM) RNN (Hochreiter and Schmidhuber, 1997) which was designed to solve the gradient explosion/vanishing problem.
- LSTM remained largely unnoticed in the community until the deep learning boom started.

Gated recurrent unit (GRU)

(Cho et al., 2014)

Gated recurrent unit (GRU)

- Motivation for gating in GRU:
  - Vanilla RNN $\mathbf{h}_t = \phi(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x} + \mathbf{b})$ re-writes all the elements of state $\mathbf{h}_{t-1}$ with new values $\mathbf{h}_t$.
  - How can we keep old values for some elements of $\mathbf{h}_{t-1}$?
- GRU uses an update gate $\mathbf{u}_t \in (0, 1)$ that controls which states should be updated:

$$\mathbf{h}_t = (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \widetilde{\mathbf{h}}_t$$
$$\mathbf{u}_t = \sigma(\mathbf{W}_u\mathbf{h}_{t-1} + \mathbf{U}_u\mathbf{x}_t + \mathbf{b}_u)$$

  where $\sigma(x) = 1/(1 + e^{-x})$ is the sigmoid function and $\widetilde{\mathbf{h}}_t$ are the new state candidates.
- The new state candidates are computed using only the states selected by the reset gate $\mathbf{r}_t$:

$$\widetilde{\mathbf{h}}_t = \phi(\mathbf{W}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{U}\mathbf{x}_t + \mathbf{b}_h)$$
$$\mathbf{r}_t = \sigma(\mathbf{W}_r\mathbf{h}_{t-1} + \mathbf{U}_r\mathbf{x}_t + \mathbf{b}_r)$$

Gated recurrent unit (GRU)

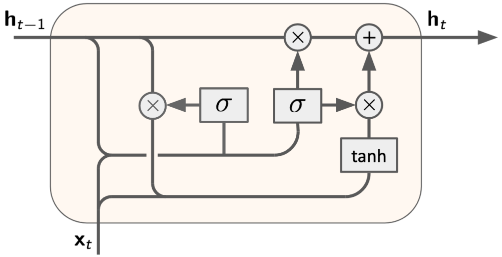
- State update:

$$\mathbf{h}_t = (1 - \mathbf{u}) \odot \mathbf{h}_{t-1} + \mathbf{u} \odot \widetilde{\mathbf{h}}_t$$

- Update gate: $\mathbf{u} = \sigma(\mathbf{W}_u \mathbf{h}_{t-1} + \mathbf{U}_u \mathbf{x}_t + \mathbf{b}_u)$
- New candidate state:

$$\widetilde{\mathbf{h}}_t = \phi(\mathbf{W}(\mathbf{r} \odot \mathbf{h}_{t-1}) + \mathbf{U}\mathbf{x}_t + \mathbf{b}_h)$$

- Reset gate: $\mathbf{r} = \sigma(\mathbf{W}_r \mathbf{h}_{t-1} + \mathbf{U}_r \mathbf{x}_t + \mathbf{b}_r)$



Does GRU help with the vanishing gradient problem?

- GRU update rule for the state:

$$\mathbf{h}_t = (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \phi(\mathbf{W}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{U}\mathbf{x}_t)$$

- Let us look at the gradient (back)propagation assuming that $\mathbf{u}_t$ and $\mathbf{r}_t$ are fixed:

$$\frac{\partial \mathbf{h}_\tau}{\partial \mathbf{h}_{\tau-1}} = \mathrm{diag}(1 - \mathbf{u}_\tau) + \mathrm{diag}(\mathbf{u}_\tau)\,\mathrm{diag}(\phi'_\tau)\mathbf{W}\,\mathrm{diag}(\mathbf{r}_\tau)$$

  where $\phi'_\tau = \phi'(\mathbf{W}(\mathbf{r}_\tau \odot \mathbf{h}_{\tau-1}) + \mathbf{U}\mathbf{x}_\tau)$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}^\top = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\dots,2} \frac{\partial \mathbf{h}_\tau}{\partial \mathbf{h}_{\tau-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^\top \prod_{\tau=t,\dots,2} \left(\mathrm{diag}(1 - \mathbf{u}_\tau) + \mathrm{diag}(\mathbf{u}_\tau)\,\mathrm{diag}(\phi'_\tau)\mathbf{W}\,\mathrm{diag}(\mathbf{r}_\tau)\right)$$

- For simplicity, let us assume that the state of an RNN is one-dimensional and all intermediate signals do not depend on time step τ:

$$\frac{\partial \mathcal{L}}{\partial h_t} \prod_{\tau=t,\dots,2} \left((1 - u_\tau) + u_\tau \phi'_\tau w r_\tau\right) = \frac{\partial \mathcal{L}}{\partial h_t} \left((1-u) + u\gamma r\right)^{t-1} = \frac{\partial \mathcal{L}}{\partial h_t} \left(\frac{1 + \gamma/2}{2}\right)^{t-1}$$

  where $\gamma = \phi'_\tau w$ and we also assumed that gates u, r are half-closed $u = r = \frac{1}{2}$.

