Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

data science

CAIDAS

Chapter 5

Sentences as a sequence

Recurrent Neural Networks

Content of this Chapter

1. Recurrent Neural Networks in Theory
    1. Vanilla RNNs
    2. Backpropagation Through Time (BPTT)
    3. The Long Term: LSTMs and Friends
2. RNNs in Practice

5.1 Recurrent Neural Networks in Theory

- Vanilla RNNs
- Backpropagation Through Time (BPTT)
- LSTM and friends

Character-based Text Generation

- Idea:
  - Given a sequence of characters $d = (d_1, d_2, \dots, d_n)$ of length n. Learn a model M that predicts the next character in the sequence, d_{n+1}
- Example:

  „The Dursleys had everything they wanted, ...“
  - M(„T“) = „h“
  - M(„Th“) = „e“
  - M(„The Dursleys ha“) = „d“

⇒ Train as a classification problem with samples $((d_1, d_2, \dots, d_k), d_{k+1})$

How can we process sequences?

Character-based Text Generation (FCN)

- Remember a fully connected (FCN) layer is defined as:

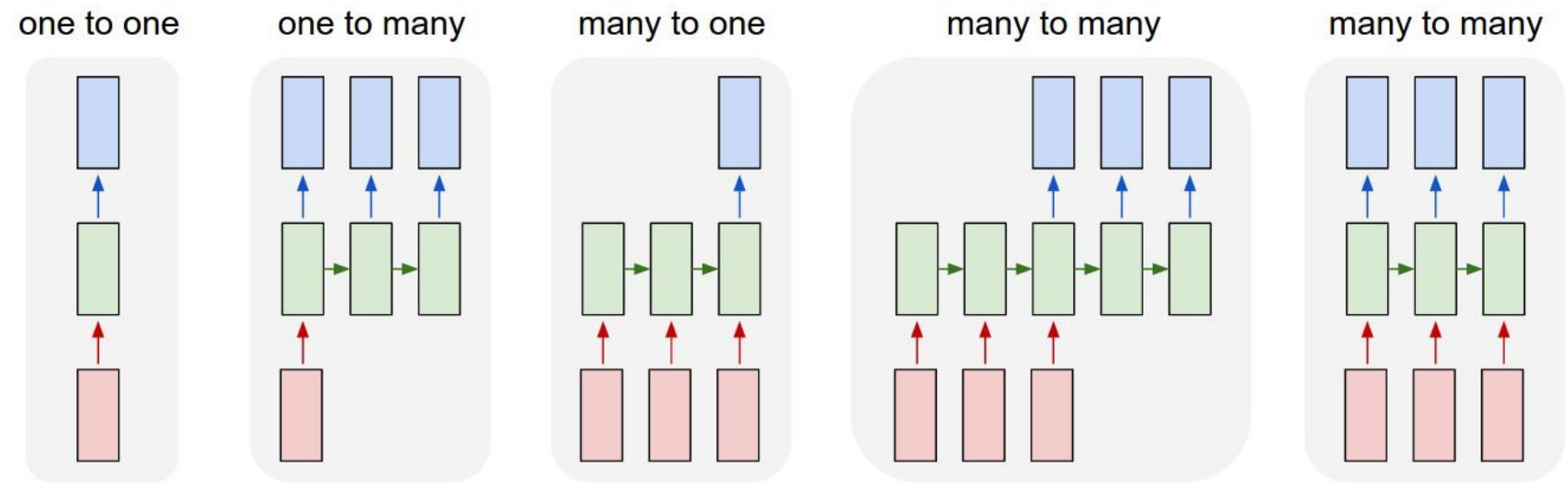
$$y = f(Wx + b)$$

- The layer takes only one input x, but we have a sequence of characters!

- Possible Solutions:
  - Concatenate the individual character vectors: $\mathrm{x} = concat(d_1, d_2, \ldots, d_k)$
    - We already saw that this leads to large weight matrices!
    - Furthermore, we can not process sequences of arbitrary length ☹
  - Average over the individual character vectors: $\mathrm{x} = \frac{1}{k}\sum_{1 \le i \le k} d_i$
    - Input vector can be kept small
    - Sequences of arbitrary length are possible
    - All structure is lost ☹
  - ...?

→ How to preserve structure, yet process sequences of arbitrary length?

Overview: Types of Neural Networks for all Purposes



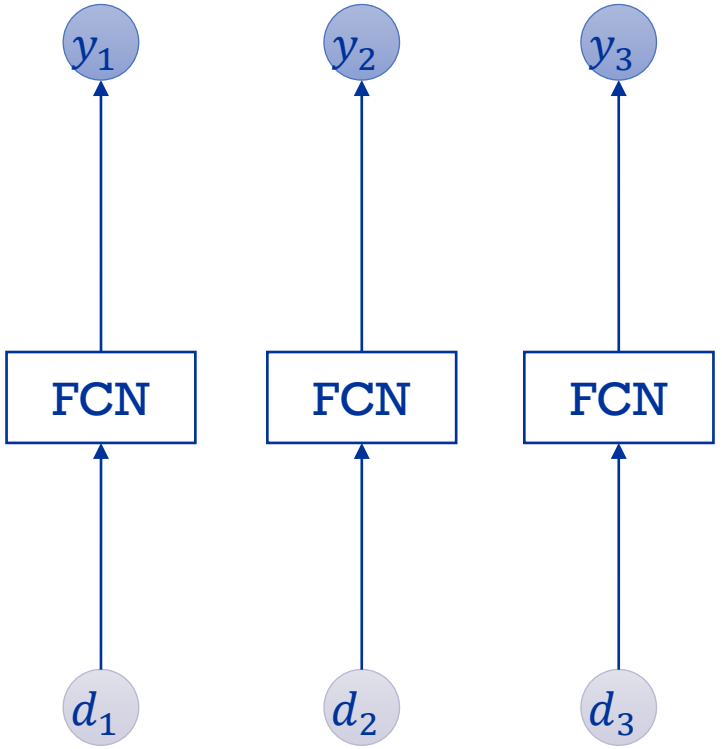
Fully Connected Network

Network mapping a sequence to one output

Network returning an output after each timestep of a sequence

Recurrent Neural Networks

Character-based Text Generation: From FCN to RNN



$$y_1 = f(W d_1 + b)$$
$$y_2 = f(W d_2 + b)$$
$$y_3 = f(W d_3 + b)$$

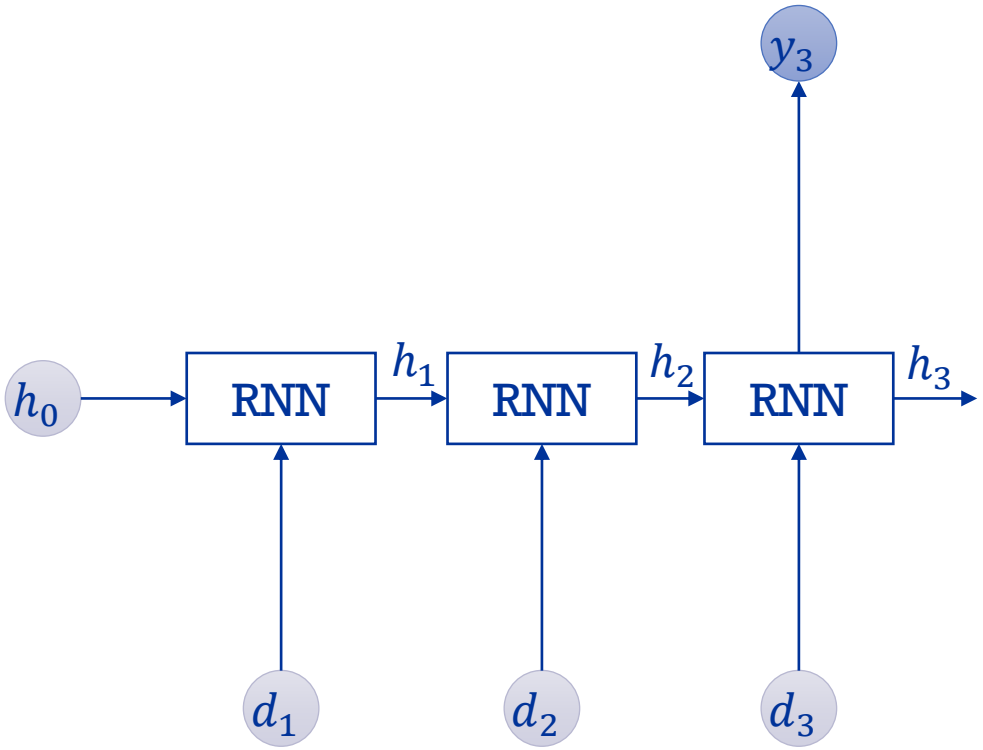
- Every character is considered individually.
- How do we preserve information about prior characters?

→ **Add connections between the networks!**

Character-based Text Generation: From FCN to RNN

Idea

- add a state h that is carried between inputs.
- Update state with current input
- extract output y from current state h
- **Share the weights between the timesteps!**



$$h_1 = \sigma_h(W_h d_1 + U_h h_0)$$

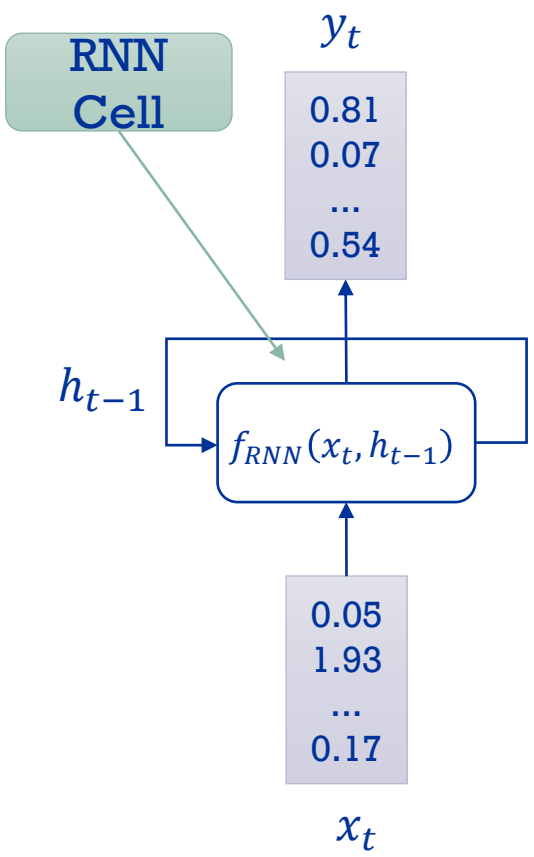
$$h_2 = \sigma_h(W_h d_2 + U_h h_1)$$

$$h_3 = \sigma_h(W_h d_3 + U_h h_2)$$

$$y_3 = \sigma_y(W_y h_3 + b_y)$$

This view is called an **unrolled RNN**

Vanilla RNN



$$h_t = f_{RNN}(x_t, h_{t-1}) = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$

