POLITECNICO MILANO 1863

Artificial Neural Networks and Deep Learning

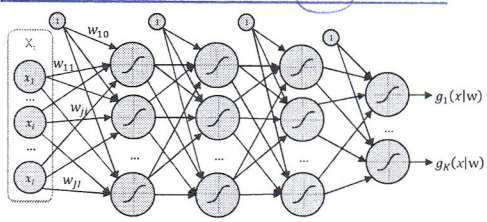
- Recurrent Neural Networks-

Matteo Matteucci, PhD (matteo.matteucci@polimi.it)
Artificial Intelligence and Robotics Laboratory
Politecnico di Milano

AIRLAB

Sequence Modeling

So far we have considered only «static» datasets

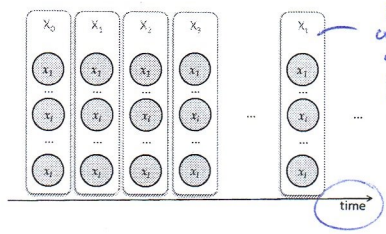


the output of this does not depend on which inputs we've seen in the past (there is no notion of dynamics, memory or dependency between the datapoint: once we get one point we'll always get the same output, no matter of what the past set of inputs was)

(image classification is something like this!)

Sequence Modeling

So far we have considered only «static» datasets



However it might be that, if we observe an input at a certain time t, the dataset can be built as a sequence of input. This means that at time 2 the input might depend on what was the input at time 1 and 0. The output now depends on the sequence of inputs we've seen in the past.

Sequence Modeling

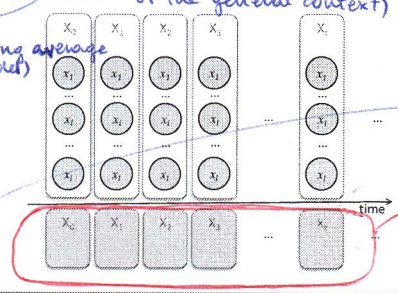
Different ways to deal with «dynamic» data:

these data are not only "temporal": a text dataset can be interpreted like this: a text is a sequence of characters, and there is a high correlation between characters in a word (and this correlation sometimes depends on a topic of the general context)

- Memoryless models:
  - Autoregressive models
  - Feedforward neural networks
- Models with memory:
  - Linear dynamical systems
  - Hidden Markov models
  - Recurrent Neural Networks
  - ...

we don't remember all the sequence: we use only the previous or the two previous inputs to predict the next one (⇒ autoregressive models that can be implemented through feedforward neural networks)

or (moving average model)



They break the feedforward assumption and they have recurrent connections.

New notation: with X_j we mean the input vector (which has size i) at time j (compact notation for inputs)

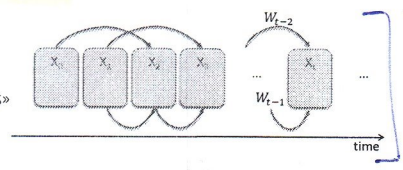
If we want to remember longer sequences we have to add memory: we add an internal state and we use it as memory

Memoryless Models for Sequences

Autoregressive models

- Predict the next input from previous ones using «delay taps»

This model can be implemented with a feedforward neural network

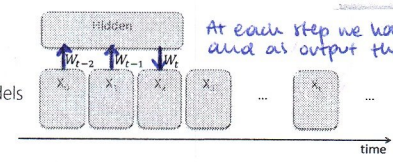


$X_t = W_{t-1} X_{t-1} + W_{t-2} X_{t-2}$

The output of the model is the weighted average of the previous time + the weighted average of two-steps in the past ("DELAY TAPS"). This model has no memory; after 2 steps behind it does not remember anything.

Feed forward neural networks

- Generalize autoregressive models using non linear hidden layers



At each step we have as input the 2 delay taps and as output the next input

Again we don't have memory: as soon as the windows moves there's no way to remember the past.

How can we add more memory? (Generally, memory?)

inputs, **state** (which evolves during time), **output** (which depends on the current state)

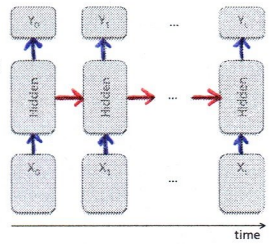
Dynamical Systems (Models with Memory)

Stochastic systems ...

a model which describes how the data has been generated

Generative models with a hidden state which cannot be observed directly

- The hidden state has some dynamics possibly affected by noise and produces the output
- To compute the output need to infer hidden state
- Input are treated as driving inputs



output (generated by the internal state = memory) (the current internal state)

memory (internal state) | The memory propagates along time

inputs (that modify the internal representation of the system)

In linear dynamical systems this becomes:

- State continuous with Gaussian uncertainty
- Transformations are assumed to be linear
- State can be estimated using *Kalman filtering*

Two examples of this general model in which we have some inputs, these inputs modify the hidden state and the output is a function of the hidden state.

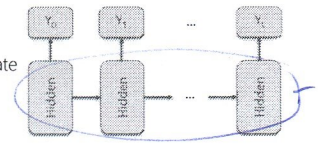
How do we represent this? With a **RECURRENT NEURAL NETWORK**

Dynamical Systems (Models with Memory)

Stochastic systems ...

Generative models with a hidden state which cannot be observed directly

- The hidden state has some dynamics possibly affected by noise and produces the output
- To compute the output need to infer hidden state
- Input are treated as driving inputs



the Markov chain is hidden, we only observe the outputs

In hidden Markov models this becomes:

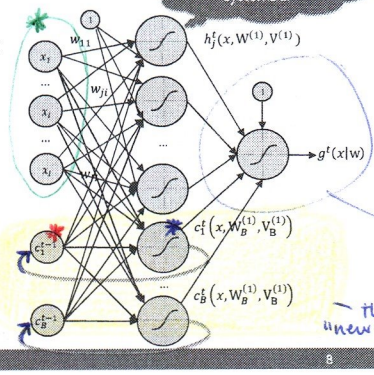
- State assumed to be discrete, state transitions are stochastic (transition matrix)
- Output is a stochastic function of hidden states
- State can be estimated via *Viterbi algorithm*.

Recurrent Neural networks

Deterministic systems ...

Memory via recurrent connections:

- Distributed hidden state allows to store a information efficiently
- Non-linear dynamics allows complex hidden state updates



this is the "new" part: the **memory**

In recurrent neural net we consider the feedback loop: this feedback loop says: the internal memory (represented by the output of ✱) is a function of the internal memory at the previous step (✱) plus a linear combination of the inputs (✱).

The output has a part that depends on the current input and a part which depends on the story of the system.

"With enough neurons and time, RNNs can compute anything that can be computed by a computer."

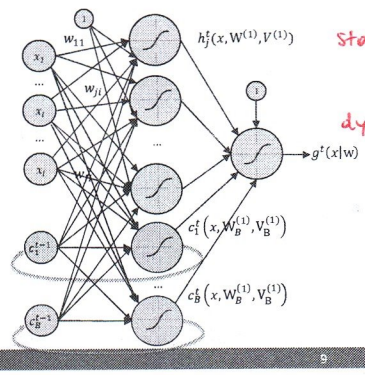
(Computation Beyond the Turing Limit Hava T. Siegelmann, 1995)

This means that the represented structure behaves like a computer with memory (and so it can do the same calculations as a computer)

Recurrent Neural networks

Memory via recurrent connections:

- Distributed hidden state allows to store a information efficiently
- Non-linear dynamics allows complex hidden state updates



Notice: the weights for the static part are in **W**, the weights for the dynamic part are in **V**.

static
- $h(\cdot)$ are weighted with **W**
- $c(\cdot)$ are weighted with **V**.

dynamic

Let's try to write the output of this network as a function of the inputs:

$$g^t(x_n|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j^t(\cdot) + \sum_{b=0}^{B} v_{1b}^{(2)} \cdot c_b^t(\cdot)\right)$$

Output of a hidden layer:

$$h_j^t(\cdot) = h_j^t\left(\sum_{j=0}^{J} w_{ji}^{(1)} \cdot x_{i,n} + \sum_{b=0}^{B} v_{jb}^{(1)} \cdot c_b^{t-1}\right)$$

"$c(\cdot)$" for "**context**":

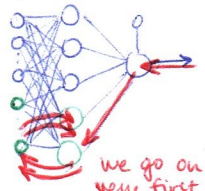
$$c_b^t(\cdot) = c_b^t\left(\sum_{j=0}^{J} v_{bi}^{(1)} \cdot x_{i,n} + \sum_{b=0}^{B} v_{bb}^{(1)} \cdot c_b^{t-1}\right)$$

in red we highlight the dependencies of the past

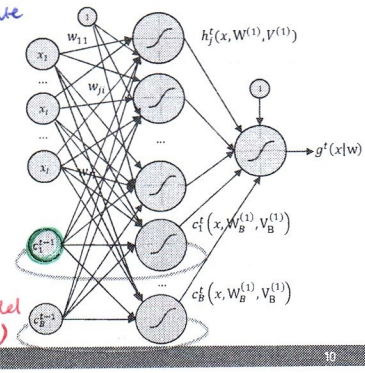
This **model** is difficult to train because we have to train weights which are not feedforward.
A critical aspect of recurrent neural net is that we cannot train them with backprop. (one of the conditions for backprop was to not have loops).

Backpropagation Through Time

This means that to backpropagate the error we have to follow the path to the very first input:



we go on till the very first input (starting point of the model to process the entire sequence)

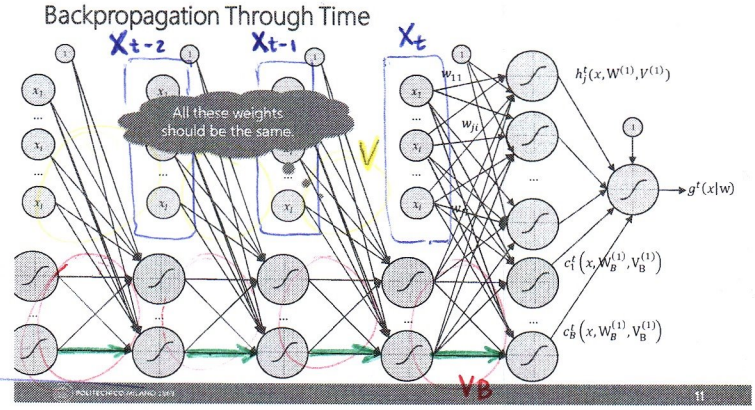


We basically **UNROLL** the recurrent part (connections)

Once we unroll the recurrent connections we get a feedforward neural network:

in this way we've backpropagating errors through time

Backpropagation Through Time



Notice that since we're unrolling, all the weights are the same: for instance all the weights must be the same.

