Recurrent neural networks

Fourth lecture

Lecturer: A. Taylor

Slides's version: February 25, 2026.

Thanks Erwan Scornet (original slides) and Pontus Giselsson

Evaluation

Graded homework:

- You should receive it via moodle by Friday (via moodle & slack).
- Deadline: 11th / 12th of March.

Exam:

- Wed 18th of March (9am to 11.30am). Place to be confirmed (see Synapses).
- Exam: no documents or calculators allowed.

Time-dependency

Objective: model time-dependent data. Examples:

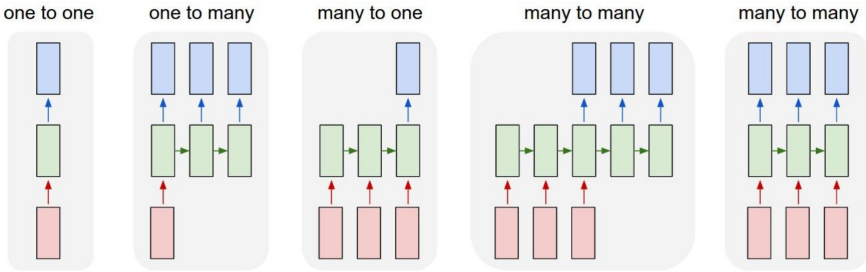
- time series,
- translation,
- videos.

Issues with conventional neural networks?

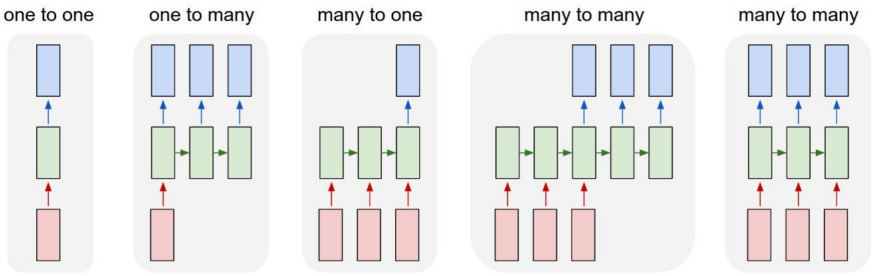
1. queries/data with variable lengths (e.g., sentences do not always contain the same number of words),
2. orderings in sequences matter.

Outline

RNNs offer a lot of variability



RNNs offer a lot of variability



- Vanilla neural networks,
- image captioning: image/sequence of words,
- sentiment classification: sequence of words/sentiments,
- translation: sequence of words/sequence of words,
- video classification on frame level: sequence of images/sequence of labels.

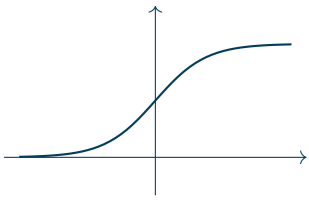
Limitations of conventional neural networks

- NNs can't deal with sequential or "temporal" data,
- NNs lack (time) memory,
- NNs have a fixed architecture: fixed input/output sizes,

In this class: recurrent neural networks (RNNs). Motivation $\rightarrow$ natural way to handle time-dependencies.

Reminders – sigmoid

Recall: sigmoid activation $\sigma(u) = \frac{1}{1+e^{-u}}$. Traditional activation in old-fashioned neural nets.

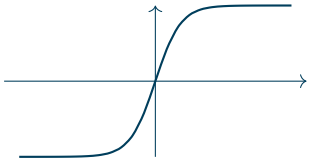


- max value: $\sup_u \sigma(u) = \lim_{u\to\infty} \sigma(u) = 1$,
- min value: $\inf_u \sigma(u) = \lim_{u\to-\infty} \sigma(u) = 0$,
- max derivative: $\max_u |\sigma'(u)| = \frac{1}{4}$.

... among the root causes of vanishing gradients.

Reminders – tanh

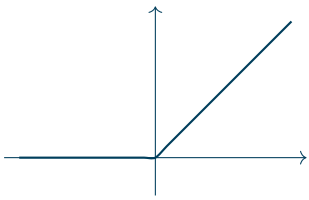
Recall: tanh activation $\tanh(u) = \frac{e^u - e^{-u}}{e^{-u} + e^u}$.



- max value: $\sup_u \tanh(u) = 1$,
- min value: $\inf_u \tanh(u) = -1$,
- max derivative: $\max_u |\tanh'(u)| = 1$.

Reminders – ReLU

Recall: ReLU activation $\mathrm{ReLU}(u) = \max\{0, u\}$. Now classical activation.



- unbounded activation: $\sup_u \mathrm{ReLU}(u) = +\infty$,
- min value: $\inf_u \mathrm{ReLU}(u) = 0$,
- max derivative: $\max_u |\mathrm{ReLU}'(u)| = 1$.

- Do not use sigmoid or tanh in networks with many layers (vanishing gradient).
- ReLU activations overcome vanishing gradient problems $\rightarrow$ learn and perform better.
- ReLU activation is default when developing MLPs and CNNs.
- ReLUs usually not used in RNNs (functions with bounded values are preferred).

Outline

Legend

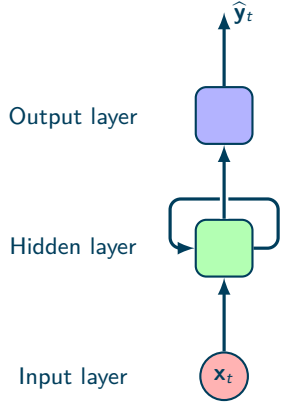
Fully connected (FC) linear layer with activation

Elementwise operation

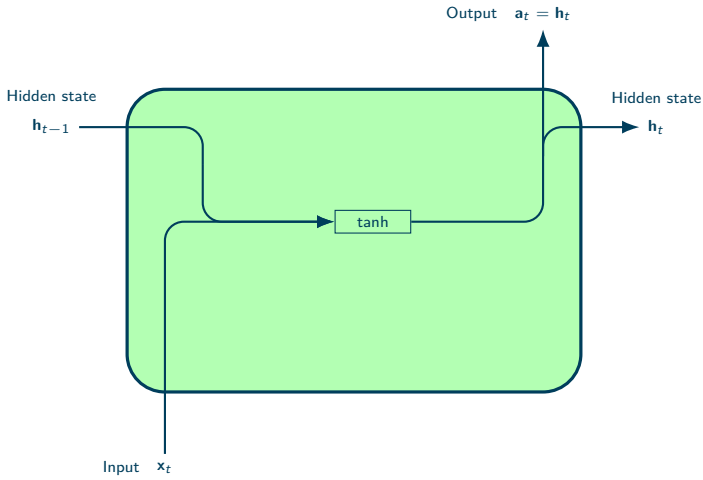
Copy

Concatenate

RNN layer

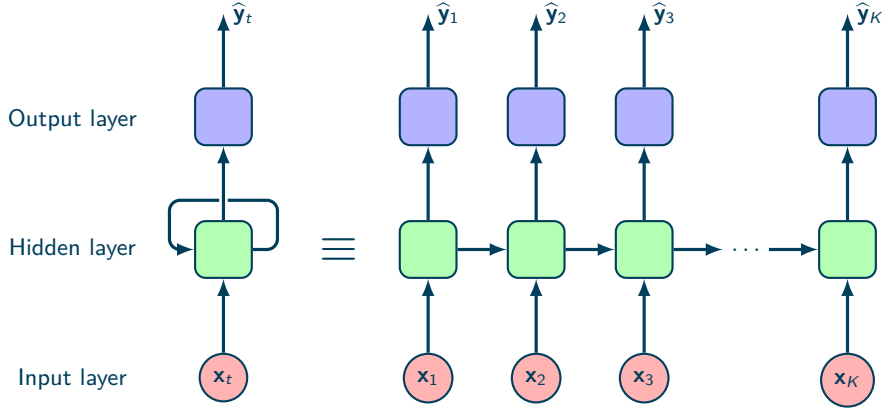

$\widehat{\mathbf{y}}_t$
Output layer
Hidden layer
Input layer
$\mathbf{x}_t$

Base recurrent layer



Why not RELU? $\rightarrow$ ReLUs might suffer from "exploding values" (ReLUs could be used, with adaptations).

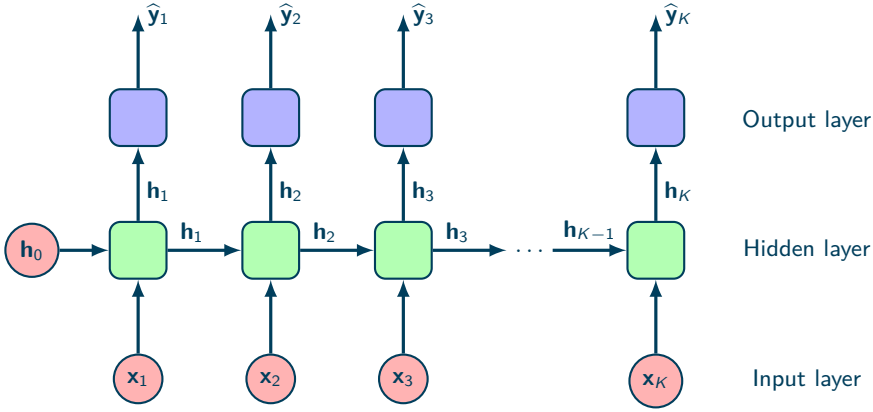
Unfolded RNN



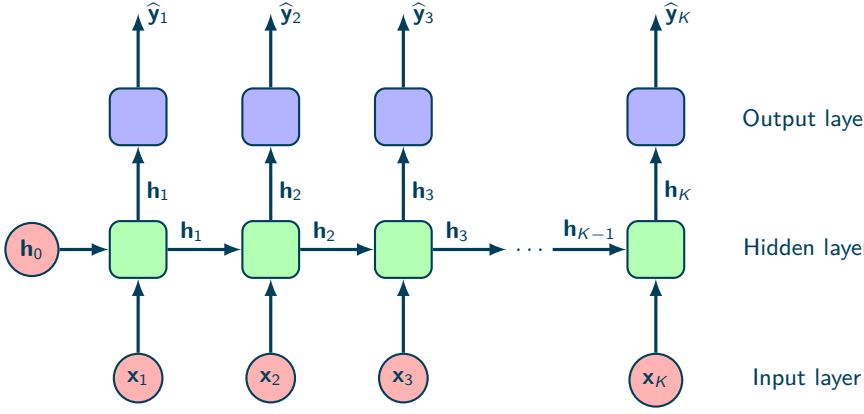
Note: last $\mathbf{x}_K$ is denoted with K (and not T, avoid confusion with transpose).