- W_h: maps input x_t into internal state space
- U_h : extracts relevant information from prior state h_{t-1}
- b_h : bias. As usual, can be omitted using the bias trick
- σ_h : internal activation function

$$y_t = f_{out}(h_t) = \sigma_y(W_y h_t + b_y)$$

- W_y : maps state h_t into output space
- b_y : bias
- σ_y : output activation function

⟹ **Basically a fully connected layer**

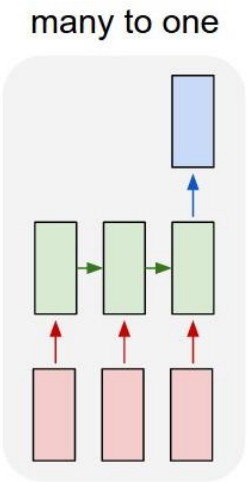
- Learnable parameters: W_h, U_h, W_y, b_h, b_y
- Initial state h_0 is commonly initialized as 0-vector

L., Elman Jeffrey: Finding Structure in Time. In: Cognitive Science, 14 (1990), Nr. 2, S. 179-211

Training Recurrent Neural Networks

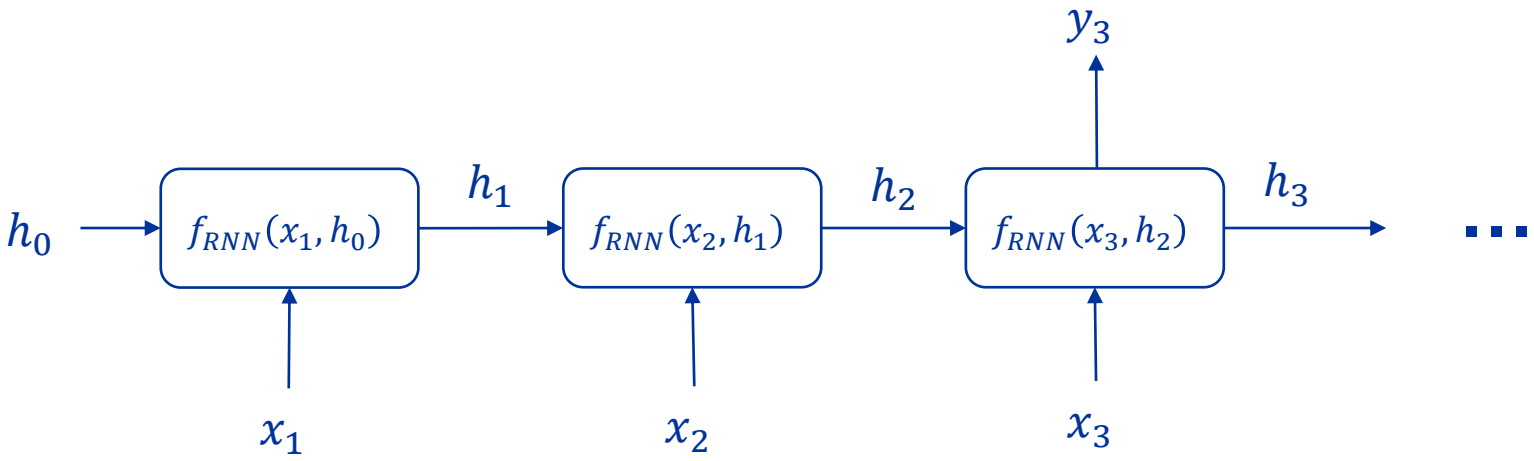
- How do we learn the parameters of the RNN?

→ Let‘s apply Backpropagation on the unrolled network!
  - Similar for all types mentioned above
  - Focus on many-to-one for „handy“ gradie



Backpropagation Through Time

Unrolling the RNN over Time

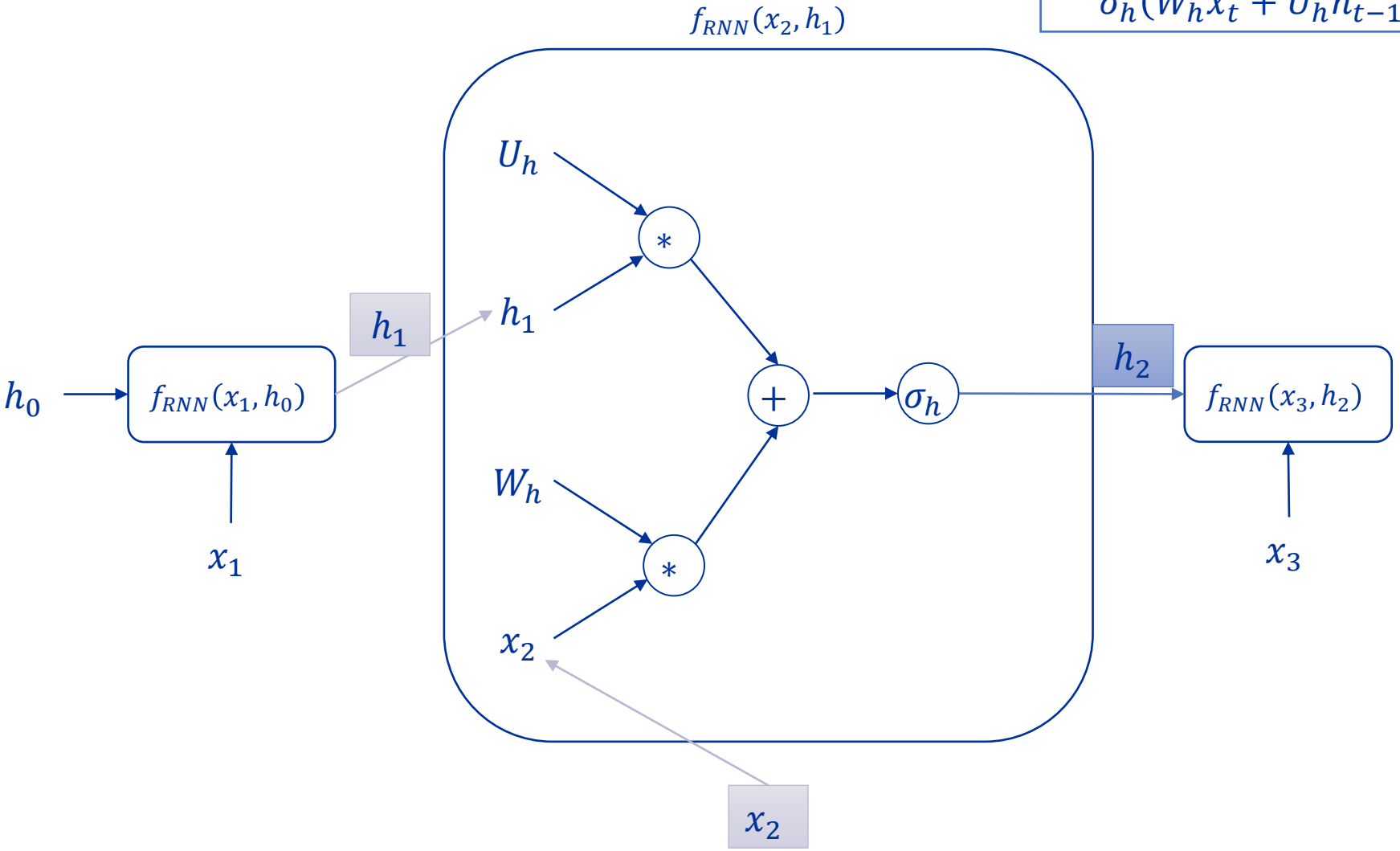


- Remember: An unrolled RNN is basically a feedforward network with some additional properties:
  - State h_t accumulates information about the sequence
  - All weights are shared between timesteps/layers
- Errors are backpropagated through h_t over all t steps of the sequence.

→ How does the gradient actually flow through an RNN cell?

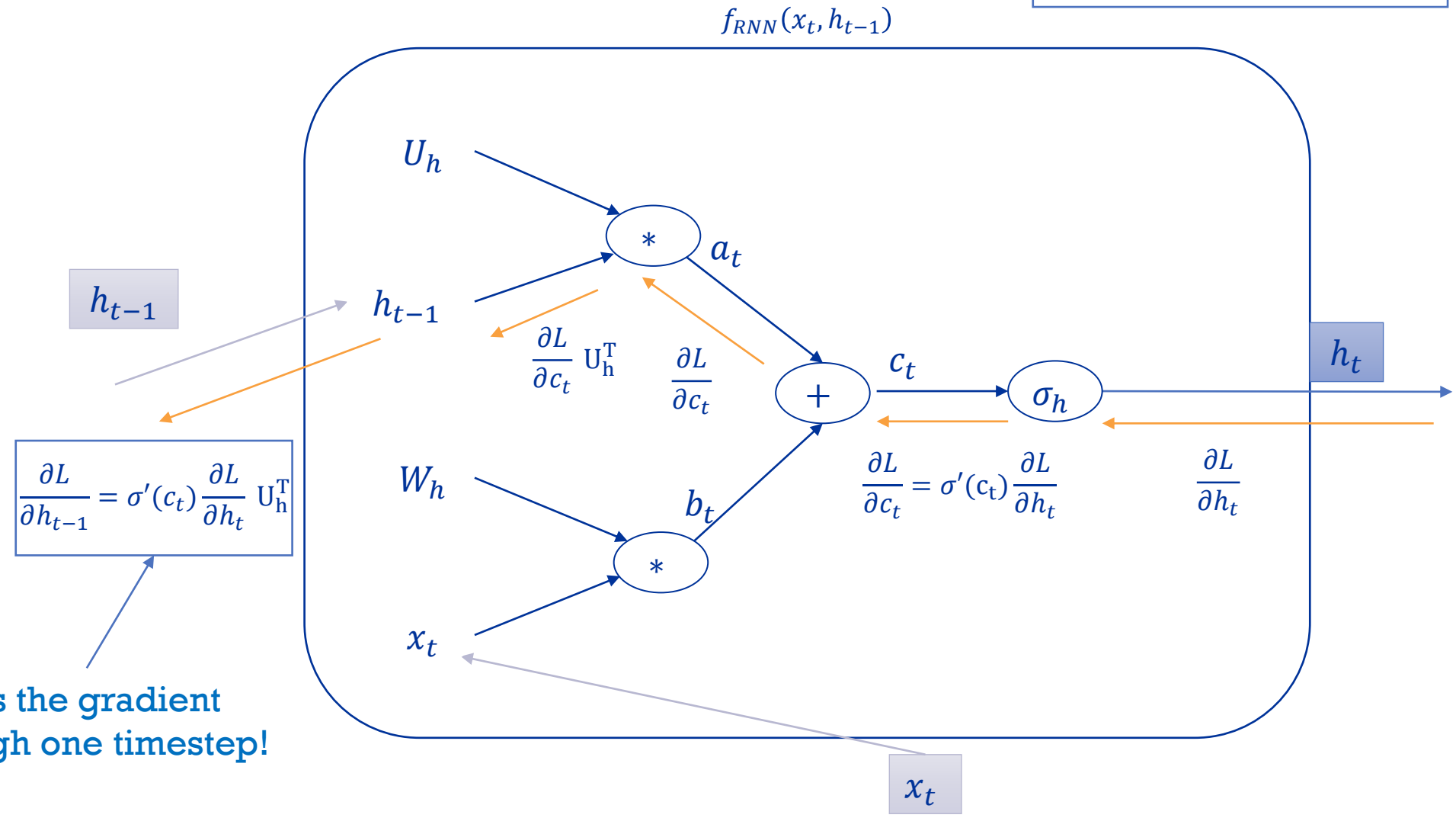
A look at the computational graph...

