Deep Network Development

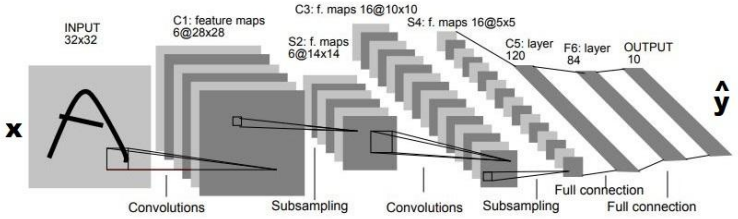
Lecture #9

Viktor Varga
Department of Artificial Intelligence, ELTE IK

Requirements

The content of the slides marked by this symbol
will not be included in the exams / tests.

Last week - PyTorch CNN




```python
class LeNet5(nn.Module):
  def __init__(self, n_categories):
    super().__init__()
    self.conv_block1 = nn.Sequential(
      nn.Conv2d(1, 6, kernel_size = (5,5)),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size = (2,2)))
    self.conv_block2 = nn.Sequential(
      nn.Conv2d(6, 16, kernel_size = (5,5)),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size = (2,2)))
    self.fc1 = nn.Linear(400, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, n_categories)
```

```python
def forward(self, x):
  # x.shape == (batch_size, 1, 32, 32)
  out = self.conv_block1(x)
  out = self.conv_block2(out)
  out = out.reshape(out.size(0), -1)
  out = self.fc1(out)
  out = nn.ReLU()(out)
  out = self.fc2(out)
  out = nn.ReLU()(out)
  out = self.fc3(out)
  return out
```

(bs, 1, 32, 32)
(bs, 6, 14, 14)
(bs, 16, 5, 5)
(bs, 400)
(bs, 120)
(bs, 120)
(bs, 84)
(bs, 84)
(bs, 10)

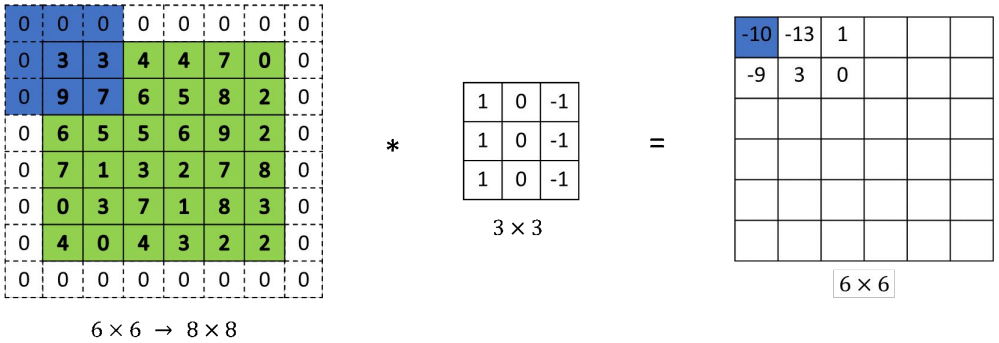
Last activation function and loss function chosen depending on the task…
(regression, binary classification, multi-class classification)

Image source: http://datahacker.rs/what-is-padding-cnn/, https://adeshpande3.github.io/

Last week - Convolutional network hyperparameters

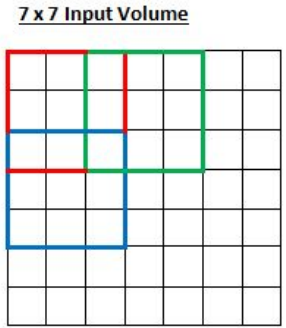
Padding:

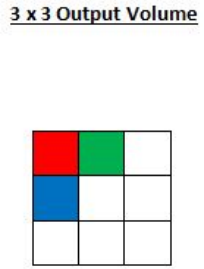
```
nn.Conv2d(..., padding=1, ...)
nn.Conv2d(..., padding="same", ...)
```



Stride (step size):

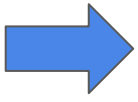
```
nn.Conv2D(..., stride=2, ...)
nn.MaxPooling2D(..., stride=(4, 3), ...)
```

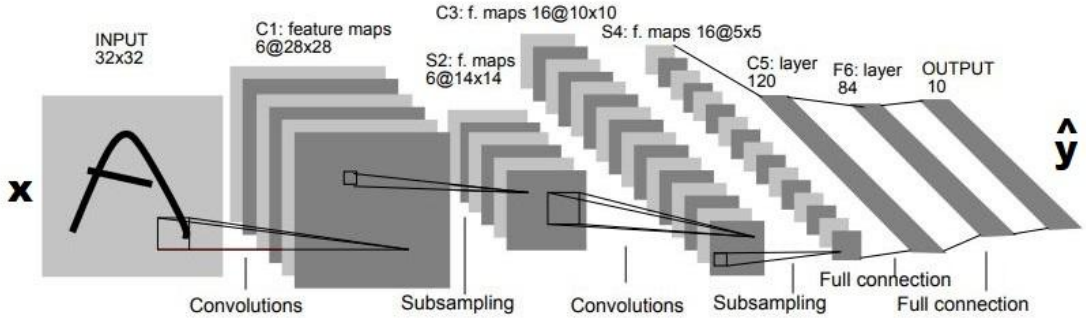


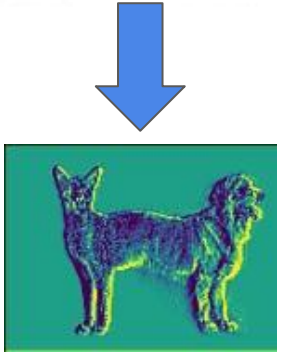


Last week - What does a ConvNet learn?

Visualizing heatmaps







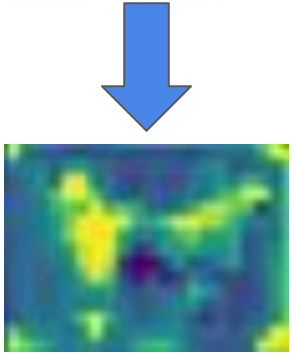


Image source: Yosinski et al. 2015, Deep Visualization Toolbox

Last week - What does a ConvNet learn?

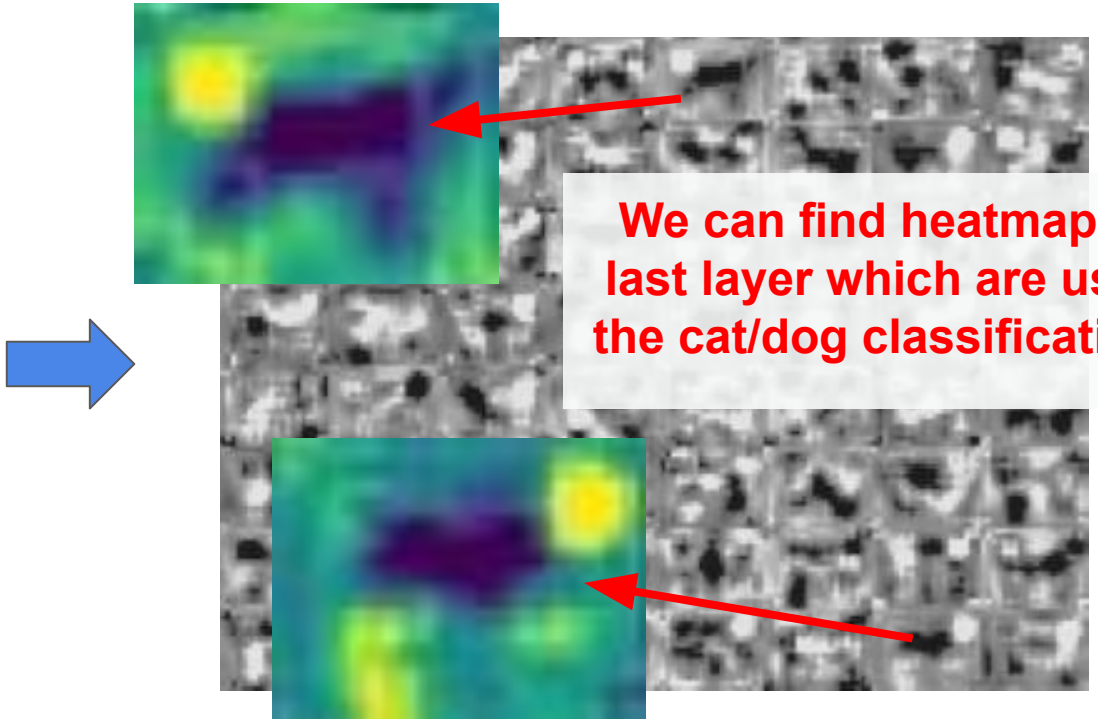
Visualizing heatmaps

First convolutional layers:
Edge-, corner-, color transition detection

Last convolutional layers:
High-level object detection

Last week - What does a ConvNet learn?

Visualizing heatmaps - Last layer output



We can find heatmaps in the last layer which are useful for the cat/dog classification task!

Image source: Zeiler et al. 2013, https://arxiv.org/pdf/1311.2901.pdf

Last week - What does a ConvNet learn?

Let's turn it around: Let's look for inputs that produce high values on individual heatmaps!

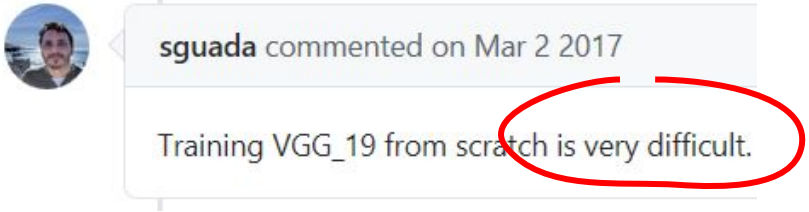
First conv. layer

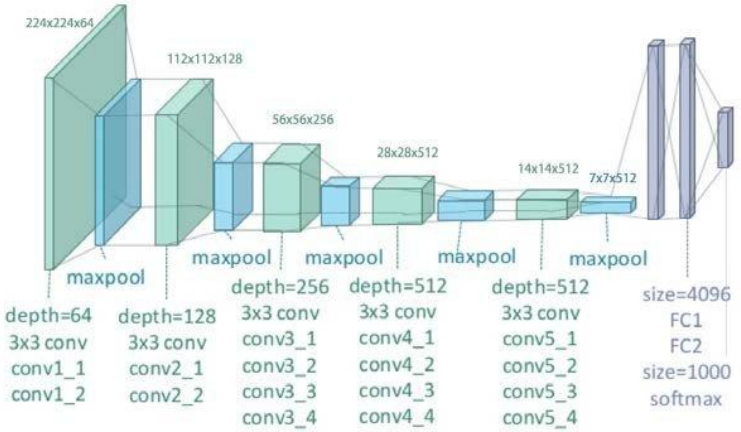
Middle conv. layer

Last conv. layer

Image source: Simonyan & Zisserman 2014

Last week - The scaling of the gradient





Training deep neural networks is getting more and more difficult as we keep adding layers to the network. **Why?**

Last week - The scaling of the gradient

Let's examine **how a layer affects the gradient** backpropagated through it!

$$\hat{y} = g(wx + b) \qquad w, b, x, \hat{y} \in \mathbb{R}$$

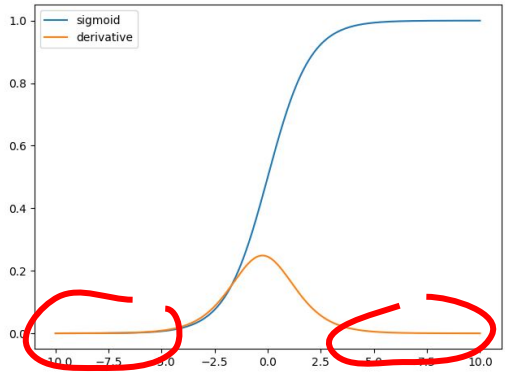
$$\frac{\partial J}{\partial x} = \boxed{\frac{\partial \hat{y}}{\partial x}} \cdot \frac{\partial J}{\partial \hat{y}}$$

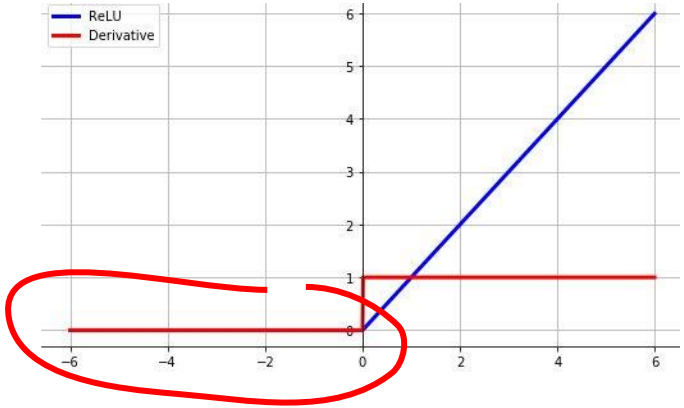
$$\frac{\partial \hat{y}}{\partial x} = \boxed{g'(wx + b)} \cdot \boxed{w}$$

During backpropagation, **each layer multiplies the backpropagated gradient** by the derivative of the activation function and the weight (matrix).

Last week - The scaling of the gradient

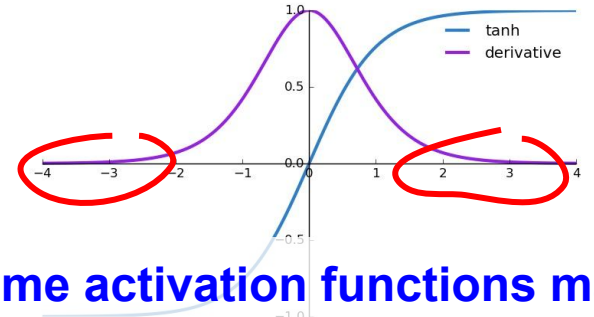
Sigmoid

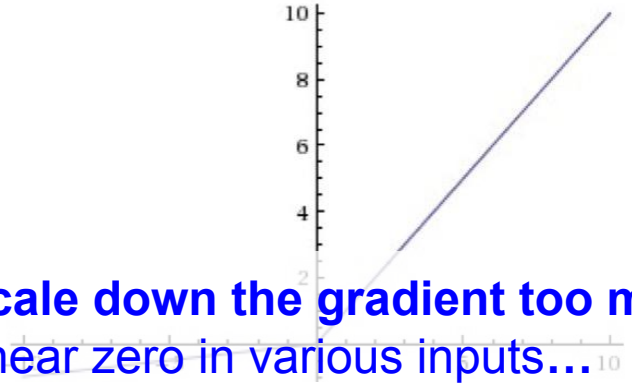




ReLU

Tanh





Leaky-ReLU

Some activation functions may scale down the gradient too much

as their derivative is zero or near zero in various inputs…

Last week - The unstable gradient problem

During backpropagation in a deep network **many of these terms are multiplied together** to get the update of a given parameter:

$$\frac{\partial \hat{y}}{\partial x} = g'(wx + b) \cdot w$$

Result: In a network with too many layers, different parts of the network may learn at completely different speeds: while the updates for some parameters are negligible, others are so large that their learning does not converge.

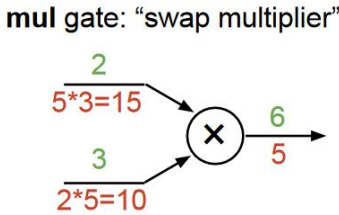
Last week - Batch normalization

A solution for the unstable gradient problem:

We rescale the gradient every few layers so that its magnitude remains similar across all sections of the network.

$$\hat{x}_i := \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \qquad \hat{y}_i := w \cdot \hat{x}_i + b \qquad w, b \in \mathbb{R}$$

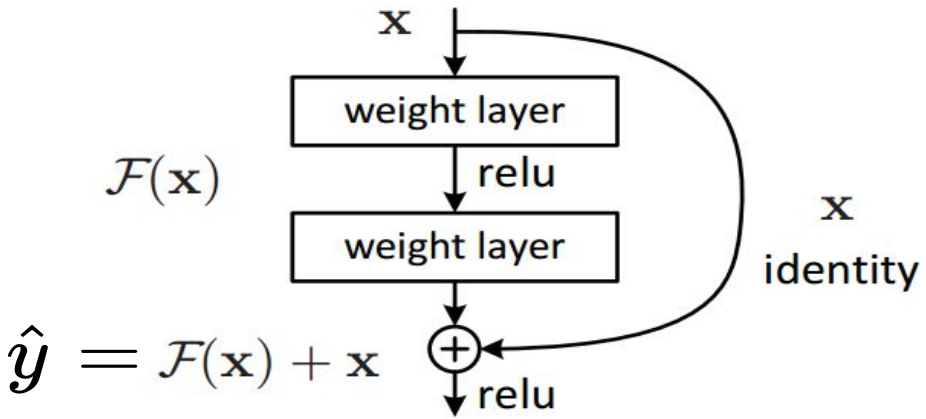
If we multiply by a constant during the forward pass, **then, the gradient is multiplied by that number during the backward pass too!**



Last week - Residual networks

Another solution for the unstable gradient problem:

Residual block: We add the input to the output



The gradient flows backwards uninterrupted through the skip (“+x”) connection.

Last week - Transfer learning

What options do we have if we want to
learn a task from a small dataset?

a) **Training a deep neural network** → Overfitting

b) **Training a small / shallow neural network** → Low accuracy

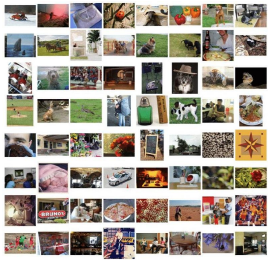
c) **Transfer learning** → Reduced overfitting, higher accuracy
IF we find a suitable pre-training task/dataset

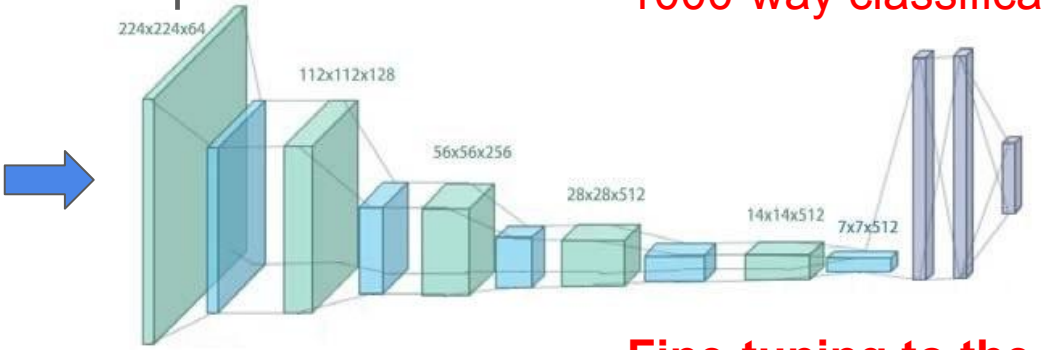
Last week - Transfer learning

Transfer learning - Example

Pre-training task:
1000-way classification

1)





Large dataset

Fine-tuning to the target task:
Binary classification

2)

A smaller dataset is enough

by freezing the pre-trained weights if needed…

Last week - Transfer learning

Transfer learning - We don't need to do the pre-training if we have access to pre-trained weights trained on an appropriate task / dataset.

