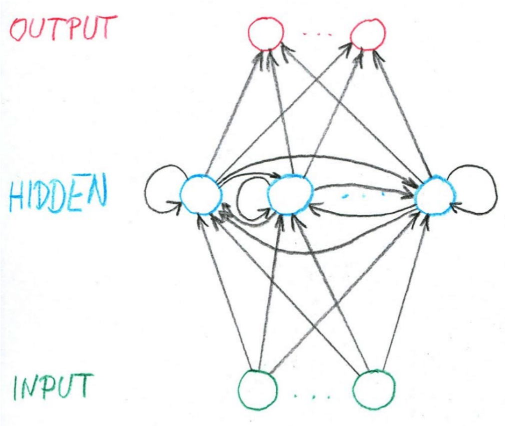
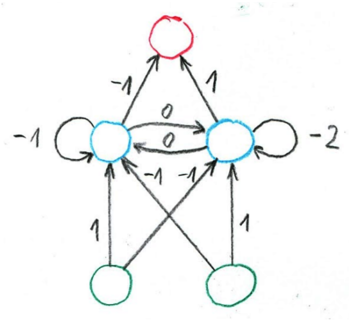
Recurrent Neural Networks - LSTM

RNN



- Input:
  $\vec{x} = (x_1, \ldots, x_M)$
- Hidden:
  $\vec{h} = (h_1, \ldots, h_H)$
- Output:
  $\vec{y} = (y_1, \ldots, y_N)$

RNN example

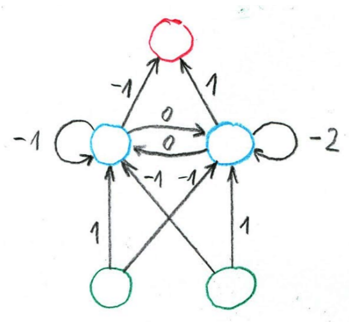


Activation function:

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0 \\ 0 & \xi < 0 \end{cases}$$

y		1	0	1	
h	(0,0)	(1,1)	(1,0)	(0,1)	$\cdots$
x		(0,0)	(1,0)	(1,1)	

RNN example

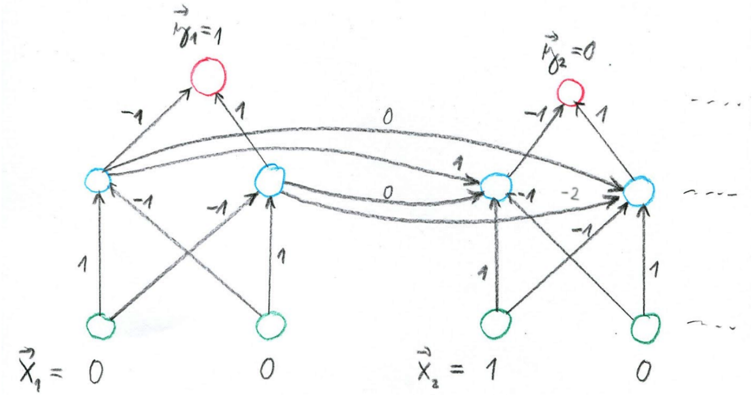


Activation function:

$$\sigma(\xi) = \begin{cases} 1 & \xi \geq 0 \\ 0 & \xi < 0 \end{cases}$$

y		$\vec{y}_1 = 1$	$\vec{y}_2 = 0$	$\vec{y}_3 = 1$	
h	$\vec{h}_0 = (0,0)$	$\vec{h}_1 = (1,1)$	$\vec{h}_2 = (1,0)$	$\vec{h}_3 = (0,1)$	$\cdots$
x		$\vec{x}_1 = (0,0)$	$\vec{x}_2 = (1,0)$	$\vec{x}_3 = (1,1)$	

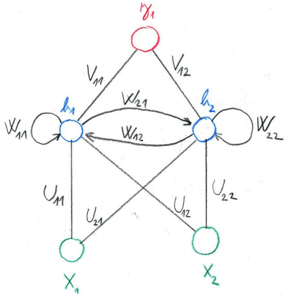
RNN example



y		$\vec{y}_1 = 1$	$\vec{y}_2 = 0$	$\vec{y}_3 = 1$	
h	$\vec{h}_0 = (0, 0)$	$\vec{h}_1 = (1, 1)$	$\vec{h}_2 = (1, 0)$	$\vec{h}_3 = (0, 1)$	$\cdots$
x		$\vec{x}_1 = (0, 0)$	$\vec{x}_2 = (1, 0)$	$\vec{x}_3 = (1, 1)$	

RNN – formally

- M inputs: $\vec{x} = (x_1, \ldots, x_M)$
- H hidden neurons: $\vec{h} = (h_1, \ldots, h_H)$
- N output neurons: $\vec{y} = (y_1, \ldots, y_N)$
- Weights:
  - $U_{kk'}$ from input $x_{k'}$ to hidden h_k
  - $W_{kk'}$ from hidden $h_{k'}$ to hidden h_k
  - $V_{kk'}$ from hidden $h_{k'}$ to output y_k



RNN – formally

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$

$$\vec{x}_t = (x_{t1}, \ldots, x_{tM})$$

RNN – formally

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$

  $$\vec{x}_t = (x_{t1}, \ldots, x_{tM})$$

- Hidden sequence: $\mathbf{h} = \vec{h}_0, \vec{h}_1, \ldots, \vec{h}_T$

  $$\vec{h}_t = (h_{t1}, \ldots, h_{tH})$$

  We have $\vec{h}_0 = (0, \ldots, 0)$ and

  $$\vec{h}_{tk} = \sigma\left(\sum_{k'=1}^{M} U_{kk'} x_{tk'} + \sum_{k'=1}^{H} W_{kk'} h_{(t-1)k'}\right)$$

RNN – formally

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$

  $$\vec{x}_t = (x_{t1}, \ldots, x_{tM})$$

- Hidden sequence: $\mathbf{h} = \vec{h}_0, \vec{h}_1, \ldots, \vec{h}_T$

  $$\vec{h}_t = (h_{t1}, \ldots, h_{tH})$$

  We have $\vec{h}_0 = (0, \ldots, 0)$ and

  $$\vec{h}_{tk} = \sigma\left(\sum_{k'=1}^{M} U_{kk'} x_{tk'} + \sum_{k'=1}^{H} W_{kk'} h_{(t-1)k'}\right)$$

- Output sequence: $\mathbf{y} = \vec{y}_1, \ldots, \vec{y}_T$

  $$\vec{y}_t = (y_{t1}, \ldots, y_{tN})$$

  where $y_{tk} = \sigma\left(\sum_{k'=1}^{H} V_{kk'} h_{tk'}\right)$.

RNN – in matrix form

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$

RNN – in matrix form

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$
- Hidden sequence: $\mathbf{h} = \vec{h}_0, \vec{h}_1, \ldots, \vec{h}_T$ where

$$\vec{h}_0 = (0, \ldots, 0)$$

and

$$\vec{h}_t = \sigma(U\vec{x}_t + W\vec{h}_{t-1})$$

RNN – in matrix form

- Input sequence: $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$
- Hidden sequence: $\mathbf{h} = \vec{h}_0, \vec{h}_1, \ldots, \vec{h}_T$ where

  $$\vec{h}_0 = (0, \ldots, 0)$$

  and

  $$\vec{h}_t = \sigma(U\vec{x}_t + W\vec{h}_{t-1})$$

- Output sequence: $\mathbf{y} = \vec{y}_1, \ldots, \vec{y}_T$ where

  $$y_t = \sigma(Vh_t)$$

RNN – Comments

- $\vec{h}_t$ is the memory of the network, captures what happened in all previous steps (with decaying quality).
- RNN **shares weights** U, V, W along the sequence.
  Note the similarity to convolutional networks where the weights were shared spatially over images, here they are shared temporally over sequences.
- RNN can deal with **sequences of variable length.**
  Compare with MLP which accepts only fixed-dimension vectors on input.