Definitions in RNN

- Input layer: data comes sequentially: $\mathbf{x}_1, \mathbf{x}_2, \ldots$.
- Hidden Layer: hidden state of the network at time t: $\mathbf{h}_t$.
- Output layer: for the input $\mathbf{x}_t$, the prediction is given by $\widehat{\mathbf{y}}_t$.



Definition of RNN



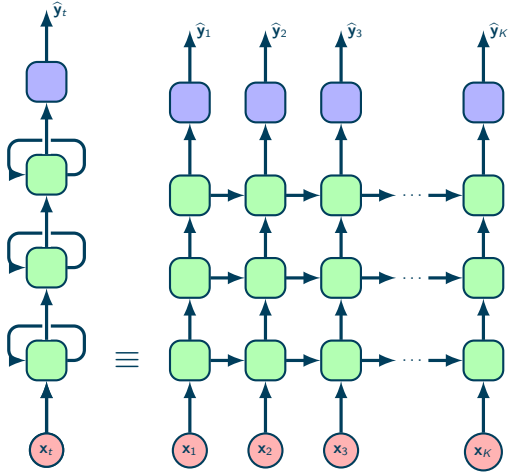
Hidden layer ($\rightarrow$ parameter sharing!)

$$\mathbf{h}_t = \tanh(W_{HH}\mathbf{h}_{t-1} + W_{IH}\mathbf{x}_t + \mathbf{b}_h)$$

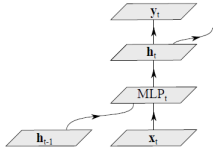
Output (example for classification here, other possibilities such as ReLU are acceptable):

$$\hat{\mathbf{y}}_t = \mathrm{softmax}(W_{HO}\mathbf{h}_t + \mathbf{b}_{out})$$

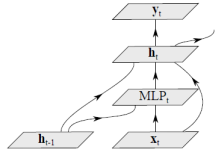
Stacking recurrent layers – deep RNN



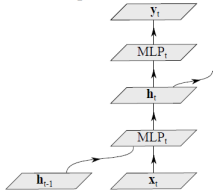
Deep RNN



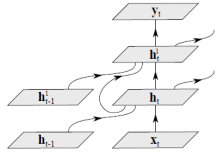
(a) Input to hidden.



(b) Input to hidden with short-cut.

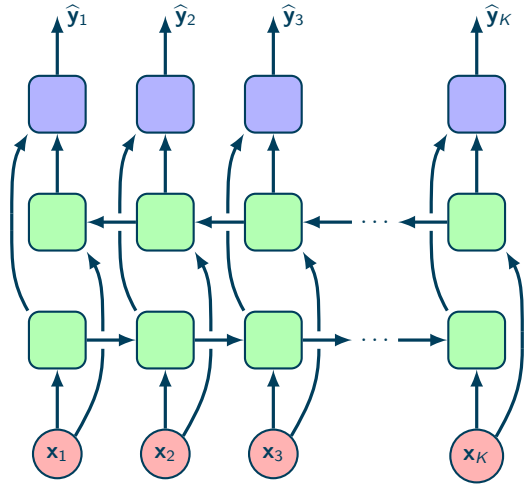


(c) Hidden to hidden and output.



(d) Stack of hidden states.

Bi-directional RNN (BRNN)



Bi-directional RNN (BRNN)

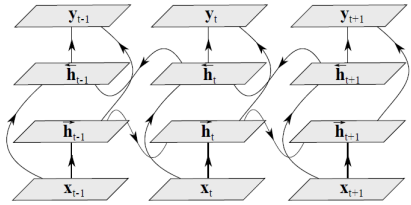


Figure: bi-directional recurrent neural network (BRNN)

If output layer is fully connected layer:

$$\widehat{\mathbf{y}}_t = W_{\overrightarrow{HO}}\overrightarrow{\mathbf{h}_t} + W_{\overleftarrow{HO}}\overleftarrow{\mathbf{h}_t} + \mathbf{b}_o$$

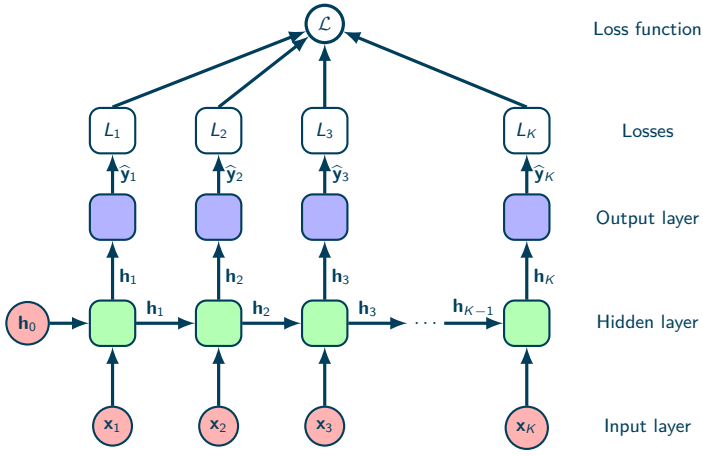
E.g., `https://www.geeksforgeeks.org/bidirectional-recurrent-neural-network/`

Review, RNNs architectures: ["Recent Advances in Recurrent Neural Networks", Salehinejad et al. 2017].

Outline

Outline

Loss


$\mathcal{L}$
Loss function
L_1
L_2
L_3
L_K
Losses
$\hat{\mathbf{y}}_1$
$\hat{\mathbf{y}}_2$
$\hat{\mathbf{y}}_3$
$\hat{\mathbf{y}}_K$
Output layer
$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_3$
$\mathbf{h}_K$
$\mathbf{h}_0$
$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_3$
$\cdots$
$\mathbf{h}_{K-1}$
Hidden layer
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_K$
Input layer

Gradients via backpropagation through time (BPTT) — scalar network

Recall:

$$\mathbf{h}_t = \tanh(W_{HH}\mathbf{h}_{t-1} + W_{IH}\mathbf{x}_t + \mathbf{b}_h)$$
$$\hat{\mathbf{y}}_t = \mathrm{softmax}(W_{HO}\mathbf{h}_t + \mathbf{b}_{out}).$$

Gradients via backpropagation through time (BPTT) — scalar network

Recall:

$$\mathbf{h}_t = \tanh(W_{HH}\mathbf{h}_{t-1} + W_{IH}\mathbf{x}_t + \mathbf{b}_h)$$
$$\hat{\mathbf{y}}_t = \mathrm{softmax}(W_{HO}\mathbf{h}_t + \mathbf{b}_{out}).$$

Derivative with respect to W_{HH}? $\frac{\partial L_K}{\partial W_{HH}}$ via usual chain rule through the different "paths" depending on W_{HH}

Gradients via backpropagation through time (BPTT) — scalar network

Recall:

$$\mathbf{h}_t = \tanh(W_{HH}\mathbf{h}_{t-1} + W_{IH}\mathbf{x}_t + \mathbf{b}_h)$$
$$\hat{\mathbf{y}}_t = \mathrm{softmax}(W_{HO}\mathbf{h}_t + \mathbf{b}_{out}).$$

Derivative with respect to W_{HH}? $\frac{\partial L_K}{\partial W_{HH}}$ via usual chain rule through the different "paths" depending on W_{HH}
Do as if all quantities were **scalar-valued** (otherwise: tensors... and a bit painful), then, it follows:

- $K = 1$:

$$\frac{\partial L_1}{\partial W_{HH}} = \frac{\partial L_1}{\partial \widehat{\mathbf{y}}_1}\frac{\partial \widehat{\mathbf{y}}_1}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W_{HH}},$$

Gradients via backpropagation through time (BPTT) — scalar network

Recall:

$$\mathbf{h}_t = \tanh(W_{HH}\mathbf{h}_{t-1} + W_{IH}\mathbf{x}_t + \mathbf{b}_h)$$
$$\hat{\mathbf{y}}_t = \mathrm{softmax}(W_{HO}\mathbf{h}_t + \mathbf{b}_{out}).$$

Derivative with respect to W_{HH}? $\frac{\partial L_K}{\partial W_{HH}}$ via usual chain rule through the different "paths" depending on W_{HH}
Do as if all quantities were **scalar-valued** (otherwise: tensors... and a bit painful), then, it follows:

- $K = 1$:

$$\frac{\partial L_1}{\partial W_{HH}} = \frac{\partial L_1}{\partial \widehat{\mathbf{y}}_1}\frac{\partial \widehat{\mathbf{y}}_1}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W_{HH}},$$

- $K = 2$:

$$\frac{\partial L_2}{\partial W_{HH}} = \frac{\partial L_2}{\partial \widehat{\mathbf{y}}_2}\frac{\partial \widehat{\mathbf{y}}_2}{\partial \mathbf{h}_2}\left(\frac{\partial \mathbf{h}_2}{\partial W_{HH}} + \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W_{HH}}\right),$$

Gradients via backpropagation through time (BPTT) — scalar network

Recall:

$$\mathbf{h}_t = \tanh(W_{HH}\mathbf{h}_{t-1} + W_{IH}\mathbf{x}_t + \mathbf{b}_h)$$
$$\hat{\mathbf{y}}_t = \mathrm{softmax}(W_{HO}\mathbf{h}_t + \mathbf{b}_{out}).$$

Derivative with respect to W_{HH}? $\frac{\partial L_K}{\partial W_{HH}}$ via usual chain rule through the different "paths" depending on W_{HH}
Do as if all quantities were **scalar-valued** (otherwise: tensors... and a bit painful), then, it follows:

- $K = 1$:

$$\frac{\partial L_1}{\partial W_{HH}} = \frac{\partial L_1}{\partial \widehat{\mathbf{y}}_1}\frac{\partial \widehat{\mathbf{y}}_1}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W_{HH}},$$