1)
```
mobilenet_model = torch.hub.load('pytorch/vision', 'mobilenet_v2', weights='MobileNet_V2_Weights.IMAGENET1K_V1')
```

Fine-tuning to the target task:
Binary classification

2) 

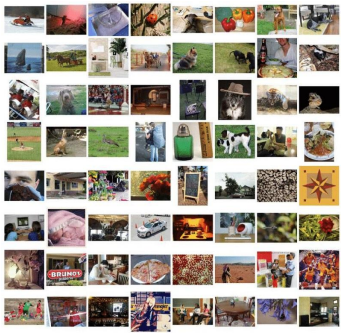
A smaller dataset is enough

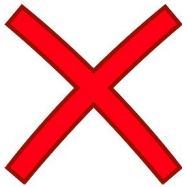
by freezing the pre-trained weights if needed…

Last week - Transfer learning

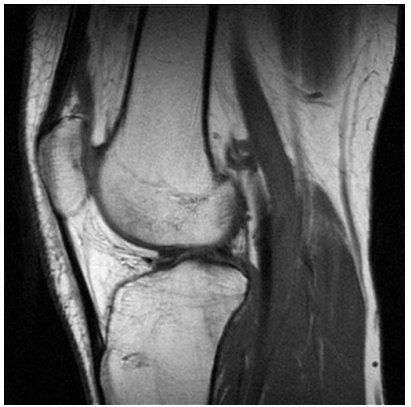
It’s not always easy to find an appropriate pre-training dataset or a pre-trained model for every task…

Pretraining task





Target task

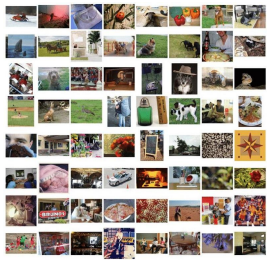


Last week - Transfer learning

Transfer Learning in Practice - General “rules”

- **The smaller the dataset for the target task:**
  The fewer layers/parameters we train during fine-tuning.

- **The more the target task differs from the pretraining task:**
  The fewer pre-trained layers we should retain for fine-tuning.



Until now - What type of data can we learn to process?

- Data that is not too high-dimensional, where **variables have a fixed, unique meaning.**
  (e.g., x_0: weight of a patient, x_1: age of a patient, etc.).

  → **Multi-Layer Perceptron**

Until now - What type of data can we learn to process?

- Data that is not too high-dimensional, where **variables have a fixed, unique meaning.**
  (e.g., x_0: weight of a patient, x_1: age of a patient, etc.).

  → **Multi-Layer Perceptron**

- Data can be high dimensional, but **variables can be indexed** along one or more axes. The **variables play an equivalent role**.
  (e.g., $x_{i,j}$: pixels of an image).

  → **Convolutional Neural Networks**

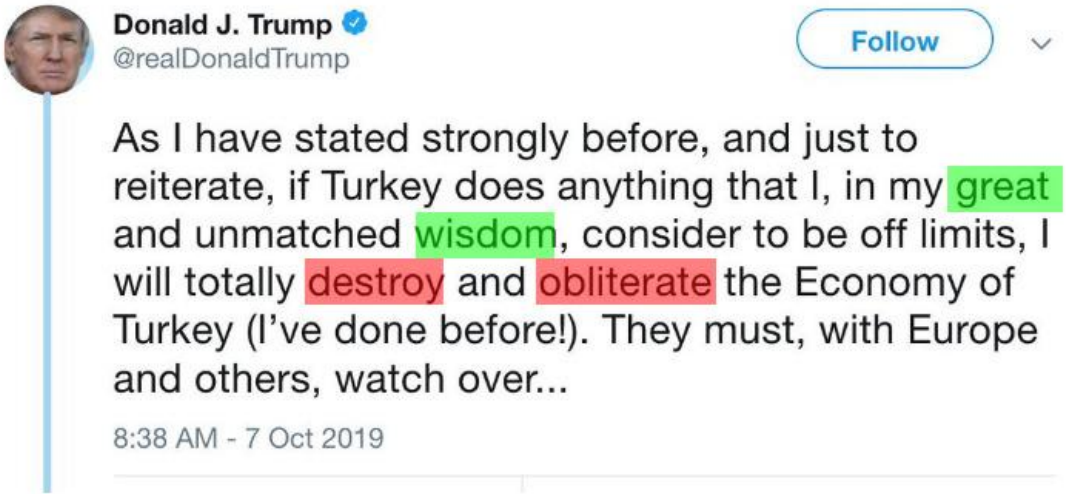
Until now - What type of data can we learn to process?

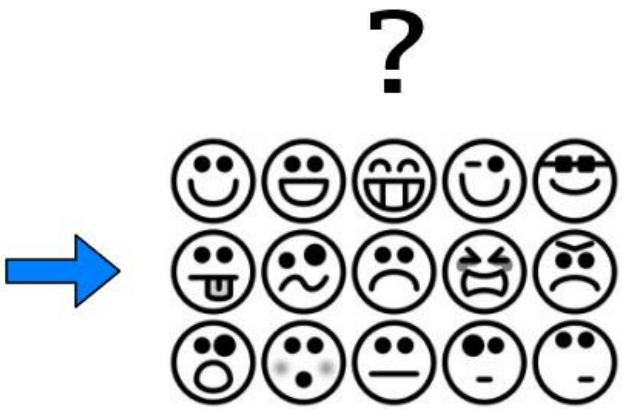
Until now: Input of known length

The length of the input is not always known in advance.
How do we process sequences of unknown length?

Applications with variable length input/output: **N → 1**

Sentiment analysis, emotion recognition





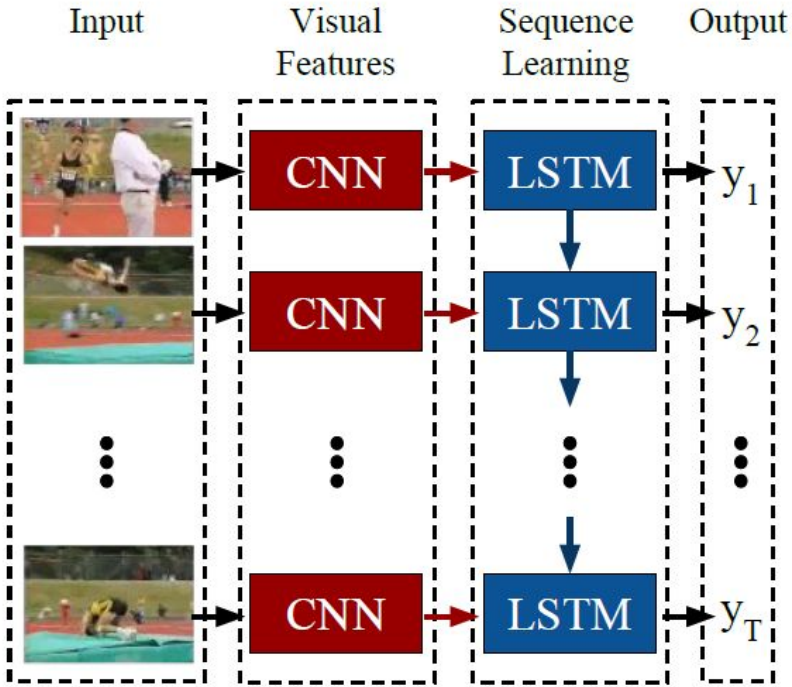
Input: Variable length text / speech / video, etc.

Label: Sentiment category (e.g., a probability vector)

Image source: https://arxiv.org/abs/1411.4389

Applications with variable length input/output: **N → N**

Labeling each frame of a video



Input: Variable length video

Label: A scalar or a vector for each frame (e.g., the result of a regression / classification for each frame)

Applications with variable length input/output: **1 → N**

Image captioning

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

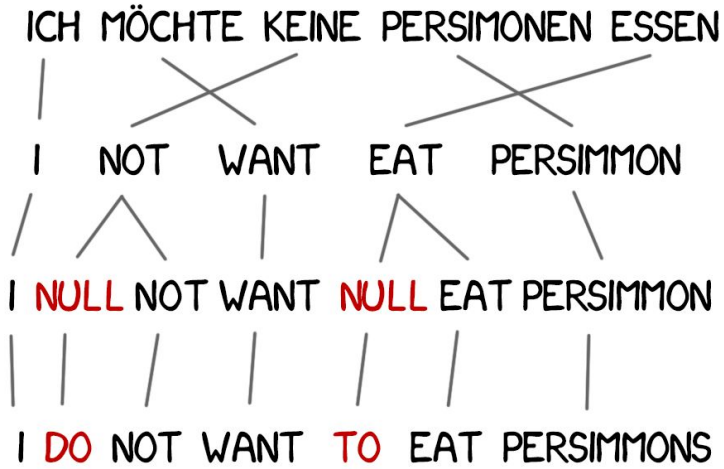
Input: An image

Label: Variable length text

Image source: http://vas3k.com/blog/machine_translation/

Applications with variable length input/output: **M → N**

Machine translation



Input:
Variable length text

Label:
Variable length text

Learning to process sequences

The neural network architectures we have studied so far only work with inputs and outputs of predetermined size.

We need a new approach!

Learning to process sequences

- Instead of processing the entire sequence in a single step,
  we examine only the next element of the sequence at each step!

Learning to process sequences

- Instead of processing the entire sequence in a single step,
  we examine only the next element of the sequence at each step!

- Let the network learn a **state representation** that can be used
  to store important information about prior elements
  while processing the sequence! (**~memory**)

Learning to process sequences

- Instead of processing the entire sequence in a single step, **we examine only the next element of the sequence at each step!**

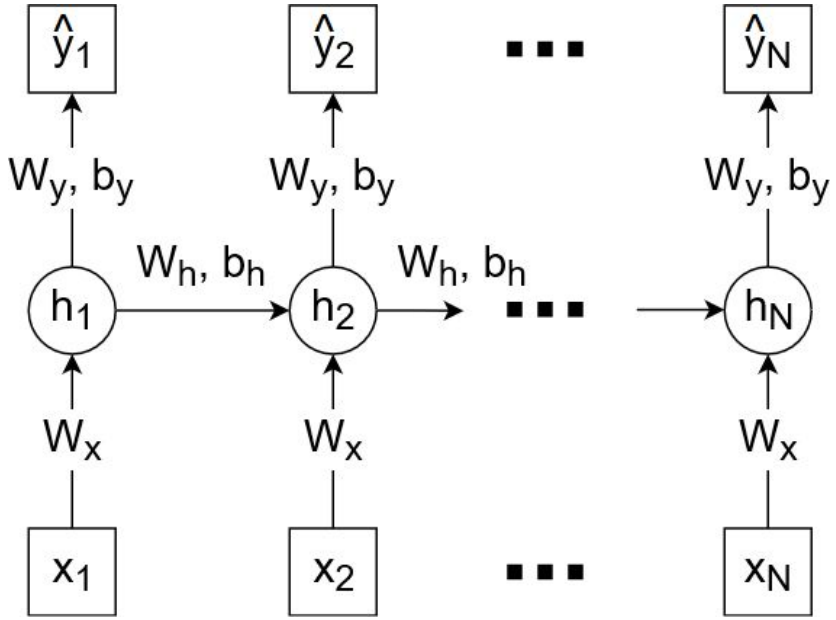
- Let the network learn a **state representation** that can be used to store important information about prior elements while processing the sequence! (**~memory**)

- The sequence can be of arbitrary length, so we use the **same weights for processing every element** (temporal invariance).

Recurrent Neural Network

Unrolled representation

"Vanilla" Recurrent Neural Network (RNN), **N → N**

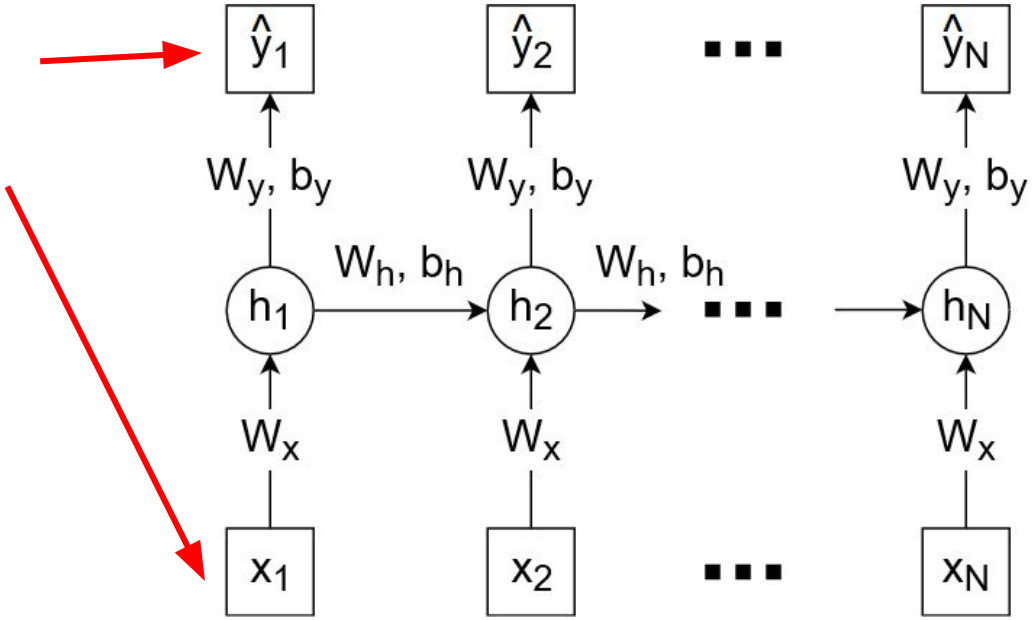


Recurrent Neural Network

Unrolled representation

"Vanilla" Recurrent Neural Network (RNN), **N → N**

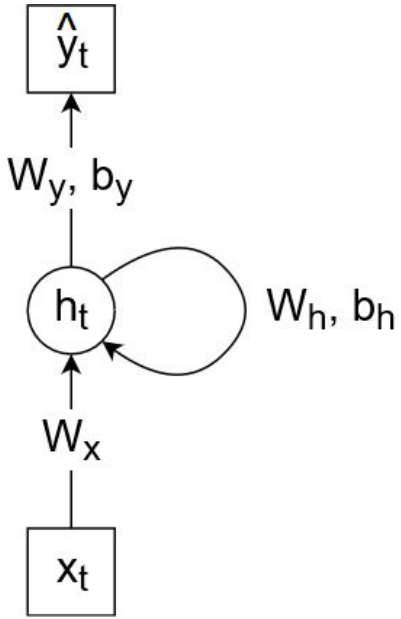
Both the input and output sequences are of length **N.** Each element of the sequences is a vector.



Recurrent Neural Network

Recurrent representation

“Vanilla” Recurrent Neural Network (RNN), **N → N**

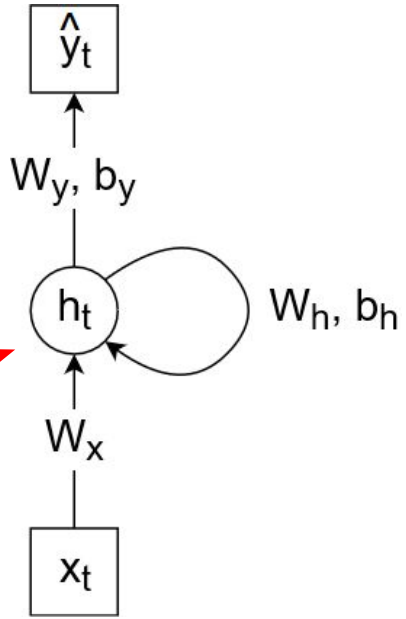


Recurrent Neural Network

Recurrent representation

"Vanilla" Recurrent Neural Network (RNN), **N → N**

As the neural network **parameters** (weight matrices, bias vectors) **are shared across time,** we can visualize the network in a simpler way:



In the entire RNN **there are a total of five parameter matrices/vectors** (regardless of the sequence length):
3 weight matrices (W_x, W_h, W_y) and 2 bias vectors (b_h, b_y)

Recurrent Neural Network

“Vanilla” Recurrent Neural Network (RNN), **N → N**

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

$$\hat{y}_t = \sigma_y(W_y h_t + b_y)$$

The activation function in a Vanilla RNN (σ_h) is tipically **tanh.**

The corresponding part should be omitted when t = 1.

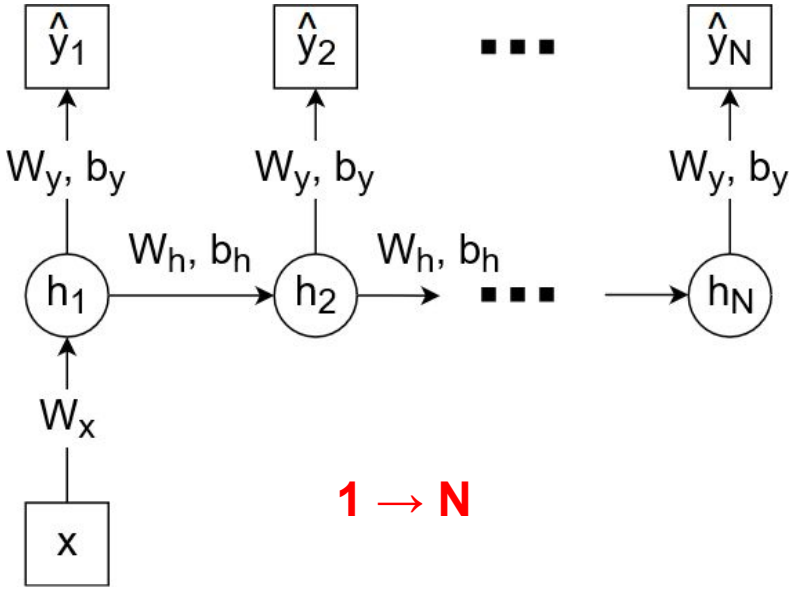
$$x_t \in \mathbb{R}^n \qquad W_x \in \mathbb{R}^{s \times n}$$

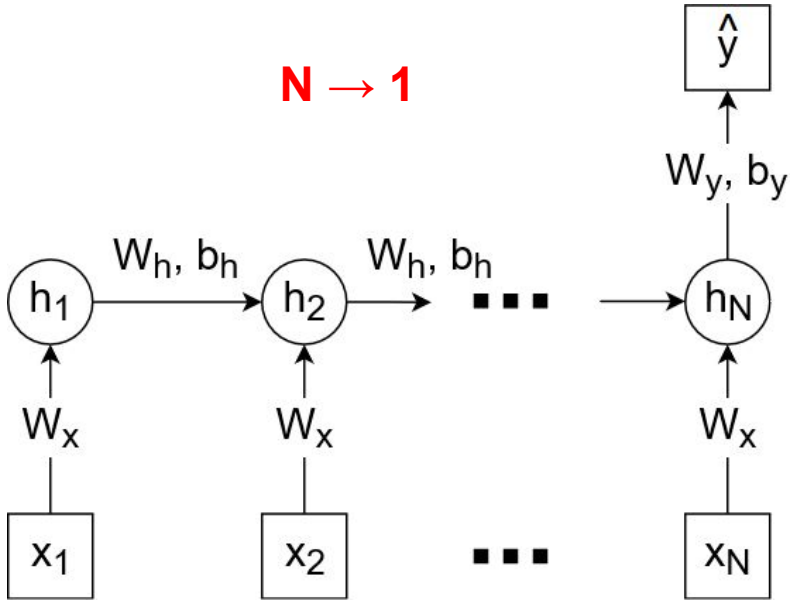
$$h_t \in \mathbb{R}^s \qquad W_h \in \mathbb{R}^{s \times s} \qquad b_h \in \mathbb{R}^s$$

$$\hat{y}_t \in \mathbb{R}^k \qquad W_y \in \mathbb{R}^{k \times s} \qquad b_y \in \mathbb{R}^k$$

RNN, “**1 → N**”, “**N → 1**” variants

Depending on the task, **other architectural variants** may be used:





Recurrent Neural Network - Loss function

Loss function for Recurrent Neural Networks?