Does GRU help with the vanishing gradient problem?

- Gradient propagation in GRU (simplified): $\frac{\partial \mathcal{L}}{\partial h_t}\left(\frac{1+\gamma/2}{2}\right)^{t-1}$
- Let us do the same simplified analysis for vanilla RNN:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}^{\top}\prod_{\tau=t,\ldots,2}\operatorname{diag}(\phi'_\tau)\mathbf{W} = \frac{\partial \mathcal{L}}{\partial h_t}\prod_{\tau=t,\ldots,2}\phi'_\tau w = \frac{\partial \mathcal{L}}{\partial h_t}\gamma^{t-1} \quad \text{where } \gamma = \phi'_\tau w$$

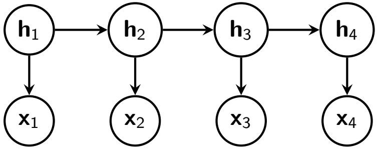
- If γ is small, the gradients in GRU decay with rate $\frac{1}{2}$ which is much better than the rate of γ in the vanilla RNN.
- If γ is large, the magnitudes of the gradients grow exponentially as $O\left(\frac{\gamma^t}{4^t}\right)$ which is better than $O\left(\gamma^t\right)$ in the vanilla RNN.
- Thus, the gating mechanism combats the problem of vanishing/exploding gradients. Gradients may explode or vanish in GRU but such problems occur more rarely compared to the vanilla RNN.

Connection to probabilistic graphical models for sequential data

Linear dynamical systems

- Consider a linear Gaussian model with temporal structure (time series):

$$\begin{aligned} p(\mathbf{h}_1) &= \mathcal{N}(\mathbf{h}_1 \mid \boldsymbol{\mu}_1, \mathbf{R}_1) \\ p(\mathbf{h}_t \mid \mathbf{h}_{t-1}) &= \mathcal{N}(\mathbf{h}_t \mid \mathbf{B}\mathbf{h}_{t-1}, \mathbf{R}) \\ p(\mathbf{x}_t \mid \mathbf{h}_t) &= \mathcal{N}(\mathbf{x}_t \mid \mathbf{A}\mathbf{h}_t, \mathbf{V}) \end{aligned}$$

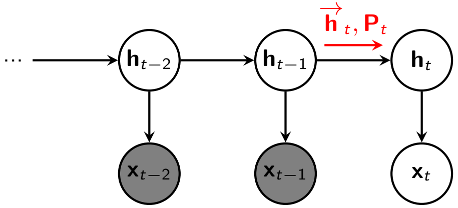


- Inference in linear dynamical systems: Find the conditional distribution $p(\mathbf{h}_t \mid \mathbf{x}_1, \dots, \mathbf{x}_t)$ of latent variables $\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_t$ given the observation sequence $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_t$.
- Since it is a linear Gaussian probabilistic model, the inference can be done using the message-passing algorithm (see, e.g., Chapter 13 of Bishop, 2006) which yields the Kalman filter.

Kalman filter: Message-passing in linear dynamical systems

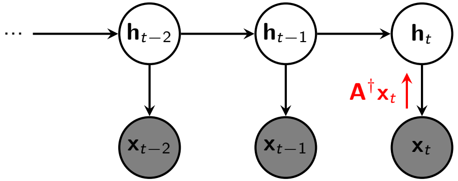
1. Prediction $p(\mathbf{h}_t \mid \mathbf{x}_1, \ldots, \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{h}_t \mid \overrightarrow{\mathbf{h}}_t, \mathbf{P}_t)$

$$\overrightarrow{\mathbf{h}}_t = \mathbf{B}\bar{\mathbf{h}}_{t-1}$$
$$\mathbf{P}_t = \mathbf{B}\mathbf{\Sigma}_{t-1}\mathbf{B}^\top + \mathbf{R}$$