RNN – training

Training set

$$\mathcal{T} = \left\{ (\mathbf{x}_1, \mathbf{d}_1), \ldots, (\mathbf{x}_p, \mathbf{d}_p) \right\}$$

here

- each $\mathbf{x}_\ell = \vec{x}_{\ell 1}, \ldots, \vec{x}_{\ell T_\ell}$ is an input sequence,
- each $\mathbf{d}_\ell = \vec{d}_{\ell 1}, \ldots, \vec{d}_{\ell T_\ell}$ is an expected output sequence.

Here each $\vec{x}_{\ell t} = (x_{\ell t 1}, \ldots, x_{\ell t M})$ is an input vector and each $\vec{d}_{\ell t} = (d_{\ell t 1}, \ldots, d_{\ell t N})$ is an expected output vector.

Error function

In what follows I will consider a training set with a **single element** $(\mathbf{x}, \mathbf{d})$. I.e. drop the index ℓ and have

- $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$ where $\vec{x}_t = (x_{t1}, \ldots, x_{tM})$
- $\mathbf{d} = \vec{d}_1, \ldots, \vec{d}_T$ where $\vec{d}_t = (d_{t1}, \ldots, d_{tN})$

The squared error of $(\mathbf{x}, \mathbf{d})$ is defined by

$$E_{(\mathbf{x},\mathbf{d})} = \sum_{t=1}^{T} \sum_{k=1}^{N} \frac{1}{2}(y_{tk} - d_{tk})^2$$

Recall that we have a sequence of network outputs $\mathbf{y} = \vec{y}_1, \ldots, \vec{y}_T$ and thus y_{tk} is the k-th component of $\vec{y}_t$

Gradient descent (single training example)

Consider a single training example $(\mathbf{x}, \mathbf{d})$.

The algorithm computes a sequence of weight matrices as follows:

Gradient descent (single training example)

Consider a single training example $(\mathbf{x}, \mathbf{d})$.

The algorithm computes a sequence of weight matrices as follows:

- Initialize all weights randomly close to 0.

Gradient descent (single training example)

Consider a single training example $(\mathbf{x}, \mathbf{d})$.

The algorithm computes a sequence of weight matrices as follows:

- Initialize all weights randomly close to 0.
- In the step $\ell + 1$ (here $\ell = 0, 1, 2, \ldots$) compute "new" weights $U^{(\ell+1)}, V^{(\ell+1)}, W^{(\ell+1)}$ from the "old" weights $U^{(\ell)}, V^{(\ell)}, W^{(\ell)}$ as follows:

$$U_{kk'}^{(\ell+1)} = U_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta U_{kk'}}$$

$$V_{kk'}^{(\ell+1)} = V_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta V_{kk'}}$$

$$W_{kk'}^{(\ell+1)} = W_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta W_{kk'}}$$

Gradient descent (single training example)

Consider a single training example $(\mathbf{x}, \mathbf{d})$.

The algorithm computes a sequence of weight matrices as follows:

- Initialize all weights randomly close to 0.
- In the step $\ell + 1$ (here $\ell = 0, 1, 2, \ldots$) compute "new" weights $U^{(\ell+1)}, V^{(\ell+1)}, W^{(\ell+1)}$ from the "old" weights $U^{(\ell)}, V^{(\ell)}, W^{(\ell)}$ as follows:

$$U_{kk'}^{(\ell+1)} = U_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta U_{kk'}}$$

$$V_{kk'}^{(\ell+1)} = V_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta V_{kk'}}$$

$$W_{kk'}^{(\ell+1)} = W_{kk'}^{(\ell)} - \varepsilon(\ell) \cdot \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta W_{kk'}}$$

The above is THE learning algorithm that modifies weights!

Backpropagation

Computes the derivatives of E, no weights are modified!

Backpropagation

Computes the derivatives of E, no weights are modified!

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta U_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} \cdot \sigma' \cdot x_{tk'} \qquad k' = 1, \ldots, M$$

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta V_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta y_{tk}} \cdot \sigma' \cdot h_{tk'} \qquad k' = 1, \ldots, H$$

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta W_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} \cdot \sigma' \cdot h_{(t-1)k'} \qquad k' = 1, \ldots, H$$

Backpropagation

Computes the derivatives of E, no weights are modified!

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta U_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} \cdot \sigma' \cdot x_{tk'} \qquad k' = 1, \ldots, M$$

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta V_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta y_{tk}} \cdot \sigma' \cdot h_{tk'} \qquad k' = 1, \ldots, H$$

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta W_{kk'}} = \sum_{t=1}^{T} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} \cdot \sigma' \cdot h_{(t-1)k'} \qquad k' = 1, \ldots, H$$

Backpropagation:

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta y_{tk}} = y_{tk} - d_{tk} \qquad \text{(assuming squared error)}$$

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} = \sum_{k'=1}^{N} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta y_{tk'}} \cdot \sigma' \cdot V_{k'k} + \sum_{k'=1}^{H} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{(t+1)k'}} \cdot \sigma' \cdot W_{k'k}$$

Long-term dependencies

$$\frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{tk}} = \sum_{k'=1}^{N} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta y_{tk'}} \cdot \sigma' \cdot V_{k'k} + \sum_{k'=1}^{H} \frac{\delta E_{(\mathbf{x},\mathbf{d})}}{\delta h_{(t+1)k'}} \cdot \sigma' \cdot W_{k'k}$$

- Unless $\sum_{k'=1}^{H} \sigma' \cdot W_{k'k} \approx 1$, the gradient either vanishes, or explodes.
- For a large T (long-term dependency), the gradient "deeper" in the past tends to be too small (large).
- A solution: LSTM

  LSTM is currently a bit obsolete. The main idea is to decompose W into several matrices, each responsible for a different task. One is concerned about memory, one is concerned about the output at each step, etc.

https://arxiv.org/pdf/2205.13504.pdf

LSTM

$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t) \qquad \text{output}$$

$$\vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t \qquad \text{memory}$$

$$\tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t) \qquad \text{new memory contents}$$

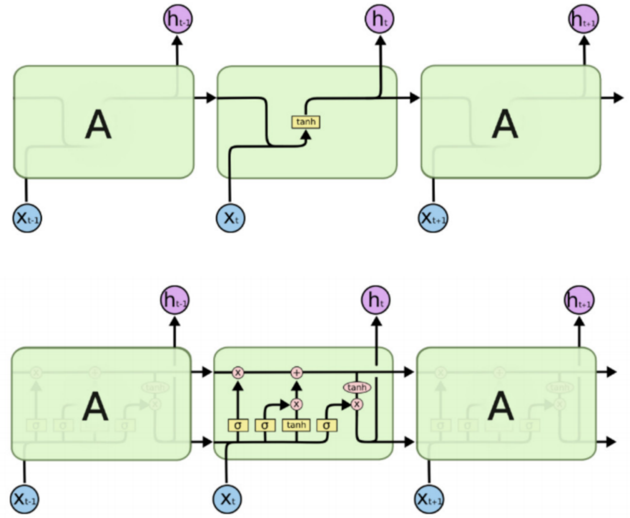
$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t) \qquad \text{output gate}$$

$$\vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t) \qquad \text{forget gate}$$

$$\vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t) \qquad \text{input gate}$$