Recurrent Neural Network - Loss function

Loss function for Recurrent Neural Networks

Usually we define the loss for the sequence as an aggregate of the losses computed over individual elements (e.g., mean of element-wise losses).

$$J(\Theta) = \frac{1}{N}\sum_{t=1}^{N} J_t(\Theta)$$

Loss for an entire sequence:
Mean of the element-wise losses.

$$J_t(\Theta) = \frac{1}{2mk}\sum_{j=1}^{m} ||\hat{y}_t^{(j)} - y_t^{(j)}||_2^2$$

MSE (element-wise)

$$J_t(\Theta) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{k} y_{t,i} log(\hat{y}_{t,i})$$

CE (element-wise)

Recurrent Neural Network - Loss function

Loss function for Recurrent Neural Networks

Usually we define the loss for the sequence as an aggregate of the losses computed over individual elements (e.g., mean of element-wise losses).

$$J(\Theta) = \frac{1}{N}\sum_{t=1}^{N} J_t(\Theta)$$

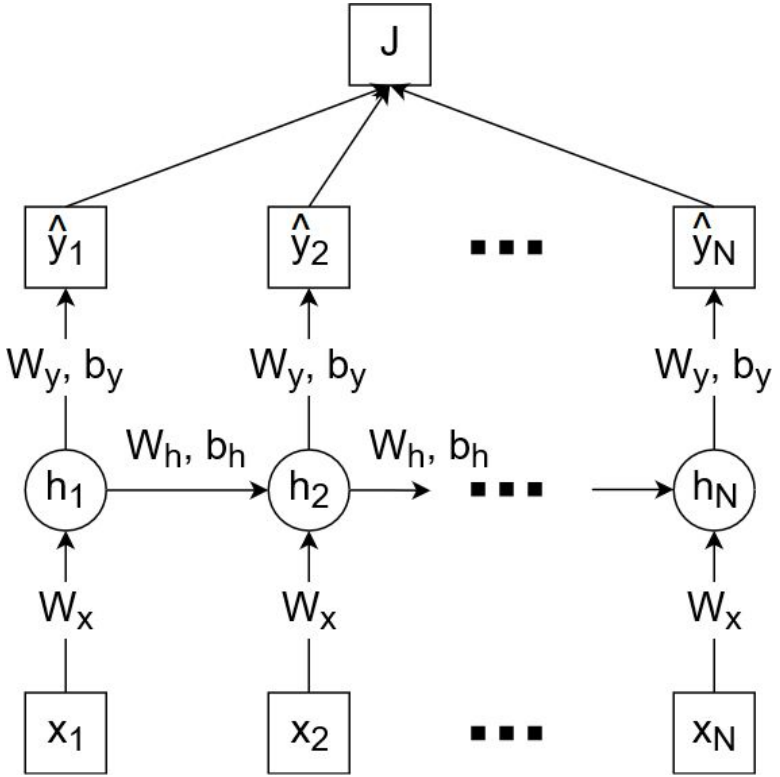
$$J_t(\Theta) = \frac{1}{2mk}\sum_{j=1}^{m} ||\hat{y}_t^{(j)} - y_t^{(j)}||_2^2$$

In some cases, element-wise losses are not ideal (e.g., machine translation, image captioning). Differences between the meaning of two sentences are not described well with their word-by-word differences.

$$J_t(\Theta) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{k} y_{t,i}\log(\hat{y}_{t,i})$$

Recurrent Neural Network - Backpropagation

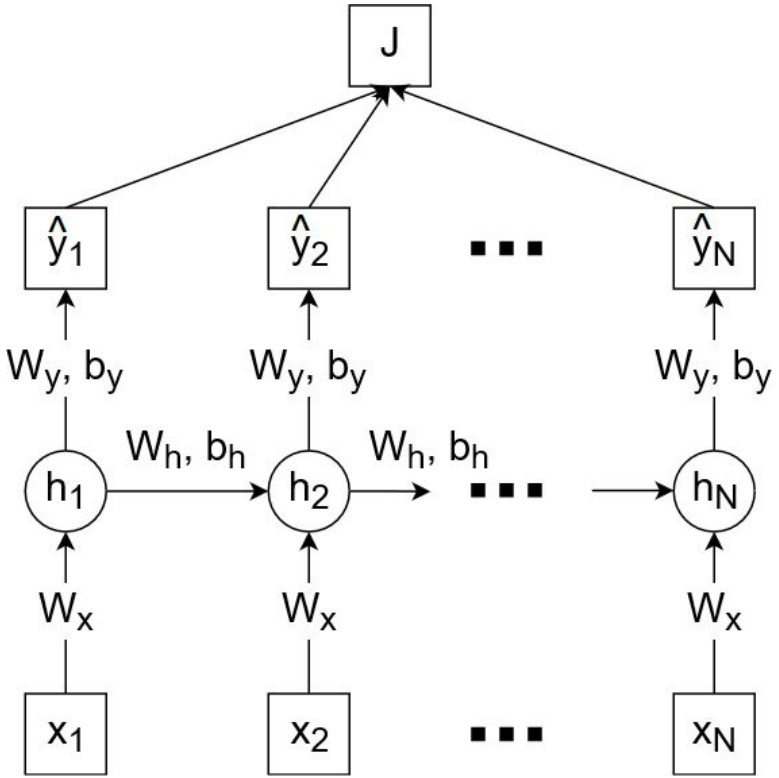
Computational graphs can be defined similarly to feed-forward neural networks.



Recurrent Neural Network - Backpropagation

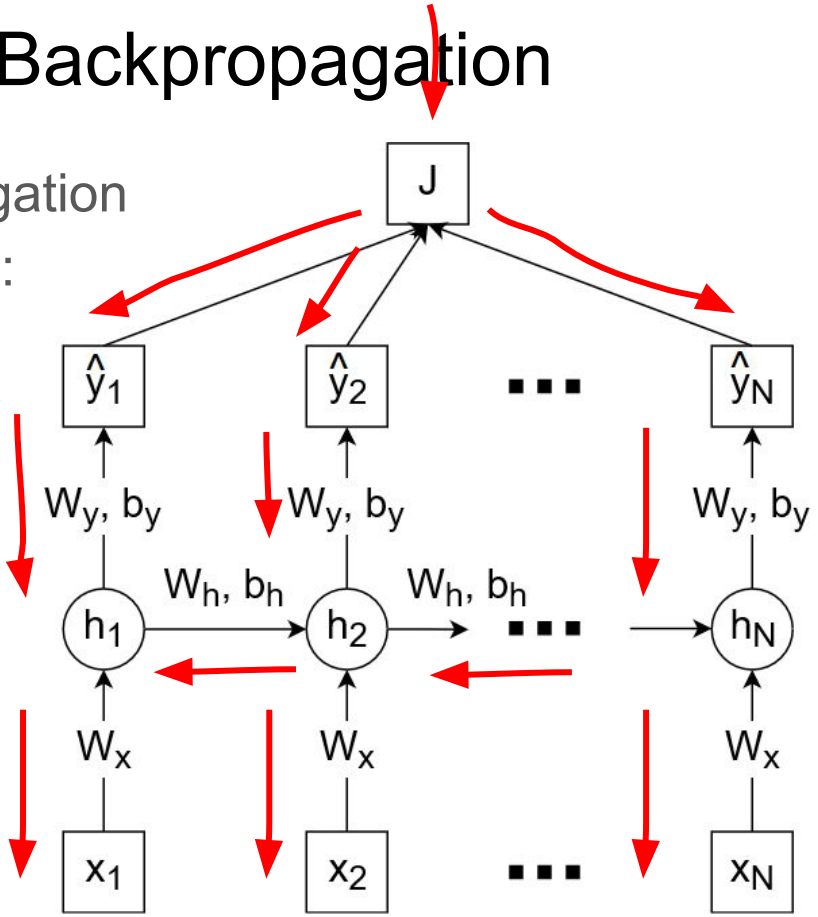
Computational graphs can be defined similarly to feed-forward neural networks.

(This graph is not following the format used in Lecture 6…)



Recurrent Neural Network - Backpropagation

Application of the traditional backpropagation algorithm to Recurrent Neural Networks: **Backpropagation Through Time** (BPTT)

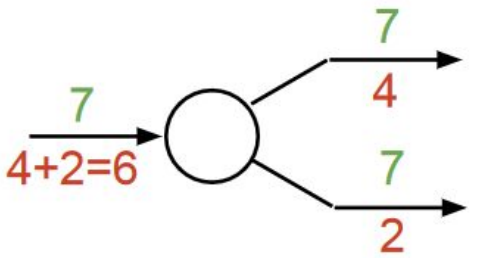


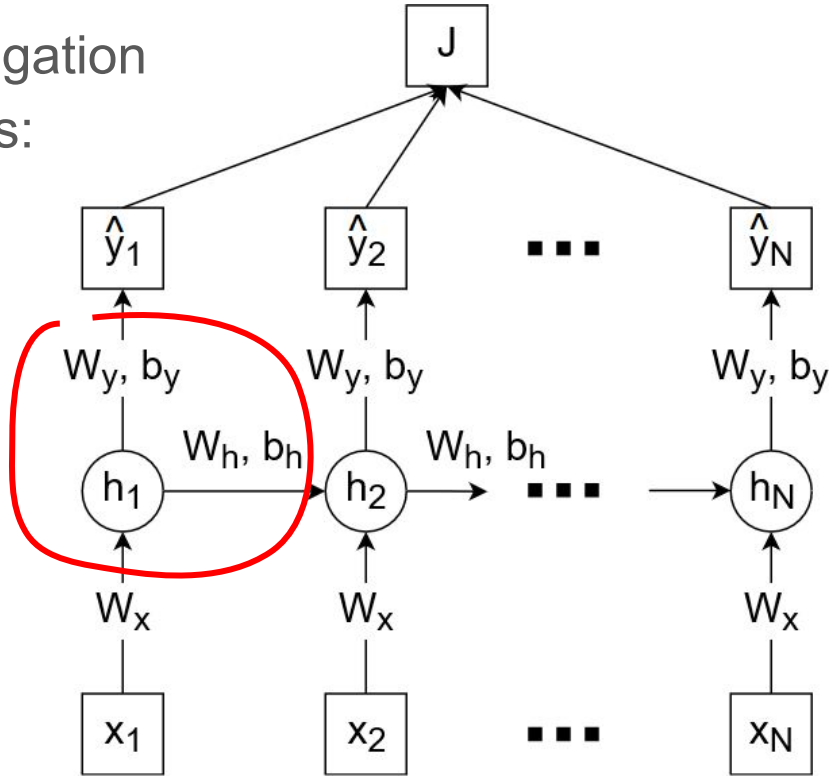
Recurrent Neural Network - Backpropagation

Application of the traditional backpropagation algorithm to Recurrent Neural Networks: **Backpropagation Through Time** (BPTT)

Backpropagation rule for **branching**:

copy gate: gradient adder





Recurrent Neural Network - Backpropagation

Backpropagation Through Time (BPTT)

Recurrent networks can have a large temporal extent, resulting in extremely long paths in the computational graph.

Consequence?

Recurrent Neural Network - Backpropagation

Backpropagation Through Time (BPTT)

Recurrent networks can have a large temporal extent, resulting in extremely long paths in the computational graph.

Consequence:

- **Training can only begin once the entire sequence is available.**
  → Cannot be used for continuous learning and prediction.
- It takes a **very long time** and a huge number of operations **to compute the entire backpropagation.**

Recurrent Neural Network - Backpropagation

Backpropagation Through Time (BPTT)

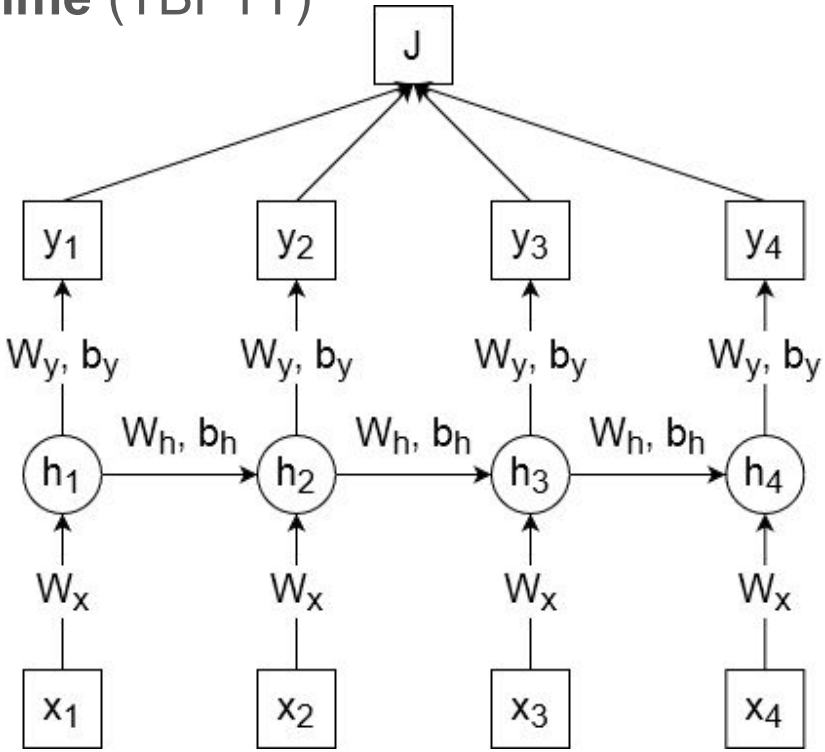
Recurrent networks can have a large temporal extent, resulting in extremely long paths in the computational graph.

Solution: Let’s modify the backpropagation algorithm slightly!

Recurrent Neural Network - Backpropagation

Truncated Backpropagation Through Time (TBPTT)

We **stop** backpropagation **after a given number of time steps**.

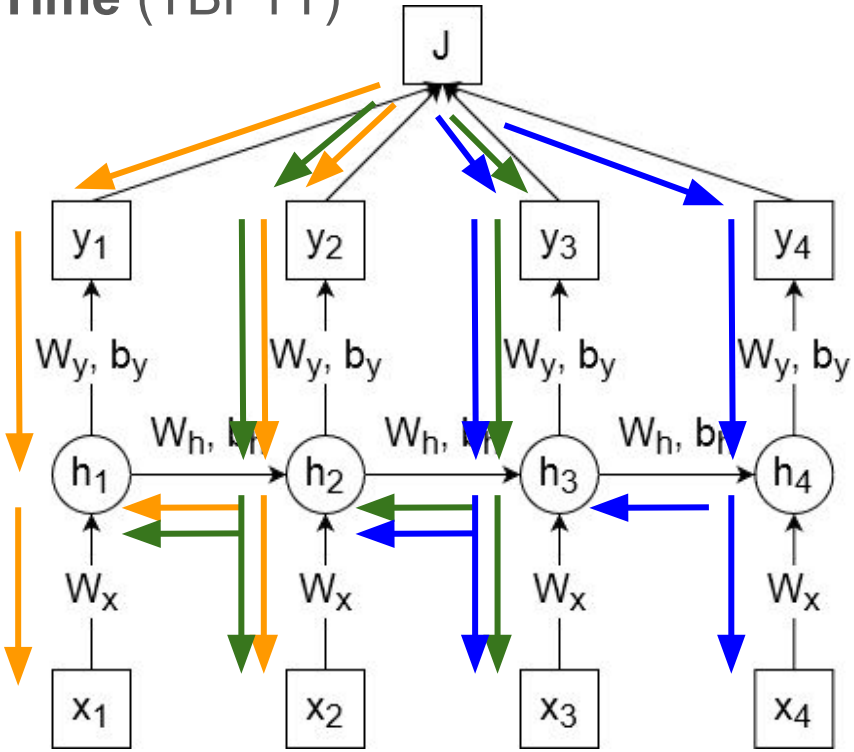


Recurrent Neural Network - Backpropagation

Truncated Backpropagation Through Time (TBPTT)

We **stop** backpropagation **after a given number of time steps**.

Implementation:
Create overlapping chunks from the time series using a sliding window approach.
One iteration is computed on one chunk or a batch of chunks, independently.

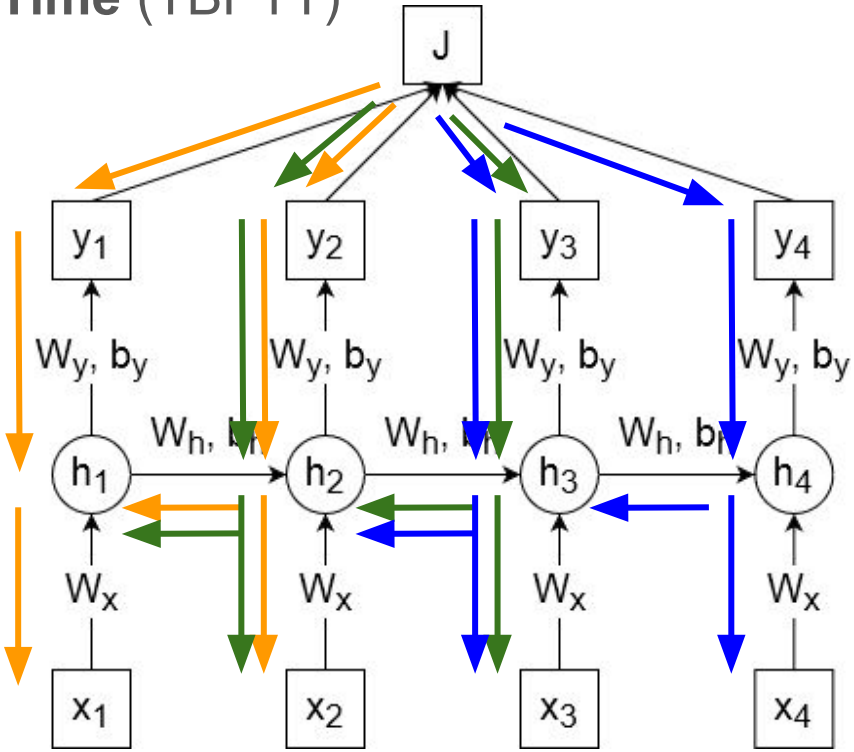


Recurrent Neural Network - Backpropagation

Truncated Backpropagation Through Time (TBPTT)

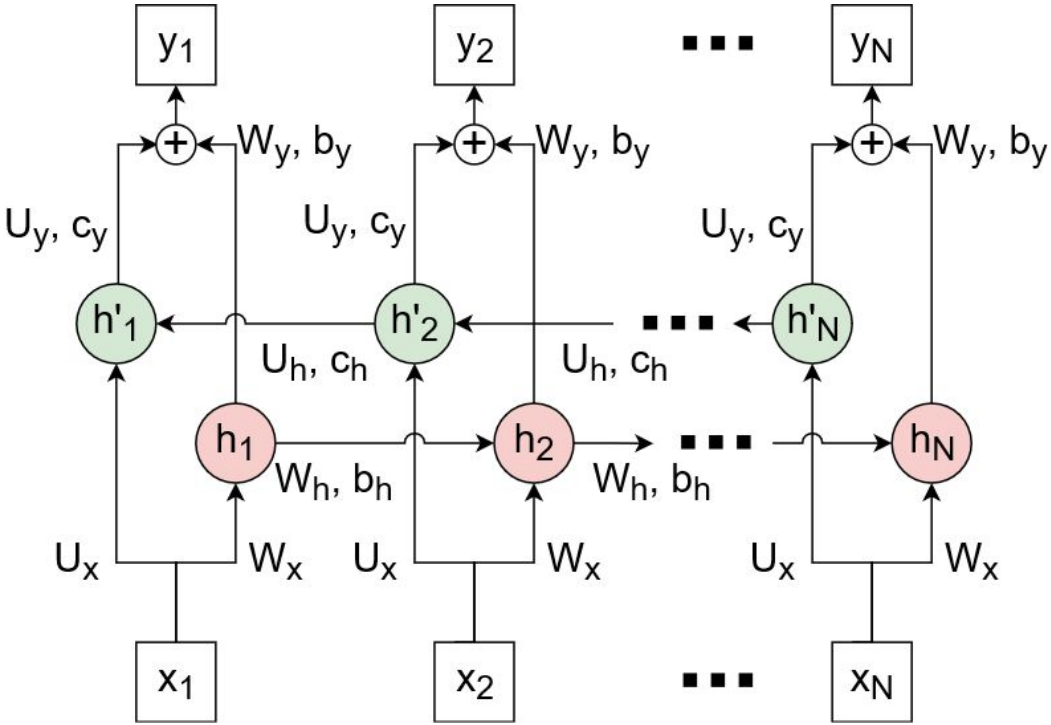
We **stop** backpropagation **after a given number of time steps**.