$$h_t = f_{RNN}(x_t, h_{t-1}) = \sigma_h(W_h x_t + U_h h_{t-1})$$



… and the gradient flow

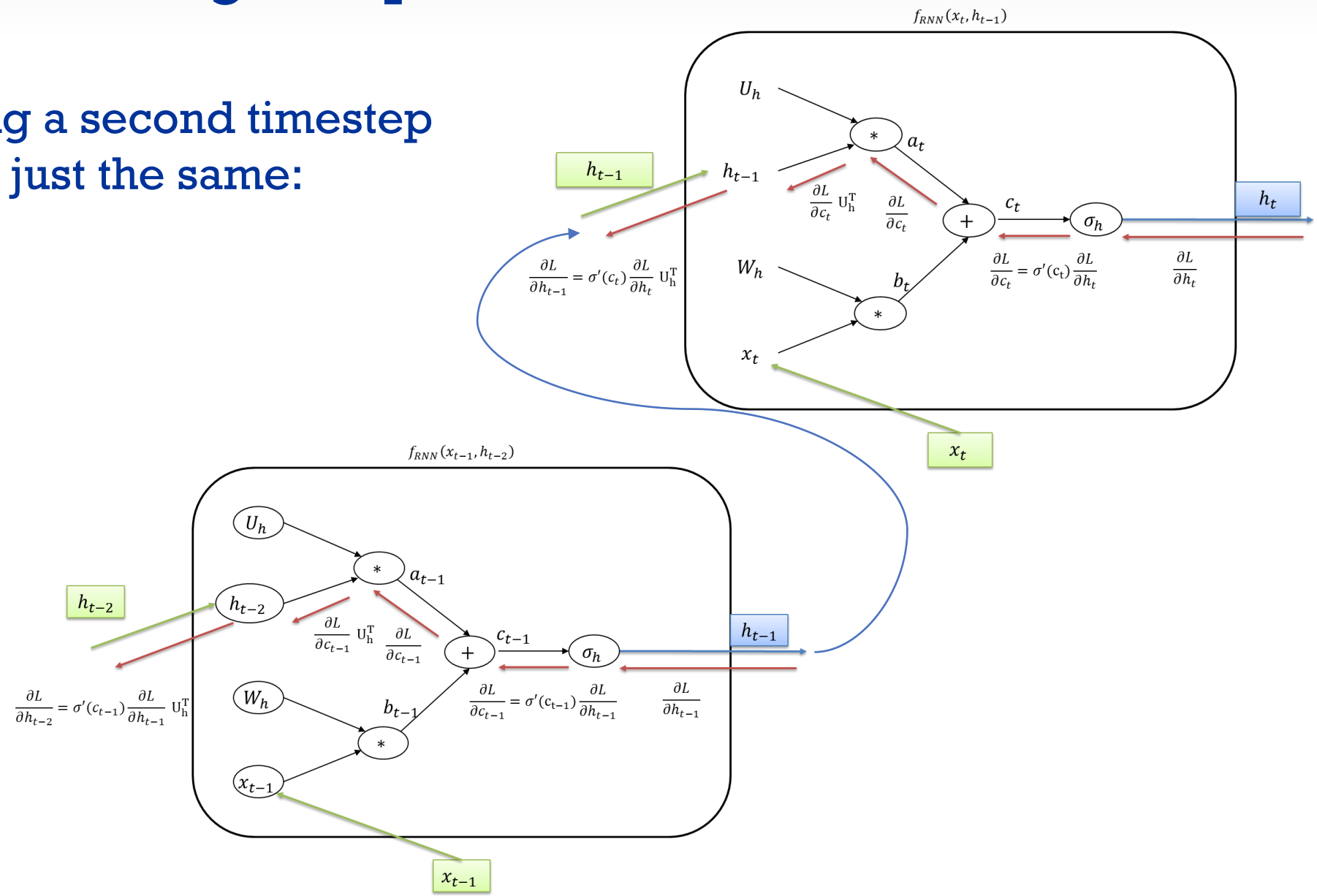
$$h_t = f_{RNN}(x_t, h_{t-1}) = \sigma_h(W_h x_t + U_h h_{t-1})$$



This is the gradient through one timestep!

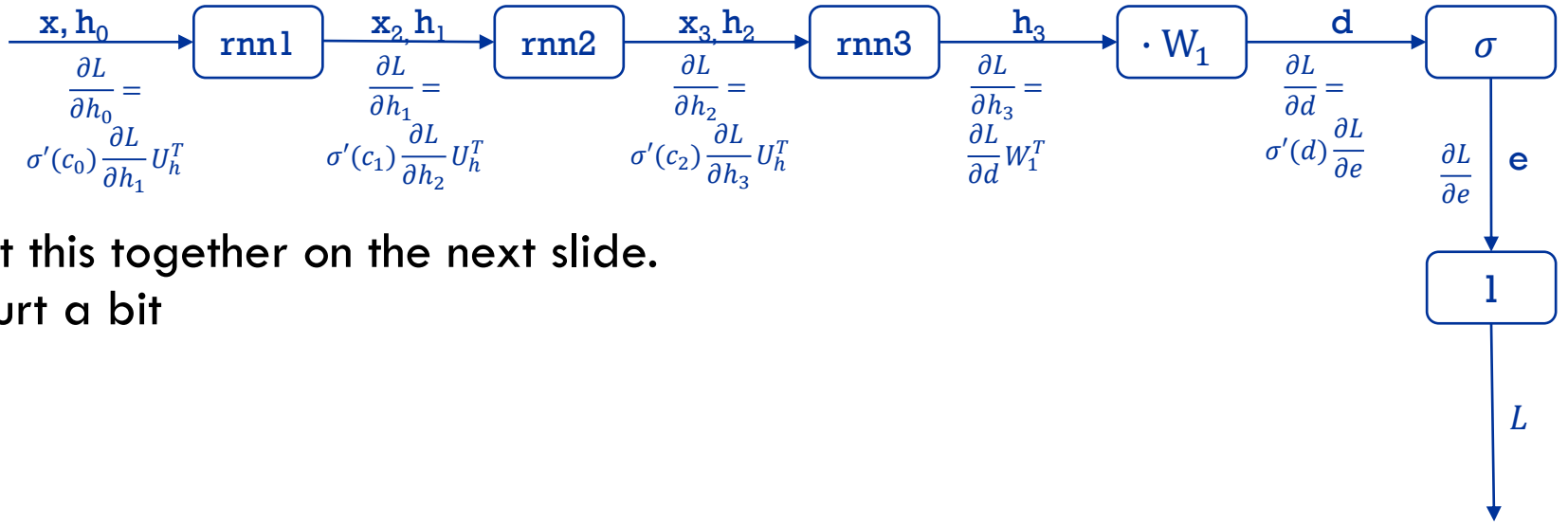
Going Deeper

Adding a second timestep works just the same:



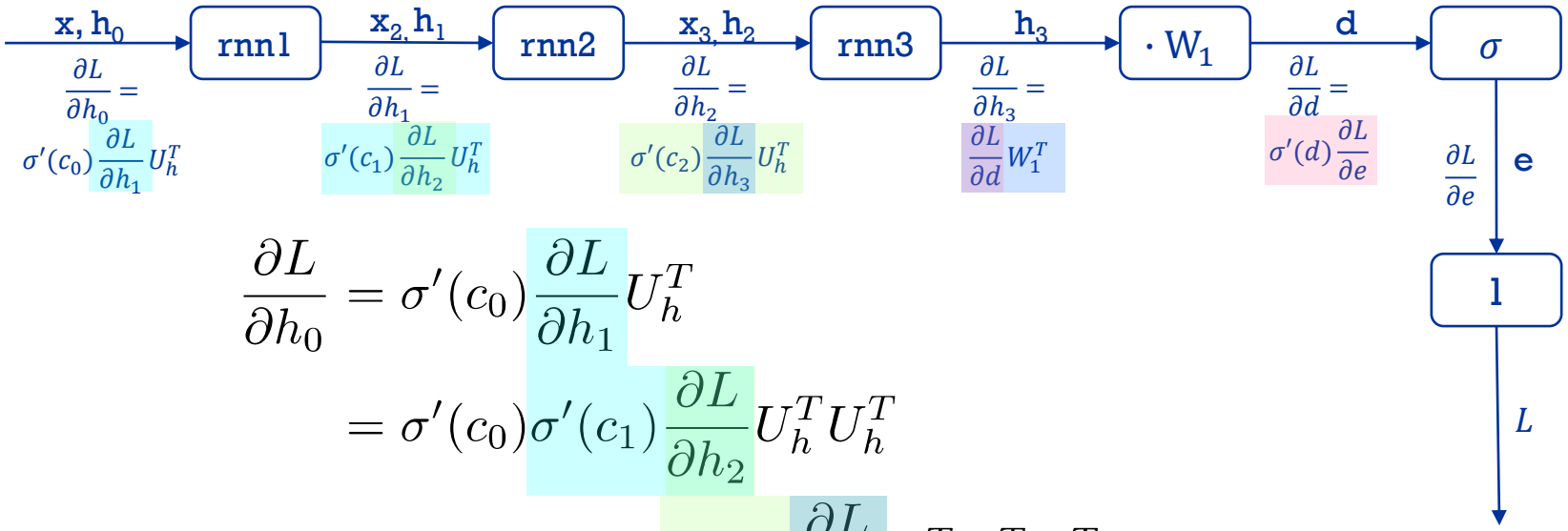
And for a Full Network

- BPTT over a network with
  - one RNN cell, unrolled over three timesteps
  - one fully connected layer
  - and a loss function at the end



- Now let‘s put this together on the next slide.
  It will only hurt a bit

Backpropagation Through Time



$$\frac{\partial L}{\partial h_0} = \sigma'(c_0)\frac{\partial L}{\partial h_1}U_h^T$$

$$= \sigma'(c_0)\sigma'(c_1)\frac{\partial L}{\partial h_2}U_h^T U_h^T$$

$$= \sigma'(c_0)\sigma'(c_1)\sigma'(c_2)\frac{\partial L}{\partial h_3}U_h^T U_h^T U_h^T$$

$$= \sigma'(x)\sigma'(c_1)\sigma'(c_2)\frac{\partial L}{\partial d}W_1^T U_h^T U_h^T U_h^T$$

$$= \sigma'(c_0)\sigma'(c_1)\sigma'(c_2)\sigma'(d)\frac{\partial L}{\partial e}W_1^T \boxed{U_h^T U_h^T U_h^T}$$

→ Matrix U_h^T appears once per timestep!