How this (→) is done in practice? →

1. We unroll for **U** steps; it's not useful to unroll all the steps, we unroll for a (fixed) finite time of steps:

TRUNCATED BACKPROPAGATION

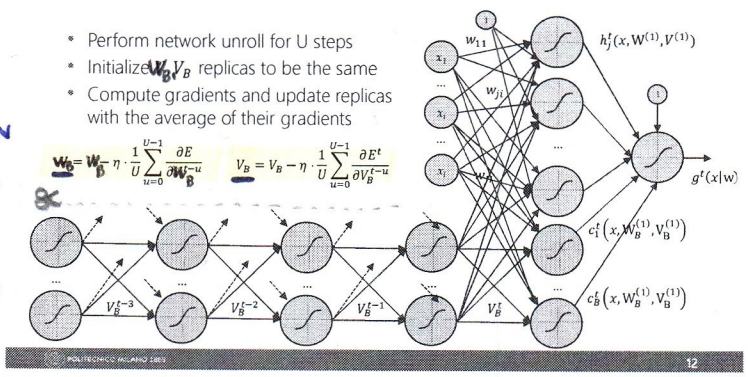
In this way the network cannot learn any connection between events that are more than **U** steps apart.

2. We initialize **V** and V_B (which are the weights that we replicate) to be the same (their replicas to be the same, **V** and V_B has no connection)

Backpropagation Through Time

- Perform network unroll for U steps
- Initialize W_B, V_B replicas to be the same
- Compute gradients and update replicas with the average of their gradients

$$W_B = W_B - \eta \cdot \frac{1}{U} \sum_{u=0}^{U-1} \frac{\partial E}{\partial W_B^{t-u}} \qquad V_B = V_B - \eta \cdot \frac{1}{U} \sum_{u=0}^{U-1} \frac{\partial E^t}{\partial V_B^{t-u}}$$



3. We compute the derivative of the output with respect to each of the gradients ($V_B^t, V_B^{t-1}, V_B^{t-2}, \ldots$) and we update **V** and V_B.

In this way, instead of updating all their copies independently (which would make them different but as we said before they cannot be different) we update them together, with the average gradient through time.

→ We don't keep all the values of the memory neurons, we compute all the gradients for all the weights in the past.

How much should we go back in time?

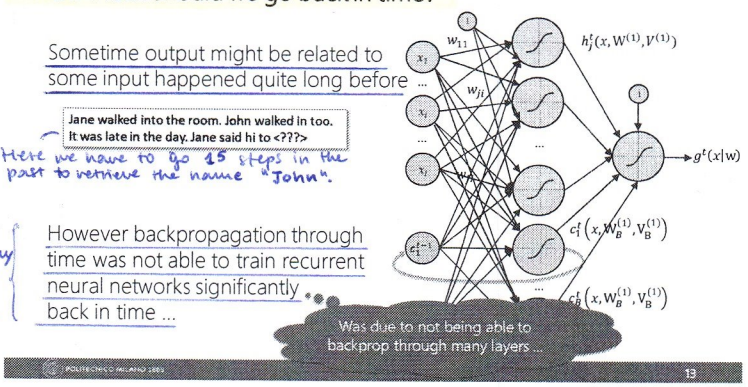
Sometime output might be related to some input happened quite long before

Jane walked into the room. John walked in too. It was late in the day. Jane said hi to <???>

Here we have to go 15 steps in the past to retrieve the name "John".

However backpropagation through time was not able to train recurrent neural networks significantly back in time ...

Was due to not being able to backprop through many layers ...

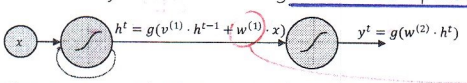


Recurrent neural net cannot learn too many steps back in the past. Why? Because of the **VANISHING GRADIENT**

(notice that vanishing gradient is a problem for all very deep neural networks, not only for recurrent neural networks)

How much can we go back in time? Not more than ~ 10 steps in the past:

To better understand why it was not working consider a simplified case:



Backpropagation over an entire sequence S is computed as

$$\frac{\partial E}{\partial w} = \sum_{t=1}^{S} \frac{\partial E^t}{\partial w} \Rightarrow \frac{\partial E^t}{\partial w} = \sum_{t=1}^{t} \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial h^t} \frac{\partial h^t}{\partial h^k} \frac{\partial h^k}{\partial w} \Rightarrow \frac{\partial h^t}{\partial h^k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} v^{(1)} g'(w^{(2)} \cdot h^{i-1})$$

we want to update this error through backpropagation (by deriving the error function): $\frac{\partial E}{\partial w}$

as many products as the number of steps we go in the past

If we consider the norm of these terms

$$\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\| \leq \|v^{(1)}\| \|g'(h^{i-1})\| \Rightarrow \left\| \frac{\partial h^t}{\partial h^k} \right\| \leq (\gamma_v \cdot \gamma_{g'})^{t-k}$$

If $(\gamma_v \cdot \gamma_{g'}) < 1$ this converges to 0 ...

With Sigmoids and Tanh we have vanishing gradients

$$\left\| \frac{\partial h^t}{\partial h^k} \right\| \leq \underbrace{(\gamma_v \cdot \gamma_{g'})}_{\text{norm of the weights}}^{t-k} \quad \text{(norm of } g' \text{)}$$

$t-k$ = how many steps we want to go in the time

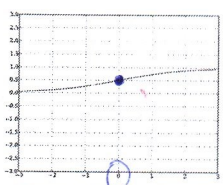
If the product $\gamma_v \cdot \gamma_{g'}$ is < 1 we can't go too much back in time, otherwise $(\gamma_v \cdot \gamma_{g'})^{t-k} \sim 0$.

→ $\partial E / \partial w \to 0$ exponentially with $t-k$ (if $\gamma_v \cdot \gamma_{g'} < 1$)

Why Sigmoids and Tanh are so bad? →

→ because they always lead to $\gamma_v \cdot \gamma_{g'} < 1$

Which Activation Function?

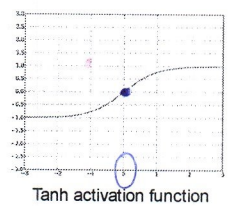


Sigmoid activation function

$$g(a) = \frac{1}{1 + \exp(-a)}$$

$$g'(a) = g(a)(1 - g(a))$$

$$g'(0) = g(0)(1 - g(0)) = \frac{1}{1 + \exp(0)} \cdot \frac{\exp(0)}{1 + \exp(0)} = 0.25$$



Tanh activation function

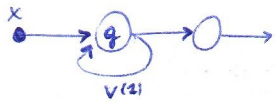
$$g(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}$$

$$g'(a) = 1 - g(a)^2$$

$$g'(0) = 1 - g(0)^2 = 1 - \left(\frac{\exp(0) - \exp(0)}{\exp(0) + \exp(0)}\right)^2 = 1$$

max. val of $g' = 0.25$ max. val of $g' = 1$

How can we avoid the problem?

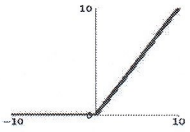


Dealing with Vanishing Gradient ⇒ ~~Sigmoid~~, ~~Tanh~~ ⇒ ReLU ✓

Force all gradients to be either 0 or 1

$$g(a) = ReLu(a) = \max(0, a)$$

$$g'(a) = 1_{a>0}$$

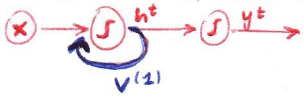


This provides: $\gamma_g = 1$:

$$\left\| \frac{\partial h^t}{\partial h^k} \right\| \leq (\gamma_v \cdot \gamma_g)^{t-k} \leq (\gamma_v)^{t-k}$$

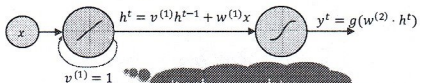
we still have to deal with this (we want $\gamma_v \geq 1$)

$\gamma_v = \|v^{(1)}\|$, where $v^{(1)}$ is:



⇒ we set $v^{(1)} = 1$.
Why don't we set it > 1?
We don't want an **EXPLODING GRADIENT** (which is a problem such as the vanishing gradient)

Build Recurrent Neural Networks using small modules that are designed to remember values for a long time.



It only accumulates the input ...

However this has a problem: it only cumulates the input, we want it to work as a full memory.

Long Short-Term Memories (LSTM) — more effective recurrent neural network

Hochreiter & Schmidhuber (1997) solved the problem of vanishing gradient designing a memory cell using logistic and linear units with multiplicative interactions:

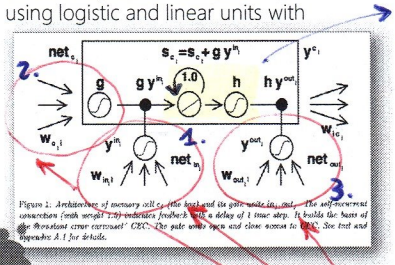


Figure 1: Architecture of memory cell c_j (the box) and its gate units in_j, out_j. The self-recurrent connection (with weight 1.0) indicates feedback with a delay of 1 time step. It builds the basis of the 'constant error carrousel' CEC. The gate units open and close access to CEC. See text and appendix A.1 for details.

- Information gets into the cell whenever its "*write*" gate is on.
- The information stays in the cell so long as its "*keep*" gate is on.
- Information is read from the cell by turning on its "*read*" gate.

Can backpropagate through this since the loop has fixed weight

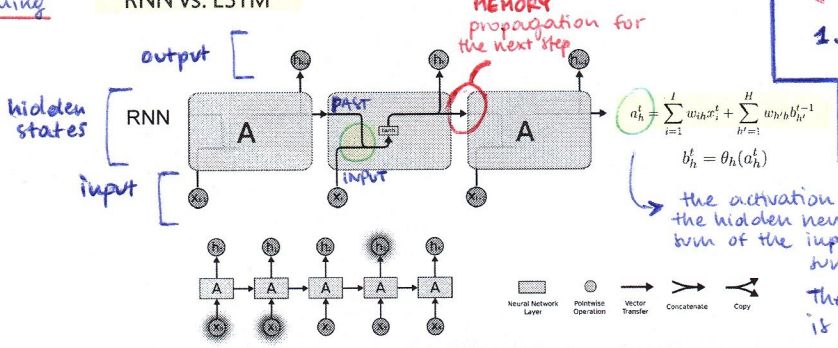
neuron with a loop (and so with memory) with weight = 1 (so it does not explode) and a linear activation function (not ReLU)

With this approach informations can propagate as much as we want.
(since the weights of the recurrent part are always fixed (in the sense that we don't need to unroll, we just sum up the inputs))
⇒ CEC solves the vanishing gradient problem.

This core elements is called **CONSTANT ERROR CAROUSEL**. However, this CEC only add the inputs, and to they added 3 gates. These gates decide:

1. when it is the case to accumulate a particular information ("write")
2. what to keep ("keep")
3. how much of the informations in the network we should read.