- $K = 2$:

$$\frac{\partial L_2}{\partial W_{HH}} = \frac{\partial L_2}{\partial \widehat{\mathbf{y}}_2}\frac{\partial \widehat{\mathbf{y}}_2}{\partial \mathbf{h}_2}\left(\frac{\partial \mathbf{h}_2}{\partial W_{HH}} + \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W_{HH}}\right),$$

- $K = 3$:

$$\frac{\partial L_3}{\partial W_{HH}} = \frac{\partial L_3}{\partial \widehat{\mathbf{y}}_3}\frac{\partial \widehat{\mathbf{y}}_3}{\partial \mathbf{h}_3}\left(\frac{\partial \mathbf{h}_3}{\partial W_{HH}} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial W_{HH}} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W_{HH}}\right),$$

Gradients via backpropagation through time (BPTT) — scalar network

Recall:

$$\mathbf{h}_t = \tanh(W_{HH}\mathbf{h}_{t-1} + W_{IH}\mathbf{x}_t + \mathbf{b}_h)$$
$$\hat{\mathbf{y}}_t = \mathrm{softmax}(W_{HO}\mathbf{h}_t + \mathbf{b}_{out}).$$

Derivative with respect to W_{HH}? $\frac{\partial L_K}{\partial W_{HH}}$ via usual chain rule through the different "paths" depending on W_{HH}
Do as if all quantities were **scalar-valued** (otherwise: tensors... and a bit painful), then, it follows:

- $K = 1$:

$$\frac{\partial L_1}{\partial W_{HH}} = \frac{\partial L_1}{\partial \widehat{\mathbf{y}}_1}\frac{\partial \widehat{\mathbf{y}}_1}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W_{HH}},$$

- $K = 2$:

$$\frac{\partial L_2}{\partial W_{HH}} = \frac{\partial L_2}{\partial \widehat{\mathbf{y}}_2}\frac{\partial \widehat{\mathbf{y}}_2}{\partial \mathbf{h}_2}\left(\frac{\partial \mathbf{h}_2}{\partial W_{HH}} + \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W_{HH}}\right),$$

- $K = 3$:

$$\frac{\partial L_3}{\partial W_{HH}} = \frac{\partial L_3}{\partial \widehat{\mathbf{y}}_3}\frac{\partial \widehat{\mathbf{y}}_3}{\partial \mathbf{h}_3}\left(\frac{\partial \mathbf{h}_3}{\partial W_{HH}} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial W_{HH}} + \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}\frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}\frac{\partial \mathbf{h}_1}{\partial W_{HH}}\right),$$

this is *backpropagation through time* (BPTT).

Gradients via backpropagation through time (BPTT) — scalar network

Backpropagation through time (BPTT) equation

$$\frac{\partial L_K}{\partial W_{HH}} = \frac{\partial L_K}{\partial \widehat{\mathbf{y}}_K} \frac{\partial \widehat{\mathbf{y}}_K}{\partial \mathbf{h}_K} \sum_{k=1}^{K} \Big(\prod_{m=k+1}^{K} \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} \Big) \frac{\partial \mathbf{h}_k}{\partial W_{HH}}$$

with convention $\prod_{i=I}^{J} = 1$ if $I > J$ and

Gradients via backpropagation through time (BPTT) — scalar network

Backpropagation through time (BPTT) equation

$$\frac{\partial L_K}{\partial W_{HH}} = \frac{\partial L_K}{\partial \widehat{\mathbf{y}}_K} \frac{\partial \widehat{\mathbf{y}}_K}{\partial \mathbf{h}_K} \sum_{k=1}^{K} \Big(\prod_{m=k+1}^{K} \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} \Big) \frac{\partial \mathbf{h}_k}{\partial W_{HH}}$$

with convention $\prod_{i=I}^{J} = 1$ if $I > J$ and

$$\frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} = W_{HH} \tanh'(W_{HH}\mathbf{h}_{m-1} + W_{IH}\mathbf{x}_m + \mathbf{b}_h),$$
$$\frac{\partial \mathbf{h}_k}{\partial W_{HH}} = \tanh'(W_{HH}\mathbf{h}_{k-1} + W_{IH}\mathbf{x}_k + \mathbf{b}_h)\mathbf{h}_{k-1}.$$

Gradients via backpropagation through time (BPTT) — scalar network

Backpropagation through time (BPTT) equation

$$\frac{\partial L_K}{\partial W_{HH}} = \frac{\partial L_K}{\partial \widehat{\mathbf{y}}_K} \frac{\partial \widehat{\mathbf{y}}_K}{\partial \mathbf{h}_K} \sum_{k=1}^{K} \Big(\prod_{m=k+1}^{K} \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} \Big) \frac{\partial \mathbf{h}_k}{\partial W_{HH}}$$

with convention $\prod_{i=I}^{J} = 1$ if $I > J$ and

$$\frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} = W_{HH} \tanh'(W_{HH}\mathbf{h}_{m-1} + W_{IH}\mathbf{x}_m + \mathbf{b}_h),$$
$$\frac{\partial \mathbf{h}_k}{\partial W_{HH}} = \tanh'(W_{HH}\mathbf{h}_{k-1} + W_{IH}\mathbf{x}_k + \mathbf{b}_h)\mathbf{h}_{k-1}.$$

Any problem?

Vanishing gradients

Recall that the 2-norm of a matrix A is given by

$$\|A\|_2 = \sup_{\mathbf{x} \neq 0} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \sup\{\|A\mathbf{x}\|_2, \|\mathbf{x}\|_2 = 1\} = \sqrt{\lambda_{max}(A^T A)}.$$

Vanishing gradients

Recall that the 2-norm of a matrix A is given by

$$\|A\|_2 = \sup_{\mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \sup\{\|A\mathbf{x}\|_2, \|\mathbf{x}\|_2 = 1\} = \sqrt{\lambda_{max}(A^T A)}.$$

Further: $\|Ax\|_2 \leqslant \|A\|_2 \, \|x\|_2$.

Vanishing gradients

Recall that the 2-norm of a matrix A is given by

$$\|A\|_2 = \sup_{\mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \sup\{\|A\mathbf{x}\|_2, \|\mathbf{x}\|_2 = 1\} = \sqrt{\lambda_{max}(A^T A)}.$$

Further: $\|Ax\|_2 \leqslant \|A\|_2 \|x\|_2$.

Note that $\tanh'(u) \leqslant 1$. Assume $\lambda_{max}({W_{HH}}^T W_{HH})$ bounded above by $\eta < 1$:

$$\left\| \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} \right\|_2 \leqslant \|W_{HH}\|_2 \left\| \mathrm{diag}(\tanh'(W_{HH}\mathbf{h}_{m-1} + W_{IH}\mathbf{x}_m + \mathbf{b}_h)) \right\|_2 < \eta.$$

Vanishing gradients

Recall that the 2-norm of a matrix A is given by

$$\|A\|_2 = \sup_{\mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \sup\{\|A\mathbf{x}\|_2, \|\mathbf{x}\|_2 = 1\} = \sqrt{\lambda_{max}(A^T A)}.$$

Further: $\|Ax\|_2 \leqslant \|A\|_2 \|x\|_2$.

Note that $\tanh'(u) \leqslant 1$. Assume $\lambda_{max}({W_{HH}}^T W_{HH})$ bounded above by $\eta < 1$:

$$\left\| \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} \right\|_2 \leqslant \|W_{HH}\|_2 \left\| \mathrm{diag}(\tanh'(W_{HH}\mathbf{h}_{m-1} + W_{IH}\mathbf{x}_m + \mathbf{b}_h)) \right\|_2 < \eta.$$

Thus,

$$\left\| \prod_{m=k+1}^{T} \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} \right\|_2 \leqslant \eta^{T-k}.$$

As $T - k$ gets larger, contribution of kth term decreases exponentially fast.

Exploding and vanishing gradients

Backpropagation through time (BPTT) equation

$$\frac{\partial L_K}{\partial W_{HH}} = \frac{\partial L_K}{\partial \widehat{\mathbf{y}}_K} \frac{\partial \widehat{\mathbf{y}}_K}{\partial \mathbf{h}_K} \sum_{k=1}^{K} \Big(\prod_{m=k+1}^{K} \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} \Big) \frac{\partial \mathbf{h}_k}{\partial W_{HH}}.$$

Exploding and vanishing gradients

Backpropagation through time (BPTT) equation

$$\frac{\partial L_K}{\partial W_{HH}} = \frac{\partial L_K}{\partial \widehat{\mathbf{y}}_K} \frac{\partial \widehat{\mathbf{y}}_K}{\partial \mathbf{h}_K} \sum_{k=1}^{K} \Big(\prod_{m=k+1}^{K} \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} \Big) \frac{\partial \mathbf{h}_k}{\partial W_{HH}}.$$

In summary:

- vanishing gradient when $\left\| \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} \right\|_2 < 1$

  influence from early timesteps is too small $\rightarrow$ learning long-term dependencies becomes impossible
- exploding gradient when $\left\| \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} \right\|_2 > 1.$

  influence from early timesteps is too large $\rightarrow$ learning becomes impossible

Problem: same matrix W_{HH} is used all along the path: small/large singular values are amplified exponentially.

Unrolling derivatives via chain rule (quater): effect of recurrence?

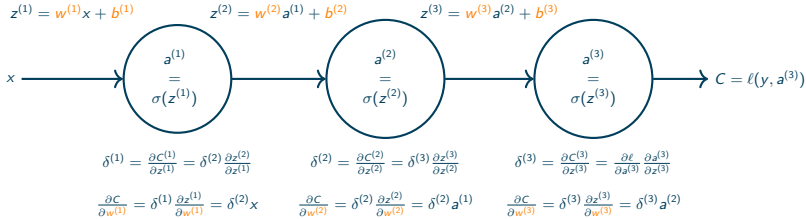
Consider a simple unidimensional network:

- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).

Unrolling derivatives via chain rule (quater): effect of recurrence?

Consider a simple unidimensional network:

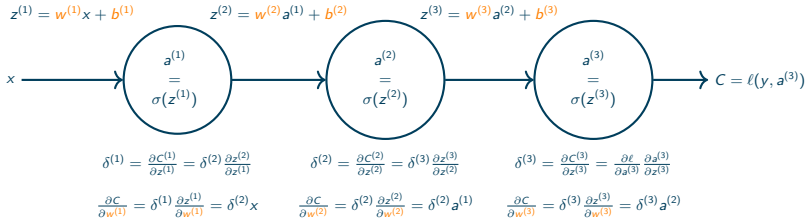
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (quater): effect of recurrence?