Exploding and Vanishing Gradients

$$\frac{\partial L}{\partial x} = \sigma'(x)\sigma'(h_1)\sigma'(h_2)\sigma'(d)\frac{\partial L}{\partial e}W_1^T \boxed{U_h^T U_h^T U_h^T}$$

- Common problem in RNNs: **vanishing** or **exploding** gradients

> **Vanishing Gradient**: Gradients from early timesteps get very small, having almost no influence on the update
> **Exploding Gradient**: Gradients from early timesteps get very large, causing the optimiser to overshoot its goal

- Why?
  → After backpropagation over k timesteps, the gradient contains $\left(U_h^T\right)^k$. This either gets very large or very small!
  (Similar to the scalar case, where $x^k \to \infty$ for $x > 1$ and $x^k \to 0$ for $x < 1$)

Combatting Exploding Gradients: Gradient Clipping

- Simple and effective remedy for exploding gradients:
  Gradient Clipping
  - During training, if a gradient gets larger than a predefined threshold, clip it to the threshold.

- Slightly more sophisticated alternative:
  Gradient Rescaling
  - Instead of clipping, rescale the gradients using their norm.

By Warren Long [CC BY 2.0 (https://creativecommons.org/licenses/by/2.0)], via Wikimedia Commons

Modifications of the Vanilla RNN

- Instead of Gradient Clipping/Rescaling, why not modify the RNN itself?

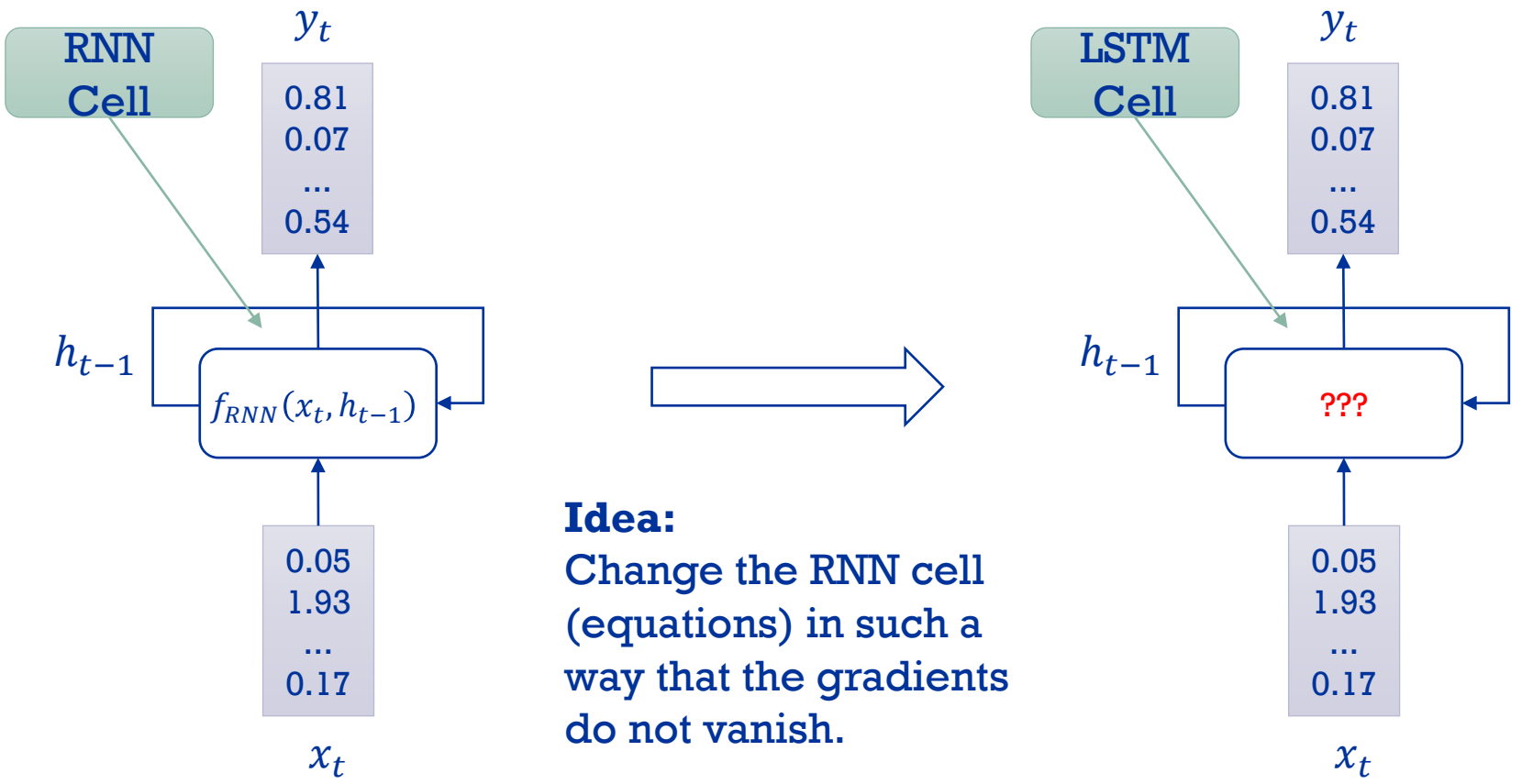
→Hochreiter and Schmidhuber (1997) did just that!

→**Long Short Term Memory** (LSTM) is a variant of RNNs that can better model long-term dependencies

Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9, 1735—1780.

LSTM and Friends

Long- Short Term Memory (LSTM)

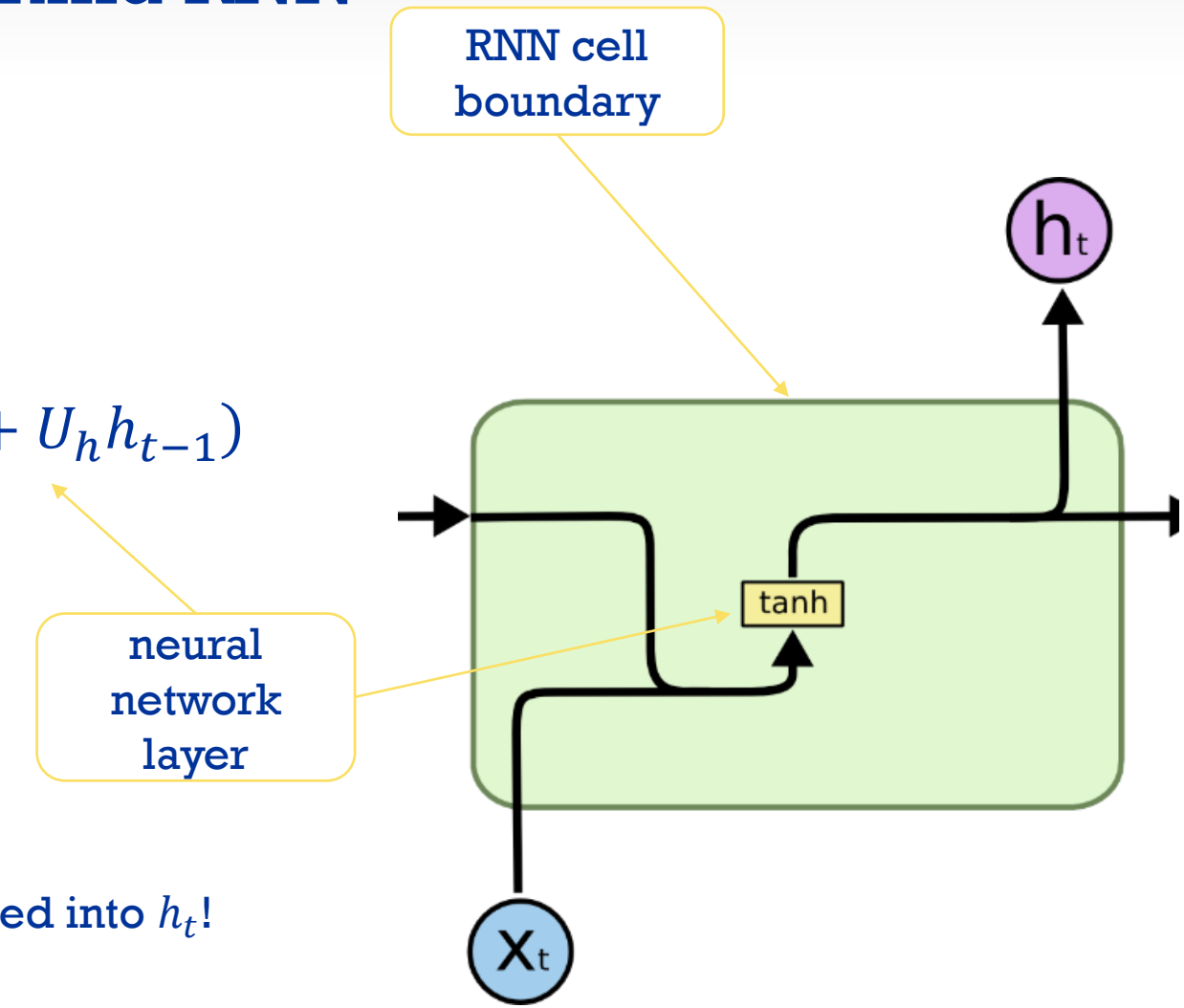


Idea:
Change the RNN cell (equations) in such a way that the gradients do not vanish.

The following slides contain illustrations taken from Christopher Olah's Blog:
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Recall Vanilla RNN

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1})$$



Note that h_{t-1} is transformed into h_t!