RNN vs. LSTM



$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1}$$

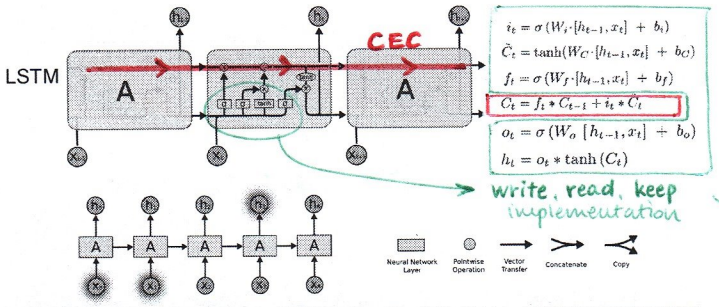
$$b_h^t = \theta_h(a_h^t)$$

the activation function of the hidden neuron is the weighted sum of the input + the weighted sum of the past.

The output ($b_h^t = \theta_h(a_h^t)$) is a nonlinear function-result of the weighted sum of the input + the weighted sum of the past.

LSTM Images from: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-Term Memory



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

they're represented by these formulas

write, read, keep implementation

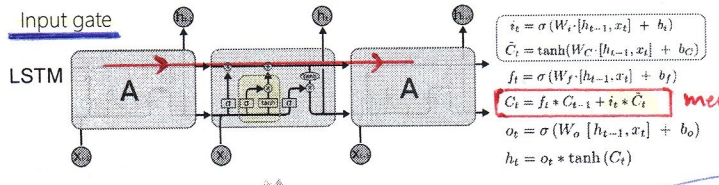
LSTM Images from: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-Term Memory

WRITE THE MEMORY

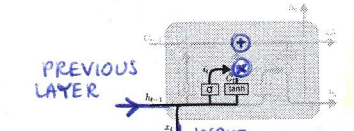
what we do when we want informations to get into the memory (and so to get into →)

Input gate



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$ — memory
$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

However, i_t (which is a sigmoidal function, so the output is $\in [0,1]$) will say how much of this new info we'll be added to the memory (if $i_t = 0$ the gate is closed)



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

this is the information that we're going to add to the memory: iperbolic tangent of a weighted combination of the input and the previous layer.

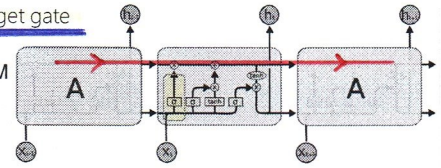
LSTM Images from: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-Term Memory

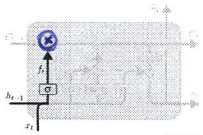
CLEAN THE MEMORY

Forget gate

LSTM



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$



$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

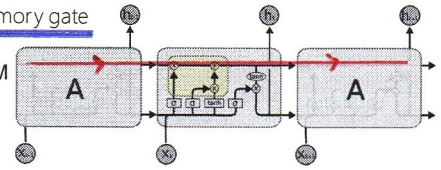
This says how much of the memory we should keep. It's a sigmoid ⇒ $\in [0,1]$. if $f = 0$ we reset the memory (we've not keeping it), if $f = 1$ we keep the entire memory.

LSTM Images from: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

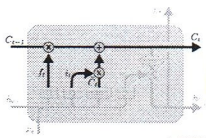
Long Short-Term Memory

Memory gate

LSTM



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

at each step we partially clean the memory and partially write the memory

LSTM Images from: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

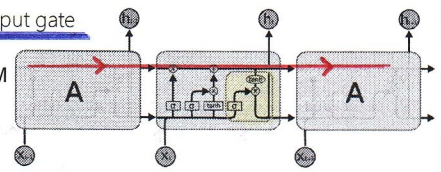
Long Short-Term Memory

in green what we learn during back propagation

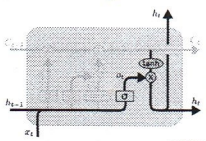
READ FROM THE MEMORY

Output gate

LSTM



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$



$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

The gate "o" decide (between 0 and 1 ($\in [0,1]$) since it's a sigmoid) wether to read or not from the memory

the output of the memory is the hiperbolic tangent of the content times "o". What is "o"? Another gate.

LSTM Images from: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

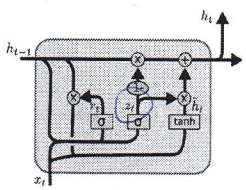
There are alternatives to the **LSTM**:

However this is not remarkably better than **LSTM** (maybe faster in training (it has fewer parameters), but it doesn't have very different performances).

Gated Recurrent Unit (GRU)

it has less gates than **LSTM** (it also re-uses a gate twice (z_t))

It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes.

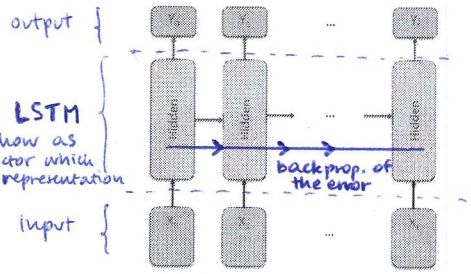


$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$
$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

LSTM Images from: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM Networks

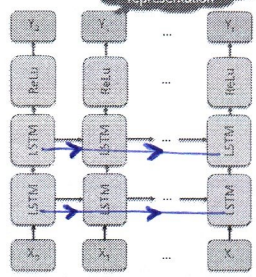
You can build a computation graph with continuous transformations.



!! (it works somehow as a feature extractor which finds an internal representation of the sequence)

Multiple Layers and Bidirectional LSTM Networks

A computati... Hierarchical representation ...with continuous transformations.



we can put multiple LSTM one after the other (just like with CNN)

Tips & Tricks

when we have available the entire sequence at the beginning (e.g. we have to translate a text and we have the whole text at the beginning)

When conditioning on full input sequence Bidirectional RNNs exploit it:

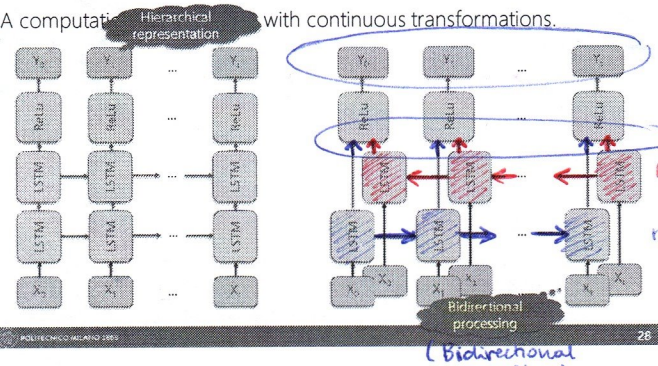
- Have one RNNs traverse the sequence left-to-right
- Have another RNN traverse the sequence right-to-left
- Use concatenation of hidden layers as feature representation

then we can use 2 RNN: one read the sequence left-to-right, the other read the sequence right-to-left and then we concatenate the hidden layers representations to produce the outputs.

for example

Multiple Layers and Bidirectional LSTM Networks

A computati... Hierarchical representation ...with continuous transformations.



3. we produce the output

2. we concatenate the 2 representations

representation from right to left

representation from left to right (simultaneously to the previous)

1.

(Bidirectional processing)

Tips & Tricks

When conditioning on full input sequence Bidirectional RNNs exploit it:

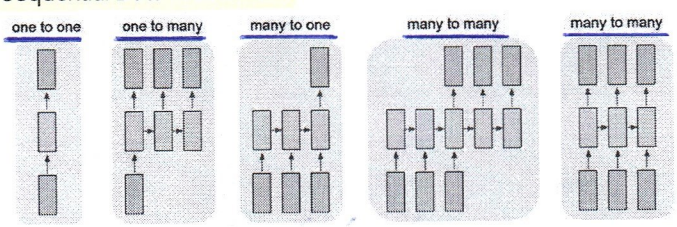
- Have one RNNs traverse the sequence left-to-right
- Have another RNN traverse the sequence right-to-left
- Use concatenation of hidden layers as feature representation

When initializing RNN we need to specify the initial state

- Could initialize them to a fixed value (such as 0)
- Better to treat the initial state as learned parameters ! better !
  - Start off with random guesses of the initial state values
  - Backpropagate the prediction error through time all the way to the initial state values and compute the gradient of the error with respect to these
  - Update these parameters by gradient descent

Different architectures that we can build using RNN:

Sequential Data Problems



Fixed-sized input to fixed-sized output (e.g. image classification)

Sequence output (e.g. image captioning takes an image and outputs a sentence of words)

Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment)

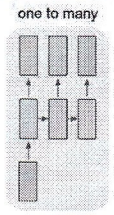
Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French)

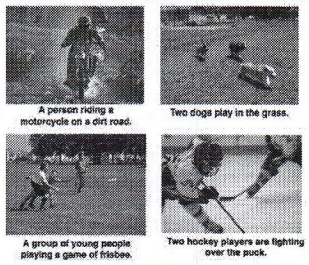
Synced sequence input and output (e.g. video classification where we wish to label each frame of the video)

LSTM Images Credits: Andrej Karpathy

Sequence to Sequence Learning Examples (1/3)

Image Captioning: input a single image and get a series or sequence of words as output which describe it. The image has a fixed size, but the output has varying length.



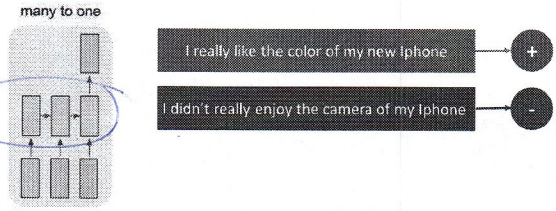


Why we need recurrence? To write text: if we already said "two" we don't want to repeat it and so we have to memorize it.

Sequence to Sequence Learning Examples (2/3)

Sentiment Classification/Analysis: input a sequence of characters or words, e.g., a tweet, and classify the sequence into positive or negative sentiment. Input has varying lengths; output is of a fixed type and size.

Remember: we can think about it as a sort of machine which is building an internal representation of the sequence

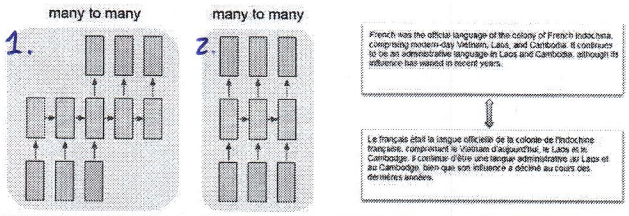


Sequence to Sequence Learning Examples (3/3)

Language Translation: having some text in a particular language, e.g., English, we wish to translate it in another, e.g., French. Each language has it's own semantics and it has varying lengths for the same sentence.