Disadvantage:
The size of the context is determined by the chunk size.
The network cannot learn to detect relationships in the data that span over a longer timescale.



Recurrent Neural Network - Variants

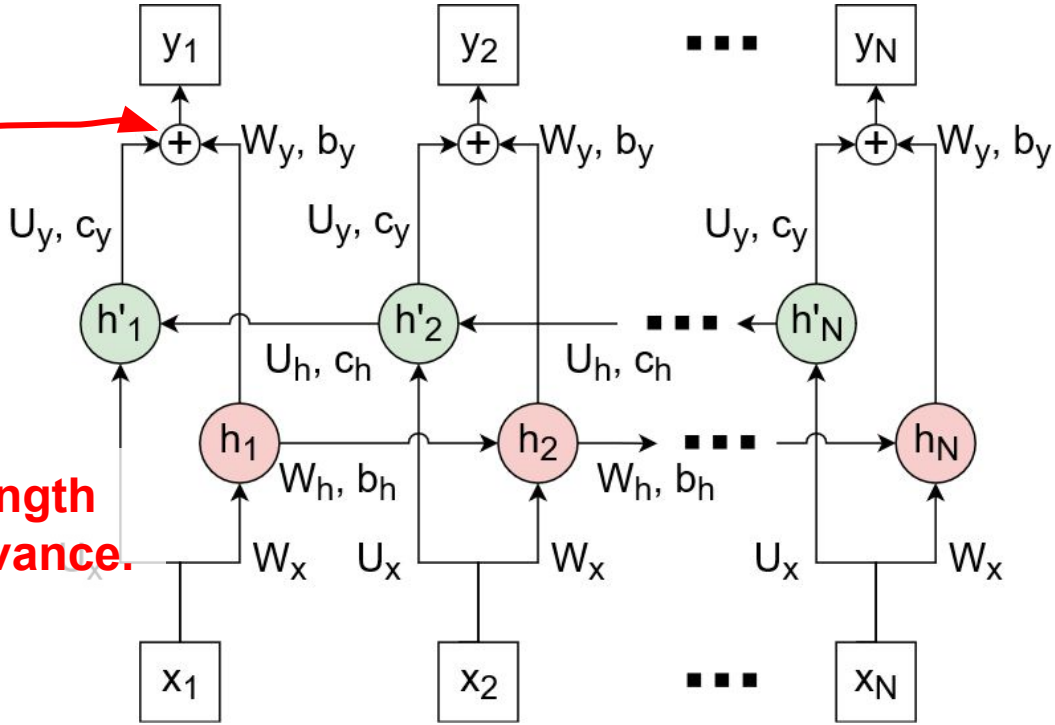
Bidirectional RNN



Recurrent Neural Network - Variants

Ideal for such **N → N** tasks, where the prediction of $\mathbf{y_t}$ requires information from later inputs.

Bidirectional RNN

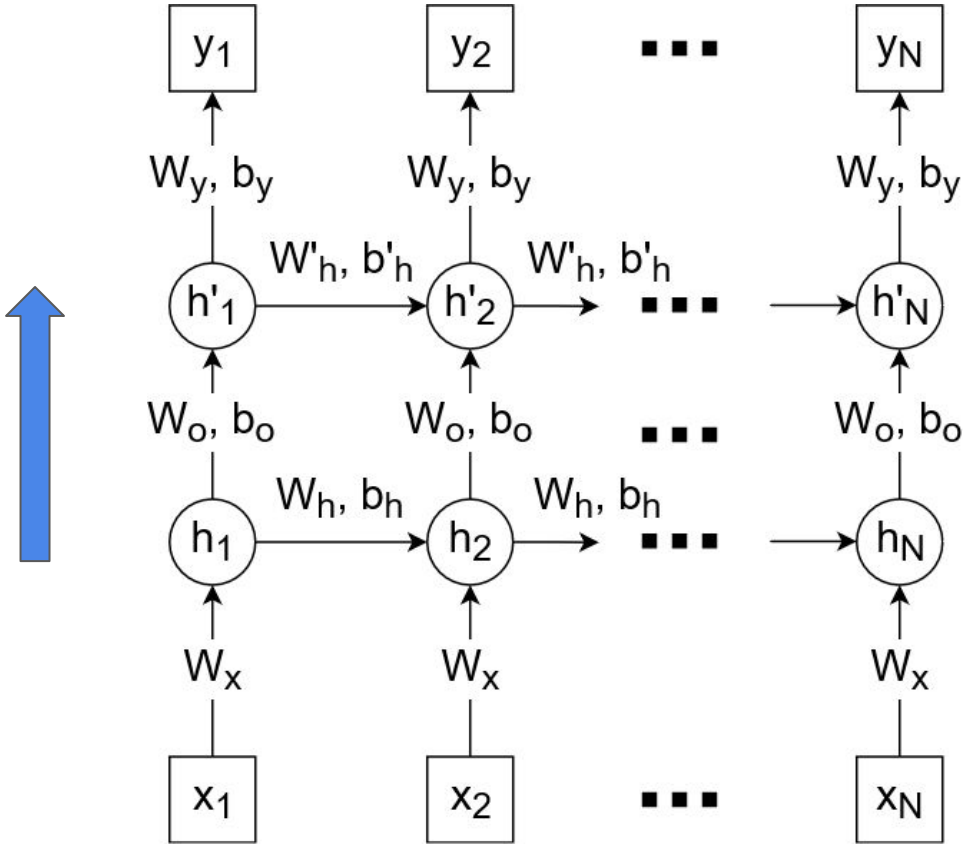


Aggregation: E.g., addition, (element-wise) multiplication or concatenation.

In case of Bidirectional RNNs, the length of the sequence must be fixed in advance.

Recurrent Neural Network - Variants

Multiple RNN layers can be stacked…



Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

Example: Let's try to predict the next character in a text based on the previous few characters!

Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

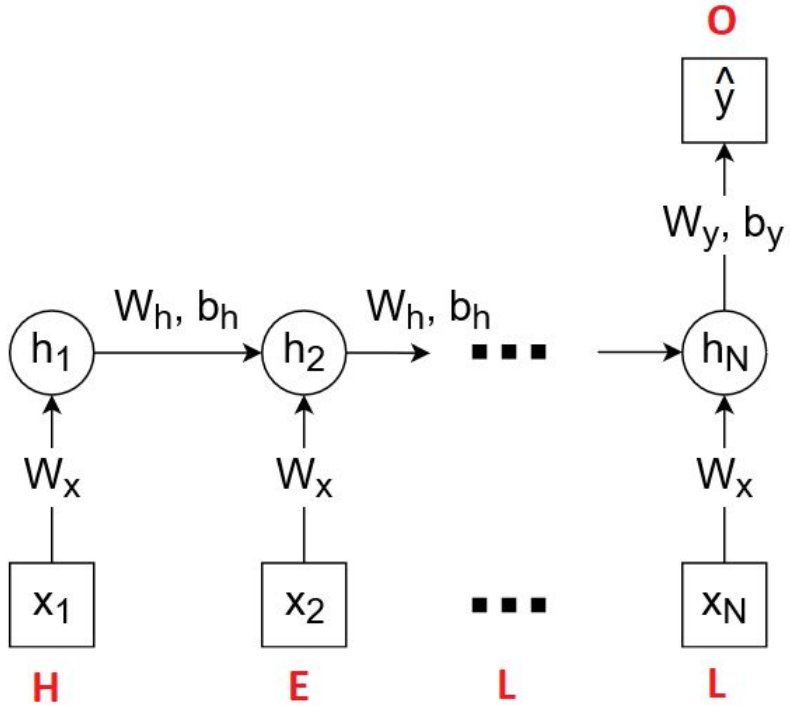
Example: Let's try to predict the next character in a text based on the previous few characters!

A naive approach to language representation: The text is treated as a time series, with letters as its elements. Letters can be represented as independent categories, such as probability vectors. For example, the English alphabet consists of 26 characters, so we can use 26-element one-hot vectors as both input and true labels (additional categories are needed for whitespace and punctuation marks).

Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

- With a **"N → 1"** model:
  Learn to **estimate element # t**
  from input **elements # t-N, …, # t-1**!

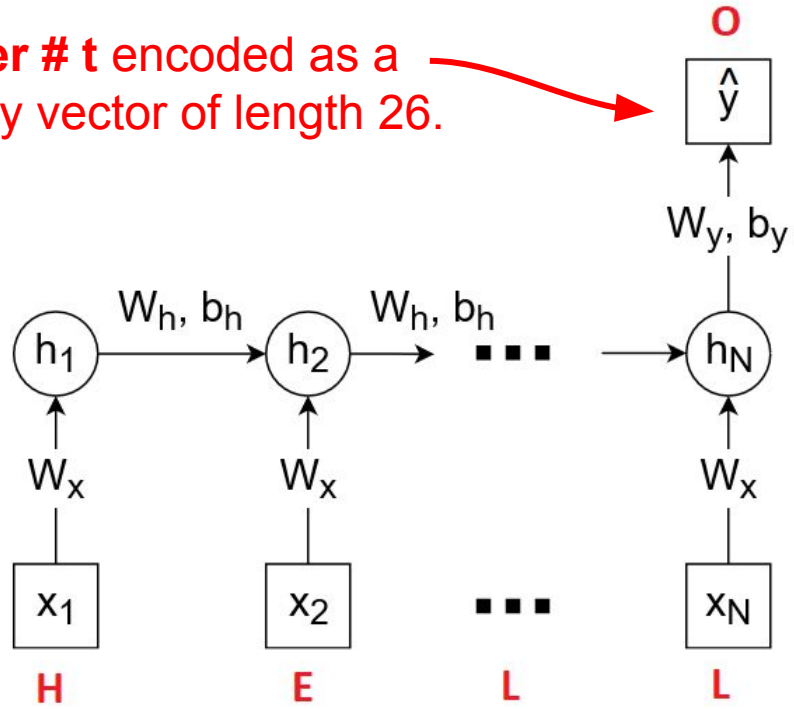


Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

- With a **"N → 1"** model:
  Learn to **estimate element # t**
  from input **elements # t-N, …, # t-1**!

Character # t encoded as a probability vector of length 26.

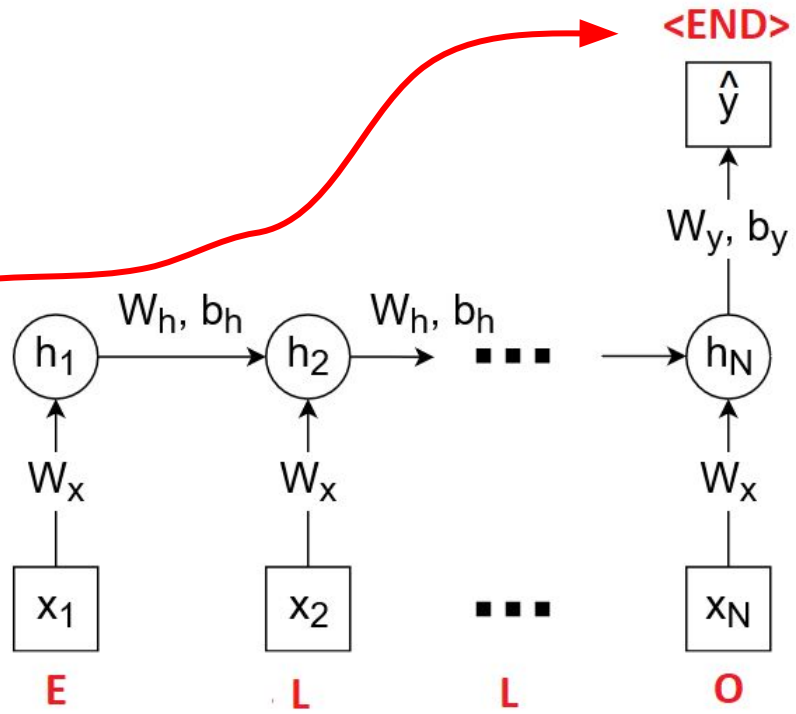


Character # t-N encoded as a probability vector of length 26.

Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

- With a **"N → 1"** model:
  Learn to **estimate element # t**
  from input **elements # t-N, …, # t-1**!



<END> token: The network needs to be able to indicate when the predicted text ends.
(Now, we need one more character category…)

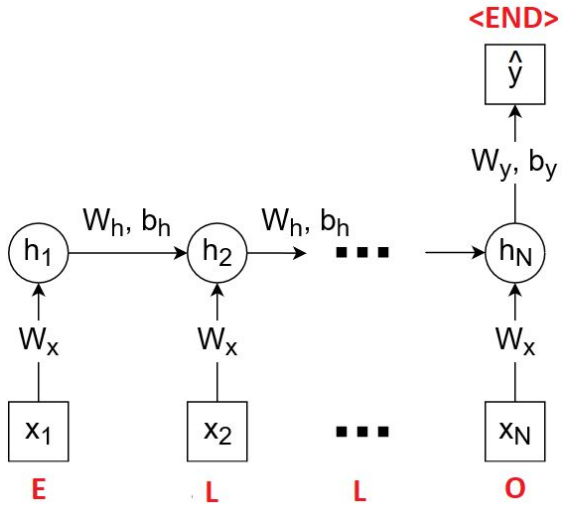
Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

- With a **“N → 1”** model: Learn to **estimate element # t from** input **elements # t-N, …, # t-1**!

This method isn't very efficient:
It takes an entire training iteration
to train the prediction for a single element.

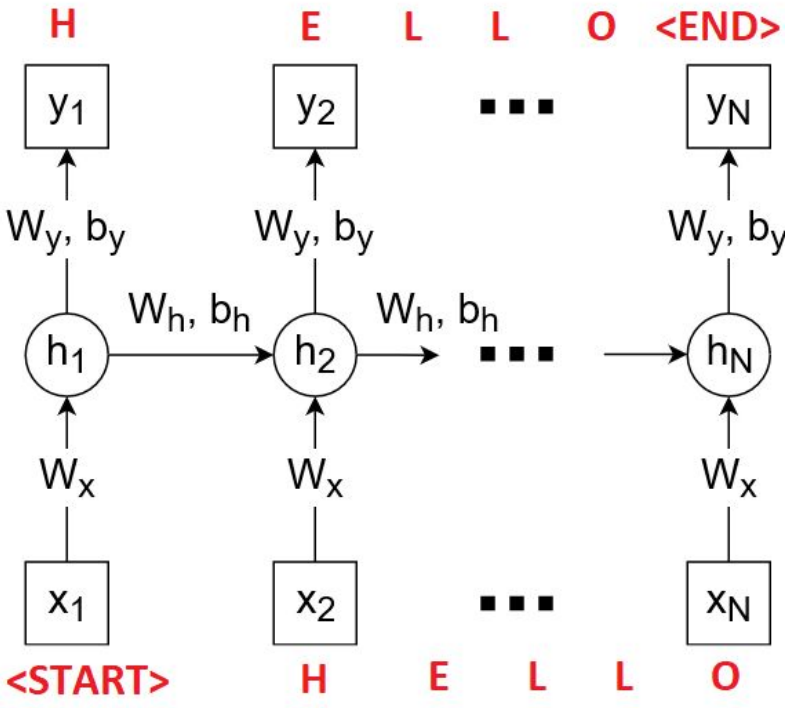
Is there a more efficient way?



Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

- With a "**N → N**" model:
  Learn to **estimate elements # t-N+1, …, # t from** input **elements # t-N, …, # t-1**!

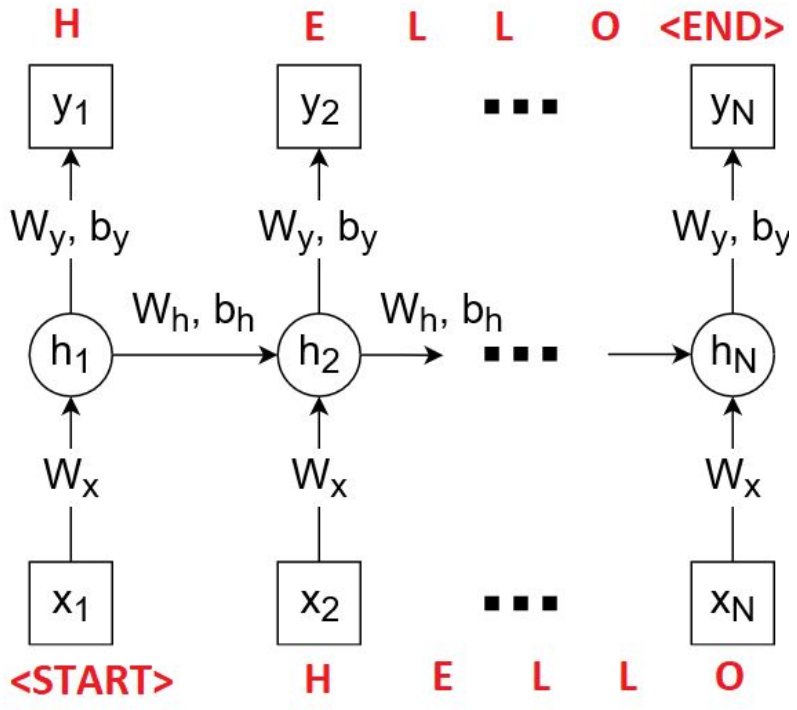


Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

- With a "**N → N**" model:
  Learn to **estimate elements # t-N+1, …, # t from** input **elements # t-N, …, # t-1**!