LSTM — Principles

- State changes are incremental: $c_{t+1} = c_t + \Delta c_{t+1}$
  - Contrast to Vanilla RNNs: Vanilla state change is a transformation/matrix multiplication!
- State updates are selective
  - We only want to write things to the state that help us
- State access is selective
  - We need a way to select the most relevant knowledge from the state
- Information can be deleted from the state
  - Some information may become out-dated and must make way for more important stuff

⇒ Each type of selectivity is modeled as a „gate“

LSTM — Gates

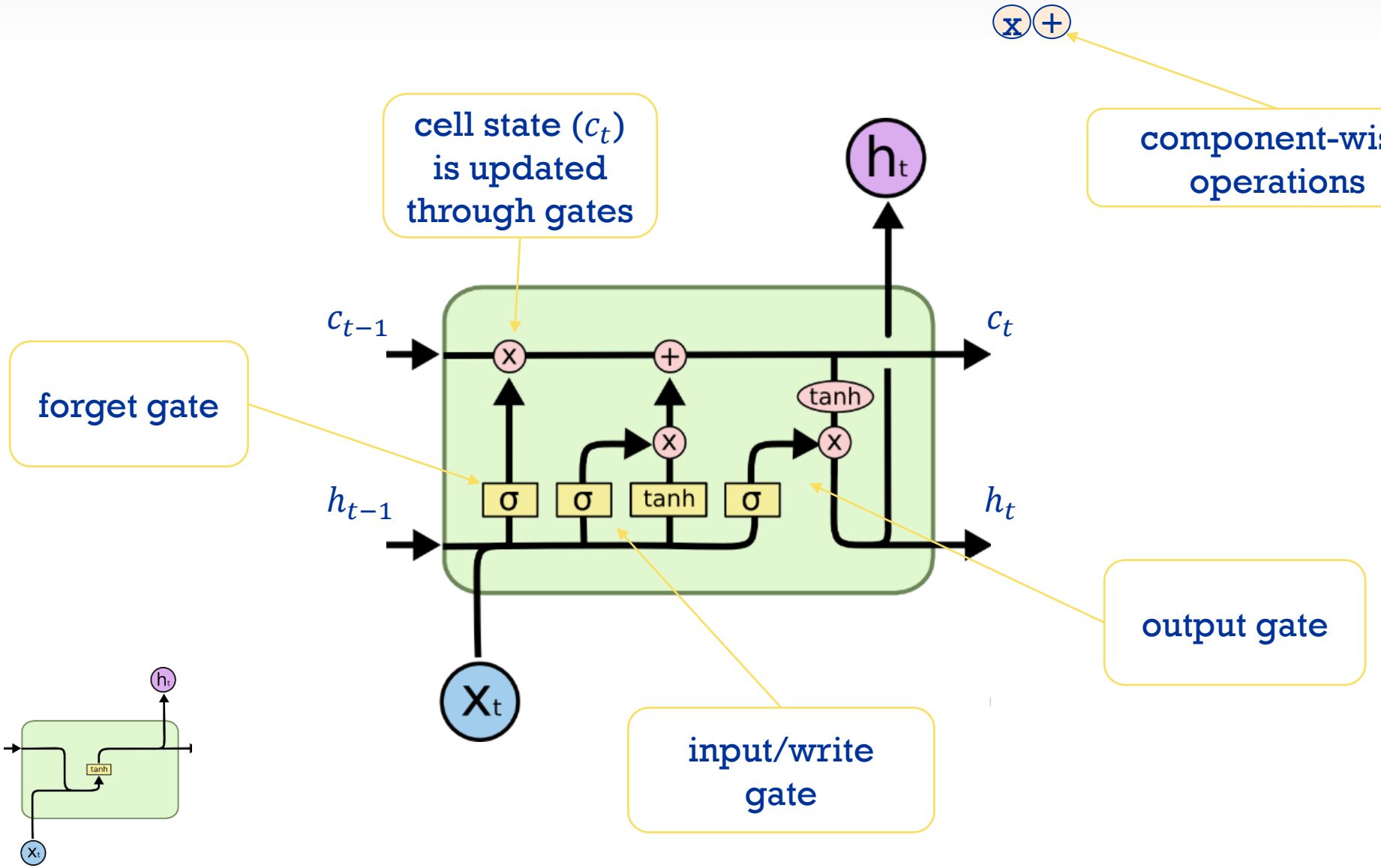
- Gates are modeled as layers similar to the Vanilla RNN
- They depend on the current input x_t and the last **shadow state** h_{t-1}
  - We will see that the LSTM cell carries two states
    1) the state (or memory) c_t
    2) the shadow state h_t
- All gates have the following form:

$$g(h_{t-1}, x_t) = \sigma(W x_t + U h_{t-1} + b)$$

Sigmoid activation function forces values in range [0, 1]

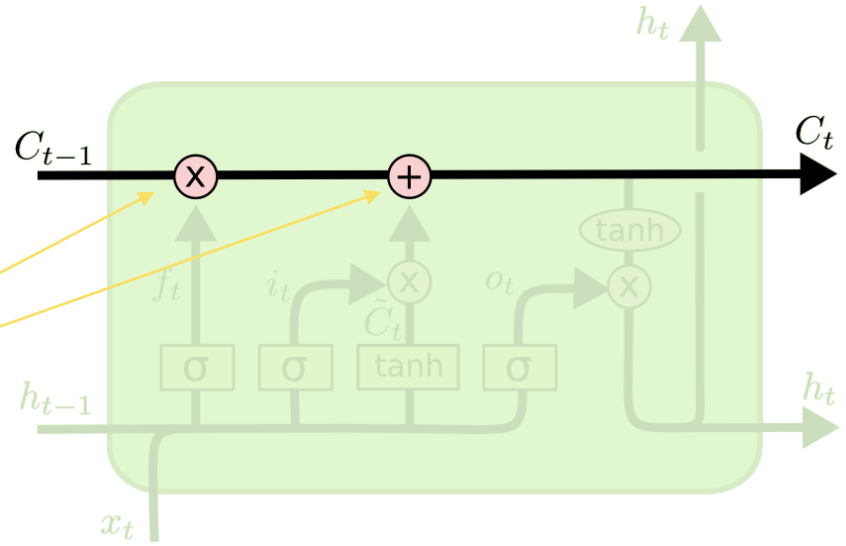
Interpretation (depending on the specific gate):
- 1 – keep all information, or read/write all information
- 0 – forget all information, or read/write none of that information

LSTM — Overview

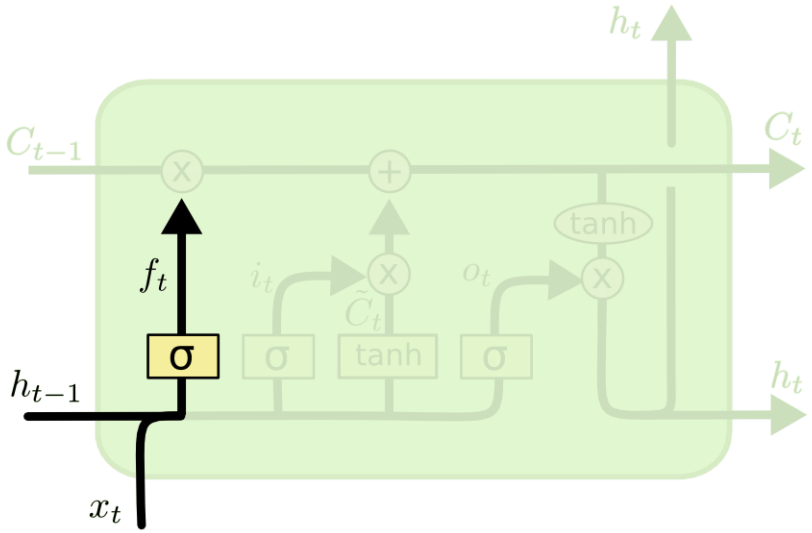


LSTM — Memory state

- The state or memory c_{t-1} is never transformed by matrix multiplication
- Both interactions are carried out component-wise



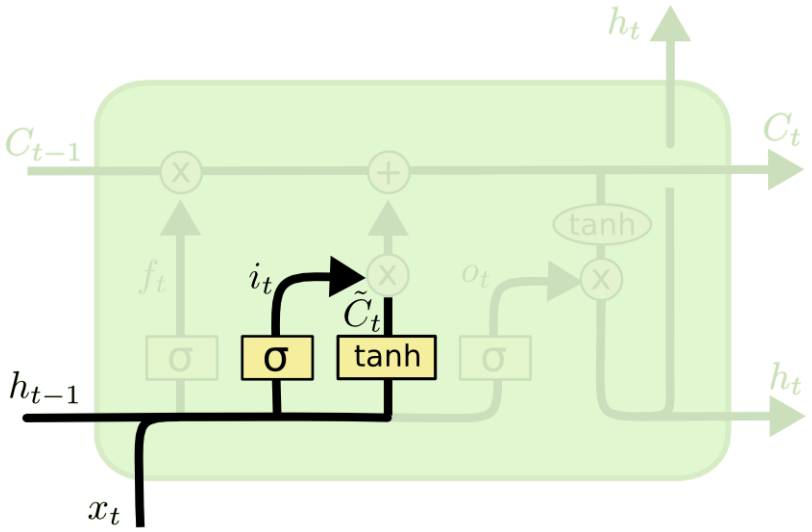
LSTM — Forget gate



$$f_t = \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right)$$

- The forget gate determines which information is outdated and can be discarded
- The update of the cell state c_t is done by component-wise multiplication, resulting in a scaling of the previous state

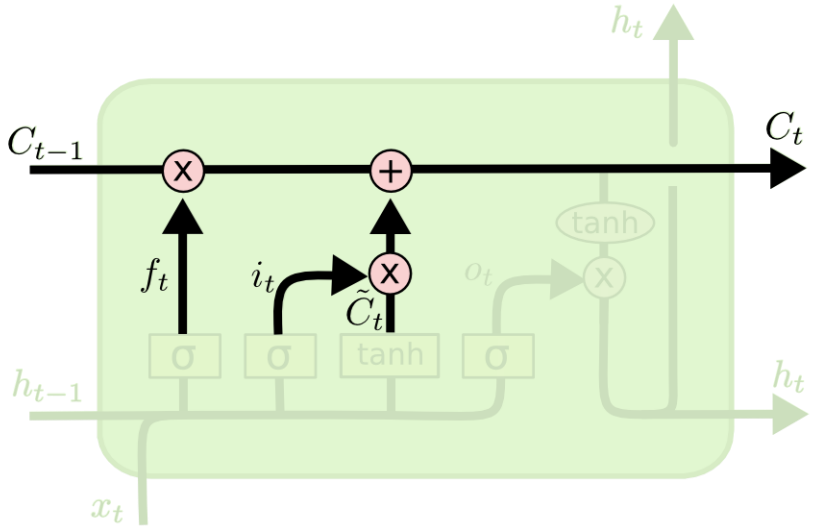
LSTM — Input gate



$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$\widetilde{c_t} = tanh(W_c x_t + U_c h_{t-1} + b_c)$$

- $\widetilde{c_t}$ consists of candidate values for updating the memory state
  - tanh is used instead of sigmoid, since it produces values in [-1, 1]
  - remember that we want to model incremental state changes, the value range enables us to add (> 0) or remove (< 0) information
- i_t is called the input gate and produces scaling factors (or weights) for the candidate values