They go under the name of **SEQUENCE TO SEQUENCE** MODELS (Seq2Seq Model (↓))

The difference between 1. and 2. is that in 2. we can produce already the first output without having seen the "future". in the case 1. we have to wait until the 3rd input (of the sequence) to predict the first output.



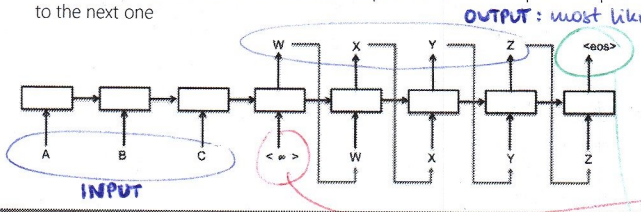
Seq2Seq Model

we encode a sentence and we generate another sentence

The Seq2Seq model follows the classical encoder decoder architecture

- At trainng time the decoder **does not** feed the output of each time step to the next; the input to the decoder time steps are the target from the training
- At inference time the decoder feeds the output of each time step as an input to the next one

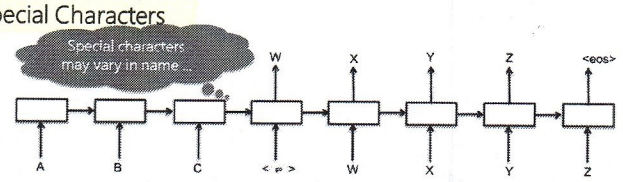
OUTPUT: most likely symbol given the sequence of symbols which came before. Once the output comes out, it's then used as the next input. The sequence is now extended and we can go on predicting.



special character which says: "From now on generate an output"

we predict as the next most likely letter after the Z the end of the sequence (<eos>)

Special Characters



All the sequences must have the same length. But we cannot force it. So, we add some PADDING to fill the sequence.

- <PAD>: During training, examples are fed to the network in batches. The inputs in these batches need to be the same width. This is used to pad shorter inputs to the same width of the batch
- <EOS>: Needed for batching on the decoder side. It tells the decoder where a sentence ends, and it allows the decoder to indicate the same thing in its outputs as well.

This character ends the prediction. = "end of sentence" = <eos>

- <UNK>: On real data, it can vastly improve the resource efficiency to ignore words that do not show up often enough in your vocabulary by replace those with this character.
- <SOS>/<GO>: This is the input to the first time step of the decoder to let the decoder know when to start generating output.

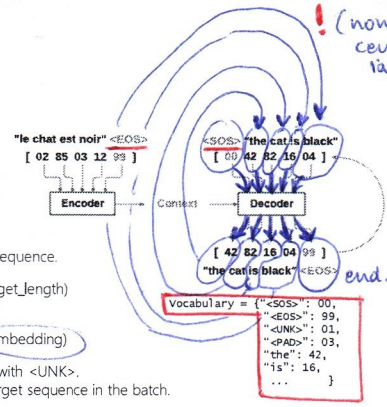
"start of sentence", used from the input sequence to the output sequence (after receiving this as input we start generating outputs)

How do we train this network?

In batches: → Dataset Batch Preparation

1. Sample batch_size pairs of (source_sequence, target_sequence).
2. Append <EOS> to the source_sequence
3. Prepend <SOS> to the target_sequence to obtain the target_input_sequence and append <EOS> to obtain target_output_sequence.
4. Pad up to the max_input_length (max_target_length) within the batch using the <PAD> token.
5. Encode tokens based of vocabulary (or embedding)
6. Replace out of vocabulary (OOV) tokens with <UNK>. Compute the length of each input and target sequence in the batch.

WORDS EMBEDDING is something **very** import. (we'll see it later)



! (non ci passavamo le frecce, ma non centrano niente le frecce blue con la parte dell'encoder) !

How is the loss computed in this case?

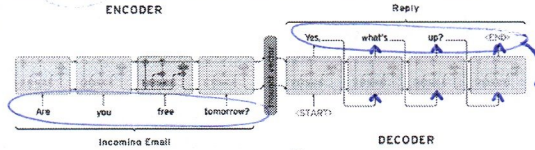
→ In this case the loss is the ... in predicting each output

Sequence to Sequence Modeling

S = source sentence, T = target sentence

Given <S, T> pairs, read S, and output T' that best matches T

for every source sentence S we predict an output sentence T' that is as close as possible to the target sentence T



desired answer

$$p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_1, \ldots, y_{t-1}) \quad (*)$$

$$1/|S| \sum_{(T,S) \in \mathcal{S}} \log p(T|S)$$

LSTM Images Credits: Andrej Karpathy

We can imagine the prediction problem as a classification problem during the sequence: what is the item in the vocabulary that we should put out now? "Yes". What item now? "what's". And so on. Basically we have a sequence of classification tasks. If we try to apply the idea of maximum likelihood (*), we can assume that, given the previous words, each output is independent of all the others.

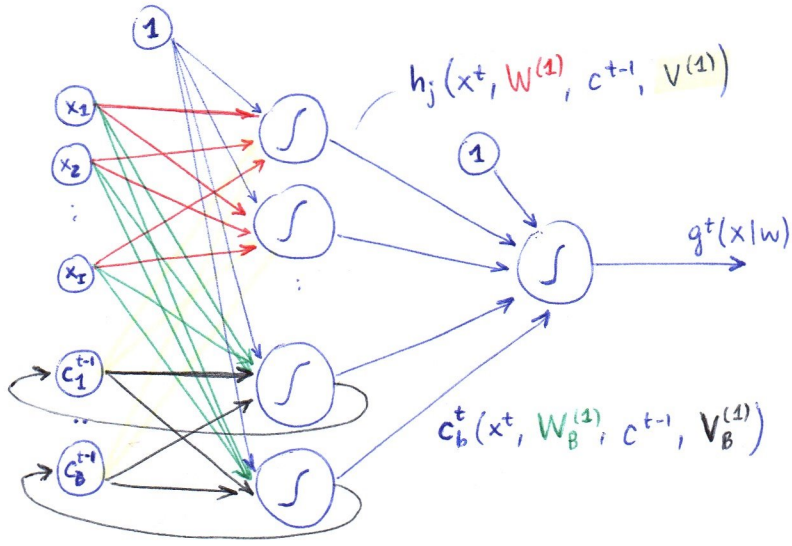
$\Rightarrow p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_{1, \ldots, y_{t-1}})$

$\Rightarrow$ likelihood $(T') = \mathbb{P}(\text{"yes"} | \langle start \rangle) \cdot \mathbb{P}(\text{"what's"} | \text{"yes"}) \cdot \mathbb{P}(\text{"up"} | \text{"what's"}) \cdot \mathbb{P}(\langle END \rangle | \text{"up"})$

current HIDDEN STATE

Because of this (→) we can compute the cross-entropy along the sequence.

Representation:



$$g^t(x|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} h_j^t(\cdot) + \sum_{b=0}^{B} v_{1b}^{(2)} c_b^t(\cdot)\right)$$

$$h_j^t(\cdot) = h_j\left(\sum_{i=0}^{J} w_{ji}^{(1)} x_{i,n}^t + \sum_{b=0}^{B} v_{jb}^{(1)} c_b^{t-1}\right)$$

$$c_b^t(\cdot) = c_b\left(\sum_{i=0}^{J} v_{bi}^{(1)} x_{i,n}^t + \sum_{k=0}^{B} v_{bk}^{(1)} c_k^{t-1}\right)$$

POLITECNICO MILANO 1863

Artificial Neural Networks and Deep Learning

- Beyond Seq2Seq Architectures-

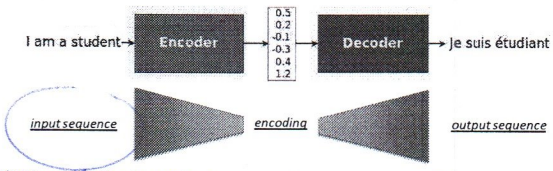
Matteo Matteucci, PhD (matteo.matteucci@polimi.it)
Artificial Intelligence and Robotics Laboratory
Politecnico di Milano

AIRLAB

Reviewed: encoding-decoding mechanism

Extending Recurrent Neural Networks

Recurrent Neural Networks have been extended with memory to cope with very long sequences and the encoding bottleneck ...



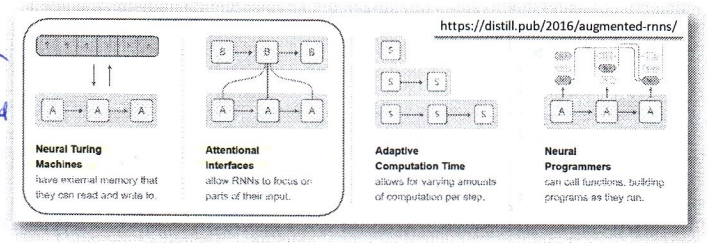
However if the input sequence is very long, the encoding fails to compress all the information

→ for this reason, for remember (or encode) very long sequences, the recurrent neural network has been extended with memory. How?

There are different ways.

Extending Recurrent Neural Networks

Recurrent Neural Networks have been extended with memory to cope with very long sequences and the encoding bottleneck ...

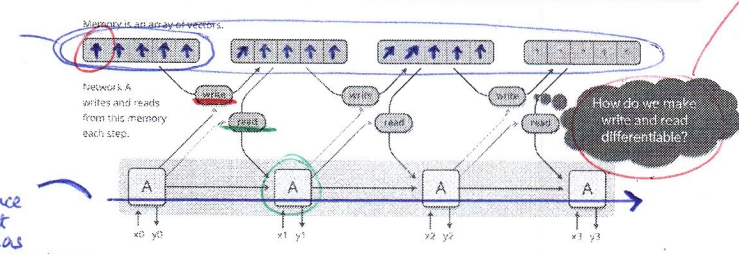


We'll focus on attentional interfaces (and to better understand them we'll go through neural Turing machines as well)

To train a neural network we have to perform backpropagation (= compute derivatives). How do we make the "write" and "read" operation differentiable?

Neural Turing Machines

Neural Turing Machines combine a RNN with an external memory bank.



Memory: array of vectors (vectors are represented by arrows)

We have a memory and a recurrent neural network which act as the reader-writer. For instance: the vector ↑ is changed by the recurrent neural network into ↗ (the rnn can modify the memory). Then the rnn can read from the memory to produce the next write: (A).

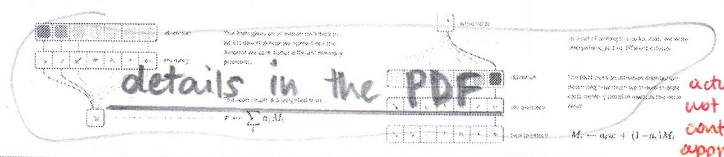
The execution is a sequence of states of this recurrent neural network which has some input x_j and some output y_j at each time step (time/execution step). The output it based on the previous state of the recurrent neural network and also on what it written in the memory.