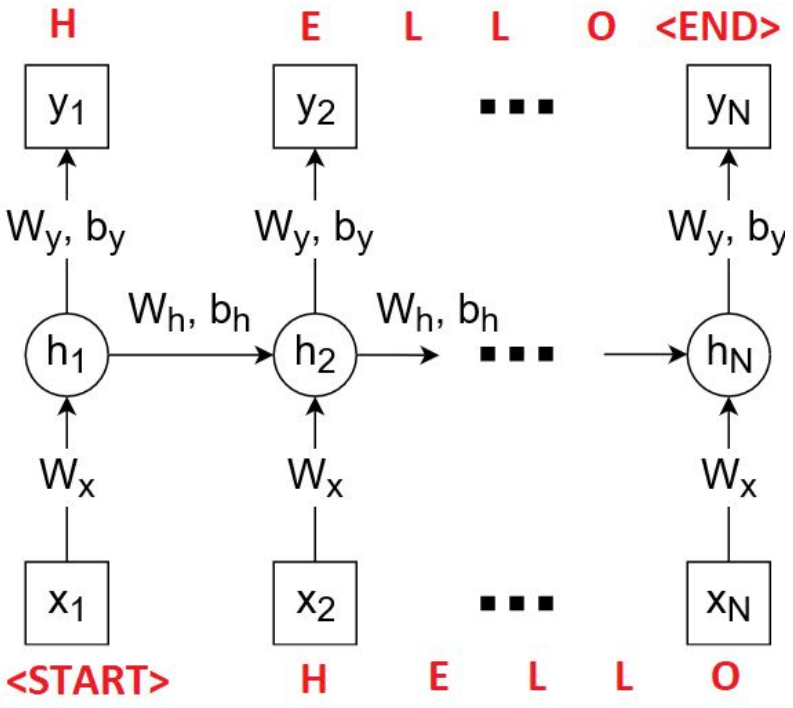
The **window is shifted to the right by one** over the time series to extract the target (output) sequence corresponding to a given input sequence.



Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

- With a "**N → N**" model:
  Learn to **estimate elements # t-N+1, …, # t from** input **elements # t-N, …, # t-1**!

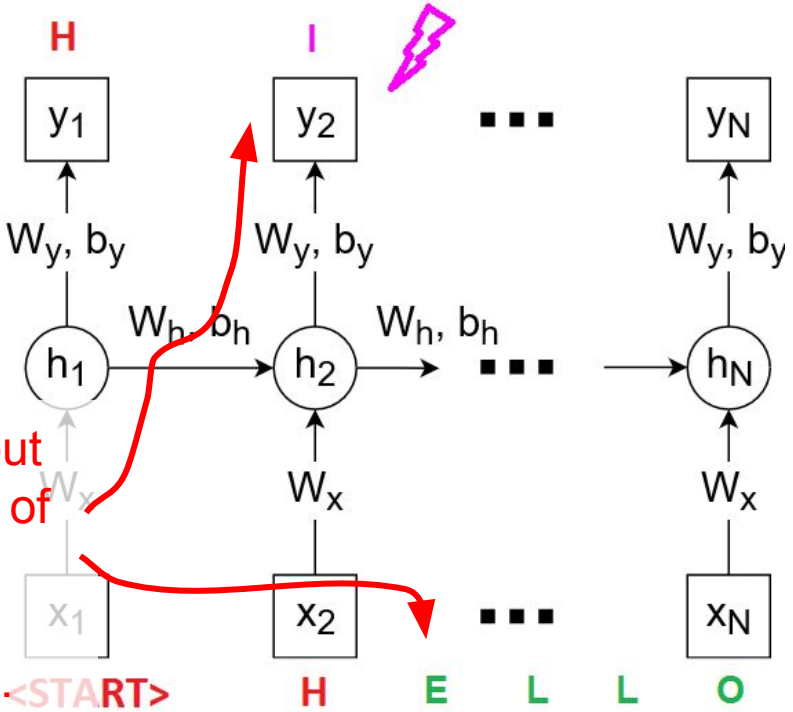


<START> token: Indication to the network that there was no preceding character.

Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

- With a "**N → N**" model:
  Learn to **estimate elements # t-N+1, …, # t from** input elements # t-N … # t-1!



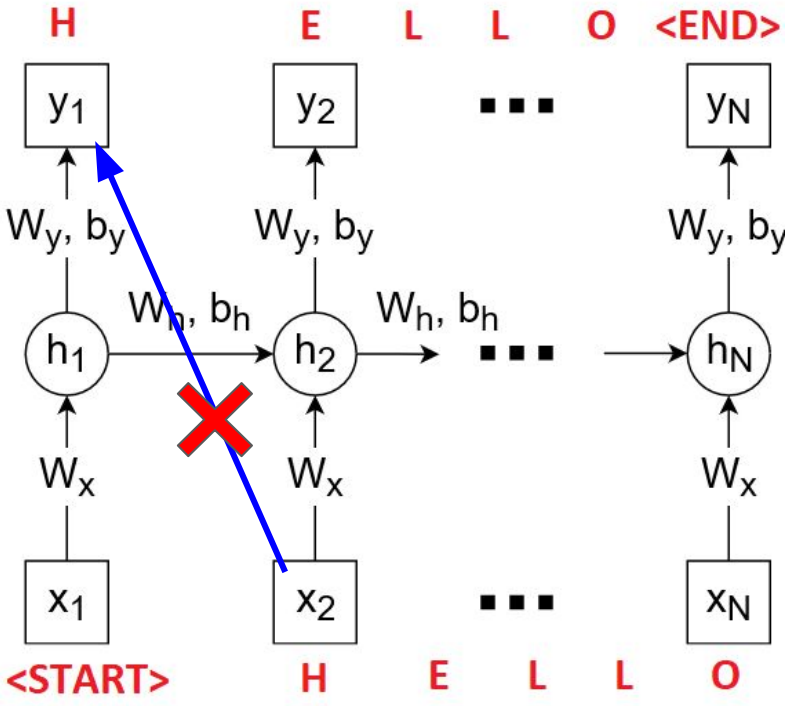
Teacher forcing: During training, the network input (and the target) is the training text. The estimates of the network are, thus, not used as the next input. During training, the model is forced to keep continuing the training text and not deviate from it.

Recurrent Neural Network - Time series prediction

Task: Predict the next element in a time series based on the last few elements!

- With a "**N → N**" model:
  Learn to **estimate**
  elements # t-N+1, …, # t
  from
  input **elements # t-N, …, # t-1**!

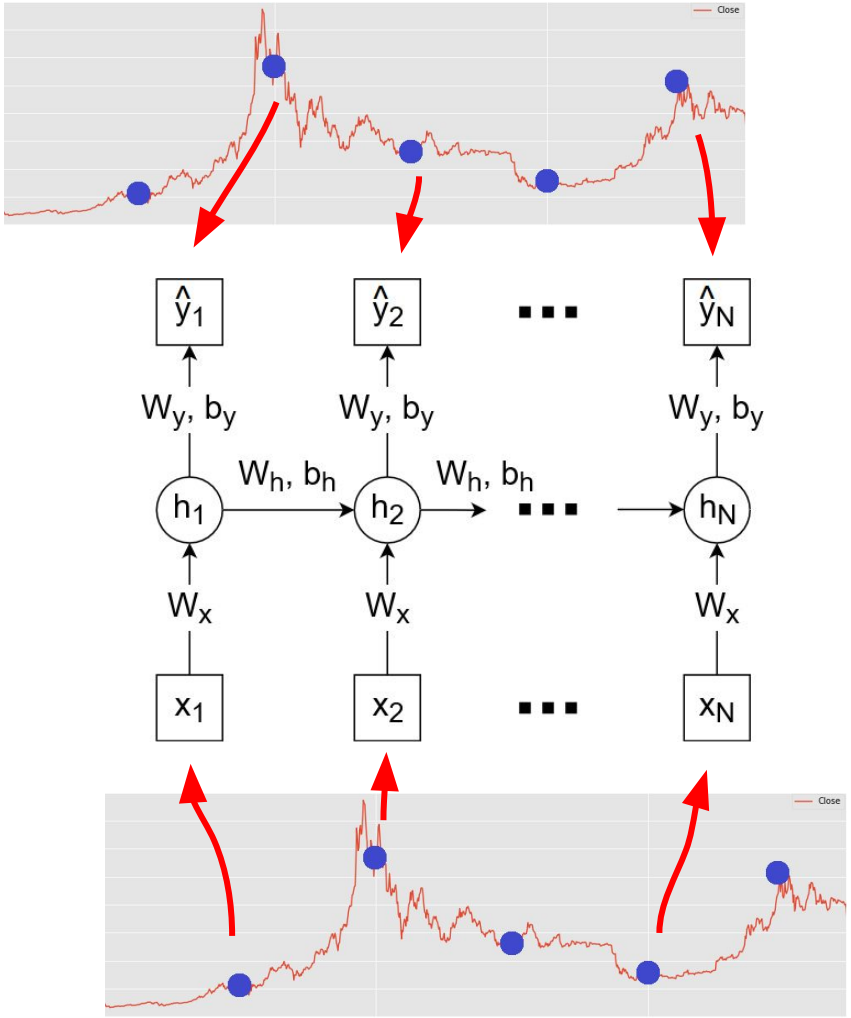
Due to **information propagating** in a **uni-directional** manner,
$\mathbf{y_1}$ cannot be "copied" from $\mathbf{x_2}$ in a trivial way.



RNN

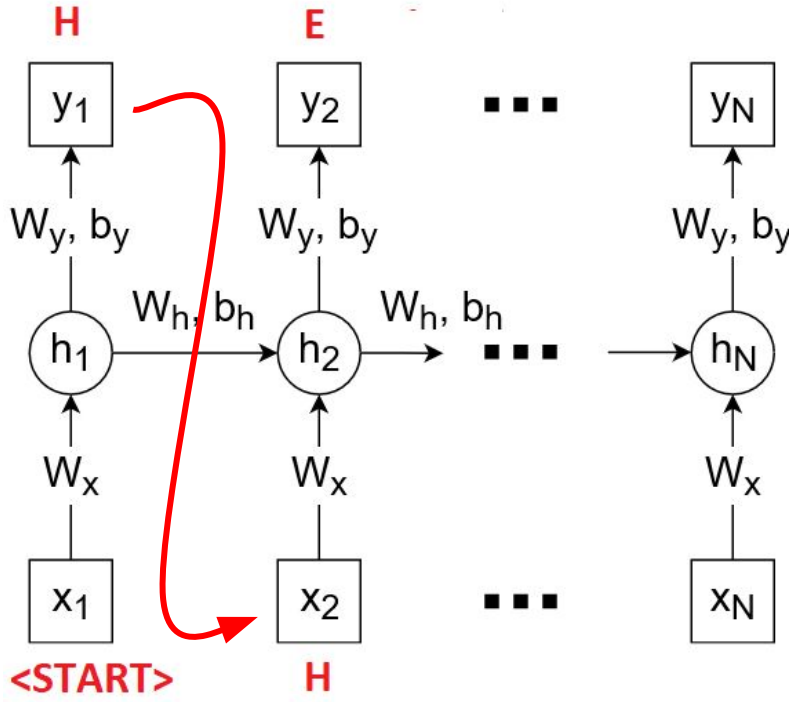
Time series prediction

Not limited to text data…



Recurrent Neural Network - Generative use

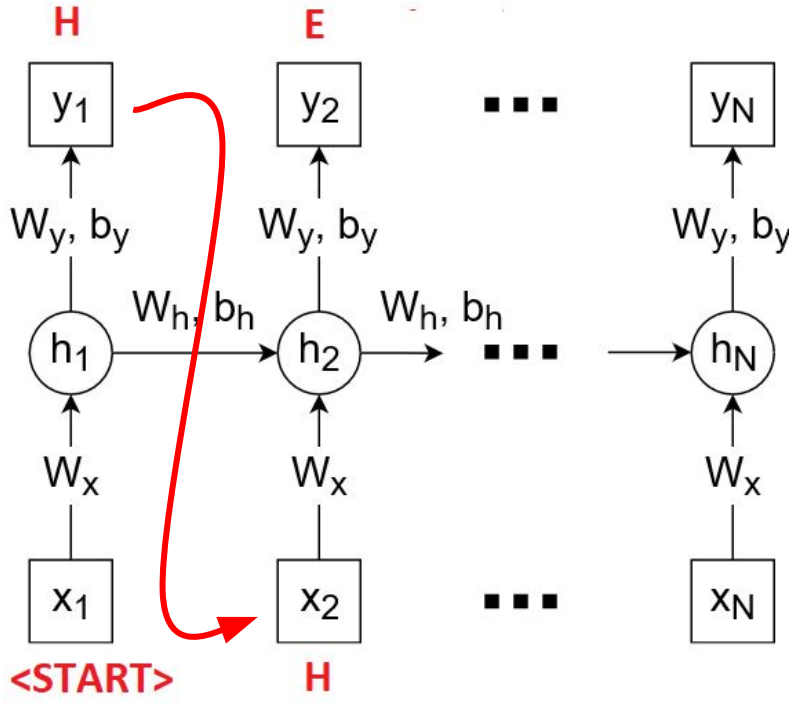
The trained RNN can be used to **generate new time series!**



Recurrent Neural Network - Generative use

The trained RNN can be used to **generate new time series!**

Let's estimate the first label, $\mathbf{y_1}$, using the input $\mathbf{x_1}$ **:= <START>**! Let's continue estimating the remaining labels one by one, substituting the previously estimated $\mathbf{y_{t-1}}$ into the input $\mathbf{x_t}$. The generation of the time series ends when the network estimates the **<END>** token.



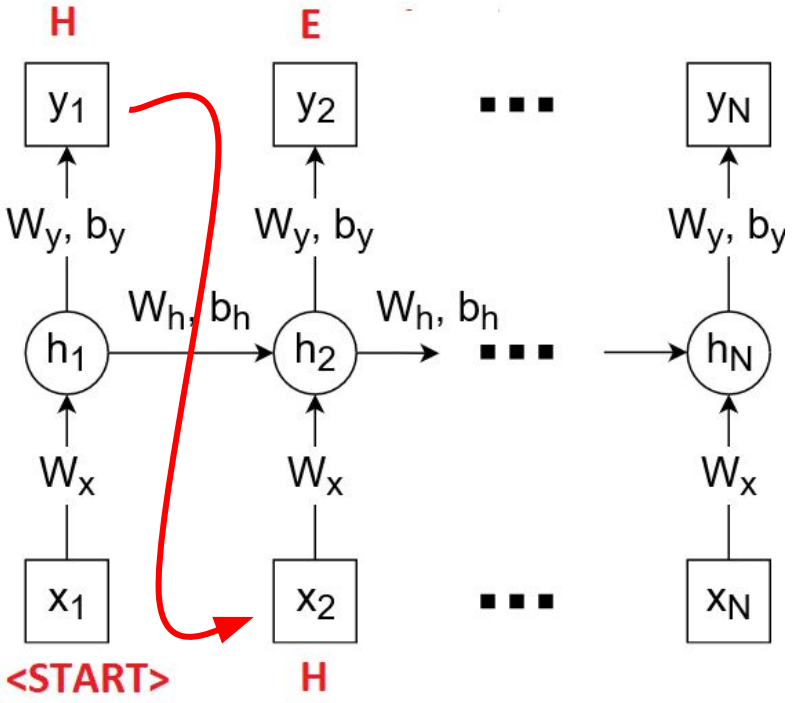
Recurrent Neural Network - Generative use

The trained RNN can be used to **generate new time series!**

This way, the trained network will **always generate exactly the same sequence…**

A different output can be obtained:

- If we use $\mathbf{y_{t-1}}$ with added noise as input in $\mathbf{x_t}$
- If we start with an initial sequence instead of just the **<START>** token.



Source: Stanford CS231n

Generative example

RNN (LSTM) **trained to predict the continuation of Shakespare plays. Then, the network was used to generate new text.**
(naive character-based language representation).

at first:

```
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
```

train more

```
"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

train more

```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.
```

train more

```
"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.
```

Source: Stanford CS231n

Generative example

The same procedure with LaTeX code…

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.*

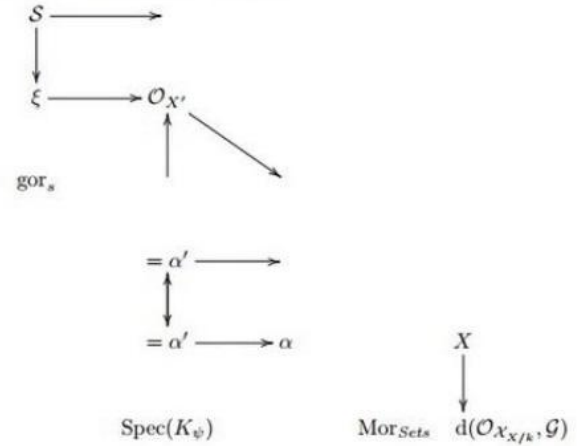
Proof. See Spaces, Lemma ??. □

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

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



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

Generative example

The same procedure with C code…