LSTM — Memory state update

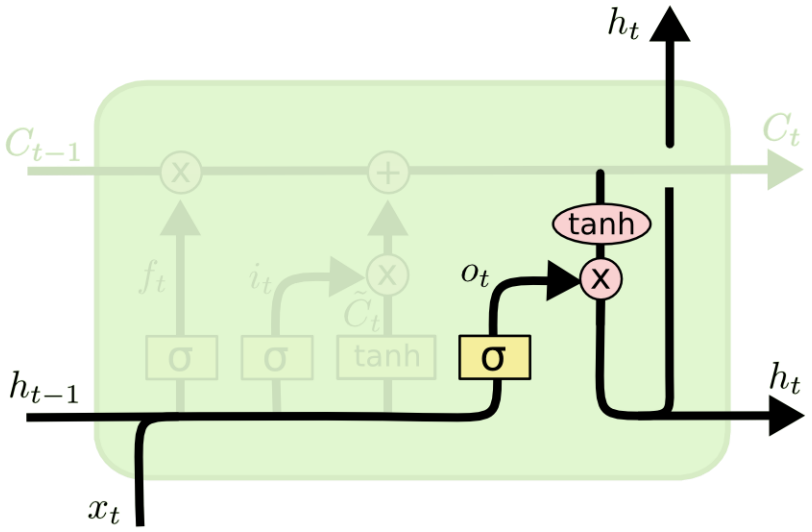


$$c_t = f_t \circ C_{t-1} + i_t \circ \tilde{c}_t$$

Component-wise multiplication

- The new memory state is computed by:
  - scaling the former memory state and thus „forgetting“ unnecessary information (forget gate)
  - adding information from the scaled candidate values (input/write gate)

LSTM — Output gate



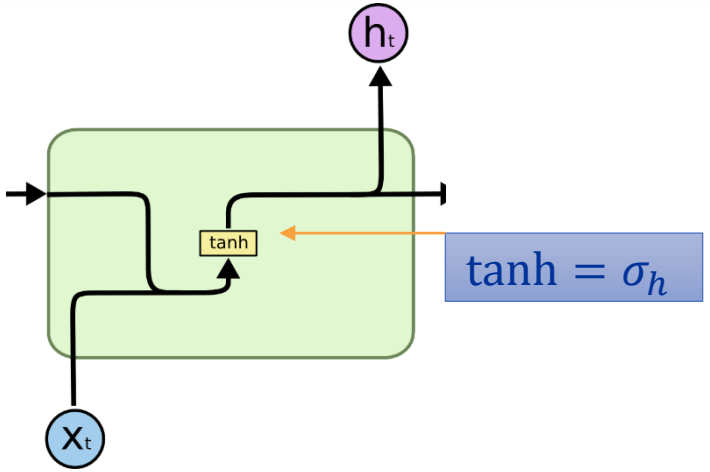
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$
$$h_t = o_t \circ tanh(c_t)$$

- the final step is to decide what the output should be, this is controlled by the output gate o_t

Final Comparison

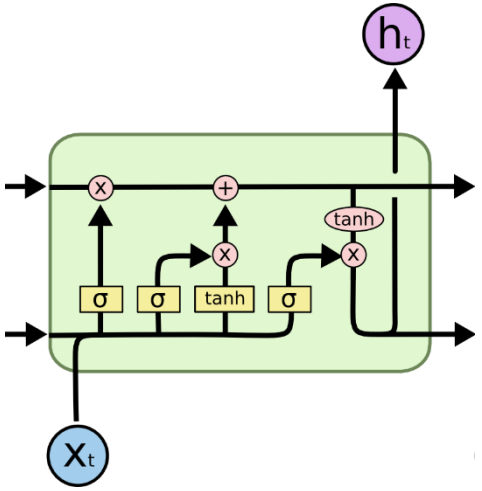
Vanilla RNN

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$

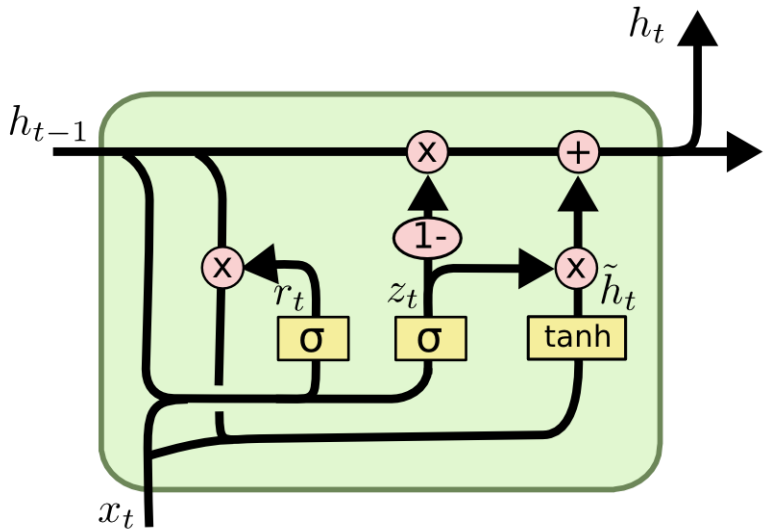


LSTM

$$f_t = \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right)$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ tanh(W_c x_t + U_c h_{t-1} + b_c)$$
$$h_t = o_t \circ tanh(c_t)$$



Gated Recurrent Unit (GRU)



$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$
$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h)$$
$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t$$

- GRU is a popular (and simpler) alternative cell
- merges input and forget gates
- uses only one state instead of two

Practical Considerations

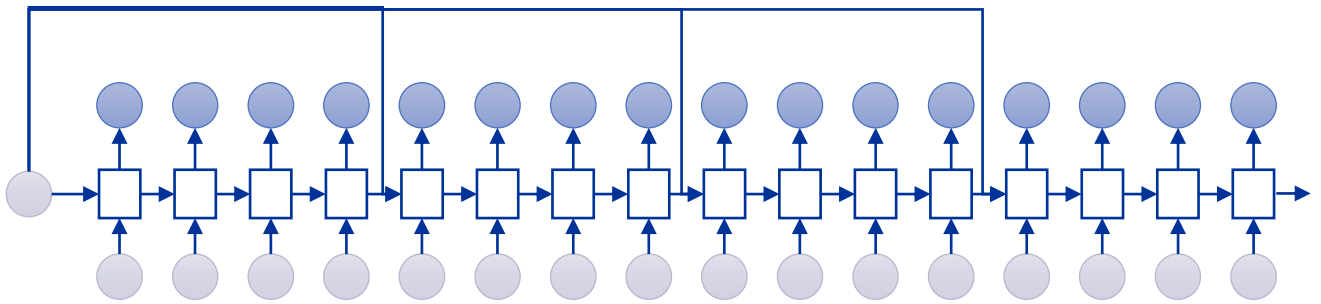
Backpropagation Through a Long Time

- Note that BPTT can get quite computationally expensive
- Especially, when the sequences get very long

→In practice, BPTT is usually not done over the full sequence, but only over a part → **Truncated BPTT**

Truncated BPTT I

- Backpropagating over the whole sequence is often not computationally feasible
- **Naive solution:**
  Split the sequence into chunks of k steps and treat every chunk as an individual training instance.
- **Drawback:**
  This prevents the RNN from learning long-range dependencies that span more than k steps.



Sutskever, Ilya: Training recurrent neural networks. In: University of Toronto, Toronto, Ont., Canada (2013)

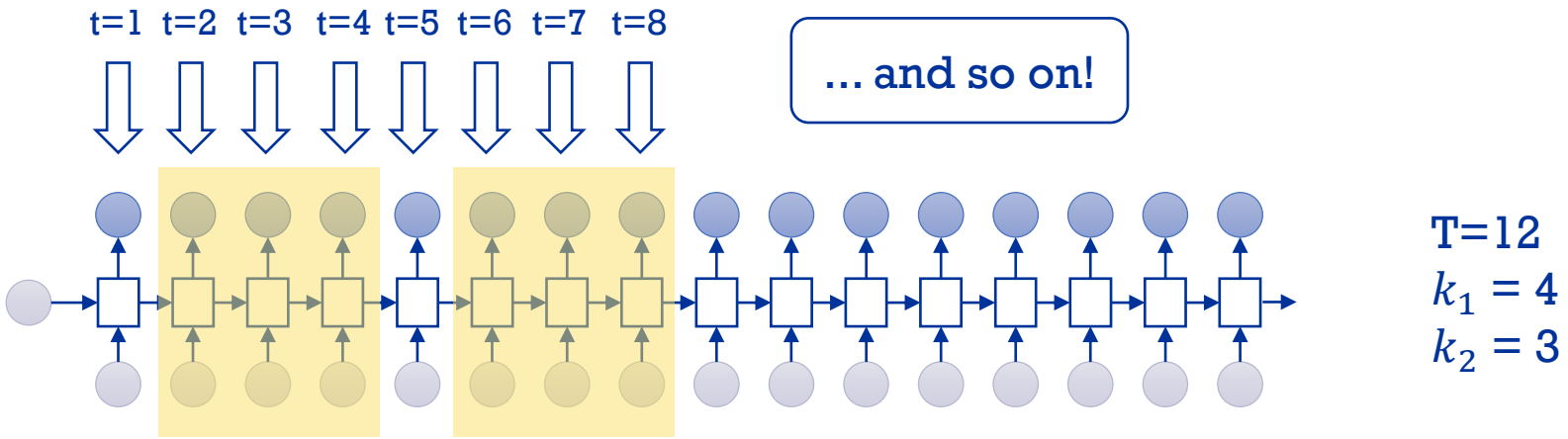
Truncated BPTT II

There is a different variant that preserves the state:

```
for t from 1 to T do
    h_t = f_RNN(x_t, h_{t-1})
    y_t = f_out(h_t)

    if t divides k_1 then
        Run BPTT on the chunk from t down to t − k_2
```

T – length of the sequence
t – current step
k_1 – controls how often BPTT occurs
k_2 – controls how many steps are included in BPTT



Sutskever, Ilya: Training recurrent neural networks. In: University of Toronto, Toronto, Ont., Canada (2013)

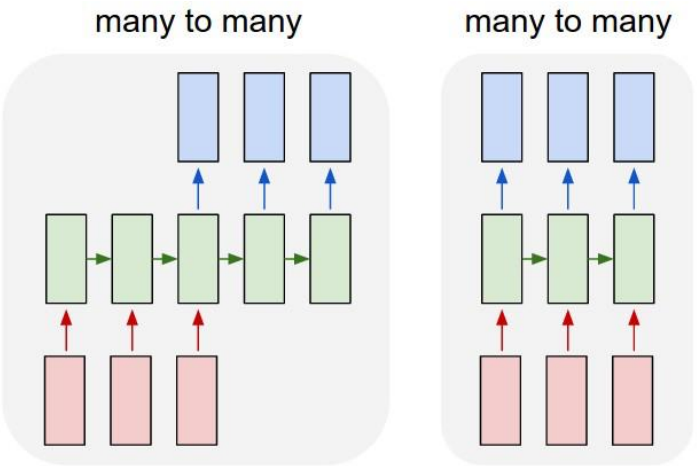
Truncated BPTT

- Note: The variants from the two previous slides only work with many-to-many networks!
- Why?

  → Need to calculate a loss at least every k or k_1 steps!

- What can we do for many-to-one tasks?