2. Correction $p(\mathbf{h}_t \mid \mathbf{x}_1, \ldots, \mathbf{x}_t) = \mathcal{N}(\mathbf{h}_t \mid \bar{\mathbf{h}}_t, \mathbf{\Sigma}_t)$

$$\bar{\mathbf{h}}_t = \overrightarrow{\mathbf{h}}_t + \mathbf{K}_t(\mathbf{x}_t - \mathbf{A}\overrightarrow{\mathbf{h}}_t)$$
$$\mathbf{\Sigma}_t = (\mathbf{I} - \mathbf{K}_t\mathbf{A})\mathbf{P}_{t-1}$$
$$\mathbf{K}_t = \mathbf{P}_{t-1}\mathbf{A}^\top(\mathbf{A}\mathbf{P}_{t-1}\mathbf{A}^\top + \mathbf{V})^{-1}$$



The message from $\mathbf{x}_t$ to $\mathbf{h}_t$ is usually not explicitly expressed in the derivations of the Kalman filter.

Kalman filter in one-dimensional case

- Let us look closer at the correction equation for the mean values of the hidden states

$$\bar{\mathbf{h}}_t = \overrightarrow{\mathbf{h}}_t + \mathbf{K}_t(\mathbf{x}_t - \mathbf{A}\overrightarrow{\mathbf{h}}_t)$$
$$\mathbf{K}_t = \mathbf{P}_{t-1}\mathbf{A}^\top(\mathbf{A}\mathbf{P}_{t-1}\mathbf{A}^\top + \mathbf{V})^{-1}$$

in the one-dimensional case:

$$\begin{aligned}\bar{h}_t &= \overrightarrow{h}_t + k_t(x_t - a\overrightarrow{h}_t) = \overrightarrow{h}_t + \frac{p_{t-1}a}{a^2p_{t-1}+v}(x_t - a\overrightarrow{h}_t)\\ &= \overrightarrow{h}_t - \frac{p_{t-1}a^2}{a^2p_{t-1}+v}\overrightarrow{h}_t + \frac{p_{t-1}a}{a^2p_{t-1}+v}x_t = \frac{v}{a^2p_{t-1}+v}\overrightarrow{h}_t + \frac{a^2p_{t-1}}{a^2p_{t-1}+v}\frac{x_t}{a}\\ &= (1-u_t)\overrightarrow{h}_t + u_t\frac{x_t}{a}\end{aligned}$$

where $u_t = \sigma\left(\log\frac{a^2p_{t-1}}{a^2p_{t-1}+v}\right)$

- The updated value of the state is a trade-off between the estimate $\overrightarrow{h}_t$ computed before observing x_t (prior) and the value $\frac{x_t}{a}$ justified by observation $\frac{x_t}{a}$ (likelihood).

Motivation of gatings in recurrent units

- Kalman filter update in the one-dimensional case:

$$\bar{h}_t = (1 - u_t)\overrightarrow{h}_t + u_t \frac{x_t}{a}$$

$$u_t = \sigma\left(\log \frac{a^2 p_{t-1}}{a^2 p_{t-1} + v}\right)$$

- Compare this with the GRU update rule:

$$\mathbf{h}_t = (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \tilde{\mathbf{h}}_t$$

$$\mathbf{u} = \sigma(\mathbf{W}_u \mathbf{h}_{t-1} + \mathbf{U}_u \mathbf{x}_t + \mathbf{b}_u)$$

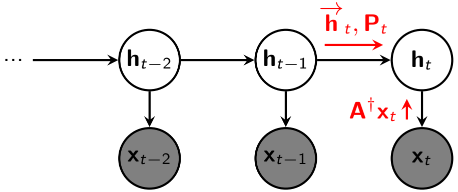
- This example justifies the use of gatings in the recurrent units: gatings allow combination of information gained from the previous observations and the current observation.
- The same intuitions hold for nonlinear dynamic systems (extended Kalman filter) which can be learned by RNNs.

Computational graph of RNN as implementation of message passing

- Message passing in linear dynamical systems:

$$\bar{h}_t = (1 - u_t)\overrightarrow{h}_t + u_t \frac{x_t}{a}$$

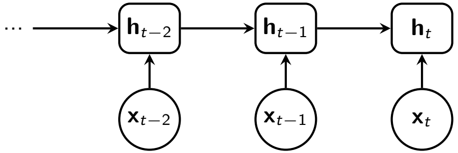
$$u_t = \sigma\left(\log \frac{a^2 p_{t-1}}{a^2 p_{t-1} + v}\right)$$



- Computational graph of an RNN with gatings:

$$\mathbf{h}_t = (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t \odot \tilde{\mathbf{h}}_t$$

$$\mathbf{u} = \sigma(\mathbf{W}_u \mathbf{h}_{t-1} + \mathbf{U}_u \mathbf{x}_t + \mathbf{b}_u)$$



- The computational graph of an RNN with gatings can be seen as implementation of an inference procedure for a probabistic graphical model with sequential data.