```
static void do_command(struct seq_file *m, void *v)
{
  int column = 32 << (cmd[2] & 0x80);
  if (state)
    cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
  else
    seq = 1;
  for (i = 0; i < 16; i++) {
    if (k & (1 << 1))
      pipe = (in_use & UMXTHREAD_UNCCA) +
        ((count & 0x00000000fffffff8) & 0x000000f) << 8;
    if (count == 0)
      sub(pid, ppc_md.kexec_handle, 0x20000000);
    pipe_set_bytes(i, 0);
  }
  /* Free our user pages pointer to place camera if all dash */
  subsystem_info = &of_changes[PAGE_SIZE];
  rek_controls(offset, idx, &soffset);
  /* Now we want to deliberately put it to device */
  control_check_polarity(&context, val, 0);
  for (i = 0; i < COUNTER; i++)
    seq_puts(s, "policy ");
}
```

Recurrent Neural Network - “**M → N**” variant

Example: Machine Translation

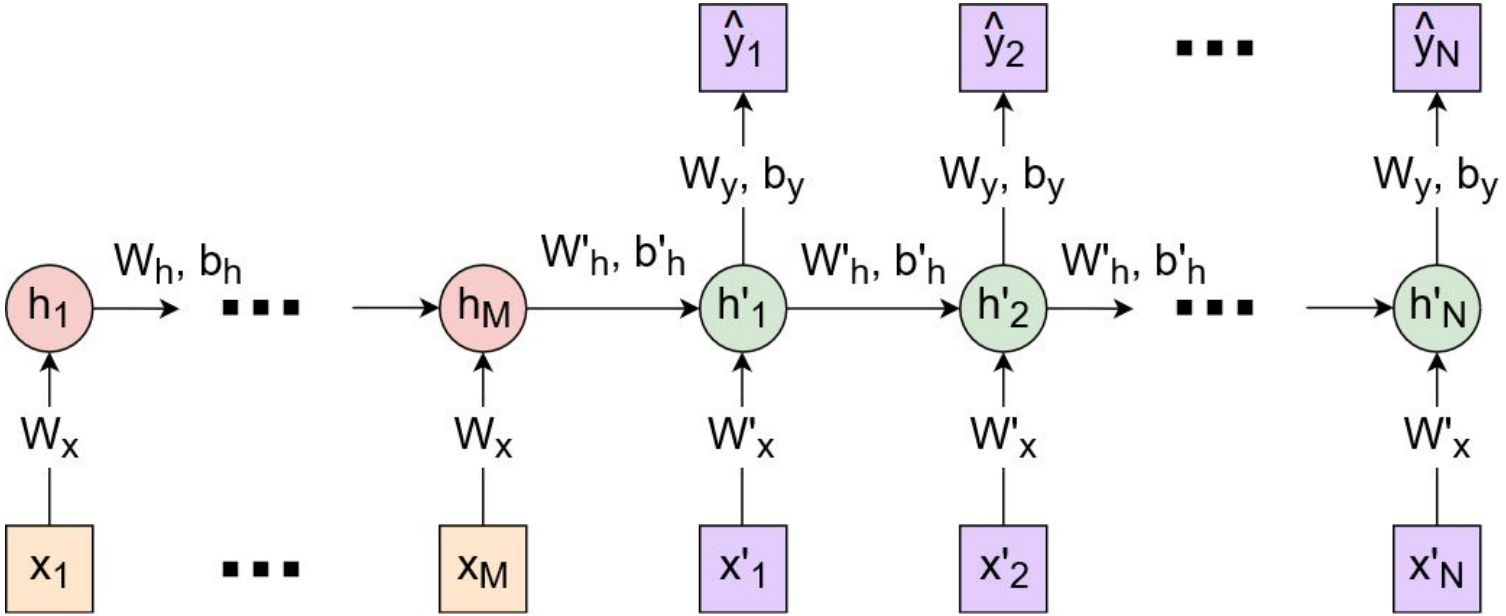
We need an architecture where:

- The **input and output** sequences are **not necessarily the same length.**
- The **length of the output** is **unknown**.
- **We must first see the entire input** (or at least one sentence) before we begin translating it, since word order may differ between the two languages.

RNN, Sequence-to-sequence (seq2seq)

Seq2seq architecture

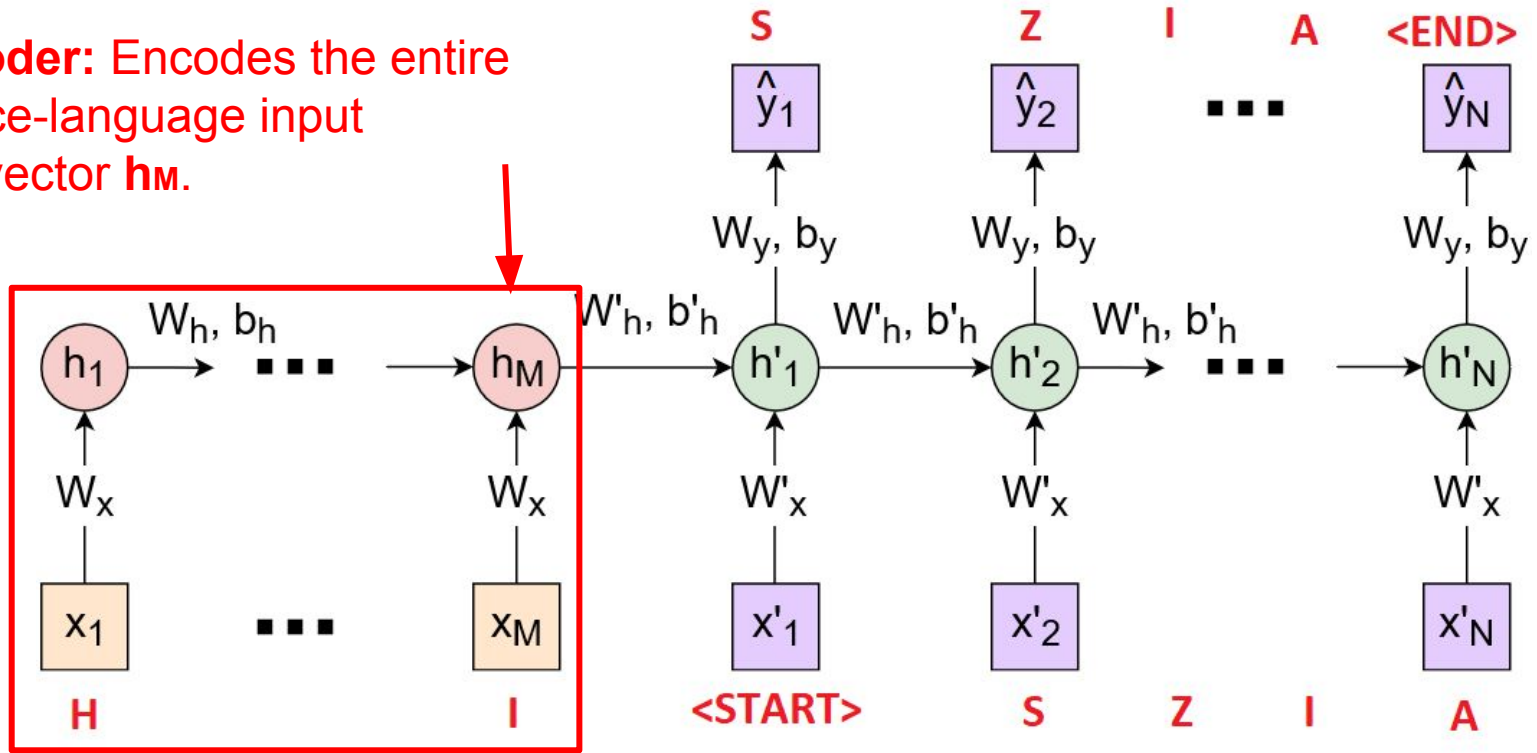
E.g., Google Translate, 2016 - 2020



RNN, Sequence-to-sequence (seq2seq)

Seq2seq architecture

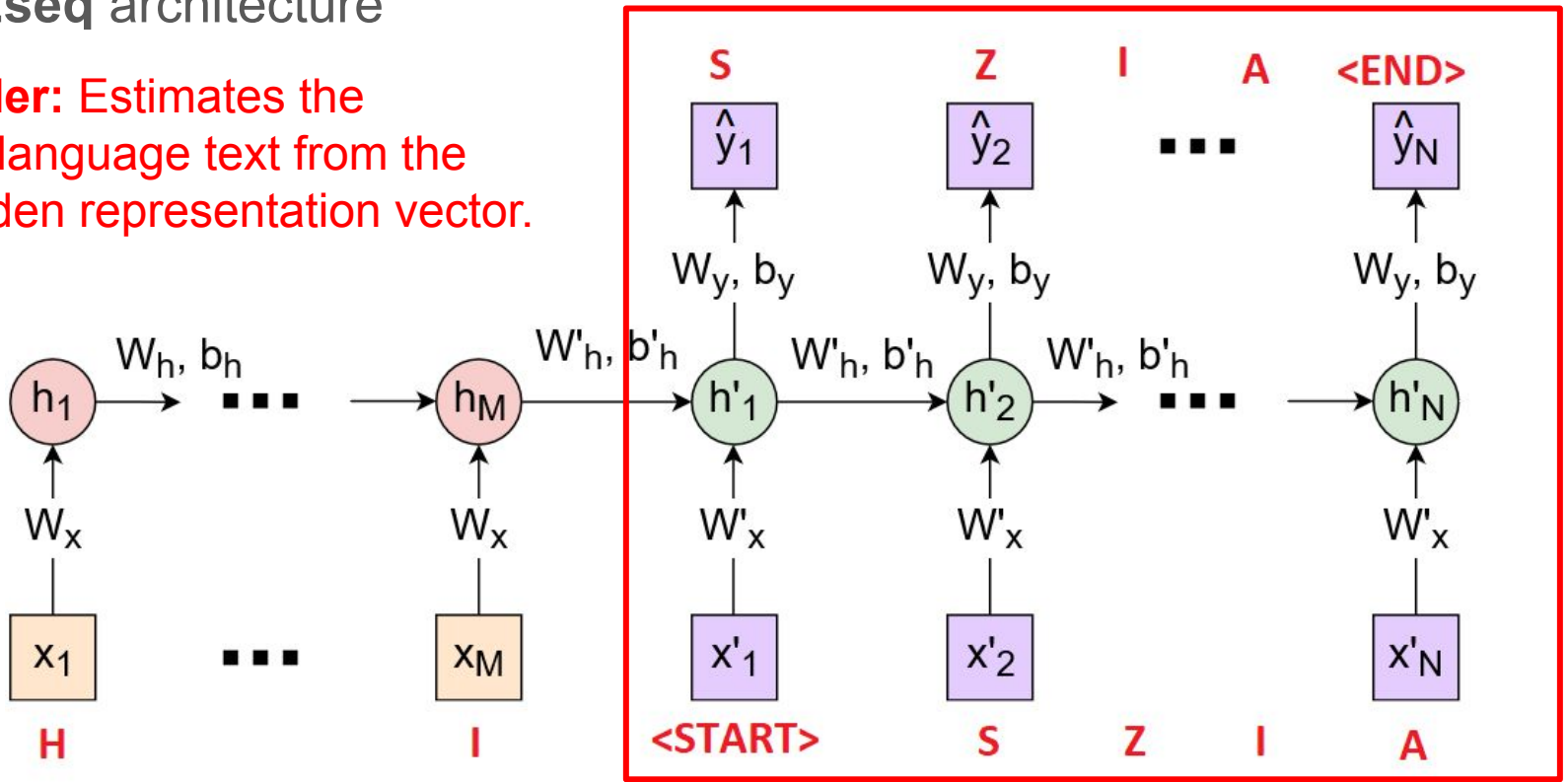
Encoder: Encodes the entire source-language input into vector $\mathbf{h_M}$.



RNN, Sequence-to-sequence (seq2seq)

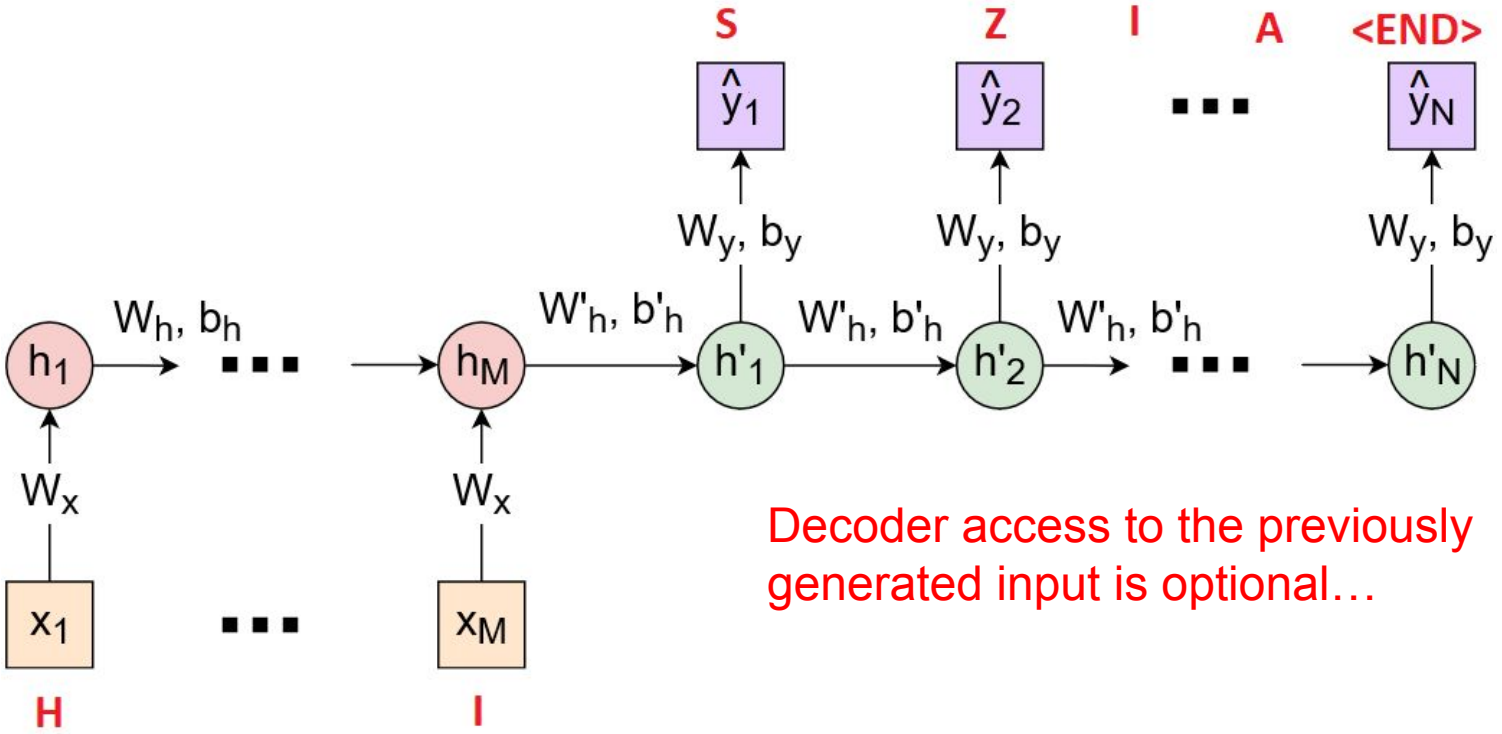
Seq2seq architecture

Decoder: Estimates the target-language text from the $\mathbf{h_M}$ hidden representation vector.



RNN, Sequence-to-sequence (seq2seq)

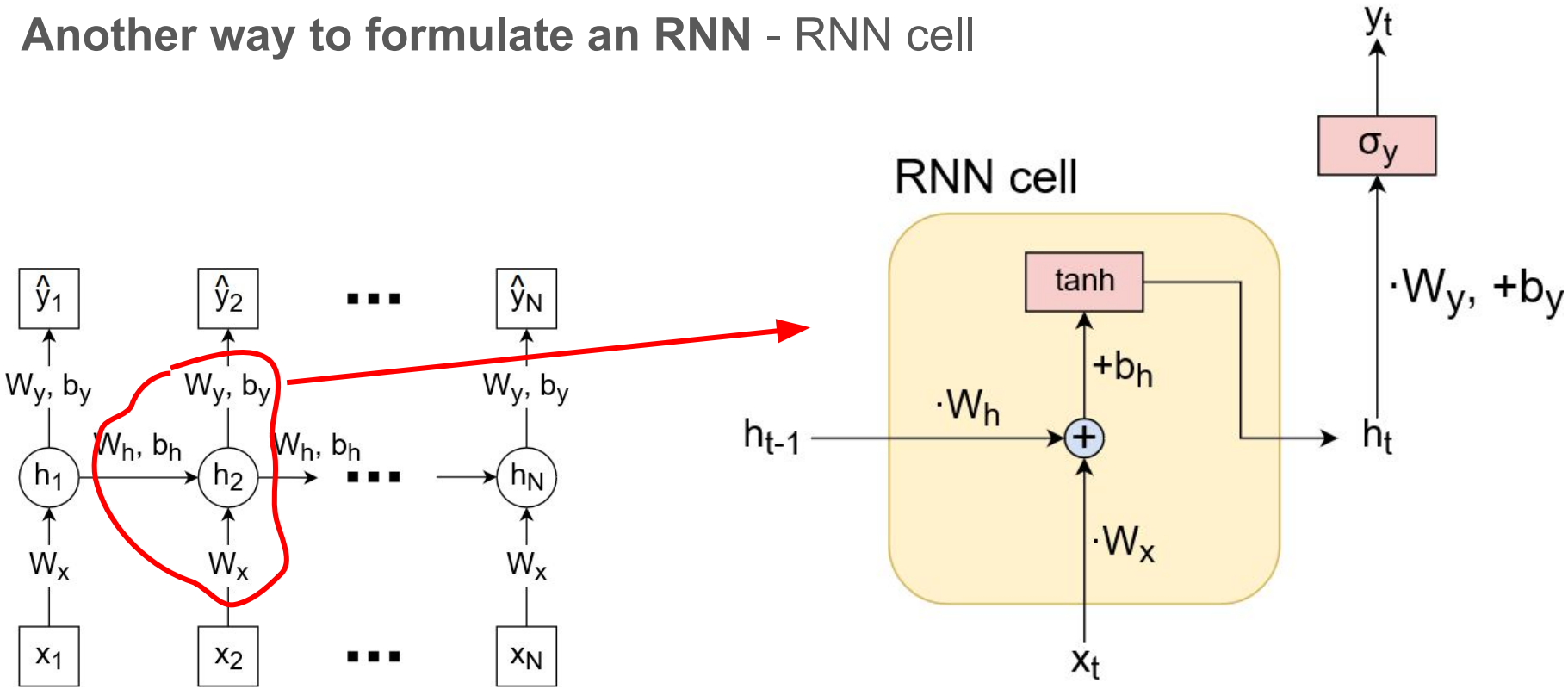
Seq2seq architecture



Decoder access to the previously generated input is optional…

Recurrent Neural Network - “RNN cell”

Another way to formulate an RNN - RNN cell

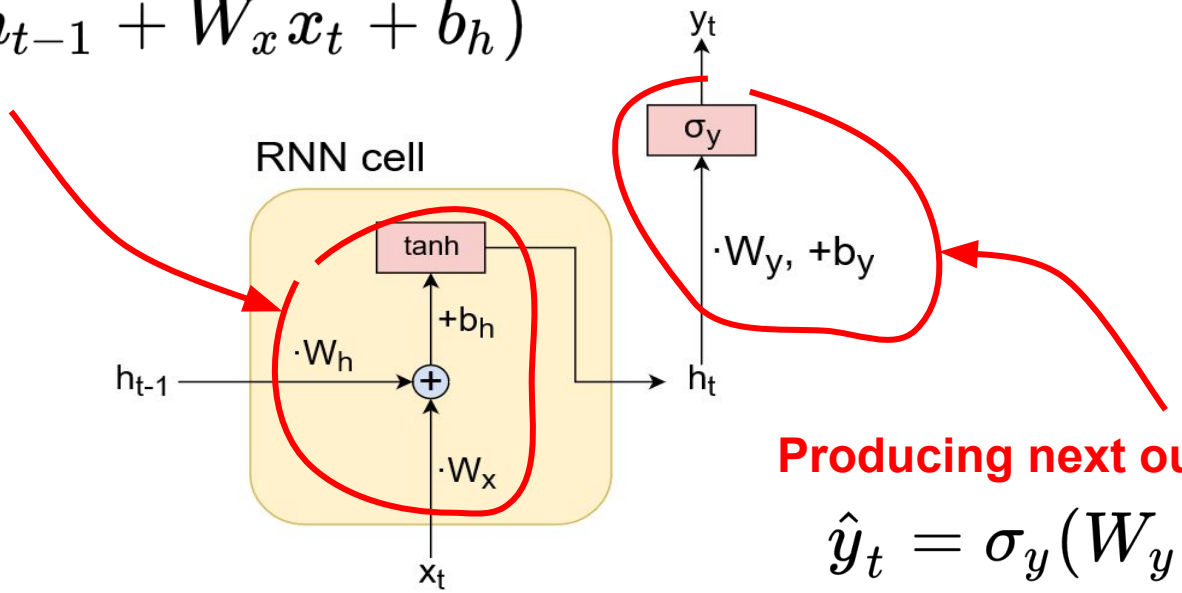


Recurrent Neural Network - “RNN cell”

RNN cell

Updating hidden representation (temporal step)

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



Producing next output

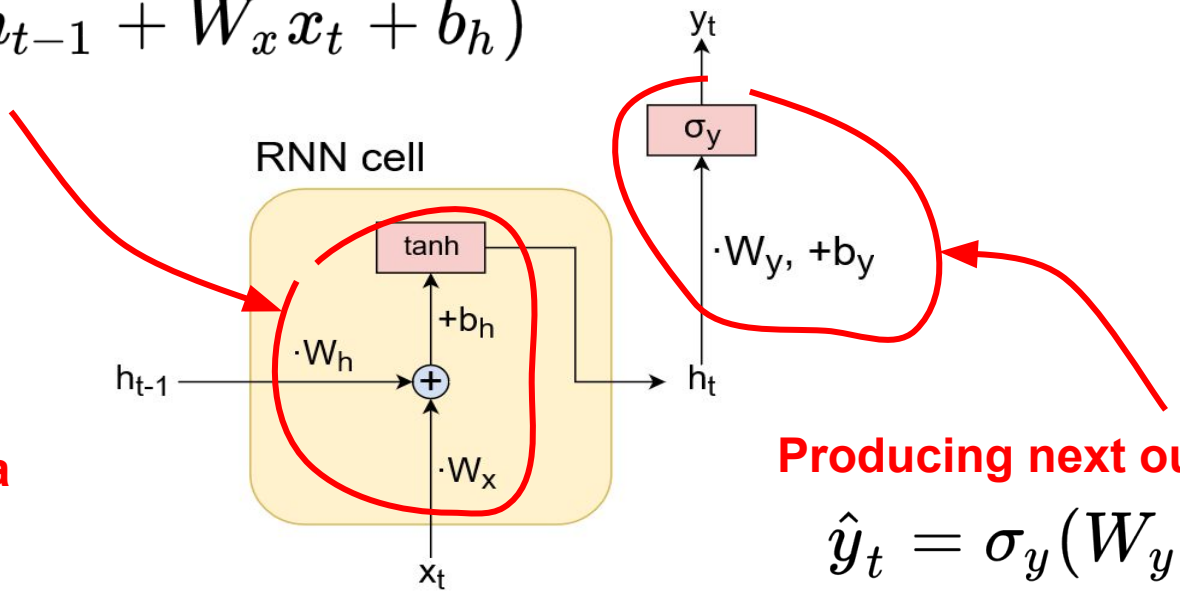
$$\hat{y}_t = \sigma_y(W_y h_t + b_y)$$

Recurrent Neural Network - “RNN cell”

RNN cell

Updating hidden representation (temporal step)

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



Is it easy to train a “Vanilla” RNN?

Producing next output

$$\hat{y}_t = \sigma_y(W_y h_t + b_y)$$

The unstable gradient problem in RNNs

Is it easy to train a “Vanilla” RNN?

With long sequences, the same problem arises as with deep networks:

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

$$\frac{\partial h_t}{\partial h_{t-1}} = ?$$

(scalar case)

The unstable gradient problem in RNNs

Is it easy to train a “Vanilla” RNN?

With long sequences, the same problem arises as with deep networks:

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

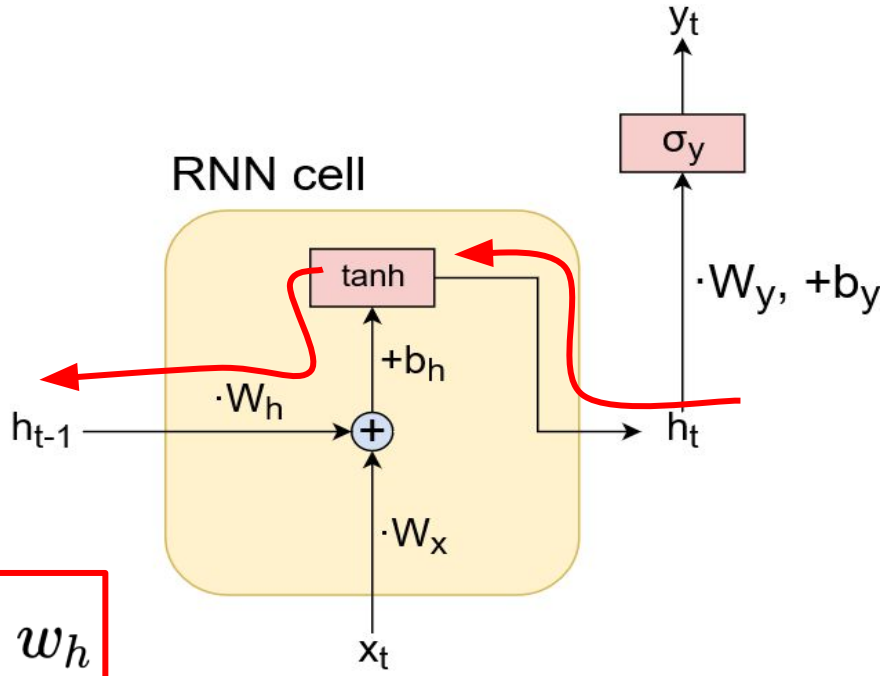
$$\frac{\partial h_t}{\partial h_{t-1}} = \sigma_h'(w_h h_{t-1} + w_x x_t + b_h) \cdot w_h$$

(scalar case)

The unstable gradient problem in RNNs

RNN cell - **Backpropagation**

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



During backpropgation, the gradient is multiplied by this in each time step:
(scalar case)

$$\frac{\partial h_t}{\partial h_{t-1}} = \sigma'_h(w_h h_{t-1} + w_x x_t + b_h) \cdot w_h$$

Reminder - The unstable gradient problem

During backpropagation in a deep network **many of these terms are multiplied together** to get the update of a given parameter:

$$\frac{\partial \hat{y}}{\partial x} = g'(wx+b) \cdot w$$

Reminder: Classic FC layer, scalar case

Result: In a network with too many layers, different parts of the network may learn at completely different speeds: while the updates for some parameters are negligible, others are so large that their learning does not converge.

The unstable gradient problem in RNNs

The problem is even more severe in case of recurrent networks:

$$\frac{\partial h_t}{\partial h_{t-1}} = \sigma_h'(w_h h_{t-1} + w_x x_t + b_h) \cdot w_h$$

The weights are shared across time, so we **multiply the gradient over and over again by the same number** (matrix) in each time step.

The unstable gradient problem in RNNs

The problem is even more severe in case of recurrent networks:

$$\frac{\partial h_t}{\partial h_{t-1}} = \sigma_h'(w_h h_{t-1} + w_x x_t + b_h) \cdot w_h$$

The weights are shared across time, so we **multiply the gradient over and over again by the same number** (matrix) in each time step.