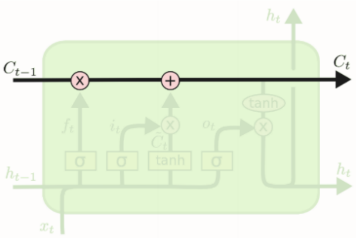
- $\circ$ is the component-wise product of vectors
- $\cdot$ is the matrix-vector product
- σ_h hyperbolic tangents (applied component-wise)
- σ_g logistic sigmoid (aplied component-wise)

RNN vs LSTM



Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM



$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$

$$\Rightarrow \vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$

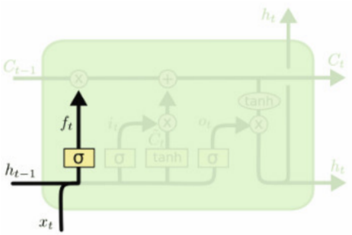
$$\tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$

$$\vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$

$$\vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

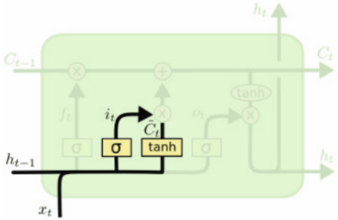
Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/



$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$
$$\vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$
$$\tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$
$$\Rightarrow \vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$
$$\vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/



$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$

$$\vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$

$$\Rightarrow \tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

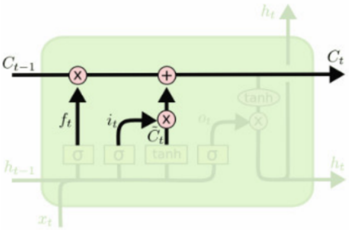
$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$

$$\vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$

$$\Rightarrow \vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM



$$\vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$

$$\Rightarrow \vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$

$$\Rightarrow \tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

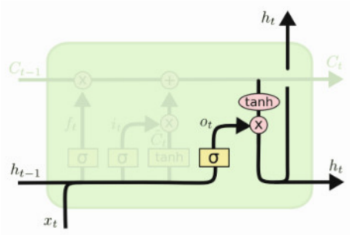
$$\vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$

$$\Rightarrow \vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$

$$\Rightarrow \vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM



$$\Rightarrow \vec{h}_t = \vec{o}_t \circ \sigma_h(\vec{C}_t)$$

$$\vec{C}_t = \vec{f}_t \circ \vec{C}_{t-1} + \vec{i}_t \circ \tilde{C}_t$$

$$\tilde{C}_t = \sigma_h(W_C \cdot \vec{h}_{t-1} + U_C \cdot \vec{x}_t)$$

$$\Rightarrow \vec{o}_t = \sigma_g(W_o \cdot \vec{h}_{t-1} + U_o \cdot \vec{x}_t)$$

$$\vec{f}_t = \sigma_g(W_f \cdot \vec{h}_{t-1} + U_f \cdot \vec{x}_t)$$

$$\vec{i}_t = \sigma_g(W_i \cdot \vec{h}_{t-1} + U_i \cdot \vec{x}_t)$$

Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

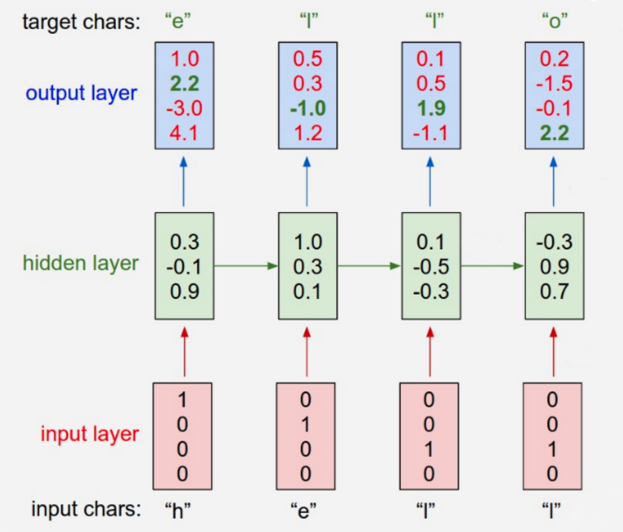
LSTM – summary

- LSTM (almost) solves the vanishing gradient problem w.r.t. the "internal" state of the network.
- Learns to control its own memory (via forget gate).
- Revolution in machine translation and text processing.

... but the development goes on ...

RNN text generator

Generating texts letter by letter.



Shakespeare

- Generating Shakespeare letter by letter.
- Trained on Shakespeare's plays (4.4MB).

VIOLA: Why, Salisbury must find his flesh and thought That which I am not aps, not a man and in fire, To show the reining of the raven and the wars To grace my hand reproach within, and not a fair are hand, That Caesar and my goodly father's world; When I was heaven of presence and our fleets, We spare with hours, but cut thy council I am great, Murdered and by thy master's ready there My power to give thee but so much as hell: Some service in the noble bondman here, Would show him to her wine.

KING LEAR: O, if you were a feeble sight, the courtesy of your law, Your sight and several breath, will wear the gods With his heads, and my hands are wonder'd at the deeds, So drop upon your lordship's head, and your opinion Shall be against your honour.

Wikipedia

- Hutter Prize 100MB dataset from Wikipedia (96MB)

Naturalism and decision for the majority of Arab countries' capitalide was grounded by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal in the [[Protestant Immineners]], which could be said to be directly in Cantonese Communication, which followed a ceremony and set inspired prison, training. The emperor travelled back to [[Antioch, Perth, October 25|21]] to note, the Kingdom of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoard]], known in western [[Scotland]], near Italy to the conquest of India with the conflict. Copyright was the succession of independence in the slop of Syrian influence that was a famous German movement based on a more popular servicious, non-doctrinal and sexual power post. Many governments recognize the military housing of the [[Civil Liberalization and Infantry Resolution 265 National Party in Hungary]], that is sympathetic to be to the [[Punjab Resolution]] (PJS)[`http://www.humah.yahoo.com/guardian.cfm/7754800786d17551963s89.htm`
Official economics Adjoint for the Nazism, Montgomery was swear to advance to the resources for those Socialism's rule, was starting to signing a

Xml halucination:


```
<page>
  <title>Antichrist</title>
  <id>865</id>
  <revision>
    <id>15900676</id>
    <timestamp>2002-08-03T18:14:12Z</timestamp>
    <contributor>
      <username>Paris</username>
      <id>23</id>
    </contributor>
    <minor />
    <comment>Automated conversion</comment>
    <text xml:space="preserve">
    #REDIRECT [[Christianity]]</text>
  </revision>
</page>
```

LaTeX

- Algebraic geometry textbook.
- LaTeX source (16MB).
- Almost compilable.

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

Linux source code

- Trained on all source files of Linux kernel concatenated into a single file (474MB of C code).

```c
/*
 * 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;
}
```

```
/*
 *  Copyright (c) 2006-2010, Intel Mobile Communications.  All rights reserved.
 *
 *   This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 as published by
 * the Free Software Foundation.
 *
 *        This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *
 *  GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *    along with this program; if not, write to the Free Software Foundation,
 *  Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/kexec.h>
#include <linux/errno.h>
#include <linux/io.h>
#include <linux/platform_device.h>
#include <linux/multi.h>
#include <linux/ckevent.h>

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/setew.h>
#include <asm/pgproto.h>
```

Evolution of Shakespeare

100 iter.:

```
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
```

300 iter.:

```
"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."
```

500 iter.:

```
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
```

700 iter.:

```
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.
```

1200 iter.:

```
"Kite vouch!" he repeated by her
door. "But I would be done and quarts, feeling, then, son is people...."
```

2000 iter.:

```
"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.
```

Attention

Consider the following task: Given a sequence of vectors

$$\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$$

generate a new sequence

$$\mathbf{y} = \vec{y}_1, \ldots, \vec{y}_{T'}$$

of possibly different length (i.e., possibly $T \neq T'$).