  → Calculate the forward pass
  over the full sequence,
  then do backpropagation
  for k_2 steps



RNNs and BPTT in PyTorch

- PyTorch provides nn.RNN, nn.LSTM, and nn.GRU modules
- They return the output and hidden state for each timestep
- Backpropagate through all timesteps as you know it
- Truncated BPTT can be implemented by hand → not easy
- Truncated BPTT is also implemented in PyTorch Ignite → easy

5.2 – Recurrent Neural Networks in Practice

- RNNs for Text Generation

RNNs for Text Generation

RNNs for Text Generation

- Text Generation = Given some training corpus, create new text that is „similar“ to the corpus

- Well-suited task for RNNs:
  - Internal State encodes the text so far
  - Output layer predicts the next token

- Possible at different levels:
  - Word level
  - Character level
  - Phoneme level
  - …

RNNs for Text Generation

- Popular blog post: „The Unreasonable Effectiveness of Recurrent Neural Networks" (Andrej Karpathy)

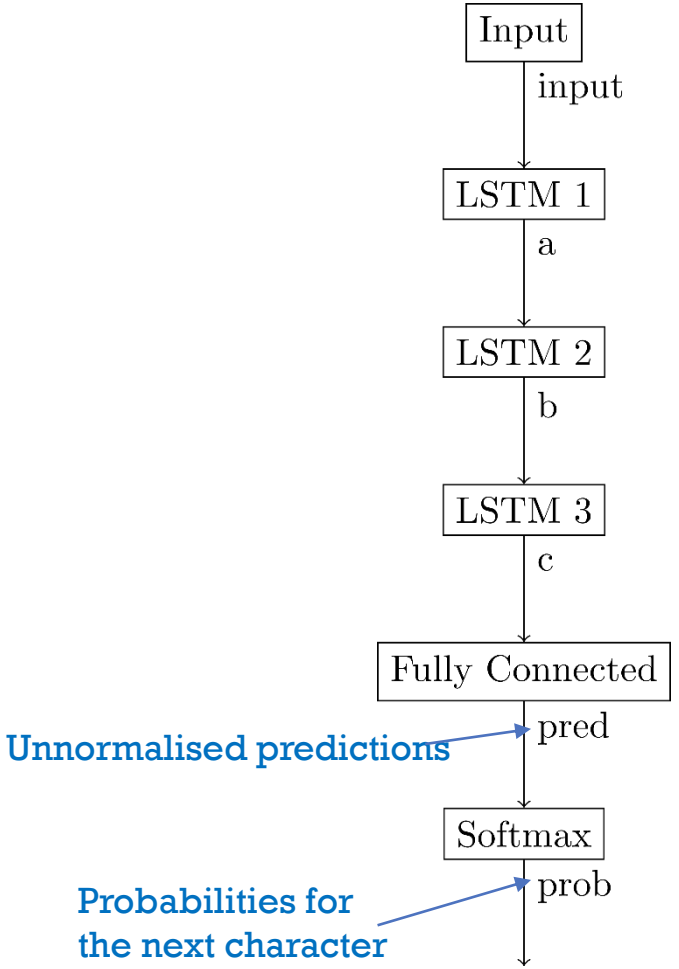
- Train a character-level RNN on different corpora
- Generate new text
- Investigate what is going on!
  - Look at different RNN cells
  - Play around with hyperparameters

```
Clown: Come, sir, I will make did
behold your worship.
VIOLA: I'll drink it.
```

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

RNNs for Text Generation — Model

- Input: The previous n characters in a 1-hot encoding
- Three stacked LSTMs
- A fully connected layer
- And a softmax activation



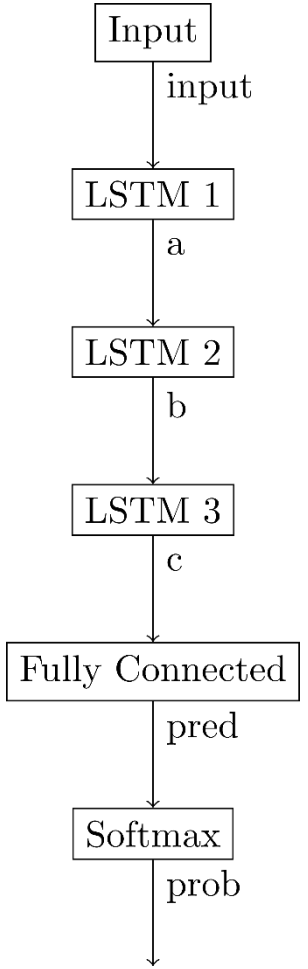
RNNs for Text Generation — Training

- Building the input:
  - Given a document $d = (d_1, d_2, \dots, d_m)$
  - Generate tuples $t_i = (x_i, y_i) \in T$, with $x_i = (d_i, \dots, d_{i+n})$, $i \in [1, m-n], y_i = d_{i+n+1}$

- Train by sliding x_i over the dataset and optimising the network to predict $\text{argmax}(\text{prob}) = y_i$

RNNs for Text Generation — Prediction

- Given some prime text $p = (p_1, \dots, p_n)$
- Predict p_{n+1}!
- Then keep going,
  predict p_{n+2} from $(p_2, \dots, p_{n+1})$, …
- Problem: Using just the most likely character will always give the same sequence!



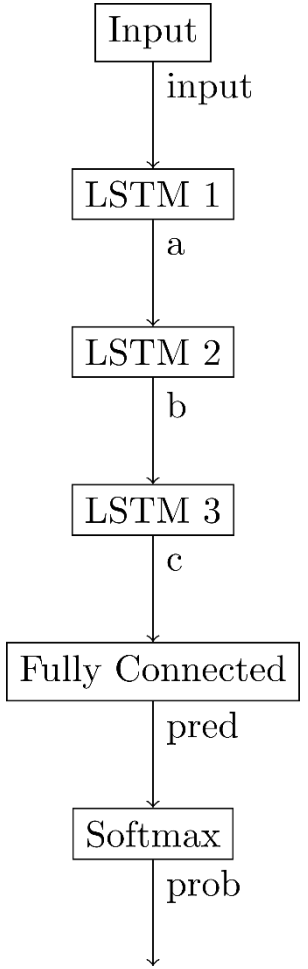
RNNs for Text Generation — Prediction with Temperature

- Network returns a probability distribution $prob$ over possible next characters

→Sample p_i from $prob$!

→May lead to more garbage predictions

→Use a „temperature“ to control variation

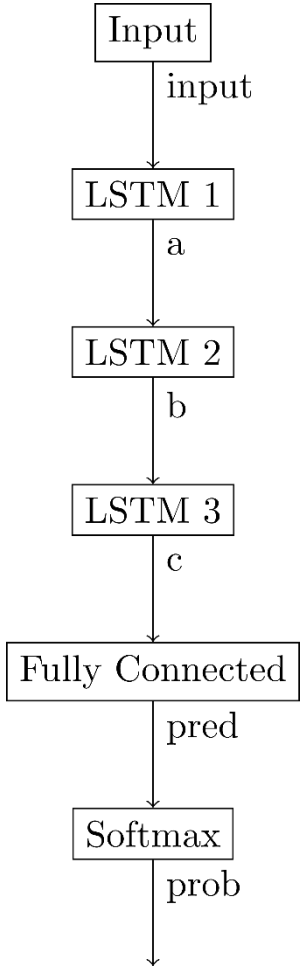


RNNs for Text Generation — Prediction with Temperature

- Output before softmax ($pred$): unnormalised probabilities

- Divide their log by a modifier $temp \in (0, 1]$ – the temperature:

$$\begin{aligned} pred &= \exp\left(\frac{\log(pred)}{temp}\right) \\ &= \exp\left(\log(pred) \cdot \frac{1}{temp}\right) \\ &= pred^{\frac{1}{temp}} \end{aligned}$$

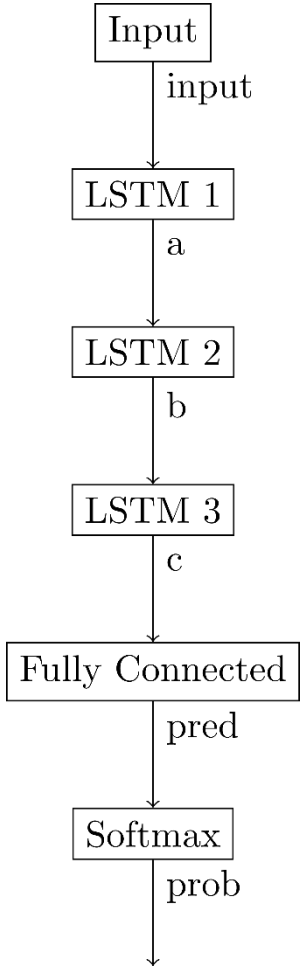


RNNs for Text Generation — Prediction with Temperature

- The smaller $temp$, the larger $pred$ gets
- More importantly: values in $pred$ move further apart

→After „scaling“ through the softmax, a smaller $temp$ leads to a more peaky probability distribution

→Sampling becomes „less random“



RNNs for Text Generation – Datasets

- Training on three different datasets:
  - Shakespeare (4.4 MB)
  - LaTeX: Algebraic Geometry (16 MB)
  - Linux source code (474 MB)

- Predict from each model!

Shakespeare!

PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.

Algebraic Geometry!

```
\begin{proof}
We may assume that $\mathcal{I}$ is an abelian sheaf on
$\mathcal{C}$.
\item Given a morphism $\Delta : \mathcal{F} \to \mathcal{I}$
is an injective and let $\mathfrak q$ be an abelian sheaf on $X$.
Let $\mathcal{F}$ be a fibered complex. Let $\mathcal{F}$ be a
category.
\begin{enumerate}
\item \hyperref[setain-construction-phantom]{Lemma}
\label{lemma-characterize-quasi-finite}
Let $\mathcal{F}$ be an abelian quasi-coherent sheaf on
$\mathcal{C}$.
Let $\mathcal{F}$ be a coherent $\mathcal{O}_X$-module. Then
$\mathcal{F}$ is an abelian catenary over $\mathcal{C}$.
\item The following are equivalent
\begin{enumerate}
\item $\mathcal{F}$ is an $\mathcal{O}_X$-module.
\end{lemma}
```

Some errors:

- Opens proof, closes lemma
- Opens enumerate, doesn‘t close it

→ Fix these manually!

Algebraic Geometry!

Proof. Omitted. ☺ □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*

Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\acute{e}tale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma ??. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

be a morphism of algebraic spaces over S and Y.

Proof. Let X be a nonzero scheme of X. Let X be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

(1) $\mathcal{F}$ is an algebraic space over S.
(2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □

This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram

$$\begin{array}{ccc} \mathcal{S} & \longrightarrow & \\ \downarrow & & \\ \xi & \longrightarrow & \mathcal{O}_{X'} \\ & & \uparrow \\ \mathrm{gor}_s & & \end{array} \qquad \begin{array}{ccc} = \alpha' & \longrightarrow & \\ \updownarrow & & \\ = \alpha' & \longrightarrow & \alpha \end{array} \qquad \begin{array}{c} X \\ \downarrow \\ \mathrm{d}(\mathcal{O}_{\mathcal{X}_{X/k}}, \mathcal{G}) \end{array}$$

$$\mathrm{Spec}(K_\psi) \qquad \mathrm{Mor}_{Sets}$$

is a limit. Then $\mathcal{G}$ is a finite type and assume S is a flat and $\mathcal{F}$ and $\mathcal{G}$ is a finite type f_*. This is of finite type diagrams, and

- the composition of $\mathcal{G}$ is a regular sequence,
- $\mathcal{O}_{X'}$ is a sheaf of rings.