Consider a simple unidimensional network:

- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



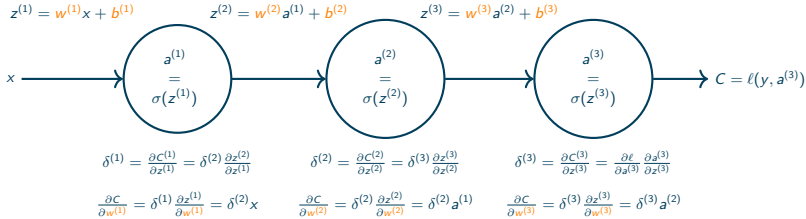
Effect of $w^{(1)} = w^{(2)} = w^{(3)} \triangleq w$ and $b^{(1)} = b^{(2)} = b^{(3)} \triangleq b$ ($\sigma(z) = z$ for simplicity)?

$$\frac{\partial C}{\partial w} =$$

Unrolling derivatives via chain rule (quater): effect of recurrence?

Consider a simple unidimensional network:

- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



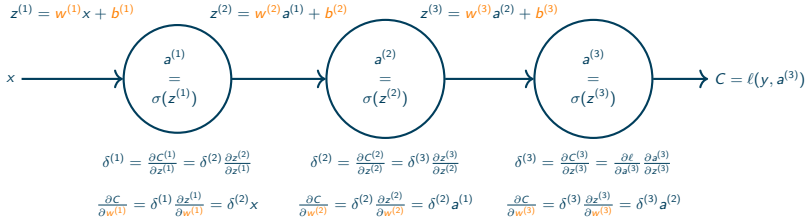
Effect of $w^{(1)} = w^{(2)} = w^{(3)} \triangleq w$ and $b^{(1)} = b^{(2)} = b^{(3)} \triangleq b$ ($\sigma(z) = z$ for simplicity)?

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial w^{(1)}} + \frac{\partial C}{\partial w^{(2)}} + \frac{\partial C}{\partial w^{(3)}} = b + 2bw + 3xw^2.$$

Unrolling derivatives via chain rule (quater): effect of recurrence?

Consider a simple unidimensional network:

- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Effect of $w^{(1)} = w^{(2)} = w^{(3)} \triangleq w$ and $b^{(1)} = b^{(2)} = b^{(3)} \triangleq b$ ($\sigma(z) = z$ for simplicity)?

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial w^{(1)}} + \frac{\partial C}{\partial w^{(2)}} + \frac{\partial C}{\partial w^{(3)}} = b + 2bw + 3xw^2.$$

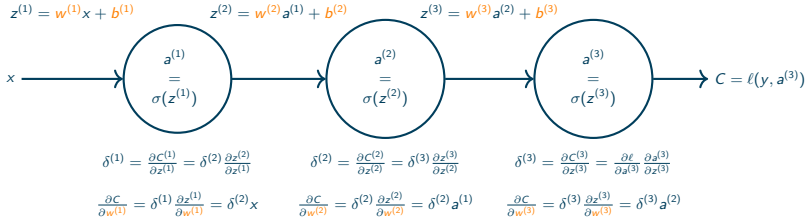
L layers: $w^{(1)} = \ldots = w^{(L)} \triangleq w$ and $b^{(1)} = \ldots = b^{(L)} \triangleq b$ (with $\sigma(z) = z$)?

$$\frac{\partial C}{\partial w} =$$

Unrolling derivatives via chain rule (quater): effect of recurrence?

Consider a simple unidimensional network:

- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Effect of $w^{(1)} = w^{(2)} = w^{(3)} \triangleq w$ and $b^{(1)} = b^{(2)} = b^{(3)} \triangleq b$ ($\sigma(z) = z$ for simplicity)?

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial w^{(1)}} + \frac{\partial C}{\partial w^{(2)}} + \frac{\partial C}{\partial w^{(3)}} = b + 2bw + 3xw^2.$$

L layers: $w^{(1)} = \ldots = w^{(L)} \triangleq w$ and $b^{(1)} = \ldots = b^{(L)} \triangleq b$ (with $\sigma(z) = z$)?

$$\frac{\partial C}{\partial w} = Lxw^{L-1} + b\sum_{k=1}^{L-1} kw^{k-1}.$$

Outline

Penalization

- Exploding gradient $\rightarrow$ gradient clipping.

  If gradient gets too large, threshold the gradient. Threshold value can be chosen by looking at statistics of the gradient over a few updates.

  ["Empirical evaluation and combination of advanced language modeling techniques", Tomáš Mikolov et al. 2011]

- Vanishing gradient via penalization. E.g., enforce gradient norms to be *preserved* as it travels time:

$$\Omega = \sum_k \left(\frac{\left\| \frac{\partial L}{\partial \mathbf{h}_{k+1}} \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k} \right\|}{\left\| \frac{\partial L}{\partial \mathbf{h}_{k+1}} \right\|} - 1 \right)^2 .$$

  ["On the difficulty of training recurrent neural networks", Pascanu et al. 2013]

Weight initialization

- The weight matrix W_{HH} is initialized at identity, biases are set to zero, with ReLU activation function.

  ["A simple way to initialize recurrent networks of rectified linear units", Le et al. 2015]

- Learn a weight matrix that is a mixture of the identity matrix and another matrix (mix of long-term and small-term dependencies).

  ["Learning longer memory in recurrent neural networks", Tomas Mikolov, Joulin, et al. 2014]

- Initialize W randomly among positive definite matrices (real and positive eigenvalues) with one eigenvalue at 1 and the other less (or equal) to 1.

  ["Improving performance of recurrent neural network with relu nonlinearity", Talathi and Vartak 2015]

Outline

Legend for what follows:

Fully connected (FC) linear layer with activation

Elementwise operation

Copy

Concatenate

Forgetting

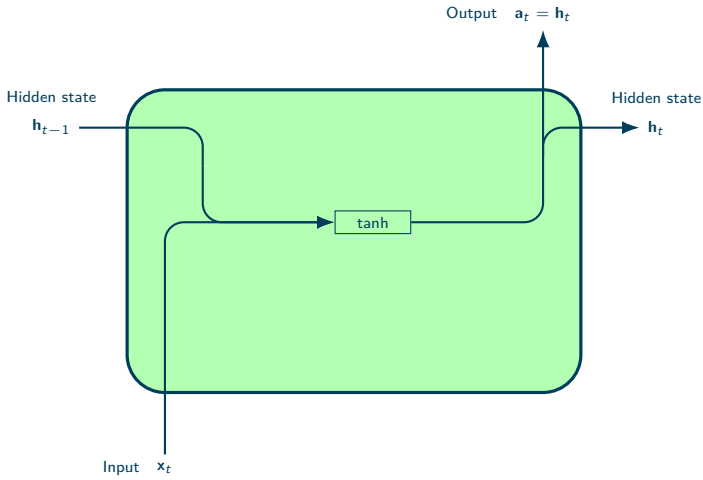
Why is forgetting an important issue?

- Example for a movie rating:

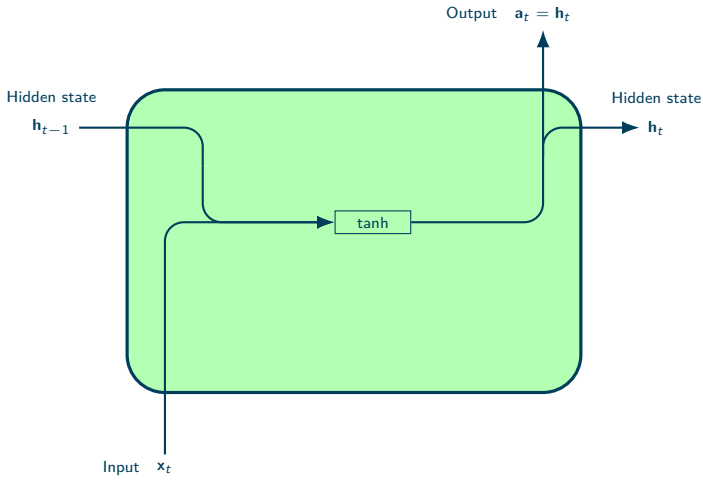
Good	movie,	the	story	is	somewhat	surprising
$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	$\mathbf{x}_6$	$\mathbf{x}_7$

$\rightarrow$ how to predict a "good" label if we forgot the first word of the sentence?

Base recurrent layer



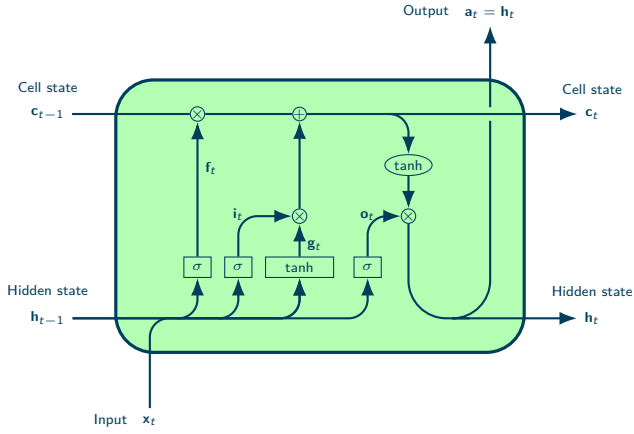
Base recurrent layer



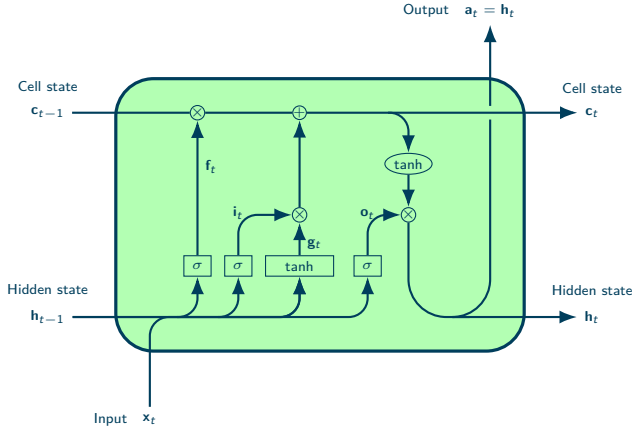
Vanishing gradient & forgetting: often attacked through **gates**.