E.g., a machine translation task, **x** is an embedding of an English sentence, **y** is a sequence of probability distributions on a German vocabulary.

Attention

Consider two recurrent networks:

- `Enc` the encoder
  - Hidden state $\vec{h}_0$ initialized by standard methods for recurrent networks
  - Reads $\vec{x}_1, \ldots, \vec{x}_T$, does not output anything but produces a sequence of hidden states $\vec{h}_1, \ldots, \vec{h}_T$

Attention

Consider two recurrent networks:

- `Enc` the encoder
  - Hidden state $\vec{h}_0$ initialized by standard methods for recurrent networks
  - Reads $\vec{x}_1, \ldots, \vec{x}_T$, does not output anything but produces a sequence of hidden states $\vec{h}_1, \ldots, \vec{h}_T$
- `Dec` the decoder
  - The initial hidden state is $\vec{h}_T$
  - Does not read anything but outputs the sequence $\vec{y}_1, \ldots, \vec{y}_{T'}$

    This is a simplification. Typically, `Dec` reads $\vec{y}_0, \vec{y}_1, \ldots, \vec{y}_{T'-1}$ where $\vec{y}_0$ is a special vector embedding a separator.

Attention

Consider two recurrent networks:

- `Enc` the encoder
  - Hidden state $\vec{h}_0$ initialized by standard methods for recurrent networks
  - Reads $\vec{x}_1, \ldots, \vec{x}_T$, does not output anything but produces a sequence of hidden states $\vec{h}_1, \ldots, \vec{h}_T$
- `Dec` the decoder
  - The initial hidden state is $\vec{h}_T$
  - Does not read anything but outputs the sequence $\vec{y}_1, \ldots, \vec{y}_{T'}$

    This is a simplification. Typically, `Dec` reads $\vec{y}_0, \vec{y}_1, \ldots, \vec{y}_{T'-1}$ where $\vec{y}_0$ is a special vector embedding a separator.

Trained on pairs of sentences, able to learn a fine translation between major languages (if the recurrent networks are LSTM).

Is not perfect because all info about $\mathbf{x} = \vec{x}_1, \ldots, \vec{x}_T$ is squeezed into the single state vector $\vec{h}_T$.

In particular, the network tends to forget the context of each word.

Attention in Recurrent Networks

What if we provide the decoder with an information about the *relevant context* of the generated word?

Attention in Recurrent Networks

What if we provide the decoder with an information about the *relevant context* of the generated word?

We use the same encoder `Enc` producing the sequence of hidden states: $\vec{h}_1, \ldots, \vec{h}_T$

Attention in Recurrent Networks

What if we provide the decoder with an information about the *relevant context* of the generated word?

We use the same encoder `Enc` producing the sequence of hidden states: $\vec{h}_1, \ldots, \vec{h}_T$

The decoder `Dec` is still a recurrent network but

- the hidden state $\vec{h}'_0$ initialized by $\vec{h}_T$ and a sequence of hidden states $\vec{h}'_0, \ldots, \vec{h}'_{T'}$ is computed,

Attention in Recurrent Networks

What if we provide the decoder with an information about the *relevant context* of the generated word?

We use the same encoder `Enc` producing the sequence of hidden states: $\vec{h}_1, \ldots, \vec{h}_T$

The decoder `Dec` is still a recurrent network but

- the hidden state $\vec{h}'_0$ initialized by $\vec{h}_T$ and a sequence of hidden states $\vec{h}'_0, \ldots, \vec{h}'_{T'}$ is computed,
- reads a sequence of context vectors $\vec{c}_1, \ldots, \vec{c}_{T'}$ where

$$\vec{c}_i = \sum_{j=1}^{T} \alpha_{ij} \vec{h}_j \qquad \text{where} \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})}$$

  where $e_{ij} = \mathtt{MLP}(\vec{h}'_{i-1}, \vec{h}_j)$
- outputs the sequence $\vec{y}_1, \ldots, \vec{y}_{T'}$

Do We Still Need the Recurrence?

- The attention mechanism extracts the information from the sequence quite well.

Do We Still Need the Recurrence?

- The attention mechanism extracts the information from the sequence quite well.
- Is there a reason for reading the input sequence sequentially?

Do We Still Need the Recurrence?

- The attention mechanism extracts the information from the sequence quite well.
- Is there a reason for reading the input sequence sequentially?
- Could we remove the recurrent network itself and preserve only the attention?

Self-Attention Layer (is all you need)

Fix an input sequence: $\vec{x}_1, \ldots, \vec{x}_T$

Consider three learnable matrices: W_q, W_k, W_v

Generate sequences of *queries, keys, and values*:

- $\vec{q}_1, \ldots, \vec{q}_T$ where $\vec{q}_k = W_q \vec{x}_k$ for all $k = 1, \ldots, T$
- $\vec{k}_1, \ldots, \vec{k}_T$ where $\vec{k}_k = W_k \vec{x}_k$ for all $k = 1, \ldots, T$
- $\vec{v}_1, \ldots, \vec{v}_T$ where $\vec{v}_k = W_v \vec{x}_k$ for all $k = 1, \ldots, T$

Self-Attention Layer (is all you need)

Fix an input sequence: $\vec{x}_1, \ldots, \vec{x}_T$

Consider three learnable matrices: W_q, W_k, W_v

Generate sequences of *queries, keys, and values*:

- $\vec{q}_1, \ldots, \vec{q}_T$ where $\vec{q}_k = W_q \vec{x}_k$ for all $k = 1, \ldots, T$
- $\vec{k}_1, \ldots, \vec{k}_T$ where $\vec{k}_k = W_k \vec{x}_k$ for all $k = 1, \ldots, T$
- $\vec{v}_1, \ldots, \vec{v}_T$ where $\vec{v}_k = W_v \vec{x}_k$ for all $k = 1, \ldots, T$

Define a vector score for all $i, j \in \{1, \ldots, T\}$ by

$$e_{ij} = \vec{q}_i \cdot \vec{k}_j$$

Intuitively, s_{ij} measures how much the input at the position i is related to the input at the position j, in other words, how much the query fits the key.

Define

$$\alpha_{ij} = \frac{\exp(e_{ij} \,/\, \sqrt{d_{attn}})}{\sum_{k=1}^{T} \exp(e_{ik} \,/\, \sqrt{d_{attn}})}$$

d_{attn} is the dimension of $\vec{v}_i$

I.e., we apply the good old softmax to $(e_{i1}, \ldots, e_{iT}) \,/\, \sqrt{d_{attn}}$

Self-Attention Layer (is all you need)

Define a vector score for all $i, j \in \{1, \ldots, T\}$ by

$$e_{ij} = \vec{q}_i \cdot \vec{k}_j$$

Intuitively, s_{ij} measures how much the input at the position i is related to the input at the position j, in other words, how much the query fits the key.

Define

$$\alpha_{ij} = \frac{\exp(e_{ij} \,/\, \sqrt{d_{attn}})}{\sum_{k=1}^{T} \exp(e_{ik} \,/\, \sqrt{d_{attn}})}$$

d_{attn} is the dimension of $\vec{v}_i$

I.e., we apply the good old softmax to $(e_{i1}, \ldots, e_{iT}) \,/\, \sqrt{d_{attn}}$

Define a sequence of outputs $\vec{y}_1, \ldots, \vec{y}_T$ by

$$\vec{y}_i = \sum_{j=1}^{T} \alpha_{ij} \cdot \vec{v}_j$$

Language Model

A sequence of *tokens* $a_1, \ldots, a_T \in \Sigma^*$

E.g. words from a vocabulary Σ.