Long short-term memory (LSTM)

(Hochreiter and Schmidhuber, 1997)

Long short-term memory (LSTM) unit

- LSTM was designed to prevent vanishing and exploding gradients.
- The unit has two states: observed state $\mathbf{h}_t$ and hidden state $\mathbf{c}_t$.
- The new hidden state is a (gated) sum of the old state and an update:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \phi_c(\mathbf{W}_c\mathbf{h}_{t-1} + \mathbf{U}_c\mathbf{x}_t + \mathbf{b}_c)$$

  where forget gate $\mathbf{f}_t \in (0,1)$ and input gate $\mathbf{i}_t \in (0,1)$.

- The gradient propagation for state $\mathbf{c}$:

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \mathrm{diag}(\mathbf{f}_t)$$

  and if we set $\mathbf{f}_t$ to 1, the gradient neither grows nor decreases.

Long short-term memory (LSTM) unit

- Update of the hidden state:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \phi_c(\mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{U}_c \mathbf{x}_t + \mathbf{b}_c)$$

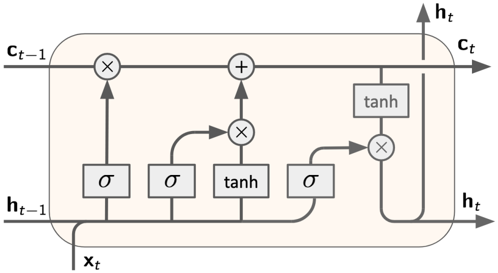
$$\text{forget gate } \mathbf{f}_t = \sigma(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f)$$

$$\text{input gate } \mathbf{i}_t = \sigma(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t + \mathbf{b}_i)$$

- The cell output (observed state):

$$\mathbf{h}_t = \mathbf{o}_t \odot \phi_h(\mathbf{c}_t)$$

$$\text{output gate } \mathbf{o}_t = \sigma(\mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_o \mathbf{x}_t + \mathbf{b}_o)$$



Initialization of the forget gates (Jozefowicz et al., 2015)

- Update gate:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f)$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \phi_c(\mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{U}_c \mathbf{x}_t + \mathbf{b}_c)$$

- Common intialization of the forget gate: small random weights for $\mathbf{b}_u$. This initialization effectively sets the forget gate to $\frac{1}{2}$ and therefore the gradient vanishes with a factor of $\frac{1}{2}$ per timestep. It works well in many problems.
- However, sometimes an RNN can fail to learn long-term dependencies. This problem can be addressed by initializing the forget gates $\mathbf{b}_u$ to large values such as 1 or 2.

Architecture search for recurrent units

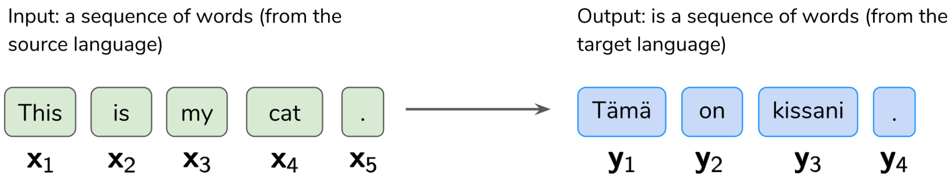
- LSTM and GRU have somewhat similar but different architectures. Can there be even better architectures of the recurrent unit?
- Jozefowicz et al. (2015) performed random search of the architecture by constructing the recurrent unit from a selected set of operations. The performance was tested on a set of standard benchmarks.
- The best architectures found in that procedure were very similar to GRU!

$$\begin{aligned}
\mathbf{z} &= \sigma(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{b}_z) \\
\mathbf{r} &= \sigma(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_t + \mathbf{b}_r) \\
\mathbf{h}_{t+1} &= \tanh(\mathbf{W}_{hh}(\mathbf{r} \odot \mathbf{h}_t) + \tanh(\mathbf{x}_t) + \mathbf{b}_h) \odot \mathbf{z} + \mathbf{h}_t \odot (1 - \mathbf{z})
\end{aligned}$$