Long short-term memory (LSTM)

LSTM: circumvent vanishing / explosions of "time-travelling" information by "gating". ["Long short-term memory", Hochreiter and Schmidhuber 1997]



Long Short Term Memory (LSTM)

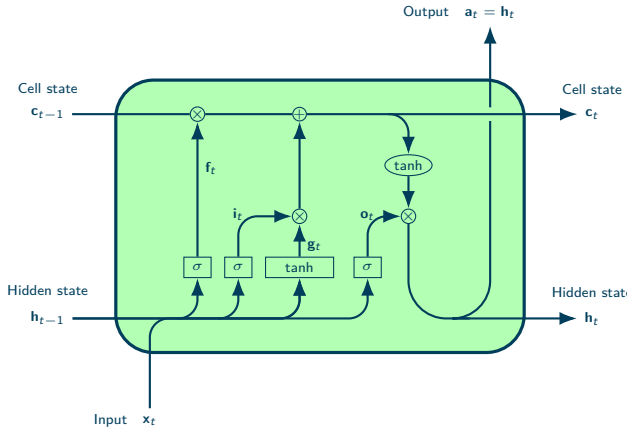


LSTM equations, forget gate:

$$\mathbf{f}_t = \sigma(W_{f,h}\mathbf{h}_{t-1} + W_{f,x}\mathbf{x}_t + b_f)$$

(proportion of the old cell state that we will remember).

Long Short Term Memory (LSTM)



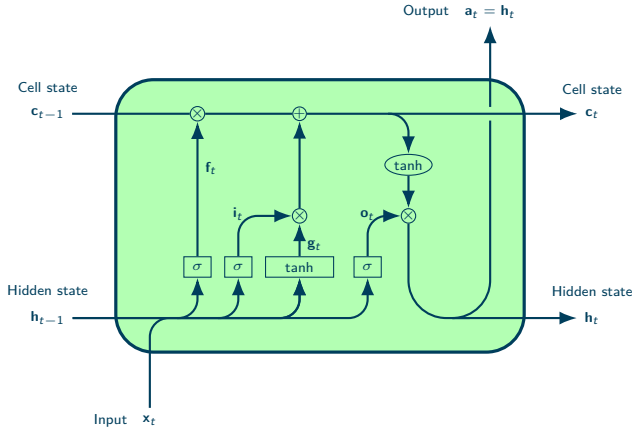
LSTM equations, input gate:

$$\mathbf{g}_t = \tanh(W_{g,h}\mathbf{h}_{t-1} + W_{g,x}\mathbf{x}_t + b_g)$$

$$\mathbf{i}_t = \sigma(W_{i,h}\mathbf{h}_{t-1} + W_{i,x}\mathbf{x}_t + b_i)$$

(candidate additional info for long-term + proportion of it added to long-term memory).

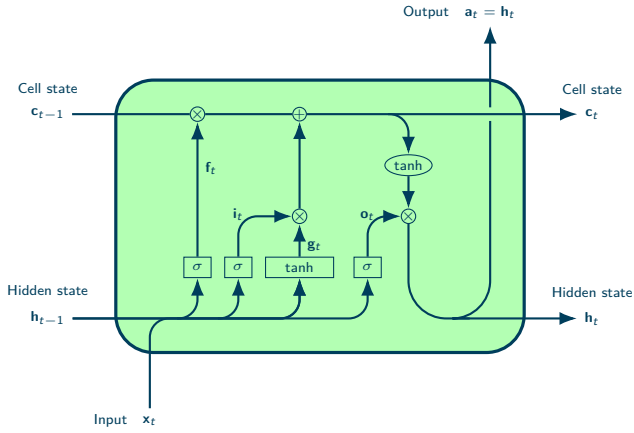
Long Short Term Memory (LSTM)



LSTM equations, new cell state:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t.$$

Long Short Term Memory (LSTM)

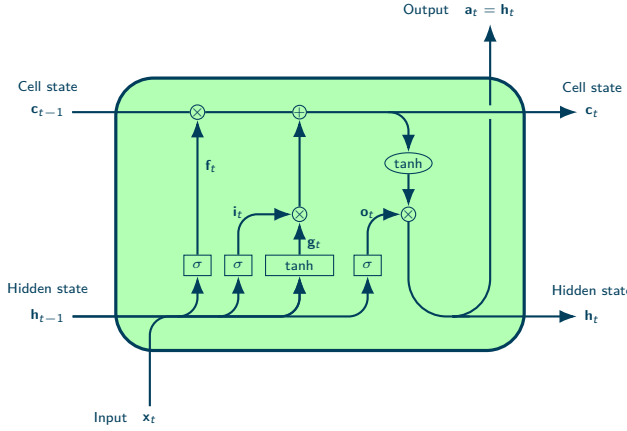


LSTM equations, output gate & new hidden state:

$$\mathbf{o}_t = \sigma(W_{o,h}\mathbf{h}_{t-1} + W_{o,x}\mathbf{x}_t + b_o)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)



LSTM equations, output gate & new hidden state:

$$\mathbf{o}_t = \sigma(W_{o,h}\mathbf{h}_{t-1} + W_{o,x}\mathbf{x}_t + b_o)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(c_t)$$

$\rightarrow$ observe: prediction of the network at time t is $\mathbf{h}_t$ (not $\mathbf{c}_t$).

LSTM and backpropagation

Recall the cell update: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$.

Gradient propagation through the cell state: $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \mathrm{diag}(\mathbf{f}_t)$.

LSTM and backpropagation

Recall the cell update: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$.

Gradient propagation through the cell state: $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \operatorname{diag}(\mathbf{f}_t)$.

Backpropagation through time:

$$\frac{\partial L_K}{\partial \mathbf{c}_k} = \Big(\prod_{m=k+1}^{K} \operatorname{diag}(\mathbf{f}_m) \Big) \frac{\partial L_K}{\partial \mathbf{c}_K}.$$

LSTM and backpropagation

Recall the cell update: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$.

Gradient propagation through the cell state: $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \mathrm{diag}(\mathbf{f}_t)$.
Backpropagation through time:

$$\frac{\partial L_K}{\partial \mathbf{c}_k} = \Big(\prod_{m=k+1}^{K} \mathrm{diag}(\mathbf{f}_m) \Big) \frac{\partial L_K}{\partial \mathbf{c}_K}.$$

- Base RNN: repeated multiplication by same matrix $\prod_m \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}}$ $\Rightarrow$ exponential effect (vanishing/exploding).

LSTM and backpropagation

Recall the cell update: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$.

Gradient propagation through the cell state: $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \mathrm{diag}(\mathbf{f}_t)$.
Backpropagation through time:

$$\frac{\partial L_K}{\partial \mathbf{c}_k} = \Big(\prod_{m=k+1}^{K} \mathrm{diag}(\mathbf{f}_m) \Big) \frac{\partial L_K}{\partial \mathbf{c}_K}.$$

- Base RNN: repeated multiplication by same matrix $\prod_m \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}}$ $\Rightarrow$ exponential effect (vanishing/exploding).
- LSTM: additive memory path $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} +$ (new information)

LSTM and backpropagation

Recall the cell update: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t$.

Gradient propagation through the cell state: $\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \mathrm{diag}(\mathbf{f}_t)$.
Backpropagation through time:

$$\frac{\partial L_K}{\partial \mathbf{c}_k} = \Big(\prod_{m=k+1}^{K} \mathrm{diag}(\mathbf{f}_m) \Big) \frac{\partial L_K}{\partial \mathbf{c}_K}.$$

- Base RNN: repeated multiplication by same matrix $\prod_m \frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}}$ $\Rightarrow$ exponential effect (vanishing/exploding).
- LSTM: additive memory path $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} +$ (new information)

Idea: vanishing/exploding gradients may still occur, but no longer structural (mitigated along memory path).

LSTM variations

There are many variations around this LSTM scheme. Examples:

- see discussions in `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`,
- gated recurrent unit (GRU), next.

Important failures in some variations:

- forget gate and the input gate are not always synchronized at beginning of training. This can cause hidden states to become large and unstable.
- Since hidden state is not bounded, gates can saturate $\rightarrow$ training difficulties!

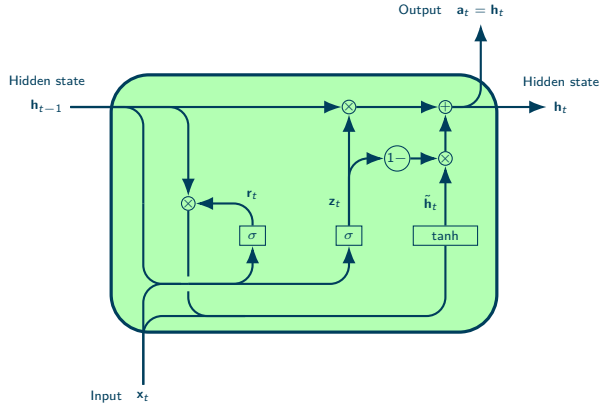
Gated recurrent unit (GRU)

["Learning phrase representations using RNN encoder-decoder for statistical machine translation", Cho et al. 2014]

The gated recurrent unit (GRU) is a common variation around LSTM.

- similar ideas (gating),
- no different hidden / cell states,
- forget, input, output gates $\rightarrow$ reset and update gate.

Gated recurrent unit (GRU)



Reset gate (read gate)

$$\mathbf{r}_t = \sigma(W_{r,h}\mathbf{h}_{t-1} + W_{r,x}\mathbf{x}_t + \mathbf{b}_r)$$

Update gate (forget gate)

$$\mathbf{z}_t = \sigma(W_{z,h}\mathbf{h}_{t-1} + W_{z,x}\mathbf{x}_t + \mathbf{b}_z)$$

Candidate hidden state

$$\tilde{\mathbf{h}}_t = \tanh(W_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + W_x\mathbf{x}_t + \mathbf{b})$$

Hidden state

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

Comparison of LSTM and GRU

- Traditional RNN: hidden states are fully replaced (new value computed via activation, current input and previous hidden state).
- Both LSTM and GRU keep existing content and add the new content on top of it.
  - structurally forces each unit to remember existence of a specific features in the input stream for a long sequence of steps. Important features, decided by either the forget gate of the LSTM unit or the update gate of the GRU, will not be overwritten but be maintained as is.
  - Shortcut paths that bypass multiple temporal steps. These shortcuts allow error to be backpropagated more easily with less vanishing effect (see ResNets and highway nets).