The goal: Maximize

$$\prod_{k=1}^{T} P(a_k \mid a_1, \ldots, a_{k-1}; W) \qquad (= P(a_1, \ldots, a_T; W))$$

where

- P is the conditional probability measure over Σ modeled using a neural network with weights W.

Language Model

A sequence of *tokens* $a_1, \ldots, a_T \in \Sigma^*$

E.g. words from a vocabulary Σ.

The goal: Maximize

$$\prod_{k=1}^{T} P(a_k \mid a_1, \ldots, a_{k-1}; W) \qquad (= P(a_1, \ldots, a_T; W))$$

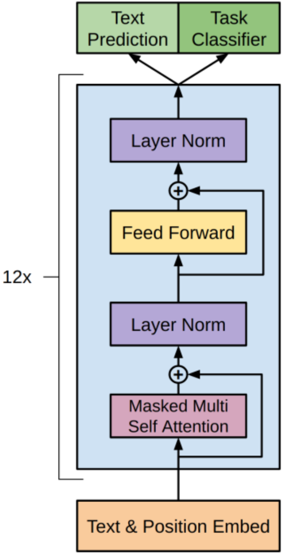
where

- P is the conditional probability measure over Σ modeled using a neural network with weights W.

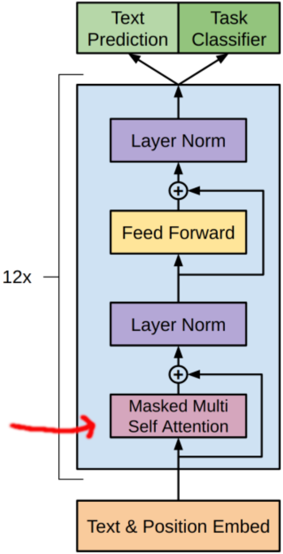
Can be used to generate text:

Given $a_1, \ldots, a_k$, sample a_{k+1} from $P(a_{k+1} \mid a_1, \ldots, a_k; W)$

GPT


Text Prediction
Task Classifier
12x
Layer Norm
Feed Forward
Layer Norm
Masked Multi Self Attention
Text & Position Embed

GPT


Text
Prediction
Task
Classifier
Layer Norm
Feed Forward
12x
Layer Norm
Masked Multi
Self Attention
Text & Position Embed

Masked Self-Attention Layer (is all you need)

Assume an attention mechanism which given an input sequence $\vec{x}_1, \ldots, \vec{x}_T$ generates $\vec{y}_1, \ldots, \vec{y}_T$.

The Problem: How to generate $\vec{y}_k$ only based on $\vec{x}_1, \ldots, \vec{x}_{k-1}$?

Masked Self-Attention Layer (is all you need)

Assume an attention mechanism which given an input sequence $\vec{x}_1, \ldots, \vec{x}_T$ generates $\vec{y}_1, \ldots, \vec{y}_T$.

The Problem: How to generate $\vec{y}_k$ only based on $\vec{x}_1, \ldots, \vec{x}_{k-1}$?

Define a vector score for all $i, j \in \{1, \ldots, T\}$ by

$$e_{ij} = \begin{cases} \vec{q}_i \cdot \vec{k}_j & \text{if } j < i \\ -\infty & \text{otherwise.} \end{cases}$$

This means that

$$\alpha_{ij} = \begin{cases} \frac{\exp(e_{ij} \,/\, \sqrt{d_{attn}})}{\sum_{k=1}^{T} \exp(e_{ik} \,/\, \sqrt{d_{attn}})} & \text{if } j < i \\ 0 & \text{otherwise.} \end{cases}$$

Masked Self-Attention Layer (is all you need)

Assume an attention mechanism which given an input sequence $\vec{x}_1, \ldots, \vec{x}_T$ generates $\vec{y}_1, \ldots, \vec{y}_T$.

The Problem: How to generate $\vec{y}_k$ only based on $\vec{x}_1, \ldots, \vec{x}_{k-1}$?

Define a vector score for all $i, j \in \{1, \ldots, T\}$ by

$$e_{ij} = \begin{cases} \vec{q}_i \cdot \vec{k}_j & \text{if } j < i \\ -\infty & \text{otherwise.} \end{cases}$$

This means that

$$\alpha_{ij} = \begin{cases} \frac{\exp(e_{ij} \,/\, \sqrt{d_{attn}})}{\sum_{k=1}^{T} \exp(e_{ik} \,/\, \sqrt{d_{attn}})} & \text{if } j < i \\ 0 & \text{otherwise.} \end{cases}$$

Define a sequence of outputs $\vec{y}_1, \ldots, \vec{y}_T$ by

$$\vec{y}_i = \sum_{j=1}^{T} \alpha_{ij} \cdot \vec{v}_j$$

Multi-head Self-Attention Layer (is all you need)

Assume the number of *heads* is H.

For $h = 1, \ldots, H$ the h-th head is an attention mechanism which given the input $\vec{x}_1, \ldots, \vec{x}_T$ produces

$$\vec{y}_1^h, \ldots, \vec{y}_T^h$$

Note that the output may be different which means that, in particular, the matrices W_q, W_k, W_v may be different for each head.

Assume that all vectors $\vec{y}_k^h$ are of the same dimension d_{mid} and consider a learnable matrix W_{out} of dimensions $d_{out} \times (H \cdot d_{mid})$.

Multi-head Self-Attention Layer (is all you need)

Assume the number of *heads* is H.

For $h = 1, \ldots, H$ the h-th head is an attention mechanism which given the input $\vec{x}_1, \ldots, \vec{x}_T$ produces

$$\vec{y}_1^h, \ldots, \vec{y}_T^h$$

Note that the output may be different which means that, in particular, the matrices W_q, W_k, W_v may be different for each head.

Assume that all vectors $\vec{y}_k^h$ are of the same dimension d_{mid} and consider a learnable matrix W_{out} of dimensions $d_{out} \times (H \cdot d_{mid})$.

The multi-head attention produces the following output:

$$\vec{y}_1, \ldots, \vec{y}_T$$

where

$$\vec{y}_k = W_{out} \cdot \left(\vec{y}_k^1 \odot \vec{y}_k^2 \odot \cdots \vec{y}_k^H \right)$$

Here $\odot$ is a concatenation of vectors.

Multi-head Self-Attention Summary

Input: A sequence $\vec{x}_1, \ldots, \vec{x}_T$
Output: A sequence $\vec{y}_1, \ldots, \vec{y}_T$

I.e., a sequence of the same length. The dimensions of $\vec{y}_k$ and $\vec{x}_k$ do not have to be equal.

Multi-head Self-Attention Summary

Input: A sequence $\vec{x}_1, \ldots, \vec{x}_T$
Output: A sequence $\vec{y}_1, \ldots, \vec{y}_T$
I.e., a sequence of the same length. The dimensions of $\vec{y}_k$ and $\vec{x}_k$ do not have to be equal.

Attention:
Learnable parameters: Matrices W_q, W_k, W_v.
These matrices are used to compute queries, keys, and values from $\vec{x}_1, \ldots, \vec{x}_T$. Output $\vec{y}_1, \ldots, \vec{y}_T$ is computed using values "scaled" by the query-key attention.

Multi-head Self-Attention Summary

Input: A sequence $\vec{x}_1, \ldots, \vec{x}_T$
Output: A sequence $\vec{y}_1, \ldots, \vec{y}_T$

I.e., a sequence of the same length. The dimensions of $\vec{y}_k$ and $\vec{x}_k$ do not have to be equal.