Neural Turing Machines Idea

Neural Turing Machines challenge:

- We want to learn what (vector) to write/read but also where (address in memory (memory addresses are discrete)) to write it
- Memory addresses are be fundamentally discrete
- Write/read differentiable w.r.t the location we read from or write → Attention mechanism! to

Solution: *Every step, read and write everywhere, just to different extents.*



actually they're not 0/1, they're continuous approximations of 0/1 (continuous and differentiable approximations of 0/1)

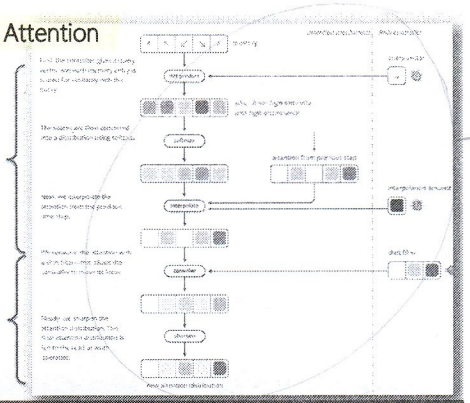
at every step we'll read mostly from one cell but a little bit also from other cells scattered (= "spaghetti") around the memory. This "everywhere but especially from here" is called **ATTENTION MECHANISM** (which is usually implemented by a **SOFTMAX** operation which says "1" where we want to read/write, "0" everywhere else.

How do we decide which positions in memory to focus their attention on? The mechanism is a sort of ASSOCIATIVE MEMORY.

Neural Turing Machines Attention

The whole process of producing weights needed to read and write can be divided in 2:

1. *Content-based attention*: searches memory and focus on places that match what they're looking for
2. *Location-based attention*: allows relative movement in memory enabling the NTM to loop.



details in the PDF

Neural Turing Machines Extensions

NTM perform algorithms, previously beyond neural networks:

- Learn to store a long sequence in memory
- Learn to loop and repeat sequences back repeatedly
- Learn to mimic a lookup table
- Learn to sort numbers ...

> we don't have to compress it in the bottleneck used in the seq2seq model

But the most interesting thing (to me) was the attention mechanism!

Some extension have been proposed to go beyond this:

- Neural GPU overcomes the NTM's inability to add and multiply numbers
- Zaremba & Sutskever train NTMs using reinforcement learning instead of the differentiable read/writes used by the original
- Neural Random Access Machines work based on pointers
- Others have explored differentiable data structures, like stacks and queues

Attention Mechanism in Seq2Seq Models

Considering the sequential dataset:

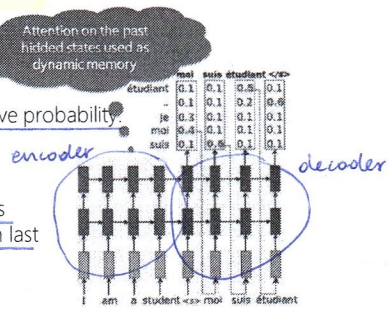
$$\{((x_1, \dots, x_n), (y_1, \dots, y_m))\}_{i=1}^{N}$$

Attention on the past hidden states used as dynamic memory

The decoder role is to model the generative probability:

$$P(y_1, \dots, y_m | x)$$

probability of generating a certain sequence $(y_1, \dots, y_m)$ given the embedding of the input sequence ($= x$).

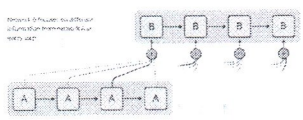


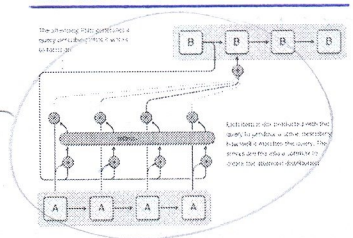
In "vanilla" seq2seq models, the decoder is conditioned initializing the initial state with last state of the encoder.

Works well for short and medium-length sentences; however, for long sentences, becomes a bottleneck ⟹ Seq2Seq is not good for longer sentences. How can we improve? Adding attention!

Attention Mechanism in Seq2Seq Models

Let's use the same idea of Neural Turing Machines to get a differentiable attention and learn where to focus attention.





details in the PDF

Attention distribution is usually generated with content-based attention.

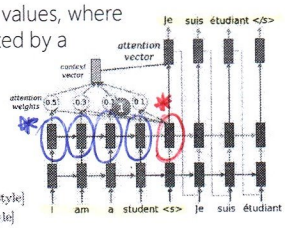
Each item is thus weighted with the query response to produce a score

Scores are fed into a softmax to create the attention distribution

Attention Mechanism in Seq2Seq Models

Attention function maps query and set of key-value pairs to an output.

Output computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function:



1. Compare current target hidden state h_t, with source states h_s to derive attention

$$\text{score}(h_t, \bar{h}_s) := \begin{cases} h_t^\top W \bar{h}_s & \text{[Luong's multiplicative style]} \\ v_a^\top \tanh(W_1 h_t + W_2 \bar{h}_s) & \text{[Bahdanau's additive style]} \end{cases}$$

We figure out how the current hidden state is similar to the past states: we compare * with every *.

Notice that: attention is something we learn, not something that we decide.

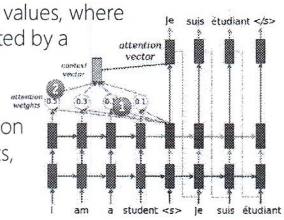
→ Based on the similarities, attention is decided.

Attention Mechanism in Seq2Seq Models

Attention function maps query and set of key-value pairs to an output.
Output computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function:

2. Apply the softmax function on the attention scores and compute the attention weights, one for each encoder token

$$\alpha_{ts} = \frac{\exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s)\right)}{\sum_{s'=1}^{S} \exp\left(\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_{s'})\right)}$$

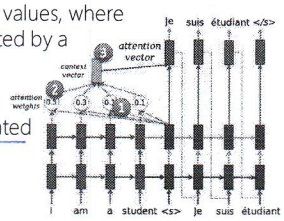


Attention Mechanism in Seq2Seq Models

Attention function maps query and set of key-value pairs to an output.
Output computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function:

3. Compute the context vector as the weighted average of the source states

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s$$

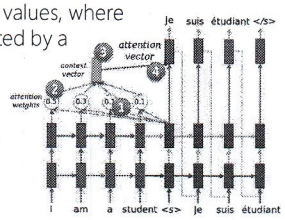


Attention Mechanism in Seq2Seq Models

Attention function maps query and set of key-value pairs to an output.
Output computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function:

4. Combine the context vector with current target hidden state to yield the final attention vector

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W}_{\boldsymbol{c}}[\boldsymbol{c}_t; \boldsymbol{h}_t])$$

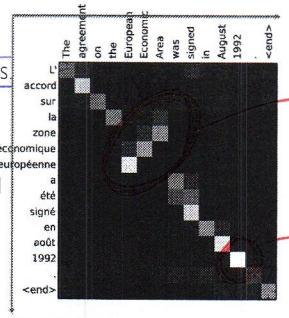


Attention Visualization

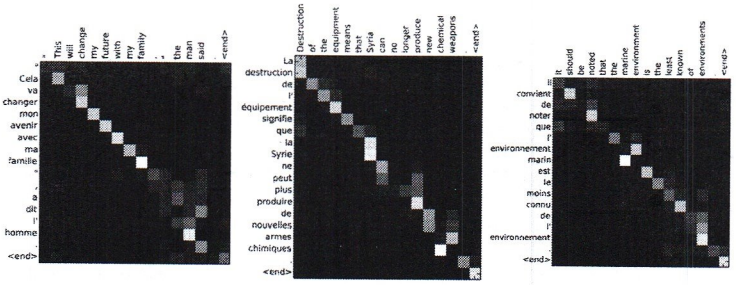
Alignment matrix is use to visualize attention weights between source and target sentences.

For each decoding step, i.e., each generated target token, describes which are the source tokens that are more present in the weighted sum that conditioned the decoding.

We can see attention as a tool in the network's bag that, while decoding, allows it to pay attention on different parts of the source sentence.



Attention Visualization



From the scores obtained in 1. we compute the alphas (which are basically computed through the softmax)

Once we know the α's we know how to read from the memory: a weighted sum (weights given by α's)

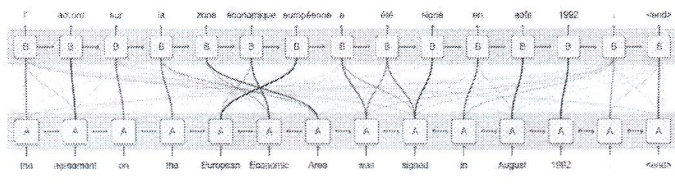
When we translate "European Economic Area" we treat it as 3 different words but highly correlated

Viceversa, "1992" translates only with "1992", it does not have correlation with other words

Attention Mechanism in Translation

Check the demo!!!

Attention allows processing the input to pass along information about each word it sees, and then for generating the output to focus on words



(Diagram derived from Fig. 3 of Bahdanau, et al. 2014)

This is an alternative representation of the matrix. For every output word we can see where the attention was in the input.

Attention Mechanism in Voice Recognition

Check the demo!!!

Attention allows one RNN to process the audio and then have another RNN skim over it, focusing on relevant parts as it generates a transcript.

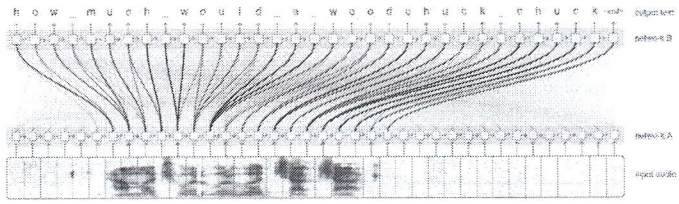


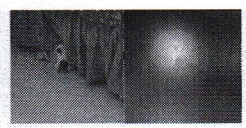
Figure derived from Chan, et al. 2015

Attention Mechanism in Image Captioning

A CNN processes the image, extracting high-level features. Then an RNN runs, generating a description of the image based on the features.

As it generates each word in the description, the RNN focuses on the CNN interpretation of the relevant parts of the image.