Comparison of LSTM and GRU

- Traditional RNN: hidden states are fully replaced (new value computed via activation, current input and previous hidden state).
- Both LSTM and GRU keep existing content and add the new content on top of it.
  - structurally forces each unit to remember existence of a specific features in the input stream for a long sequence of steps. Important features, decided by either the forget gate of the LSTM unit or the update gate of the GRU, will not be overwritten but be maintained as is.
  - Shortcut paths that bypass multiple temporal steps. These shortcuts allow error to be backpropagated more easily with less vanishing effect (see ResNets and highway nets).

Empirical results.

- GRU / LSTM: comparable performance,
- GRU / LSTM: better than standard RNN,
- No clear consensus between GRU and LSTM ["An empirical exploration of recurrent network architectures", Jozefowicz et al. 2015].

Poll!

https://app.wooclap.com/CGYRYZ?from=instruction-slide

Poll!

`https://app.wooclap.com/CGYRYZ?from=instruction-slide`

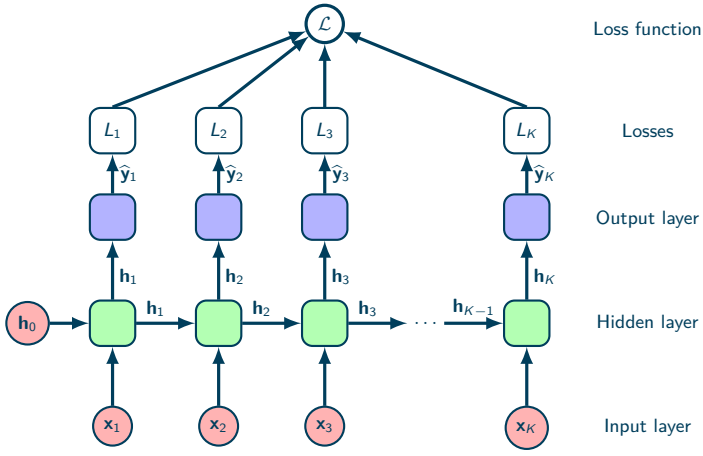
How many parameters (weights, biases) are there in an LSTM with input size n and cell-state size m?

- good sanity check for understanding
- good sanity check, but also check docs!
  - e.g.: two biases in Pytorch `https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html`

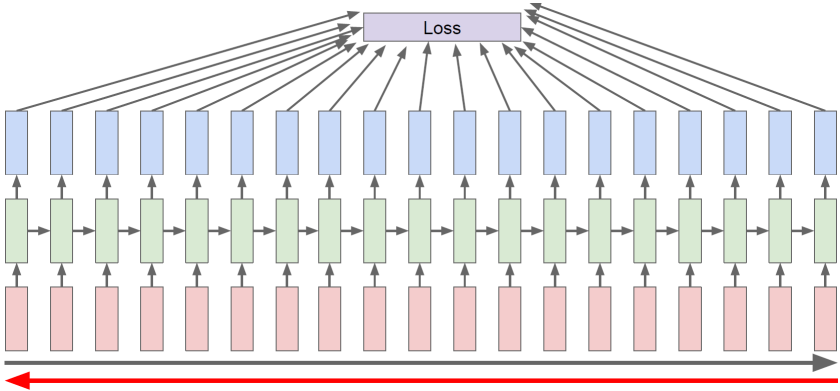
Outline

Outline

Loss


$\mathcal{L}$
Loss function
L_1
L_2
L_3
L_K
Losses
$\hat{\mathbf{y}}_1$
$\hat{\mathbf{y}}_2$
$\hat{\mathbf{y}}_3$
$\hat{\mathbf{y}}_K$
Output layer
$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_3$
$\mathbf{h}_K$
$\mathbf{h}_0$
$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_3$
$\cdots$
$\mathbf{h}_{K-1}$
Hidden layer
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_K$
Input layer

Backpropagation through time (BPTT)



Problem: one gradient step is too costly:

- cost of gradient for sequence of length K equivalent to cost of network with K layers,
- ... as it requires a pass through the entire sequence.

Backpropagation through time (BPTT): alternatives?

Say: compute gradient for sequence of length K.

Backpropagation through time (BPTT): alternatives?

Say: compute gradient for sequence of length K.

Naive approach: divide the sequence in s subsequences of length K/s.

$\rightarrow$ sensible and cheaper approach,

$\rightarrow$ but... blind to long temporal dependencies.

Backpropagation through time (BPTT): alternatives?

Say: compute gradient for sequence of length K.

Naive approach: divide the sequence in s subsequences of length K/s.

$\rightarrow$ sensible and cheaper approach,

$\rightarrow$ but... blind to long temporal dependencies.

Truncated backpropagation through time:

1. run RNN for k_1 steps forward (compute hidden states, outputs, losses)
2. run BPTT for k_2 steps backward (compute gradient wrt to losses for those k_2 steps),
3. repeat until reaching end of sequence (K).

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

- TBPTT(K,K):

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

- TBPTT(K,K): classical BPTT.

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

- TBPTT(K,K): classical BPTT.
- TBPTT(1,K):

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

- TBPTT(K,K): classical BPTT.
- TBPTT(1,K): timesteps processed one at a time followed by update of all timesteps seen so far.

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

- TBPTT(K,K): classical BPTT.
- TBPTT(1,K): timesteps processed one at a time followed by update of all timesteps seen so far.
- TBPTT(k_1,1):

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

- TBPTT(K,K): classical BPTT.
- TBPTT(1,K): timesteps processed one at a time followed by update of all timesteps seen so far.
- TBPTT(k_1,1): network likely does not have enough temporal context to learn.

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

- TBPTT(K,K): classical BPTT.
- TBPTT(1,K): timesteps processed one at a time followed by update of all timesteps seen so far.
- TBPTT(k_1,1): network likely does not have enough temporal context to learn.
- TBPTT(k_1,k_2), with $k_1 < k_2 < K$:

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

- TBPTT(K,K): classical BPTT.
- TBPTT(1,K): timesteps processed one at a time followed by update of all timesteps seen so far.
- TBPTT(k_1,1): network likely does not have enough temporal context to learn.
- TBPTT(k_1,k_2), with $k_1 < k_2 < K$: multiple updates are performed per sequence.

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

- TBPTT(K,K): classical BPTT.
- TBPTT(1,K): timesteps processed one at a time followed by update of all timesteps seen so far.
- TBPTT(k_1,1): network likely does not have enough temporal context to learn.
- TBPTT(k_1,k_2), with $k_1 < k_2 < K$: multiple updates are performed per sequence.
- TBPTT(k_1,k_2), with $k_1 = k_2$:

Truncated backpropagation through time (TBPTT)

Examples of TBPTT(k_1, k_2):

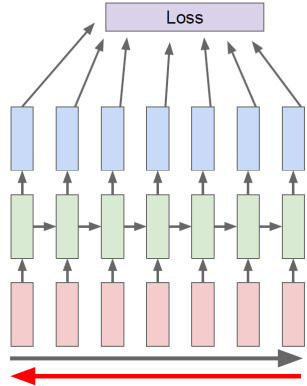
- TBPTT(K,K): classical BPTT.
- TBPTT(1,K): timesteps processed one at a time followed by update of all timesteps seen so far.
- TBPTT(k_1,1): network likely does not have enough temporal context to learn.
- TBPTT(k_1,k_2), with $k_1 < k_2 < K$: multiple updates are performed per sequence.
- TBPTT(k_1,k_2), with $k_1 = k_2$: a common configuration (fixed number of timesteps are used for both forward and backward-pass timesteps).

TBPTT can help:

- lower memory pressure,
- better GPU efficiency,
- faster training (depends on quality of gradient information),
- parameters might be updated multiple times within a single long sequence.

Truncated backpropagation through time (TBPTT)

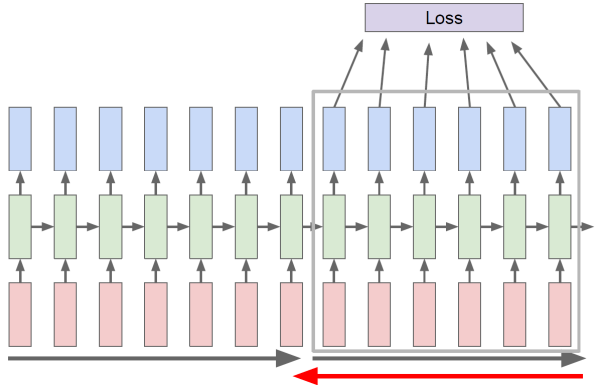
Example with $k_1 = k_2$:



Choose a small number of steps and back-propagate only onto these data.

Truncated backpropagation through time (TBPTT)

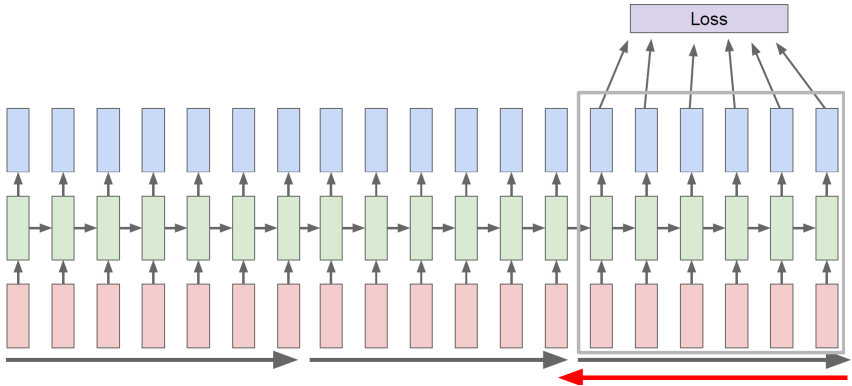
Example with $k_1 = k_2$:



Propagate the weights and use backpropagation on the second batch of data.