Attention:

Learnable parameters: Matrices W_q, W_k, W_v.

These matrices are used to compute queries, keys, and values from $\vec{x}_1, \ldots, \vec{x}_T$. Output $\vec{y}_1, \ldots, \vec{y}_T$ is computed using values "scaled" by the query-key attention.

Multi-head attention:

Learnable parameters:

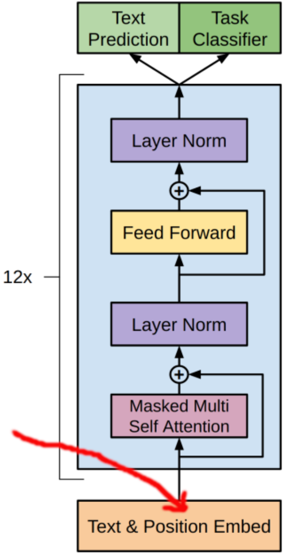
- Matrices W_q^h, W_k^h, W_v^h where $h = 1, \ldots, H$ and H is the number of heads.

  Each attention head operates independently on the input $\vec{x}_1, \ldots, \vec{x}_T$.

- Matrix W_{out}.

  Linearly transforms the concatenated results of the attention heads.

GPT - transformer



Positional encoding

The Goal: To encode a position (index) $k \in \{1, \ldots, T\}$ into a vector $\vec{P}_k$ of real numbers.

Positional encoding

The Goal: To encode a position (index) $k \in \{1, \ldots, T\}$ into a vector $\vec{P}_k$ of real numbers.

Assume that $\vec{P}_k$ should have a dimension d.
Given a position $k \in \{1, \ldots, T\}$ and $i \in \{0, \ldots, d/2\}$ define

$$P_{k,2i} = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P_{k,(2i+1)} = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Here $n = 10000$.

A user defined constant, the original paper suggests $n = 10000$.

Positional encoding

The Goal: To encode a position (index) $k \in \{1, \ldots, T\}$ into a vector $\vec{P}_k$ of real numbers.

Assume that $\vec{P}_k$ should have a dimension d.
Given a position $k \in \{1, \ldots, T\}$ and $i \in \{0, \ldots, d/2\}$ define

$$P_{k,2i} = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P_{k,(2i+1)} = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Here $n = 10000$.

A user defined constant, the original paper suggests $n = 10000$.

Given an input sequence $\vec{x}_1, \ldots, \vec{x}_T$ we add the position embedding to each $\vec{x}_k$ obtaining a new input sequence $\vec{x}'_1, \ldots, \vec{x}'_T$ where

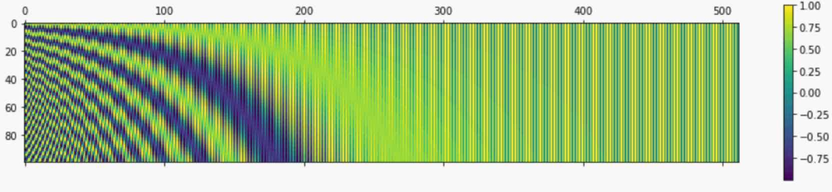
$$\vec{x}'_k = \vec{x}_k + \vec{P}_k$$

Positional encoding/embedding

Positional Encoding Matrix with $d=4$, $n=100$

Sequence	Index of token, k	$i=0$	$i=0$	$i=1$	$i=1$
I	0	$P_{00}=\sin(0) = 0$	$P_{01}=\cos(0) = 1$	$P_{02}=\sin(0) = 0$	$P_{03}=\cos(0) = 1$
am	1	$P_{10}=\sin(1/1) = 0.84$	$P_{11}=\cos(1/1) = 0.54$	$P_{12}=\sin(1/10) = 0.10$	$P_{13}=\cos(1/10) = 1.0$
a	2	$P_{20}=\sin(2/1) = 0.91$	$P_{21}=\cos(2/1) = -0.42$	$P_{22}=\sin(2/10) = 0.20$	$P_{23}=\cos(2/10) = 0.98$
Robot	3	$P_{30}=\sin(3/1) = 0.14$	$P_{31}=\cos(3/1) = -0.99$	$P_{32}=\sin(3/10) = 0.30$	$P_{33}=\cos(3/10) = 0.96$

Positional encoding/embedding



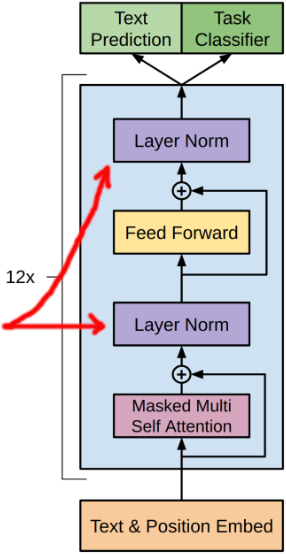
The positional encoding matrix for n=10,000, d=512, sequence length=100

- Vertically: Sinusoidal functions
- Horizontally: Decreasing frequency

For any offset $o \in \{1, \ldots, T\}$ there is a linear transformation M such that for any $k \in \{1, \ldots, T - o\}$ we have $M\vec{P}_k = \vec{P}_{k+o}$.

Intuitively, just rotate each component of the $\vec{P}_k$ appropriately.

GPT-2 - transformer



Layer normalization

Given a vector $\vec{x} \in \mathbb{R}^d$, the *layer normalization* computes:

$$\vec{x}' = \gamma \cdot \frac{(\vec{x} - \mu)}{\sigma} + \beta$$

Here

- $\mu = \frac{1}{d} \sum_{i=1}^{d} x_i$ and $\sigma^2 = \frac{1}{d} \sum_{i=1}^{d} (x_i - \mu)^2$
- $\gamma, \beta \in \mathbb{R}^d$ are vectors of trainable parameters

Layer normalization

Given a vector $\vec{x} \in \mathbb{R}^d$, the *layer normalization* computes:

$$\vec{x}' = \gamma \cdot \frac{(\vec{x} - \mu)}{\sigma} + \beta$$

Here

- $\mu = \frac{1}{d} \sum_{i=1}^{d} x_i$ and $\sigma^2 = \frac{1}{d} \sum_{i=1}^{d} (x_i - \mu)^2$
- $\gamma, \beta \in \mathbb{R}^d$ are vectors of trainable parameters

In Transformer:
The input to the layer normalization is a sequence of vectors: $\vec{x}_1, \ldots, \vec{x}_T$. The layer normalization is applied to each $\vec{x}_k$, producing a sequence of "normalized" vectors.

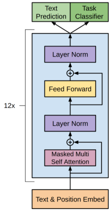
GPT - learning

A sequence of tokens $a_1, \ldots, a_T \in \Sigma$ and their one-hot encodings $\vec{u}_1, \ldots, \vec{u}_T \in \{0,1\}^{|\Sigma|}$

We assume that a_1 is a special token marking the start of the sequence.

Embed to vectors and add the position encoding (W_e is an embedding matrix):

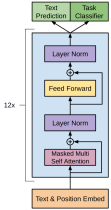
$$\vec{x}_k = W_e \cdot \vec{u}_k + P_k \in Rset^d$$



GPT - learning

A sequence of tokens $a_1, \ldots, a_T \in \Sigma$ and their one-hot encodings $\vec{u}_1, \ldots, \vec{u}_T \in \{0,1\}^{|\Sigma|}$

We assume that a_1 is a special token marking the start of the sequence.