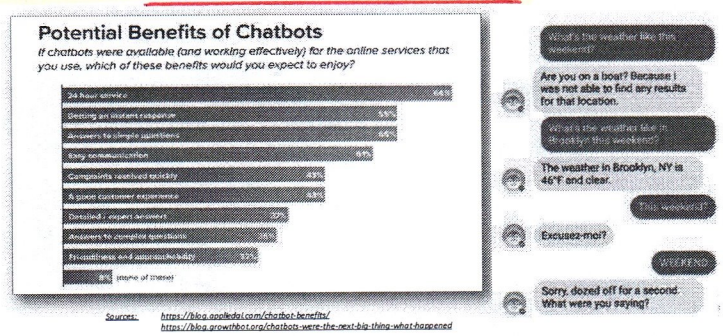
A woman is throwing a frisbee in a park.



A dog is standing on a hardwood floor.

A stop sign is on a road with a mountain in the background.

Attention in Response Generation (i.e., Chatbots)

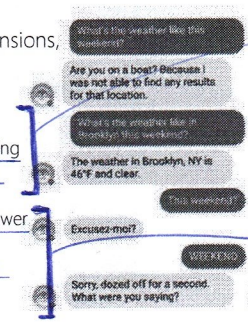


Sources: https://blog.appliedai.com/chatbot-benefits/
https://blog.growthbot.org/chatbots-were-the-next-big-thing-what-happened

Attention in Response Generation (i.e., Chatbots)

Chatbots can be defined along at least two dimensions, *core algorithm* and context handling:

- Generative: encode the question into a context vector and generate the answer word by word using conditioned probability distribution over answer's vocabulary. E.g., an encoder-decoder model.
- Retrieval: rely on knowledge base of question-answer pairs. When a new question comes in, inference phase encodes it in a context vector and by using similarity measure retrieves the top-k neighbor knowledge base items.



algorithm that implements the dialog

what it taken into account when we implement the dialog

(related to the Decoder)
CORE ALGORITHM:
There are 2 ways to implement a chatbot

it can answer questions it never saw just by understanding the meaning of the question. However, sometimes the answer is not properly structured.

we don't generate text, we only do classification (by similarity search (which we do through the embedding trick)). Problem: it only generates answers that are in the knowledge space.

The generative is the most advanced, however the training is very demanding (needs a lot of data)

Attention in Response Generation (i.e., Chatbots)

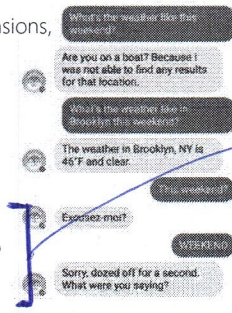
Chatbots can be defined along at least two dimensions, core algorithm and *context handling*:

- Single-turn: build the input vector by considering the incoming question. They may lose important information about the history of the conversation and generate irrelevant responses.

$$\{(q_i, a_i)\}$$

- Multi-turn: the input vector is built by considering a multi-turn conversational context, containing also incoming question.

$$\{([q_{i-2}; a_{i-2}; q_{i-1}; a_{i-1}; q_i], a_i)\}$$



The training is made by pairs (question, answear)

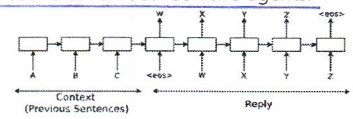
By knowing also the previous answears we can avoid the chatbot to repeat itself.

Here the training it more complex

Generative Chatbots

Vinyals and Le, 2015 and Shang et al., 2015 proposed to directly apply sequence to sequence models to the conversation between two agents:

- The first person utters "ABC"
- The second person replies "WXYZ"



Generative chatbots use an RNN and train it to map "ABC" to "WXYZ":

- We can borrow the model from machine translation
- A flat model simple and general
- Attention mechanisms apply as usual

We interpret the answear as if it is the translation of the question.

To handle also the situation of multi-turn we introduce:

Generative Hierarchical Chatbots

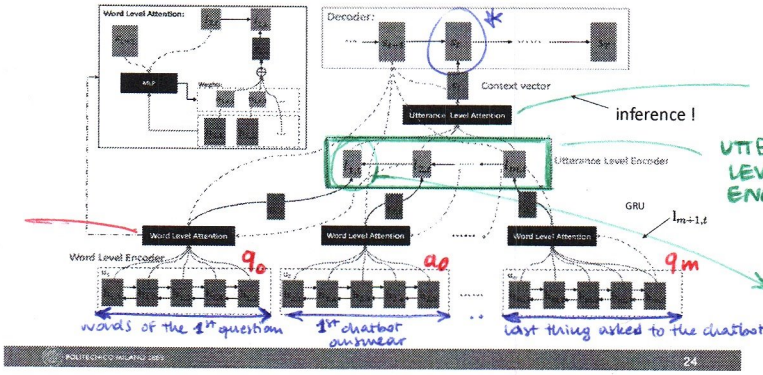
The idea could be concatenating multiple turns into a single long input sequence, but this probably results in poor performances.

- LSTM cells often fail to catch the long term dependencies within input sequences that are longer than 100 tokens (1 token is 1 "invio")
- No explicit representation of turns can be exploited by the attention mechanism

Xing et al., in 2017, extended attention mechanism from single-turn response generation to a hierarchical attention mechanism

- Hierarchical attention networks (e.g., characters -> words -> sentences)
- Generate hidden representation of a sequence from contextualized words

Hierarchical Generative Multi-turn Chatbots



UTTERANCE LEVEL ATTENTION

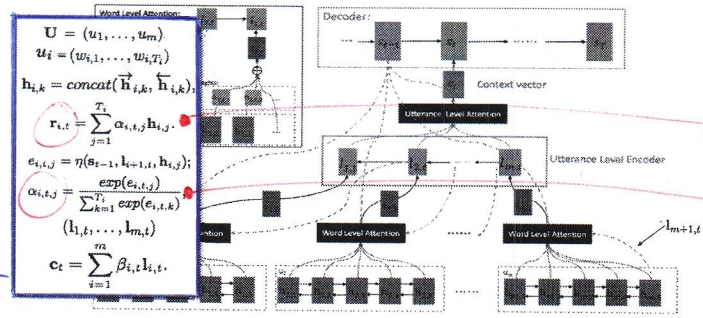
inference !

UTTERANCE LEVEL ENCODER

WORD LEVEL ATTENTION

encoding of the 1st question

Hierarchical Generative Multi-turn Chatbots



$$\mathbf{U} = (u_1, \ldots, u_m)$$
$$u_i = (w_{i,1}, \ldots, w_{i,T_i})$$
$$\mathbf{h}_{i,k} = concat(\overrightarrow{\mathbf{h}}_{i,k}, \overleftarrow{\mathbf{h}}_{i,k});$$
$$\mathbf{r}_{i,t} = \sum_{j=1}^{T_i} \alpha_{i,t,j} \mathbf{h}_{i,j}.$$
$$e_{i,t,j} = \eta(\mathbf{s}_{t-1}, \mathbf{l}_{i-1,t}, \mathbf{h}_{i,j});$$
$$\alpha_{i,t,j} = \frac{exp(e_{i,t,j})}{\sum_{k=1}^{T_i} exp(e_{i,t,k})}$$
$$(\mathbf{l}_{1,t}, \ldots, \mathbf{l}_{m,t})$$
$$\mathbf{c}_t = \sum_{i=1}^{m} \beta_{i,t} \mathbf{l}_{i,t}.$$

context at the level of the word

softmax to predict the weights

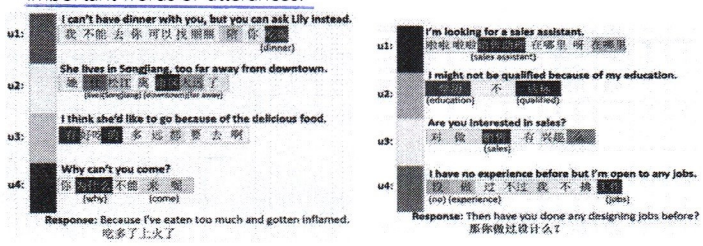
In conclusion every state * consider the previous state and the context vector. The context vector puts weights on the encodings of the question/answers. Each encoding as well puts weights on the words of the specific question/answer. We have an attention among the questions/answers and an attention among the words of each question/answear. (That's why HIERARCHICAL)

mathematical description of how to implement the hierarchical gen —.

Hierarchical Generative Multi-turn Chatbots

We can visualize hierarchical attention weights, darker color means more important words or utterances.

This gives an explanation on why the model choose an answer. (This can be inspected when we look for what went wrong)

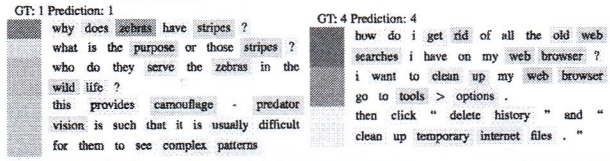


Hierarchical Document Classification

Hierarchical attention networks have been used for topic classification (e.g., Yahoo Answer data set).

- Left document denotes Science and Mathematics; model accurately localizes the words *zebra*, *stripes*, *camouflage*, *predator* and corresponding sentences.
- Right document denotes Computers and Internet; the model focuses on *web*, *searches*, *browsers* and their corresponding sentences.

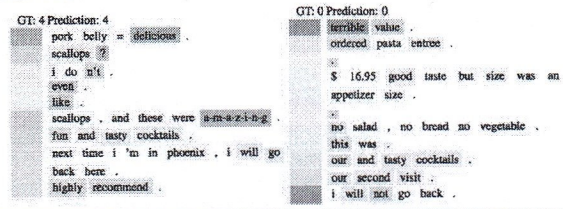
means that we have to find an embedding of the document that is close to the topic.
(We want to apply a hierarchical version since we look for paragraphs, then sentences and lastly for words)



We can embed a document by selecting relevant words from a sentence and weight each paragraph based on the relevance.
(The weights of the words depend also on the weights of the paragraphs they belong to)

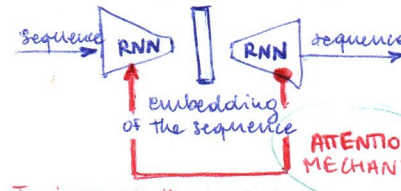
Hierarchical Document Classification

In Sentiment Analysis, the model can select words carrying strong sentiment like *delicious*, *amazing*, *terrible* and corresponding sentences.
Sentences containing useless words like cocktails, pasta, entree are disregarded.



Attention is all you need!

Standard Seq2Seq: it takes a sequence as input and it has a sequence as output:



To improve the quality we include an attention that looks at the sequence of states in RNN.
This process can be done at multiple levels: we can do it at the level of a sentence, at the level of the utterance or at the level of the document.

Attention is all you need!

Having seen attention is what makes things working you start wondering:

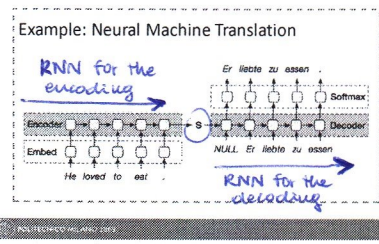
- Sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples.
- Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks. Can we base solely on attention mechanisms, dispensing with recurrence and convolutions entirely?
- !! Without recurrence, nor convolution, in order for the model to make use of the order of the sequence, we must **inject** some information about the relative or absolute position of the tokens in the sequence.