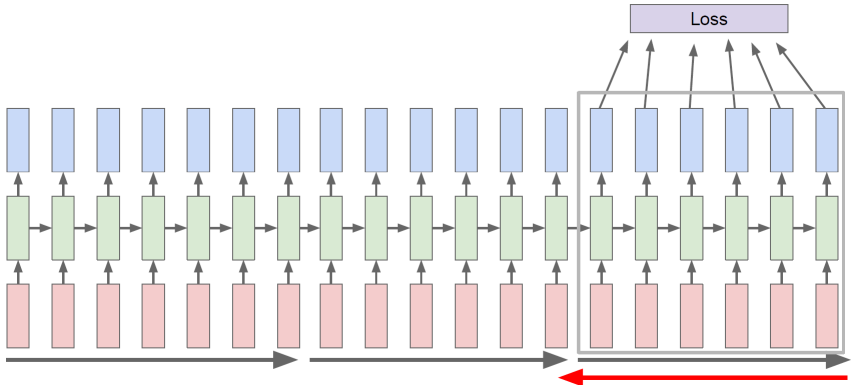
Truncated backpropagation through time (TBPTT)

Example with $k_1 = k_2$:



Truncated backpropagation through time (TBPTT)

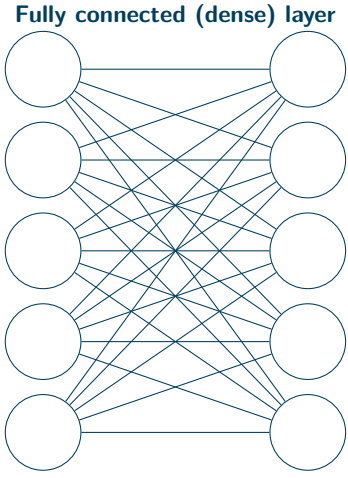
Example with $k_1 = k_2$:



More details [*Training recurrent neural networks*, Sutskever 2013].

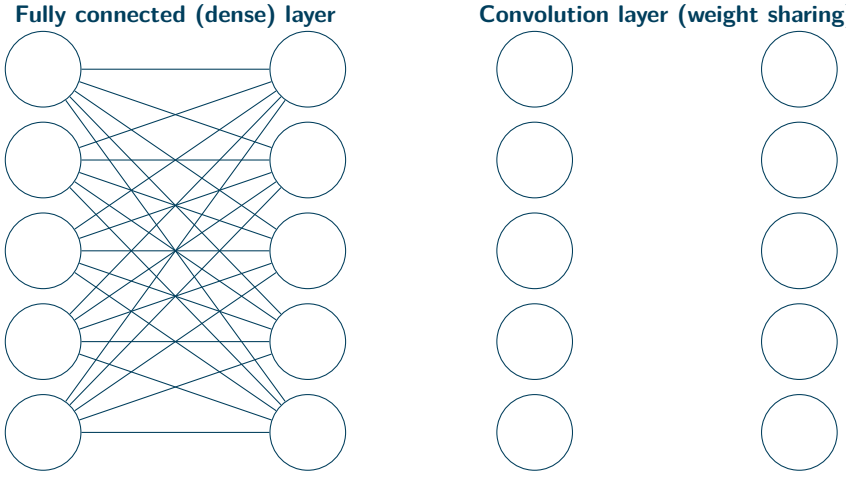
Outline

1-D convolutional layers



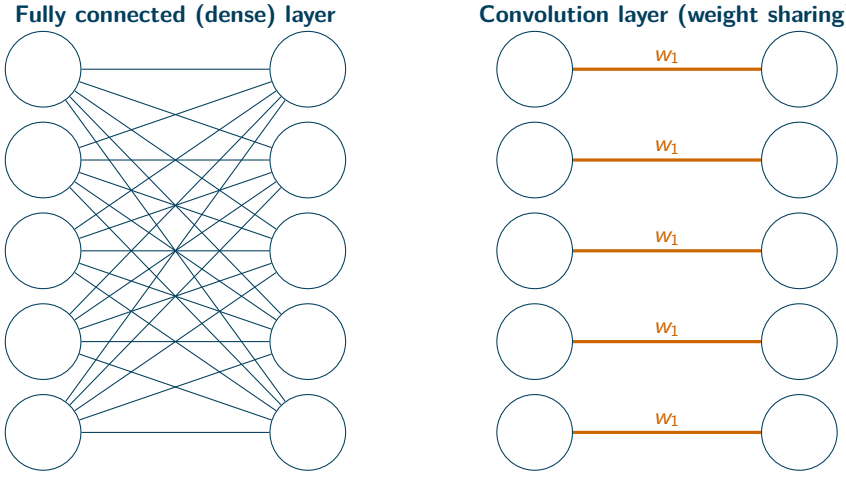
- Left: all outputs connected to all inputs. Connectivity is **dense** (dense layer).
- Right: convolution with kernel (width 3). Connectivity is **sparse**.

1-D convolutional layers



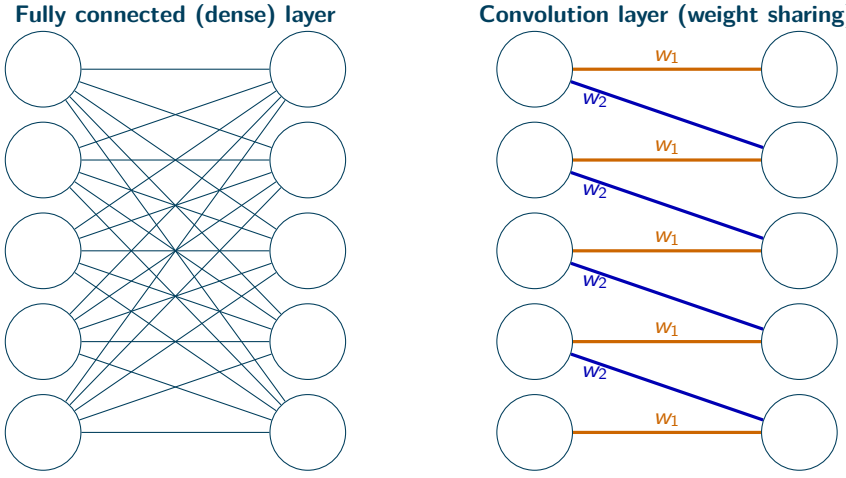
- Left: all outputs connected to all inputs. Connectivity is **dense** (dense layer).
- Right: convolution with kernel (width 3). Connectivity is **sparse**.

1-D convolutional layers



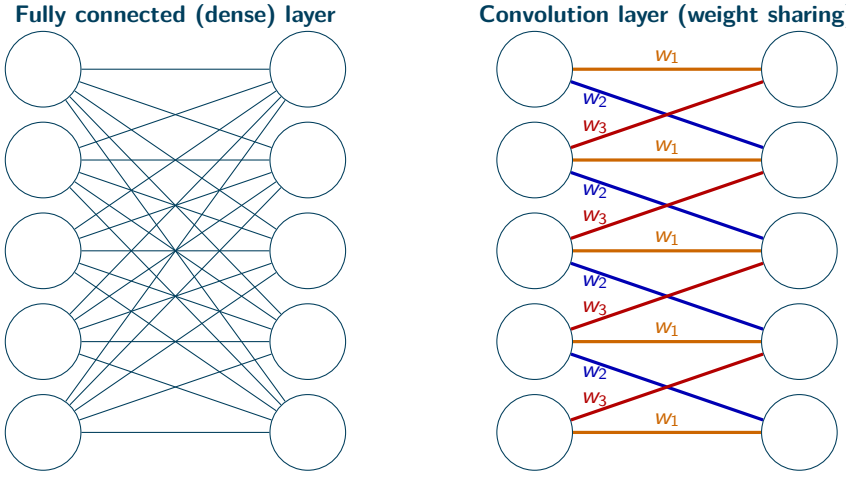
- Left: all outputs connected to all inputs. Connectivity is **dense** (dense layer).
- Right: convolution with kernel (width 3). Connectivity is **sparse**.

1-D convolutional layers



- Left: all outputs connected to all inputs. Connectivity is **dense** (dense layer).
- Right: convolution with kernel (width 3). Connectivity is **sparse**.

1-D convolutional layers



- Left: all outputs connected to all inputs. Connectivity is **dense** (dense layer).
- Right: convolution with kernel (width 3). Connectivity is **sparse**.

WaveNet

Through the use of dilated causal convolutions, WaveNet can model long-range temporal dependencies by increasing it's receptive field of input.

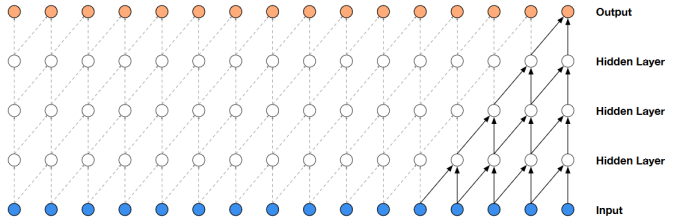


Figure: (Causal) convolutions

WaveNet

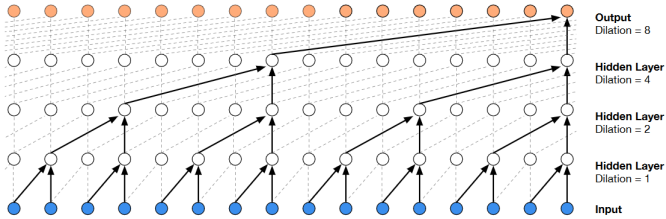


Figure: Dilated convolutions

`https://deepmind.com/blog/wavenet-generative-model-raw-audio/`

["WaveNet: A generative model for raw audio.", Van Den Oord et al. 2016].

WaveNet is CNN capable of generating speech, using dilated convolutions.

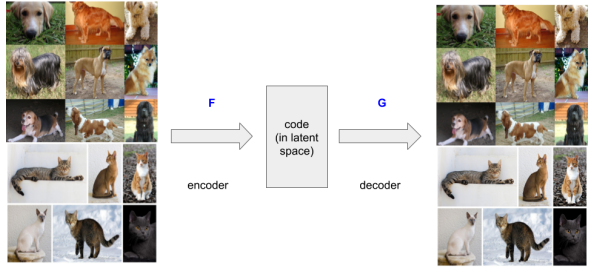
Outline

Outline

Autoencoders

Autoencoders (often used for visualization in low-dimension)

- learn compressed data representation in *unsupervised manner*,
- can be seen as self-supervised learning: the data provides the supervision!
- An autoencoder maps a space to itself and is [close to] the identity on the data.