Embed to vectors and add the position encoding (W_e is an embedding matrix):

$$\vec{x}_k = W_e \cdot \vec{u}_k + P_k \in Rset^d$$

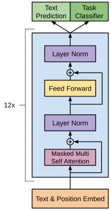
Apply the network (with the transformer block repeated 12x) to $\vec{x}_1, \ldots, \vec{x}_T$ and obtain $\vec{y}_1, \ldots, \vec{y}_T$

(Here assume that each $\vec{y}_k \in [0,1]^\Sigma$ is a probability distribution on Σ)

GPT - learning

A sequence of tokens $a_1, \ldots, a_T \in \Sigma$ and their one-hot encodings $\vec{u}_1, \ldots, \vec{u}_T \in \{0,1\}^{|\Sigma|}$

We assume that a_1 is a special token marking the start of the sequence.



Embed to vectors and add the position encoding (W_e is an embedding matrix):

$$\vec{x}_k = W_e \cdot \vec{u}_k + P_k \in Rset^d$$

Apply the network (with the transformer block repeated 12x) to $\vec{x}_1, \ldots, \vec{x}_T$ and obtain $\vec{y}_1, \ldots, \vec{y}_T$

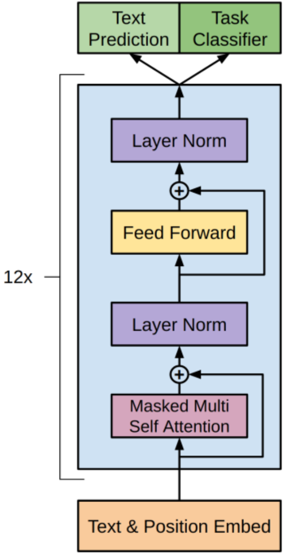
(Here assume that each $\vec{y}_k \in [0,1]^{\Sigma}$ is a probability distribution on Σ)

Compute the error:

$$-\sum_{\ell=1}^{T-1} \log\left(\vec{y}_\ell[a_{\ell+1}]\right)$$

Here $\vec{y}_\ell[a_{k+1}]$ is the probability of a_{k+1} in the distribution $\vec{y}_k$.

GPT - inference



https://transformer.huggingface.co/doc/distil-gpt2

A sequence of tokens $a_1, \dots, a_\ell \in \Sigma$ and their one-hot encodings $\vec{u}_1, \dots, \vec{u}_\ell \in \{0,1\}^{|\Sigma|}$

Embed to vectors and add the position encoding:

$$\vec{x}_k = W_e \cdot \vec{u}_k + P_k \in Rset^d$$

Apply the network to $\vec{x}_1, \dots, \vec{x}_\ell$ and obtain $\vec{y}_1, \dots, \vec{y}_\ell$

(Assume that each $\vec{y}_k \in [0,1]^\Sigma$ is a probability distribution on Σ)

Sample the next token from

$$a_{\ell+1} \sim \vec{y}_\ell$$

Feed-forward networks summary

Architectures:

- Multi-layer perceptron (MLP):
  - dense connections between layers
- Convolutional networks (CNN):
  - local receptors, feature maps
  - pooling
- Recurrent networks (RNN):
  - self-loops but still feed-forward through time
- Transformer
  - Attention, query-key-value

Training:

- gradient descent algorithm + heuristics

Autoencoders

An autoencoder consists of two parts:

- $\phi : \mathbb{R}^n \to \mathbb{R}^m$ the encoder
- $\psi : \mathbb{R}^m \to \mathbb{R}^n$ the decoder

The goal is to find ϕ, ψ so that $\psi \circ \phi$ is (almost) identity.

The value $\vec{h} = \phi(\vec{x})$ is called the *latent representation* of $\vec{x}$.

Autoencoders – training

Assume

$$\mathcal{T} = \{\vec{x}_1, \ldots, \vec{x}_p\}$$

where $\vec{x}_i \in \mathbb{R}^n$ for all $i \in \{1, \ldots, n\}$.

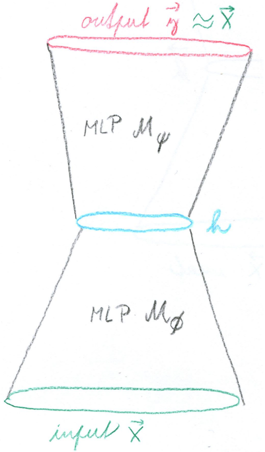
Minimize the **reconstruction** error

$$E = \sum_{i=1}^{p} (\vec{x}_i - \psi(\phi(\vec{x}_i)))^2$$

Autoencoders – neural networks

Both ϕ and ψ can be represented using MLP $\mathcal{M}_\phi$ and $\mathcal{M}_\psi$, respectively.

$\mathcal{M}_\phi$ and $\mathcal{M}_\psi$ can be connected into a single network.



Autoencoders – Usage

- Compression – from $\vec{x}$ to $\vec{h}$.
- Dimensionality reduction – the latent representation $\vec{h}$ has a smaller dimension.
- Pretraining (next slides)
- Generative versions – (roughly) generate $\vec{h}$ from a known distribution, let $\mathcal{M}_\psi$ generate realistic inputs $\vec{x}$

Autoencoder – compression – historical implementation

Architecture: MLP 64 – 16 – 64

Activity: activation function: hyperbolic tangens with limits -1 and 1

Autoencoder – compression – historical implementation

Architecture: MLP 64 – 16 – 64

Activity: activation function: hyperbolic tangens with limits -1 and 1

Data:

- Images 256×256, 8 bits per pixel.
- Samples: input and output is a frame 8×8, randomly selected in the image.
- Inputs normalized to $[-1, 1]$.

Autoencoder – compression – historical implementation

Architecture: MLP 64 – 16 – 64

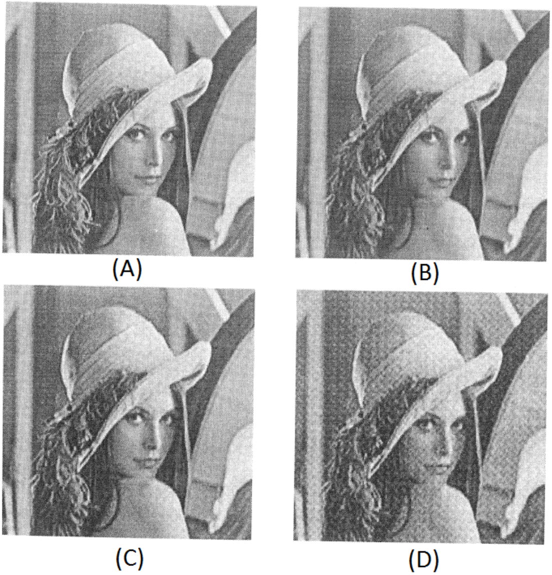
Activity: activation function: hyperbolic tangens with limits -1 and 1

Data:

- Images 256×256, 8 bits per pixel.
- Samples: input and output is a frame 8×8, randomly selected in the image.
- Inputs normalized to $[-1, 1]$.

The goal was to compress images to smaller data size.