Sequence-to-sequence models for neural machine translation

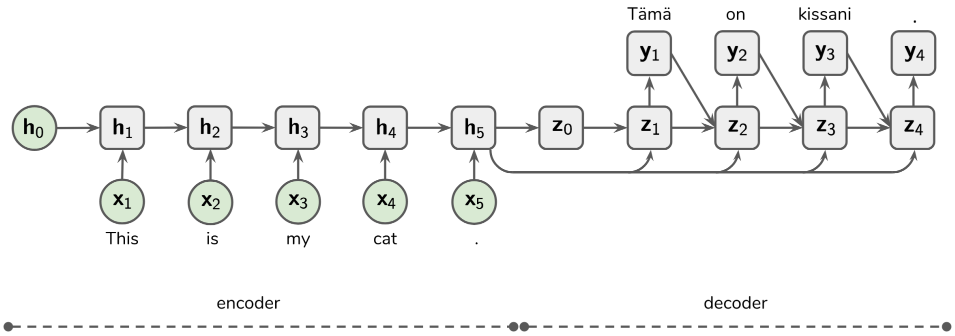
Neural machine translation

- The task is to translate a sentence from a source language to a target language.
- Inputs and outputs are sequences of words. We need a model that transforms input sequences into output sequences (a sequence-to-sequence model).
- Input and output sequences may be of different lengths.



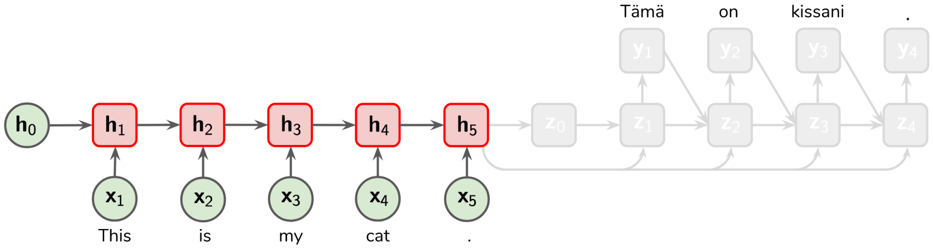
Simple sequence-to-sequence model

- The simplest sequence-to-sequence model uses two RNNs: encoder and decoder.



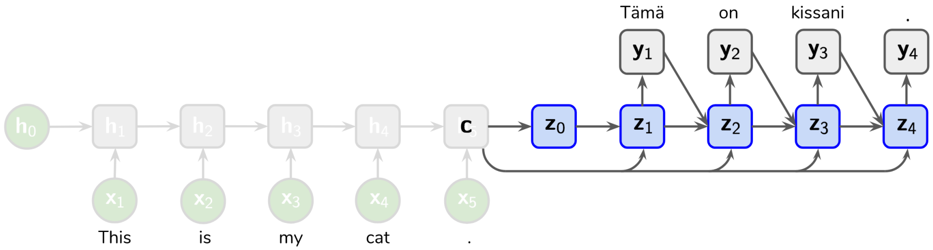
Simple sequence-to-sequence model

- The encoder is an RNN that encodes the input sentence into a vector $\mathbf{c} = \mathbf{h}_5$.
- The whole sentence is represented as a vector (a vector of thought).



Simple sequence-to-sequence model

- The decoder is an RNN that converts the developed representation **c** into the output sentence:

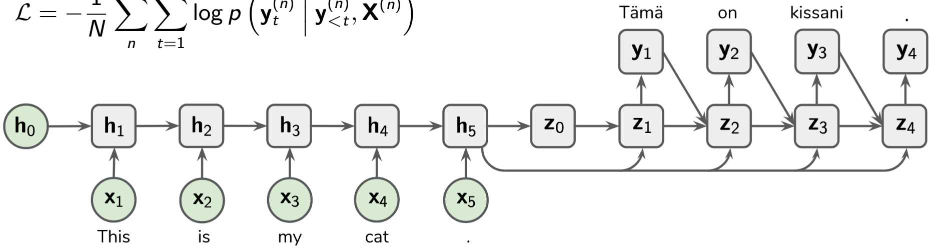


- Each neuron also receives the previous word and the input-sequence representation **c** as inputs.