□

Proof. We have see that $X = \mathrm{Spec}(R)$ and $\mathcal{F}$ is a finite type representable by algebraic space. The property $\mathcal{F}$ is a finite morphism of algebraic stacks. Then the cohomology of X is an open neighbourhood of U. □

Proof. This is clear that $\mathcal{G}$ is a finite presentation, see Lemmas ??.
A *reduced above* we conclude that U is an open covering of $\mathcal{C}$. The functor $\mathcal{F}$ is a "field

$$\mathcal{O}_{X,x} \longrightarrow \mathcal{F}_{\overline{x}} \quad \text{-1}(\mathcal{O}_{X_{\acute{e}tale}}) \longrightarrow \mathcal{O}_{X_\ell}^{-1}\mathcal{O}_{X_\lambda}(\mathcal{O}_{X_\eta}^{\mathbb{V}})$$

is an isomorphism of covering of $\mathcal{O}_{X_i}$. If $\mathcal{F}$ is the unique element of $\mathcal{F}$ such that X is an isomorphism.
The property $\mathcal{F}$ is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme $\mathcal{O}_X$-algebra with $\mathcal{F}$ are opens of finite type over S.
If $\mathcal{F}$ is a scheme theoretic image points. □

If $\mathcal{F}$ is a finite direct sum $\mathcal{O}_{X_\lambda}$ is a closed immersion, see Lemma ??. This is a sequence of $\mathcal{F}$ is a similar morphism.

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECONDS << 12;
  return segtable;
}
```

Linux Source Code!

RNNs for Text Generation – Some insights

- Looking at samples generated after n iterations over "War and Peace":
  - $n = 100$:
  - tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e plia tklrgd t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaenns lng

  - Nothing useful yet, just random characters
  - Some space-separation, but no real structure

RNNs for Text Generation – Some insights

- Looking at samples generated after n iterations over “War and Peace”:
  - $n = 300$
  - "Tmont thithey" fomesscerliundKeushey. Thom heresheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlomecoaniogennc Phe lism thond hon at. MeiDimorotion in ther thize.“

  - Starts to use quotes, spaces and periods somewhat correctly
  - Still gibberish words

RNNs for Text Generation – Some insights

- Looking at samples generated after n iterations over "War and Peace":
  - $n = 500$
  - we counter. He stutn co des. His stanted out one ofler that concossions and was to gearang reay Jotrets and with fre colt otf paitt thin wall. Which das stimn

  - Looks like real text!
  - Some short words start to form (we, he, …)

RNNs for Text Generation – Some insights

- Looking at samples generated after n iterations over "War and Peace":
  - $n = 700$
  - Aftair fall unsuch that the hall for Prince Velzonski's that me ofher hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort how, and Gogition is so overelical and ofter.
  - More and more correct words

RNNs for Text Generation – Some insights

- Looking at samples generated after n iterations over “War and Peace”:
  - $n = 1200$
  - "Kite vouch!" he repeated by herdoor. "But I would be done and quarts, feeling, then, son is people....“

  - Direct speech in quotes, even with a marker (he repeated)
  - Mostly correct words

RNNs for Text Generation – Some insights

- Looking at samples generated after n iterations over “War and Peace”:
  - $n = 2000$
  - "Why do what that day," replied Natasha, and wishing to himself the fact the princess, Princess Mary was easier, fed in had oftened him. Pierre aking his soul came to the packs and drove up his father-in-law women.

  - Acceptable sentence structure, correct words, just no coherent content.

RNNs for Text Generation – Some insights

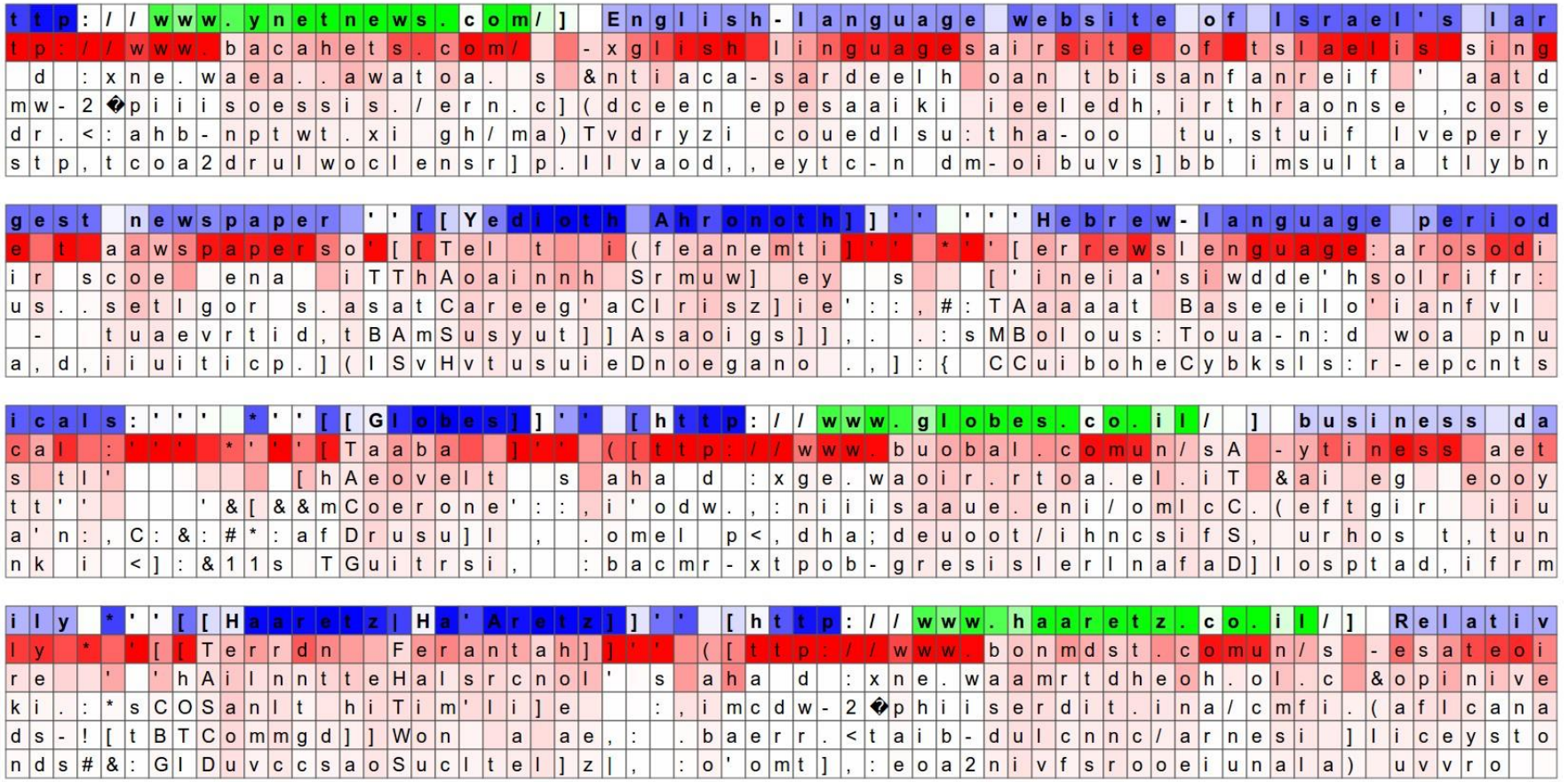
- Model learns things in ascending order of complexity:
  1. Word-space structure
  2. Short words
  3. Longer words
  4. Grammatical constructs
  5. Much later: Some degree of content/coherence

RNNs for Text Generation – Some insights

- Karpathy finds that some neurons in the LSTMs are interpretable:
  - Visualise the activation of every single neuron at each position in the text
  - Some turn on in specific circumstances
  - Some get gradually more „excited“ inside quotes, …
- Following slides: Some examples

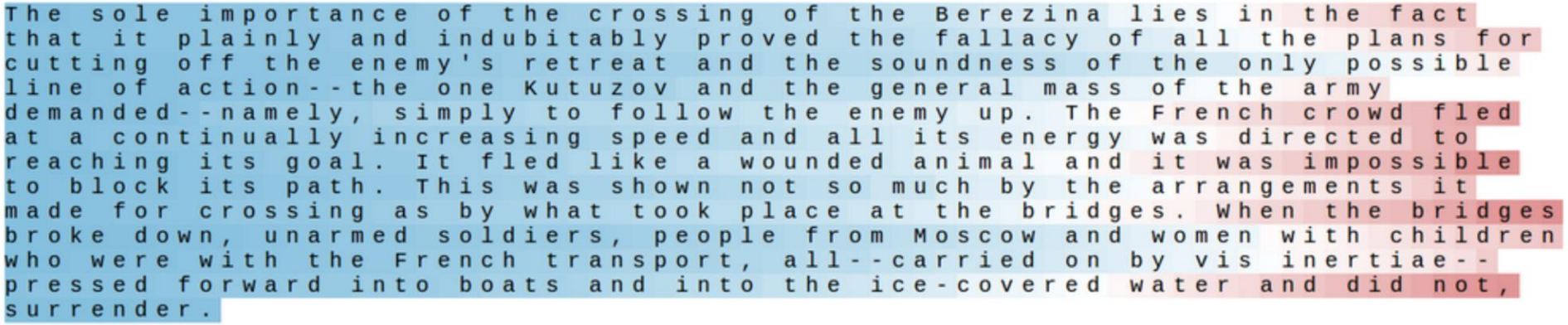
RNNs for Text Generation – Some insights

- „URL-Neuron“:



RNNs for Text Generation – Some insights

- „Line Break-Neuron“:



RNNs for Text Generation – Some insights

- „if-Neuron“:

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
        siginfo_t *info)
{
  int sig = next_signal(pending, mask);
  if (sig) {
    if (current->notifier) {
      if (sigismember(current->notifier_mask, sig)) {
        if (!(current->notifier)(current->notifier_data)) {
          clear_thread_flag(TIF_SIGPENDING);
          return 0;
        }
      }
    }
    collect_signal(sig, pending, info);
  }
  return sig;
}
```

RNNs for Text Generation – Some insights

- „Garbage-Neuron“:
- Just a reminder that most neurons are **not** easily interpretable!

```
/*  Unpack  a  filter  field's  string  representation  from  user-space
 *  buffer.  */
char  *audit_unpack_string(void  **bufp,  size_t  *remain,  size_t  len)
{
  char  *str;
  if  (!*bufp  ||  (len  ==  0)  ||  (len  >  *remain))
    return  ERR_PTR(-EINVAL);
  /*  Of  the  currently  implemented  string  fields,  PATH_MAX
   *  defines  the  longest  valid  length.
   */
```