Courtesy to `https://dataflowr.github.io/website/`.

Encoder F: original space $\to$ latent space. Decoder G: latent space $\to$ original space.
Dimension reduction if latent space is lower dimensional.

Example: baby autoencoder

```python
class AutoEncoder(nn.Module):
    def __init__(self, input_dim, encoding_dim):
        super(AutoEncoder, self).__init__()
        self.encoder = nn.Linear(input_dim, encoding_dim)
        self.decoder = nn.Linear(encoding_dim, input_dim)

    def forward(self, x):
        encoded = F.relu(self.encoder(x))
        decoded = self.decoder(encoded)
        return decoded
```

Example: https://github.com/dataflowr/notebooks/blob/master/Module9/09_AE_NoisyAE.ipynb

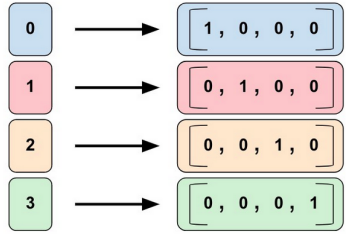
Outline

One-hot encoding

What to do when we face *symbolic* features?

- colors?
- words?

Encoding colors:



Courtesy to `https://dataflowr.github.io`.

Pro/cons of one-hot encoding?

- Each axis have a meaning,
- distances are more meaningful than categorical numbers ...
- discrete, sparse, high-dimensional representation (number of symbols).

Embeddings

Instead of one-hot encoded **features**, use **continuous vector representation of the discrete elements**.

More formally: let $W \in \mathbb{R}^{d \times n}$ (n is number of categories, d embedding dimension):

$$\text{embedding}(x) = W \text{ one-hot}(x).$$

Usually: W randomly initialized and then trained.

- Continuous and dense representation.
- Can represent a huge vocabulary in low dimension.
- Axis have no meaning a priori but once trained, embedding metric can capture semantic distance.

Outline

Example: word embedding

Example: represent words as elements of $\mathbb{R}^d$ (word embedding).
Interpretation: describe words through simple characteristics.

A (dummy) manual word embedding:

Features $\rightarrow$ Categories $\downarrow$	Gender	Royal	Age	Food	...
Man	-1.00	0.01	0.03	-0.01	
Woman	1.00	0.02	0.03	-0.02	
King	-0.95	0.99	0.12	-0.01	
Queen	0.97	0.96	0.06	-0.01	
Apple	0.00	-0.01	0.01	.98	
Orange	0.01	0.02	0.01	.99	

Example: word embedding

Example: represent words as elements of $\mathbb{R}^d$ (word embedding).
Interpretation: describe words through simple characteristics.

A (dummy) manual word embedding:

Features $\rightarrow$ Categories $\downarrow$	Gender	Royal	Age	Food	...
Man	-1.00	0.01	0.03	-0.01	
Woman	1.00	0.02	0.03	-0.02	
King	-0.95	0.99	0.12	-0.01	
Queen	0.97	0.96	0.06	-0.01	
Apple	0.00	-0.01	0.01	.98	
Orange	0.01	0.02	0.01	.99	

Many other possible features: is this a verb? is it big? etc.

Example: word embedding

Example: represent words as elements of $\mathbb{R}^d$ (word embedding).
Interpretation: describe words through simple characteristics.

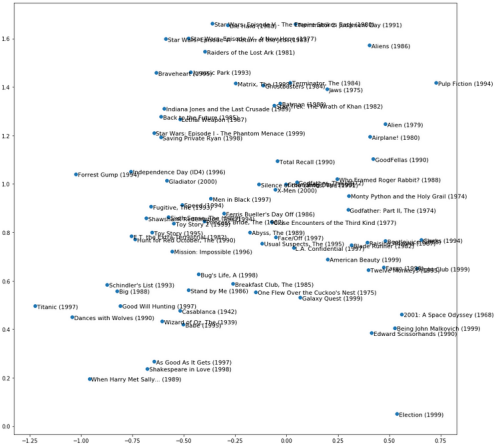
A (dummy) manual word embedding:

Features $\rightarrow$ Categories $\downarrow$	Gender	Royal	Age	Food	...
Man	-1.00	0.01	0.03	-0.01	
Woman	1.00	0.02	0.03	-0.02	
King	-0.95	0.99	0.12	-0.01	
Queen	0.97	0.96	0.06	-0.01	
Apple	0.00	-0.01	0.01	.98	
Orange	0.01	0.02	0.01	.99	

Many other possible features: is this a verb? is it big? etc.

In practice, the embedding is trained.

Example: 2D movie embedding



Courtesy to `https://dataflowr.github.io`.

Word embedding

Word embedding: represent each word as an element of $\mathbb{R}^d$ (continuous representation of discrete elements).

Different ways of creating word vectors:

- Skip-gram: predict surrounding words of a given word.
- Continuous bag of words (CBOW): predict a word given surrounding words.

See, e.g., `https://datascientest.com/nlp-word-embedding-word2vec`.

Different types of algorithms:

- Word2vec

  ["Distributed representations of words and phrases and their compositionality", Tomas Mikolov, Sutskever, et al. 2013]
- Glove

  ["Glove: Global vectors for word representation", Pennington et al. 2014]
- FastText

  ["Bag of tricks for efficient text classification", Joulin et al. 2016]

Poll!

https://app.wooclap.com/PETIEM?from=instruction-slide

Outline

Pipeline for neural networks

Pipeline for neural networks

- Step 1: Preprocessing the data (substract mean, divide by standard deviation).

Pipeline for neural networks

- Step 1: Preprocessing the data (substract mean, divide by standard deviation).
- Step 2: Choose the architecture (number of layers, number of nodes per layer)

Pipeline for neural networks

- Step 1: Preprocessing the data (substract mean, divide by standard deviation).
- Step 2: Choose the architecture (number of layers, number of nodes per layer)
- Step 3:
  1. First, run the network and see if the loss is reasonable (compare with dumb classifier: uniform for classification, mean for regression)
  2. Add some regularization and check that the error on the training set increases.
  3. On a small portion of data, make sure you can overfit when turning down the regularization.
  4. Find the best learning rate
     1. The error does not change too much $\rightarrow$ learning rate too small
     2. The error explodes, NaN $\rightarrow$ learning rate too high
     3. Find a rough range $[10^{-5}, 10^{-3}]$.

$\rightarrow$ ... and use available ressources! (pre-trained models, etc.)

What did we do (today)?

- RNNs: base architectures, issues.
- LSTMs/GRUs: to mitigate vanishing/exploding gradients.
- Backpropagation through time.
- Encoders/decoders (high-level).
- Embeddings.

What did we do (today)?

- RNNs: base architectures, issues.
- LSTMs/GRUs: to mitigate vanishing/exploding gradients.
- Backpropagation through time.
- Encoders/decoders (high-level).
- Embeddings.

This afternoon: embeddings + RNNs.

What did we do (in this course)?

Over the last classes, we spend time on techniques for limiting the effect of overfitting, improving the training capabilities, and of making our models more adapted to specific tasks (images, time dependence).

What did we do (in this course)?

Over the last classes, we spend time on techniques for limiting the effect of overfitting, improving the training capabilities, and of making our models more adapted to specific tasks (images, time dependence).

Among others:

- ingredients of neural nets: activations, number of layers, etc.
- weight sharing (both CNN and RNN),
- regularization techniques (early stopping, dropout, explicit regularization, etc.),
- computing gradients: backpropagation,
- improving scaling: batch normalization,
- etc.

What did we do (in this course)?

Over the last classes, we spend time on techniques for limiting the effect of overfitting, improving the training capabilities, and of making our models more adapted to specific tasks (images, time dependence).

Among others:

- ingredients of neural nets: activations, number of layers, etc.
- weight sharing (both CNN and RNN),
- regularization techniques (early stopping, dropout, explicit regularization, etc.),
- computing gradients: backpropagation,
- improving scaling: batch normalization,
- etc.

Many ways to go further, among them:

- explore network architectures (e.g., transformers: attention mechanisms & parallelism, etc.),
- explore optimization/regularization techniques for learning (e.g., layer normalization, dropout entire layers),
- explore embedding techniques, encoder/decoders, etc.
- unsupervised learning, generative models, etc.

... but also classical "shallow"/"symbolic" learning techniques!

Exams/questions

- Wooclap (at your own pace): will be available from Slack.
- Old exam will be available on moodle (for types of questions).
  - ⚠ we did not insist on exactly the same points this year.
- Hint: review elements from the labs!
- Examples of classical lab mistakes:
  - in backprop: forget zero grad
  - due to object-oriented programming: re-train an already-trained model (instead of training from scratch)
  - loss is negative (why?).

Questions?

[Cho+14] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[Jou+16] Armand Joulin et al. "Bag of tricks for efficient text classification". In: *arXiv preprint arXiv:1607.01759* (2016).

[JZS15] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. "An empirical exploration of recurrent network architectures". In: *International conference on machine learning*. 2015.

[LJH15] Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. "A simple way to initialize recurrent networks of rectified linear units". In: *arXiv preprint arXiv:1504.00941* (2015).

[Mik+11] Tomáš Mikolov et al. "Empirical evaluation and combination of advanced language modeling techniques". In: *Twelfth Annual Conference of the International Speech Communication Association*. 2011.

[Mik+13] Tomas Mikolov, Ilya Sutskever, et al. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems*. 2013, pp. 3111–3119.

[Mik+14] Tomas Mikolov, Armand Joulin, et al. "Learning longer memory in recurrent neural networks". In: *arXiv preprint arXiv:1412.7753* (2014).

[PMB13] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks”. In: *International conference on machine learning*. 2013.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[Sal+17] Hojjat Salehinejad et al. “Recent Advances in Recurrent Neural Networks”. In: *arXiv preprint arXiv:1801.01078* (2017).

[Sut13] Ilya Sutskever. *Training recurrent neural networks*. University of Toronto Toronto, ON, Canada, 2013.

[TV15] Sachin S Talathi and Aniket Vartak. “Improving performance of recurrent neural network with relu nonlinearity”. In: *arXiv preprint arXiv:1511.03771* (2015).

[Van+16] Aäron Van Den Oord et al. “WaveNet: A generative model for raw audio.”. In: *SSW*. 2016, p. 125.