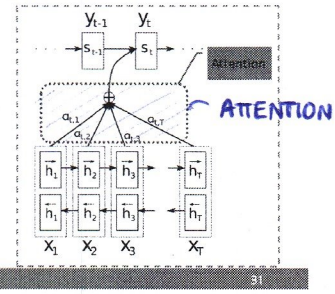
Can we remove the recurrent part and use only convolutions?
TRANSFORMERS don't use recurrency

Critical when we process complex sequences

Current State of the Art

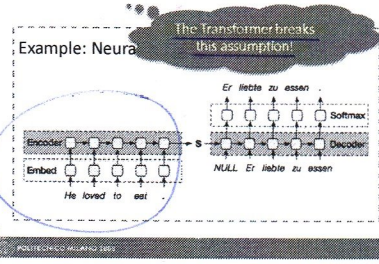
There has been a running joke in the NLP community that **an LSTM with attention** will yield state-of-the-art performance on any task.
Attention is built upon RNN ...

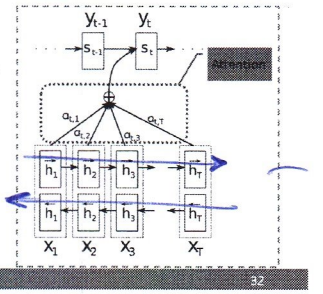




Current State of the Art

There has been a running joke in the NLP community that **an LSTM with attention** will yield state-of-the-art performance on any task.
Attention is built upon RNN ...





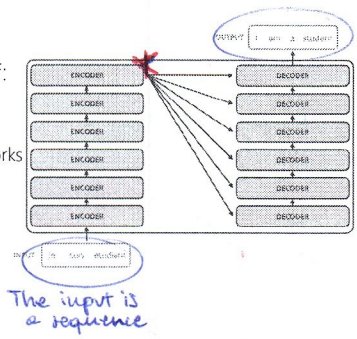
The transformer wants to break the assumption of attention being built upon RNN. The transformer wants to remove the idea of using recurrence to encode this part. *

* Why we want to remove the recursion? Because we have to pass through the entire input twice before we can start with the attention (this complicates the training).

Transformer

A Transformer model is make out of:

- Scaled Dot-Product Attention
- Multi-Head Attention
- Position-wise Feed-Forward Networks
- Embeddings and Softmax
- Positional Encoding



The input is a sequence

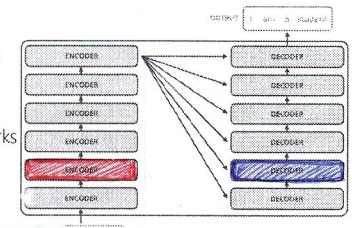
The **transformer** takes the input sequence and it encodes this sequence with a stack of layers which are performing the encoding. Then it does the same (starting from the encoding at *): it decodes the encoding into the output.

⇒ instead of having a recurrent encoding it has a non recurrent encoder and then a non recurrent decoder

Transformer

A Transformer model is make out of:

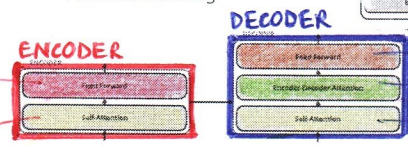
- Scaled Dot-Product Attention
- Multi-Head Attention
- Position-wise Feed-Forward Networks
- Embeddings and Softmax
- Positional Encoding



Feedforward because we don't want convolutions or recurrency

Feedforward model based on the input

Self-attention mechanism



Feedforward neural net

Encoder to Decoder attention (= attention of the input based on the encoding)

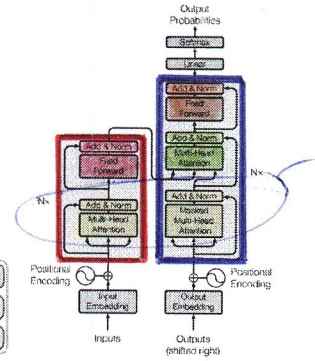
Self-attention

Transformer

These 2 blocks (self attention) are the ones that makes it possible to remove the convolutions and the recurrencies

A Transformer model is made out of:

- Scaled Dot-Product Attention
- Multi-Head Attention
- Position-wise Feed-Forward Networks
- Embeddings and Softmax
- Positional Encoding



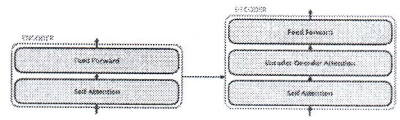


Figure 1: The Transformer - model architecture.

How does the **SELF-ATTENTION** work? We give some attention to a word based on the other words that are in the same input. We have a sequence of **n** words → each word will have an amount of attention based on **n-1** words.

How can we decide the attention on one word based on all the other?

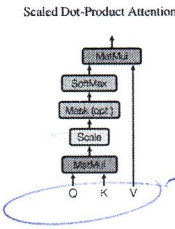
Transformer

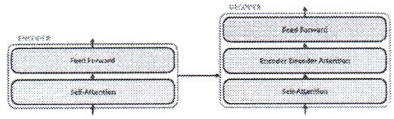
A Transformer model is made out of:

- Scaled Dot-Product Attention
- Multi-Head Attention
- Position-wise Feed-Forward Networks
- Embeddings and Softmax
- Positional Encoding

Attention mechanism based on the dot product

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$





What are Q, K, V?

Q is called "query". Suppose that we have the word: "thinking" which has its embedding (*). We associate to this specific word 3 vectors: one is the **QUERY** vector. The query is a smaller-size encoding of the word. (*). Then we have another vector: K, which is "**KEY**". The key is the representation of the word as a key in the dataset of the transformer. (*). Then we have another vector: V, which is the "**VALUE**" of the word. (*).

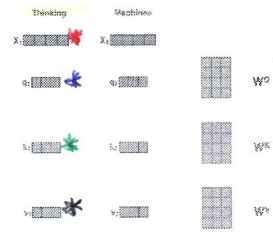
Transformer

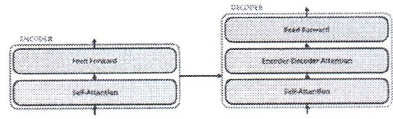
Based on what are Q, V, K we make attention as: we multiply the query and the key to know how much they're similar, we scale the result and we put it in the softmax. The ultimate attention for the word it the output of the softmax weighted by the value of the word.

A Transformer model is made out of:

- Scaled Dot-Product Attention
- Multi-Head Attention
- Position-wise Feed-Forward Networks
- Embeddings and Softmax
- Positional Encoding

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$





QK^T is to say: "how much is important the word when it appears with the other words"

At the end, the weighted sum of all the values (weighted by the attention) in the sentence becomes the encoding of the sentence.

We can say that the embedding of a word becomes the weighted sum of the values of the other words, weighted by these self-attention.

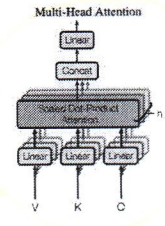
Transformer

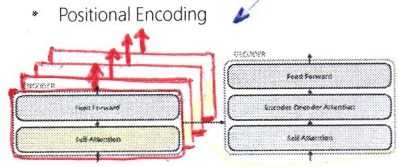
A Transformer model is made out of:

- Scaled Dot-Product Attention
- Multi-Head Attention
- Position-wise Feed-Forward Networks
- Embeddings and Softmax
- Positional Encoding

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Sometimes we want to have different encodings and so we put in parallel the encoders (since we initialize them randomly we obtain different embeddings)





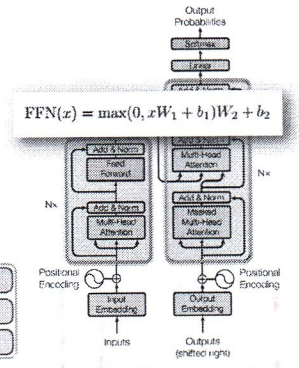
Transformer

A Transformer model is made out of:

- Scaled Dot-Product Attention
- Multi-Head Attention
- Position-wise Feed-Forward Networks
- Embeddings and Softmax
- Positional Encoding

for each word it computes a nonlinear function

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$



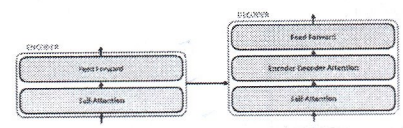
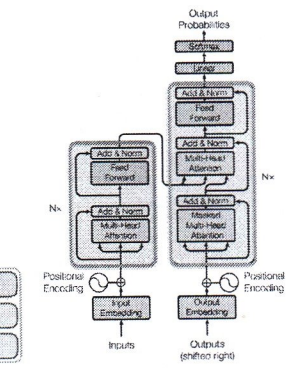


Figure 1: The Transformer - model architecture.

Transformer

A Transformer model is made out of:

- Scaled Dot-Product Attention
- Multi-Head Attention
- Position-wise Feed-Forward Networks
- Embeddings and Softmax
- Positional Encoding



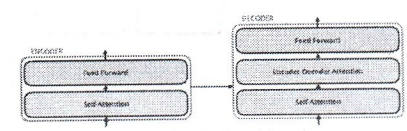


Figure 1: The Transformer - model architecture.

Transformer

A Transformer model is made out of:

- Scaled Dot-Product Attention
- Multi-Head Attention
- Position-wise Feed-Forward Networks
- Embeddings and Softmax
- Positional Encoding

After we shuffle words in an input sentence we have nothing that keeps track of the positions (original) ⟹ we introduce the positional encoding

Reason: no RNN to model the sequence position

Two types:

- learned positional embeddings (arXiv:1705.03122v2)
- Sinusoid:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

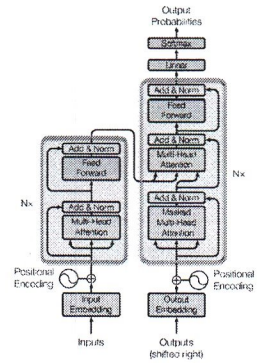


Figure 1: The Transformer - model architecture.

Transformer Complexity

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Observations:

- Self-Attention has O(1) maximum path length (capture long range dependency easily)
- When n<d, Self-Attention has lower complexity per layer

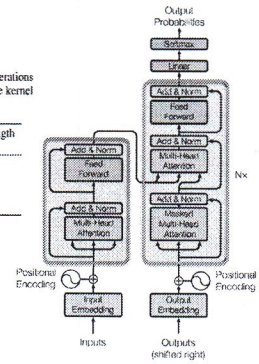


Figure 1: The Transformer - model architecture.