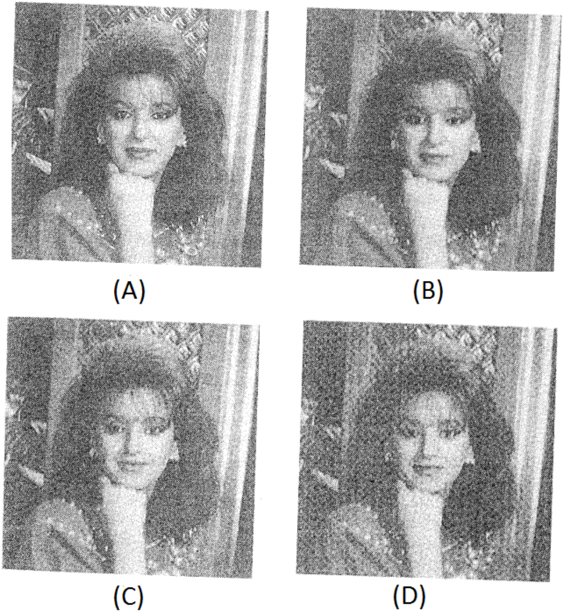
Autoencoder – compression – historical implementation



A frame 8×8 passes through the image 256×256 (no overlap)

- **(A)** original
- **(B)** compression
- **(C)** compression + rounding to 6 bits (1.5 bit per pixel)
- **(D)** compression + rounding to 4 bits (1 bit per pixel)

Dimensionality reduction – compression



New image (trained on the previous one):

- **(A)** original
- **(B)** compression
- **(C)** compression + rounding to 6 bits (1.5 bit per pixel)
- **(D)** compression + rounding to 4 bits (1 bit per pixel)

Application – dimensionality reduction

- Dimensionality reduction: A mapping R from $\mathbb{R}^n$ to $\mathbb{R}^m$ where
  - $m < n$,
  - for every example $\vec{x}$ we have that $\vec{x}$ can be "reconstructed" from $R(\vec{x})$.

Application – dimensionality reduction

- Dimensionality reduction: A mapping R from $\mathbb{R}^n$ to $\mathbb{R}^m$ where
  - $m < n$,
  - for every example $\vec{x}$ we have that $\vec{x}$ can be "reconstructed" from $R(\vec{x})$.
- Standard method: PCA (there are many linear as well as non-linear variants)

Reconstruction – PCA

1024 pixels compressed to 100 dimensions (i.e. 100 numbers).

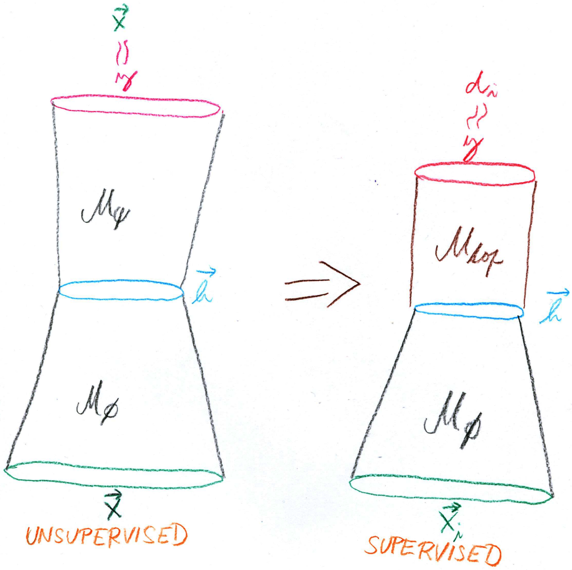
PCA vs Autoencoders

Autoencoders – Pretraining

- An autoencoder is (pre)trained on input data $\vec{x}_i$ *without* desired outputs (unsupervised)

  typically much larger datasets of unlabelled data
- the encoder $\mathcal{M}_\phi$ computes a latent representation for every input vector, it is supposed to extract important features (controversial)
- A new part of the model $\mathcal{M}_{top}$ is added on top of $\mathcal{M}_\phi$ (e.g. a MLP taking the output of $\mathcal{M}_\phi$ as an input).
- Subsequently, labels are added and the whole model (composed of $\mathcal{M}_\phi$ and $\mathcal{M}_{top}$) is trained on labelled data.

Autoencoders – Pretraining



Generative adversarial networks

Generative adversarial Nets, Goodfellow et al, NIPS 2014

An unsupervised generative model.

Two networks:

- **Generator**: A network computing a function $G : \mathbb{R}^k \to \mathbb{R}^n$ which takes a *random* input $\vec{z}$ with a distribution $p_{\vec{z}}$ (e.g., multivariate normal distribution) and returns $G(\vec{z})$ which should follow the target probability distribution.
  E.g., $G(\vec{z})$ could be realistically looking faces.
- **Discriminator**: A network computing a function $D : \mathbb{R}^n \to [0, 1]$ that given $\vec{x} \in \mathbb{R}^n$ gives a probability $D(x)$ that $\vec{x}$ is *not* "generated" by G.
  E.g., $\vec{x}$ can be an image, $D(\vec{x})$ is a probability that it is a true face of an existing person.

What error function will "motivate" G to generate realistically and D to discriminate appropriately?

Generative adversarial networks – error function

Let $\mathcal{T} = \{\vec{x}_1, \ldots, \vec{x}_p\}$ be a training multiset (or a minibatch).

Intuition: G should produce outputs similar to elements of $\mathcal{T}$.
D should recognize that its input is not from $\mathcal{T}$.

Generative adversarial networks – error function

Let $\mathcal{T} = \{\vec{x}_1, \ldots, \vec{x}_p\}$ be a training multiset (or a minibatch).

Intuition: G should produce outputs similar to elements of $\mathcal{T}$.
D should recognize that its input is not from $\mathcal{T}$.

Generate a multiset of noise samples: $\mathcal{F} = \{\vec{z}_1, \ldots, \vec{z}_p\}$ from the distribution $p_{\vec{z}}$.

$$E_{\mathcal{T},\mathcal{F}}(G, D) = -\frac{1}{p}\sum_{i=1}^{p}\Big(\ln D(\vec{x}_i) + \ln(1 - D(G(\vec{z}_i)))\Big)$$

This is just the binary cross entropy error of D which classifies the input as either real, or fake.

The problem can be seen as a game: The discriminator wants to minimize E, the generator wants to maximize E!

The learning algorithm

Denote by W_G and W_D the weights of G and D, respectively.

In every iteration of the training, modify weights of the discriminator and the generator as follows:

The learning algorithm

Denote by W_G and W_D the weights of G and D, respectively.

In every iteration of the training, modify weights of the discriminator and the generator as follows:

For k steps (here k is a hyperparameter) update the discriminator as follows:

- Sample a minibatch $T = \{\vec{x}_1, \ldots, \vec{x}_m\}$ from the training set $\mathcal{T}$.
- Sample a minibatch $F = \{\vec{z}_1, \ldots, \vec{z}_m\}$ from the distribution p_z.
- Update W_D using the *gradient descent* w.r.t. E:

$$W_D := W_D - \alpha \cdot \nabla_{W_D} E_{T,F}(G, D)$$

The learning algorithm

Denote by W_G and W_D the weights of G and D, respectively.

In every iteration of the training, modify weights of the discriminator and the generator as follows:

For k steps (here k is a hyperparameter) update the discriminator as follows:

- Sample a minibatch $T = \{\vec{x}_1, \ldots, \vec{x}_m\}$ from the training set $\mathcal{T}$.
- Sample a minibatch $F = \{\vec{z}_1, \ldots, \vec{z}_m\}$ from the distribution p_z.
- Update W_D using the *gradient descent* w.r.t. E:

$$W_D := W_D - \alpha \cdot \nabla_{W_D} E_{T,F}(G, D)$$

Now update the generator:

- Sample a minibatch $F = \{\vec{z}_1, \ldots, \vec{z}_m\}$ from the distribution p_z.
- Update the *generator* by *gradient ascent*:

$$W_G := W_G - \alpha \cdot \nabla_{W_G} \left(\frac{1}{p} \sum_{i=1}^{p} \ln(1 - D(G(\vec{z}_i))) \right)$$

(The updates may also use momentum, adaptive learning rate etc.)

GAN MNIST

GAN faces

... from the original paper.

GAN refined

... after some refinements.

... none of these people ever lived.