Consequence: Classic “Vanilla” RNNs will usually be **unable to learn to detect relationships** in the data that span a timescale **of more than 10–15 steps.**

The unstable gradient problem in RNNs

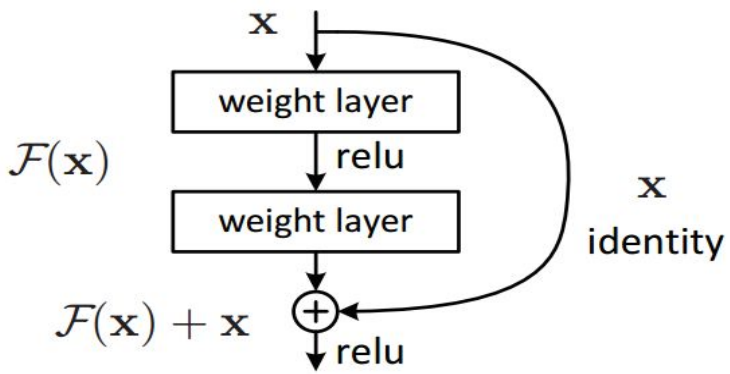
How could we reduce the impact of the unstable gradient problem in recurrent neural networks?

The unstable gradient problem in RNNs

How could we reduce the impact of the unstable gradient problem in recurrent neural networks?

Reminder: Residual networks

The gradient flows backwards uninterrupted through the skip (“+x”) connection.

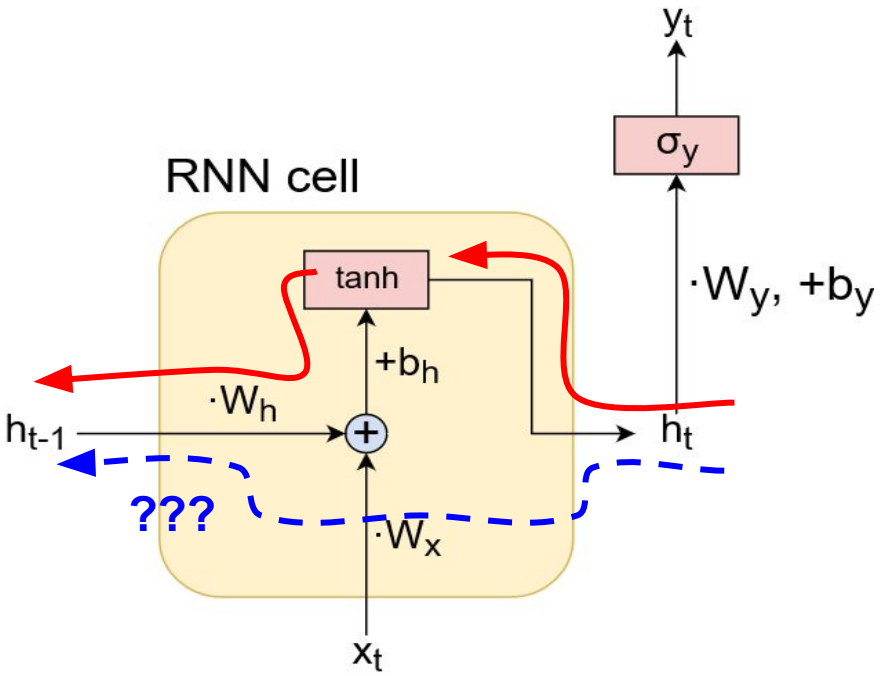


The unstable gradient problem in RNNs

How could we reduce the impact of the unstable gradient problem in recurrent neural networks?

Reminder: Residual networks

Similarly, a **skip connection in RNNs**?

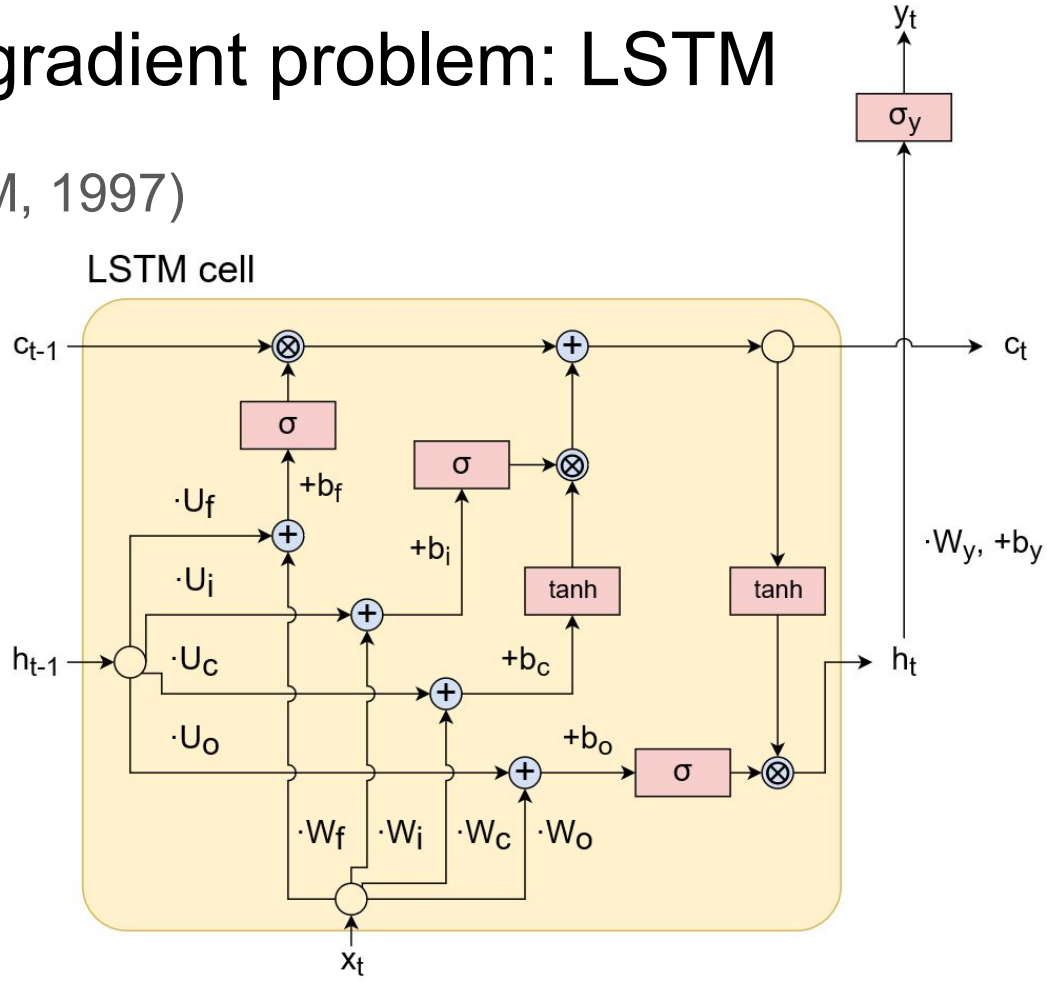


Mitigating the unstable gradient problem: LSTM

Long short-term memory (LSTM, 1997)

Mitigating the unstable gradient problem: LSTM

Long short-term memory (LSTM, 1997)

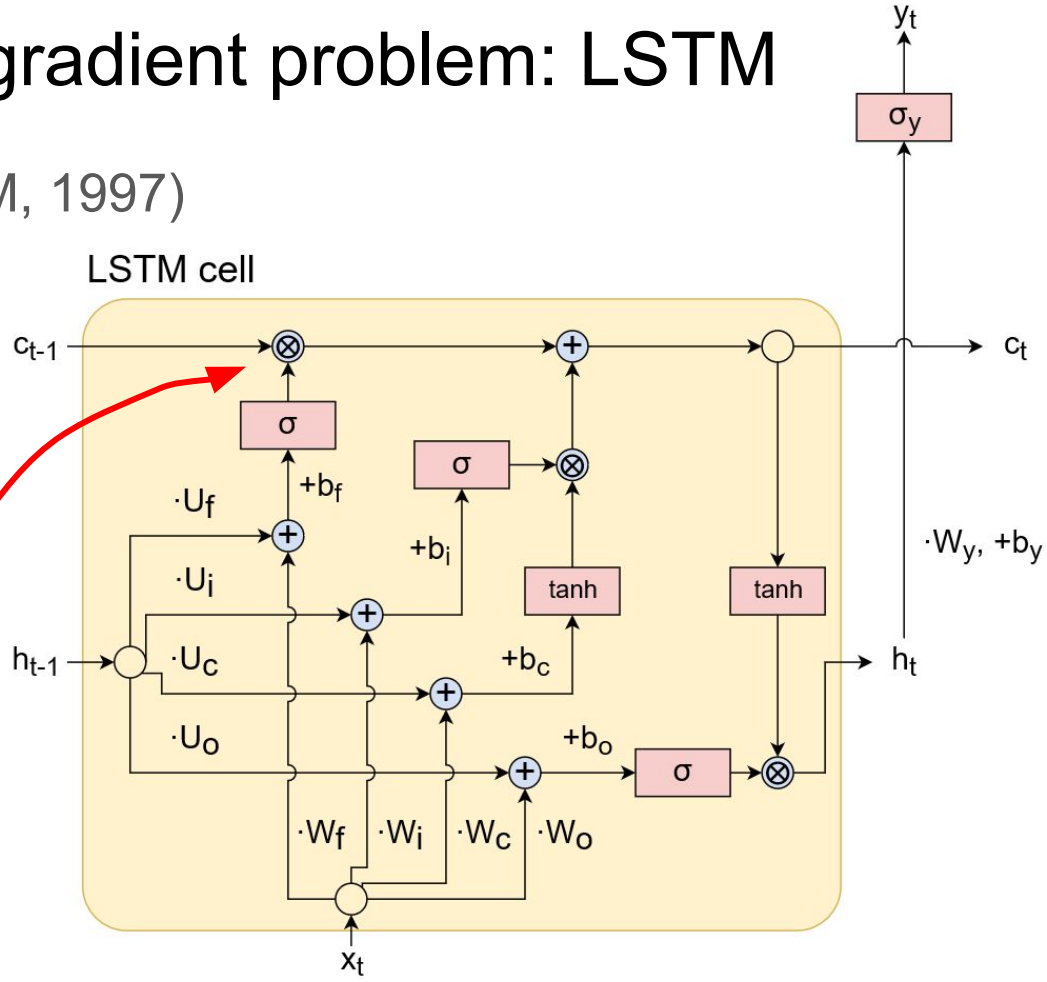


Mitigating the unstable gradient problem: LSTM

Long short-term memory (LSTM, 1997)

c (cell state): Another hidden representation vector implementing a “memory”.

Hadamard (or Schur) **- product**: Element-wise multiplication of two vectors.



Mitigating the unstable gradient problem: LSTM

LSTM equations

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$\tilde{c}_t = tanh(W_c x_t + U_c h_{t-1} + b_c)$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$
$$h_t = o_t \circ tanh(c_t)$$

Hadamard (or Schur) **- product**

$$x_t \in \mathbb{R}^n \quad f_t, i_t, o_t, h_t, \tilde{c}_t, c_t \in \mathbb{R}^s$$
$$W_f, W_i, W_o, W_c \in \mathbb{R}^{s \times n} \quad U_f, U_i, U_o, U_c \in \mathbb{R}^{s \times s} \quad b_f, b_i, b_o, b_c \in \mathbb{R}^s$$

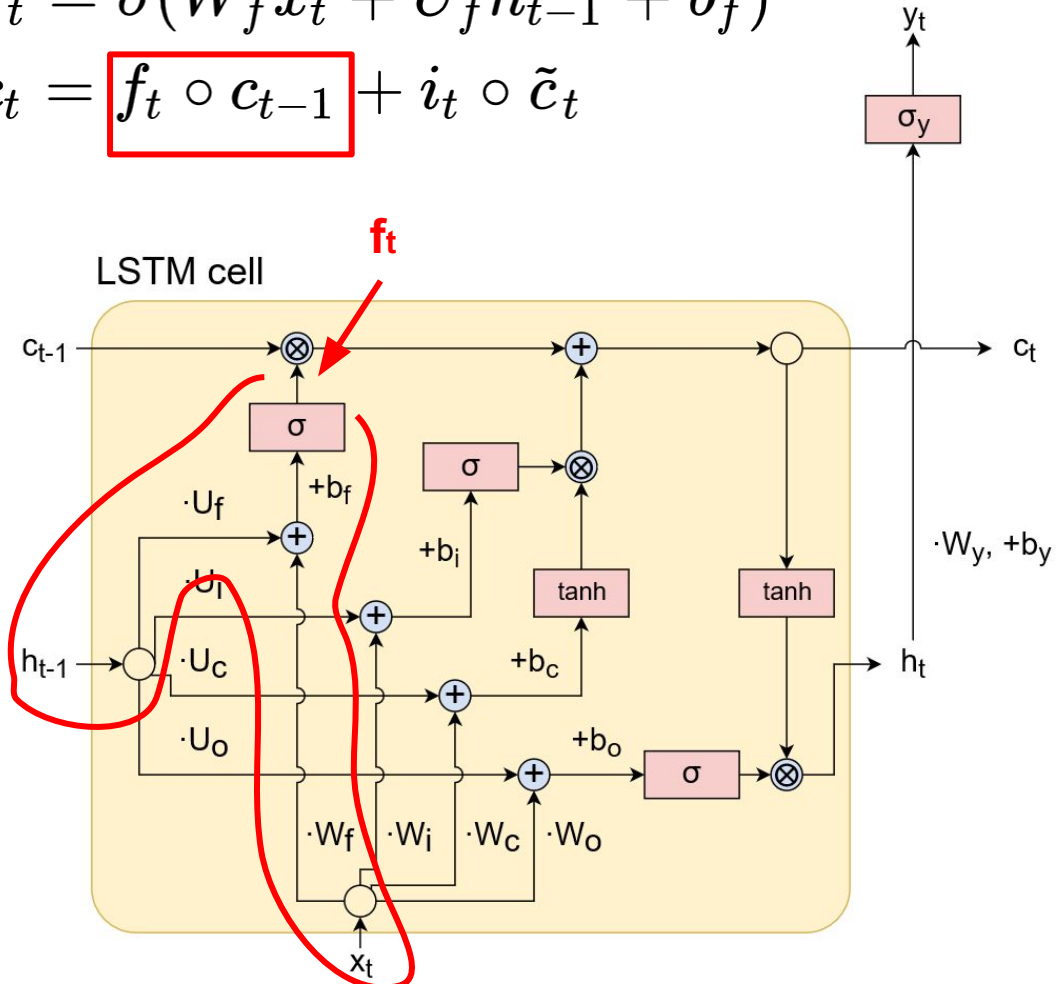
LSTM - Details

Three step update

1. **Forget gate:** Estimating what to “forget” from the previous cell state, c_{t-1}.

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

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



LSTM - Details

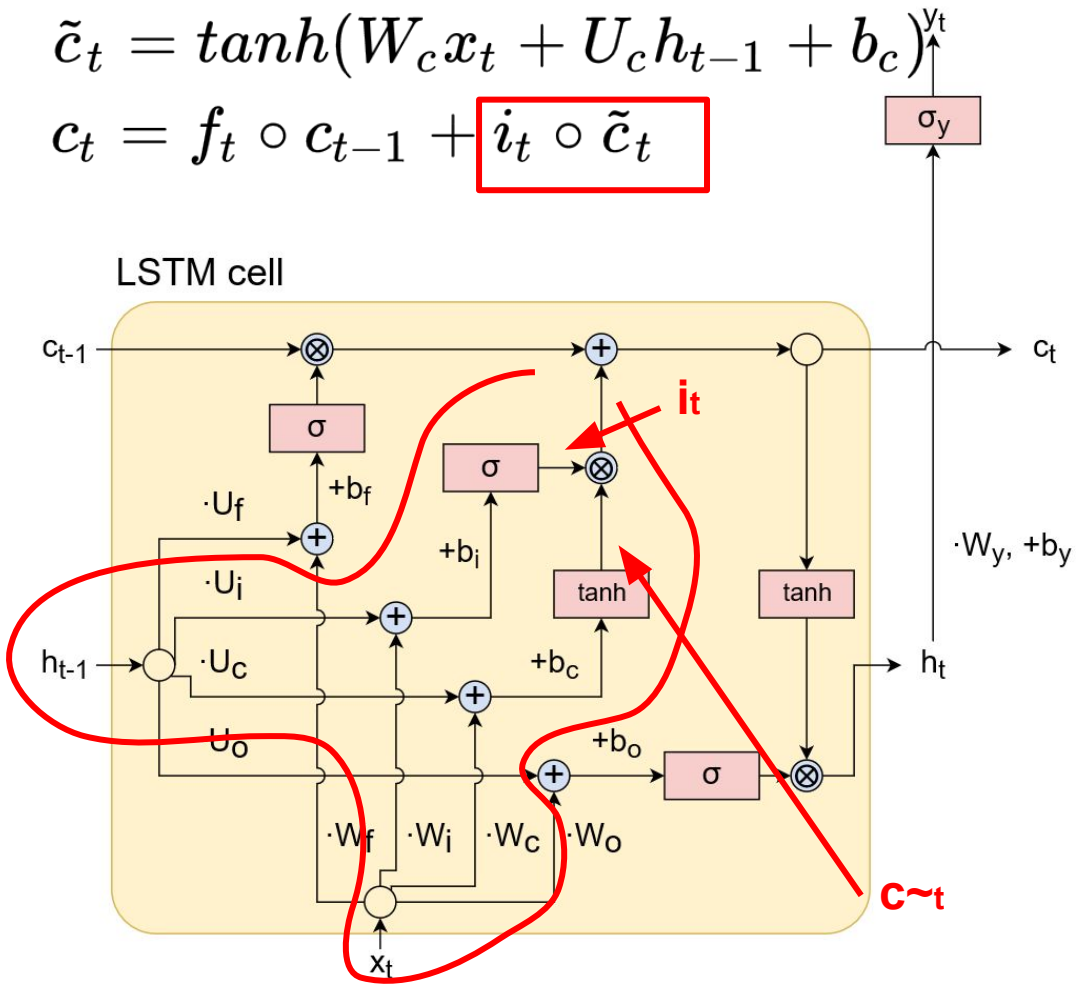
Three step update

1. Forget gate
2. **Overwriting** parts of the cell state.

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

$$\tilde{c}_t = tanh(W_c x_t + U_c h_{t-1} + b_c)$$

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

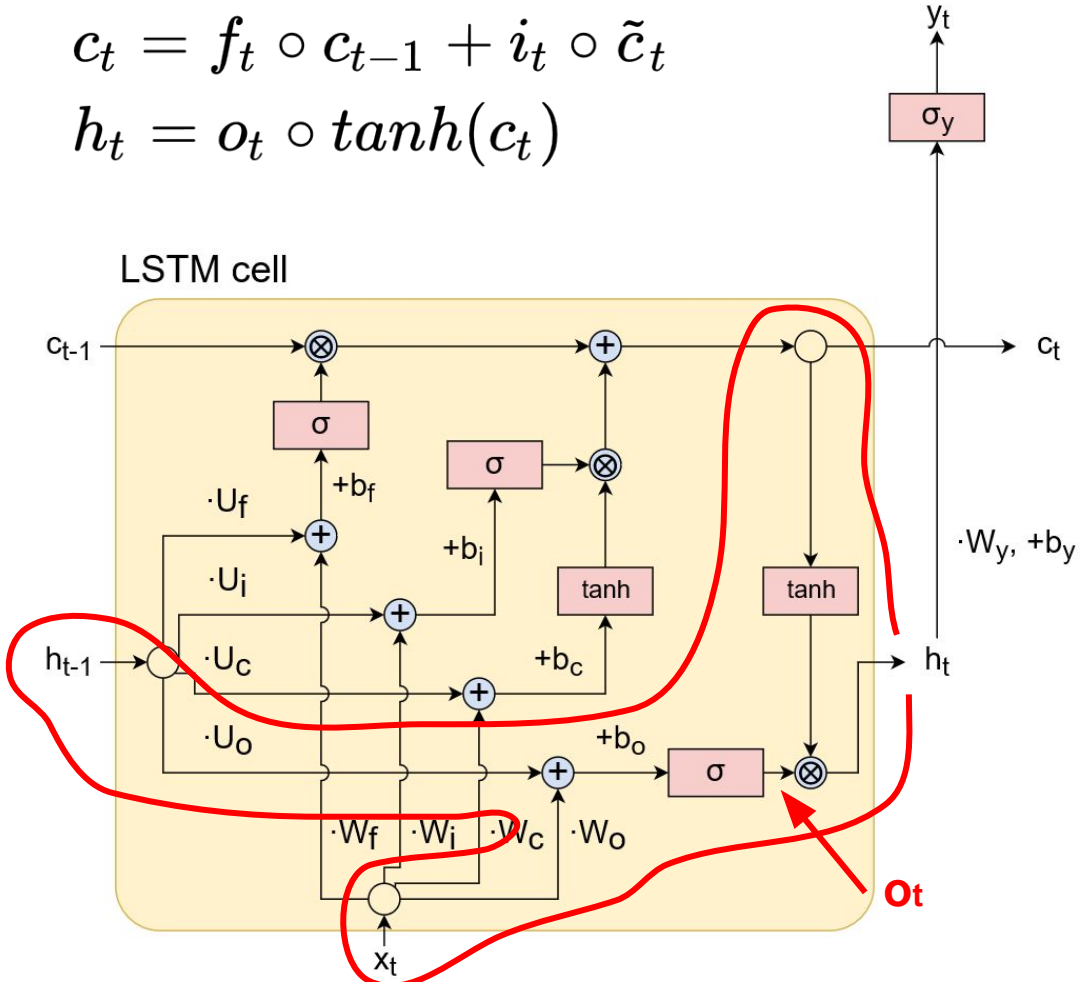


LSTM - Details

Three step update

1. Forget gate
2. Overwriting cell state
3. **Producing the output**
   for the current time step

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$
$$h_t = o_t \circ tanh(c_t)$$



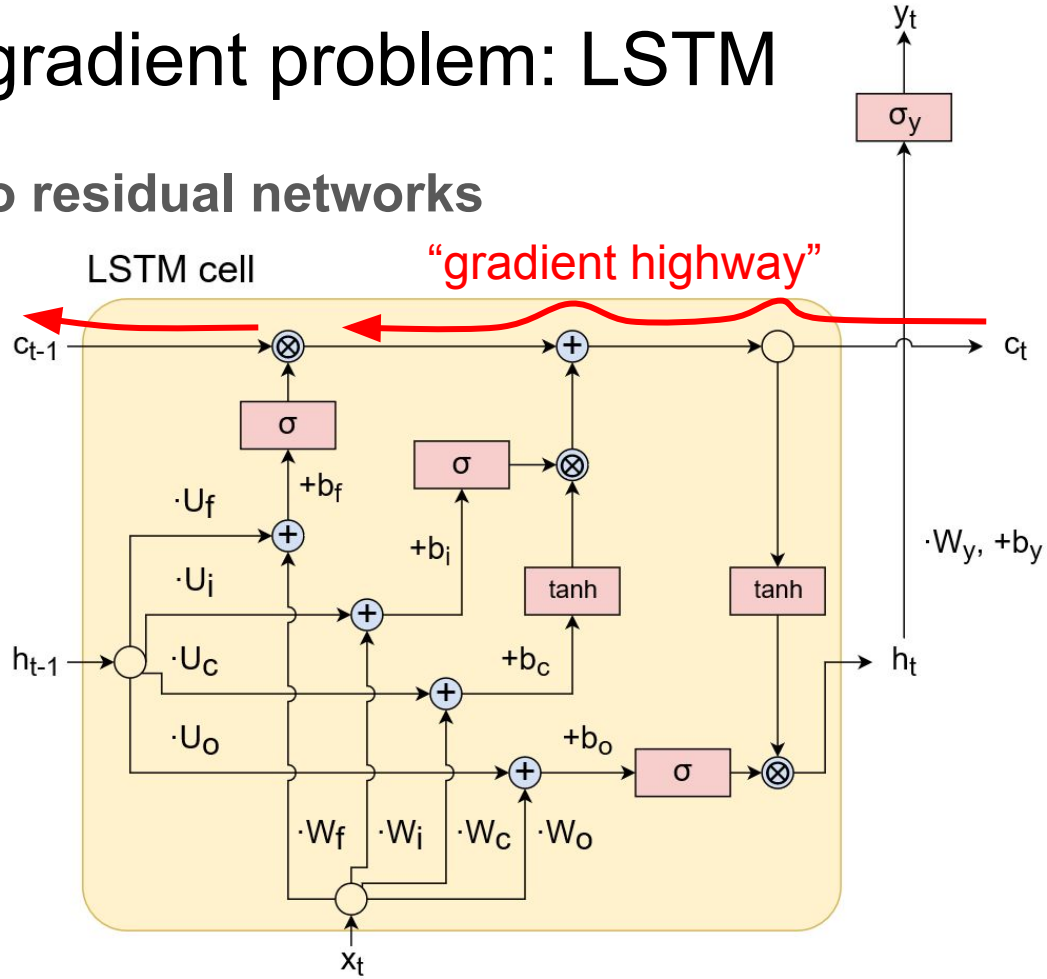
*but almost 20 years older

Mitigating the unstable gradient problem: LSTM

LSTM is a **similar*** architecture **to residual networks**

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

If f_t is all ones, the **gradient is backpropgated uninterrupted** towards the previous time step…



Mitigating the unstable gradient problem: LSTM

Long short-term memory (LSTM, 1997)

Advantages:

- The cell state hidden representation **enables an uninterrupted flow of gradient during backpropagation**, except for the 'forget gate'. The forget gate enables the network to explicitly control what information is retained in the cell state and what is discarded in each time step.

Mitigating the unstable gradient problem: GRU

A more recent alternative to LSTM:

Gated Recurrent Unit (GRU, 2014)

If vector **z** is all zero, the gradient flows back in time from $\mathbf{h}_t$ to $\mathbf{h}_{t-1}$ uninterrupted.

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

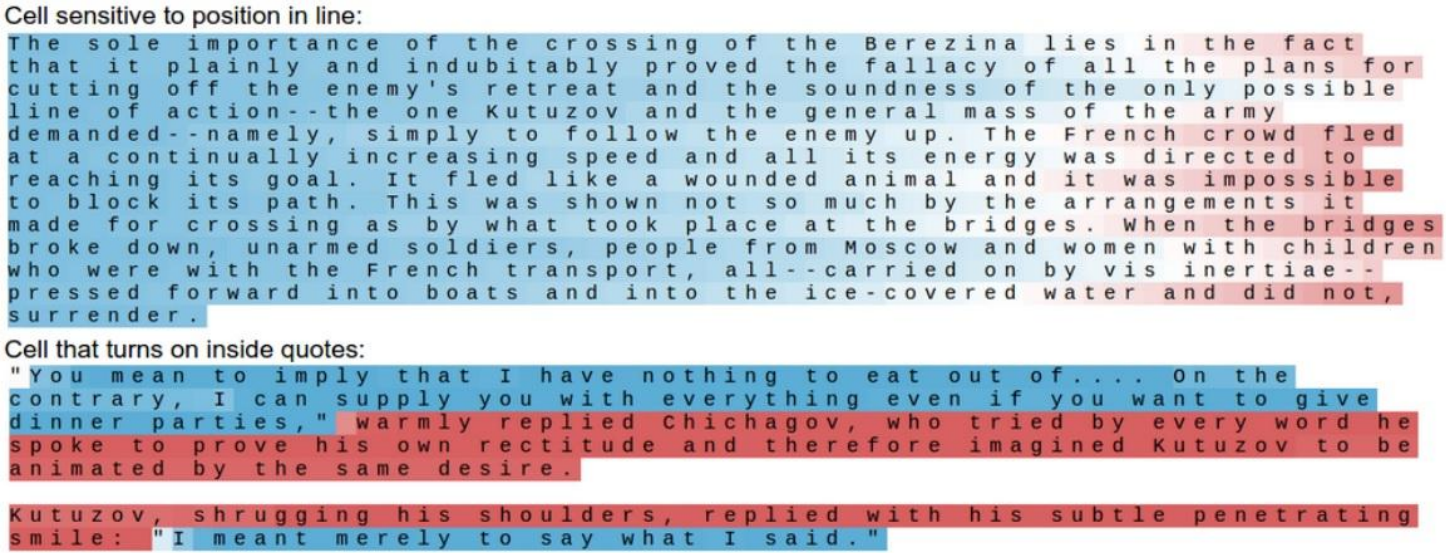
Only a single hidden representation is maintained (**h**), no extra cell state.

Advantage: fewer parameters → faster training

Source: Karpathy et al., 2016

LSTM - Visualizing neuron activations

An analysis of the neurons in the hidden layers of the LSTM trained on the novel “War and Peace”: **Which neurons activate at which locations?**



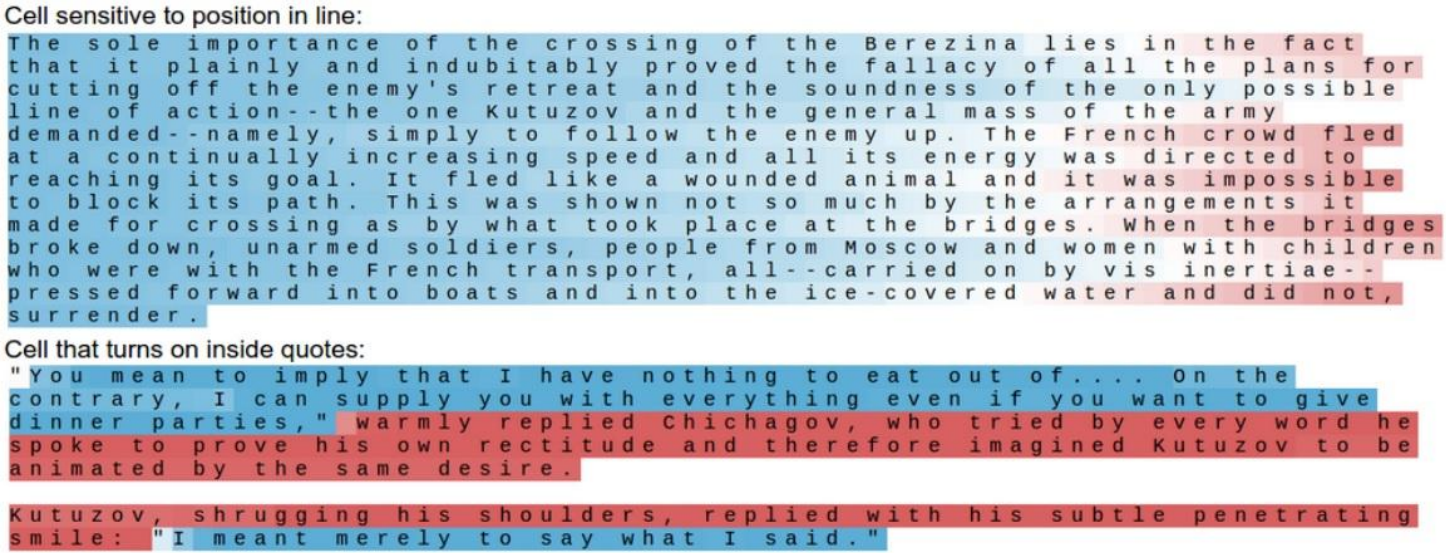
Source: Karpathy et al., 2016

LSTM - Visualizing neuron activations

An analysis of the neurons in the hidden layers of the LSTM trained on the novel “War and Peace”: **Which neurons activate at which locations?**

One neuron was activated in proportion to its **distance from the line break.**

Another kept track of the **opened quotes**.



Source: Karpathy et al., 2016

The same visualizations with a model trained on Linux kernel files.

LSTM - Visualizing neuron activations