Simple sequence-to-sequence model: Training

- The minimized cost is the negative log-likelihood of the output sequence:

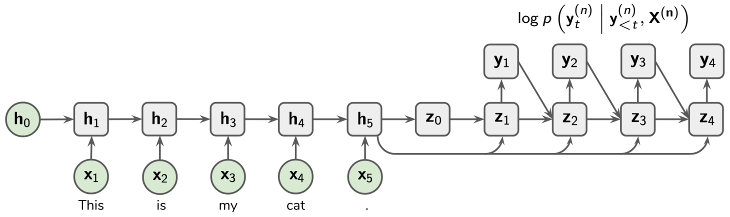
$$\mathcal{L} = -\frac{1}{N}\sum_{n}\sum_{t=1}^{T_n}\log p\left(\mathbf{y}_t^{(n)} \,\middle|\, \mathbf{y}_{<t}^{(n)}, \mathbf{X}^{(n)}\right)$$



- To produce categorical distribution over words, we apply softmax function to the hidden states of the decoder RNN: $p(\mathbf{y}_t = i \,|\, \mathbf{y}_{<t}, \mathbf{X}) \propto \exp(\mathbf{w}_i^\top \mathbf{z}_t)$.

Simple sequence-to-sequence model: Test time

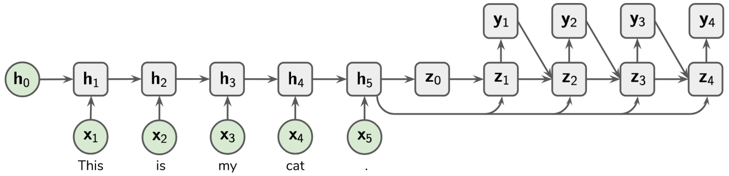
- How to generate the output sequence for a given input sequence?
- We can sample a sequence of words using the predicted categorical distribution:



- This is suboptimal: we are interested in the whole sequence that has the highest probability, sampling from the output distribution is greedy search.
- The most likely sequence is usually found with beamsearch (see, e.g., Cho, 2015).

Teacher forcing

- Training time: Feed correct words as inputs of the decoder (this is called teacher forcing).
- Test time: Feed the decoder's own predictions as inputs (generation mode).

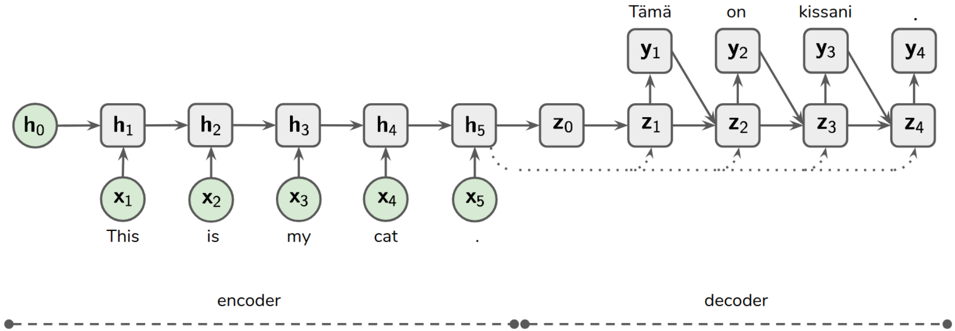


- The decoder needs to learn to work in the generation mode (without teacher forcing).
- To enable this, we can toggle teacher forcing on and off during training.

Home assignment

Assignment 05_rnn

- In the home assignment, you need to implement a sequence-to-sequence model for statistical machine translation:



Building a computational graph with an RNN in PyTorch

- There are two way to build a computational graph with RNNs in PyTorch.
- In simple cases, the whole sequence can be processed with one call:

```
h = torch.zeros(...)
h = rnn.forward(x, h)
```

- In more difficult cases, you need to build a graph with a for-loop:

```
h = torch.zeros(...)
for x_t in x:
    h = rnn.forward(x_t, h)
```

- The initial states of RNNs are often initialized with zeros.

How to represent words

- A simple word representation is one-hot vector. Word i is represented with vector $\mathbf{z}$ such that $z_i = 1$, $z_{j \neq i} = 0$.
- Better representaion:
  - represent each word i as a vector $\mathbf{w}_i$
  - treat all vectors $\mathbf{w}_i$ as model parameters and tune them in the training procedure
  - this is equivalent to $\mathbf{Wz}$ where $\mathbf{W}$ is a matrix of word embeddings (word vectors $\mathbf{w}_i$ in its columns).
- This is implemented in `torch.nn.Embedding(num_embeddings, embedding_dim)`
  - `num_embeddings` is the size of the dictionary
  - `embedding_dim` is the size of each embedding vector $\mathbf{w}_i$

Recommended reading

- Chapter 10 of the Deep Learning book.
- C. Olah. Understanding LSTM Networks.
- K. Cho. Natural Language Understanding with Distributed Representation.