```
if (unlikely(!lsm_str))
  return -ENOMEM;
df->lsm_str = lsm_str;
/* our own (refreshed) copy of lsm_rule */
ret = security_audit_rule_init(df->type, df->op, df->lsm_str,
                               (void **)&df->lsm_rule);
/* Keep currently invalid fields around in case they
 * become valid after a policy reload. */
if (ret == -EINVAL) {
  pr_warn("audit rule for LSM \'%s\' is invalid\n",
    df->lsm_str);
  ret = 0;
}
return ret;
}
```

Cell that is sensitive to the depth of an expression:

```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
  int i;
  if (classes[class]) {
    for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
      if (mask[i] & classes[class][i])
        return 0;
  }
  return 1;
}
```

Source: Karpathy et al., 2016

The same visualizations with a model trained on Linux kernel files.

LSTM - Visualizing neuron activations

The first neuron was activated **inside strings and block comments**.

```
if (unlikely(!lsm_str))
  return -ENOMEM;
df->lsm_str = lsm_str;
/* our own (refreshed) copy of lsm_rule */
ret = security_audit_rule_init(df->type, df->op, df->lsm_str,
                               (void **)&df->lsm_rule);
/* Keep currently invalid fields around in case they
 * become valid after a policy reload. */
if (ret == -EINVAL) {
  pr_warn("audit rule for LSM \'%s\' is invalid\n",
    df->lsm_str);
  ret = 0;
}
return ret;
}
```

Another one activated in proportion to **code block depth**.

Cell that is sensitive to the depth of an expression:

```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
  int i;
  if (classes[class]) {
    for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
      if (mask[i] & classes[class][i])
        return 0;
  }
  return 1;
}
```

However, the authors could not interpret the behaviour of most of the neurons…