Transformer Performance

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [17]	23.75			
Deep-Att + PosUnk [37]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [36]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [31]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [37]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [36]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

- Eng-to-De: new state-of-the-art
- Eng-to-Fr: new single-model state-of-the-art
- Less training cost

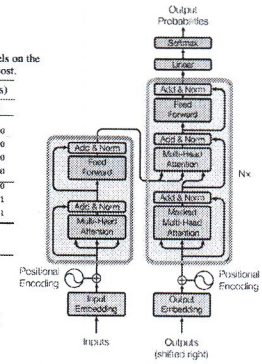


Figure 1: The Transformer - model architecture.

Transformer Performance

- source: Aber ich habe es nicht hingekriegt
- expected: But I didn't handle it
- got: But I didn't <UNK> it

- source: Wir könnten zum Mars fliegen wenn wir wolle
- expected: We could go to Mars if we want
- got: We could fly to Mars when we want

- source: Dies ist nicht meine Meinung Das sind Fakten
- expected: This is not my opinion These are the facts
- got: This is not my opinion These are facts

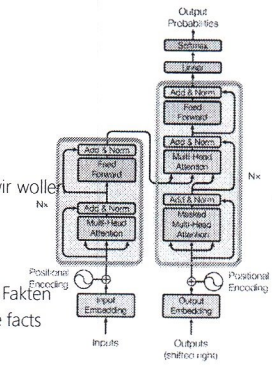


Figure 1: The Transformer - model architecture.

Acknowledgements

These slides are highly based on material taken from the following websites/blogs:

- https://www.analyticsvidhya.com/blog/2017/12/introduction-to-recurrent-neural-networks/
- https://medium.com/@Aj.Cheng/seq2seq-18a0730d1d77
- https://distill.pub/2016/augmented-rnns/
- http://jalammar.github.io/illustrated-transformer/

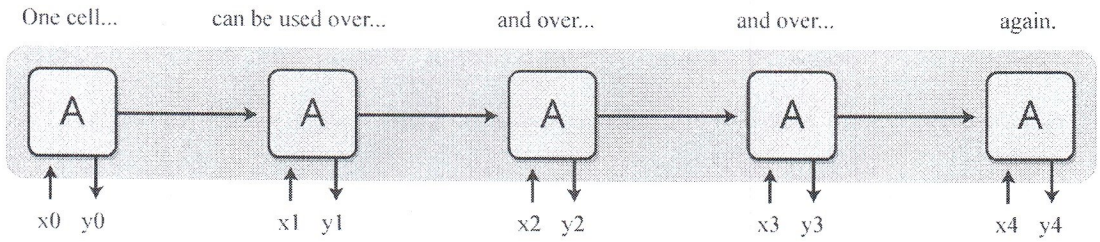
Amazing images, and part of content, about attention mechanisms from

Olah & Carter, "Attention and Augmented Recurrent Neural Networks", Distill, 2016. http://doi.org/10.23915/distill.00001

Attention and Augmented Recurrent Neural Networks

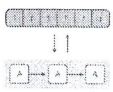
CHRIS OLAH Google Brain
SHAN CARTER Google Brain
Sept. 8 2016
Citation: Olah & Carter, 2016

Recurrent neural networks are one of the staples of deep learning, allowing neural networks to work with sequences of data like text, audio and video. They can be used to boil a sequence down into a high-level understanding, to annotate sequences, and even to generate new sequences from scratch!

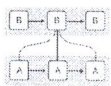


The basic RNN design struggles with longer sequences, but a special variant—"long short-term memory" networks [1]—can even work with these. Such models have been found to be very powerful, achieving remarkable results in many tasks including translation, voice recognition, and image captioning. As a result, recurrent neural networks have become very widespread in the last few years.

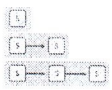
As this has happened, we've seen a growing number of attempts to augment RNNs with new properties. Four directions stand out as particularly exciting:



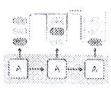
Neural Turing Machines have external memory that they can read and write to.



Attentional Interfaces allow RNNs to focus on parts of their input.



Adaptive Computation Time allows for varying amounts of computation per step.



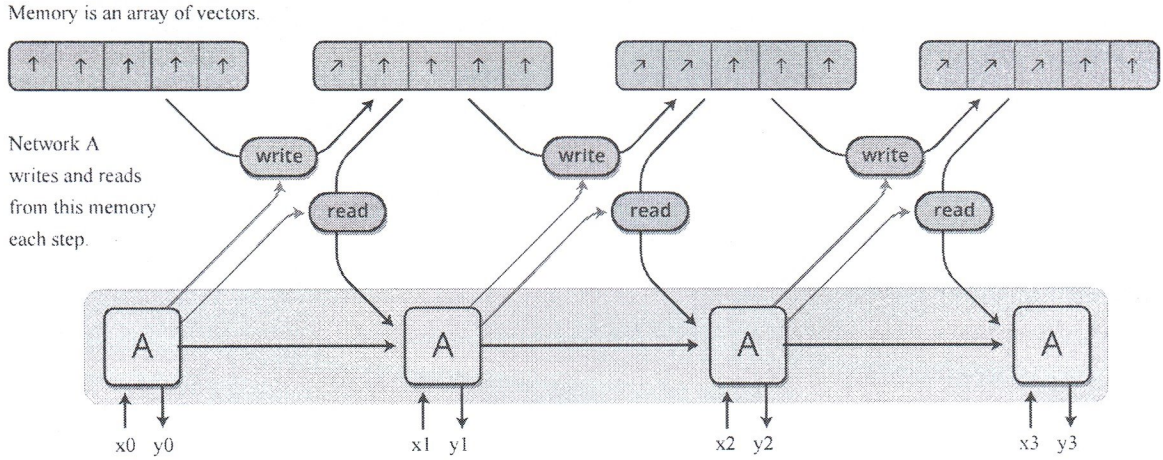
Neural Programmers can call functions, building programs as they run.

Individually, these techniques are all potent extensions of RNNs, but the really striking thing is that they can be combined, and seem to just be points in a broader space. Further, they all rely on the same underlying trick—something called attention—to work.

Our guess is that these "augmented RNNs" will have an important role to play in extending deep learning's capabilities over the coming years.

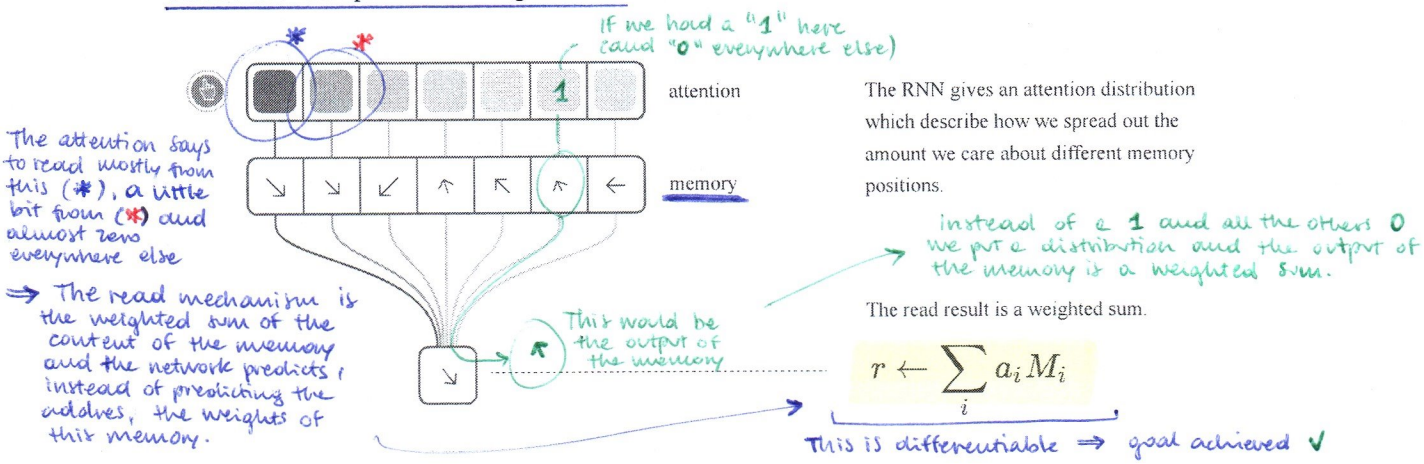
Neural Turing Machines

Neural Turing Machines [2] combine a RNN with an external memory bank. Since vectors are the natural language of neural networks, the memory is an array of vectors:

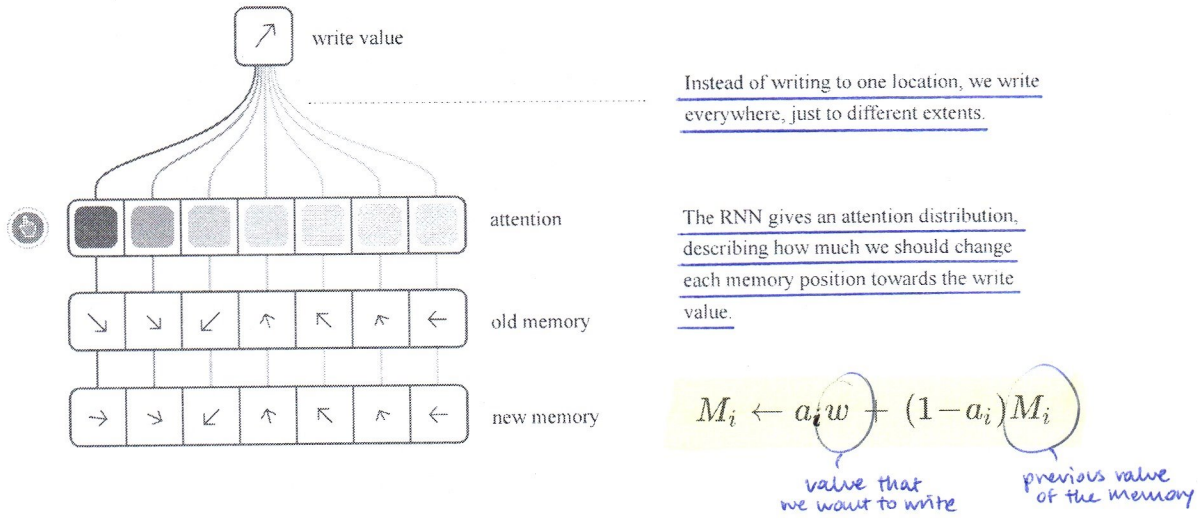


But how does reading and writing work? The challenge is that we want to make them differentiable. In particular, we want to make them differentiable with respect to the location we read from or write to, so that we can learn where to read and write. This is tricky because memory addresses seem to be fundamentally discrete. NTMs take a very clever solution to this: every step, they read and write everywhere, just to different extents.

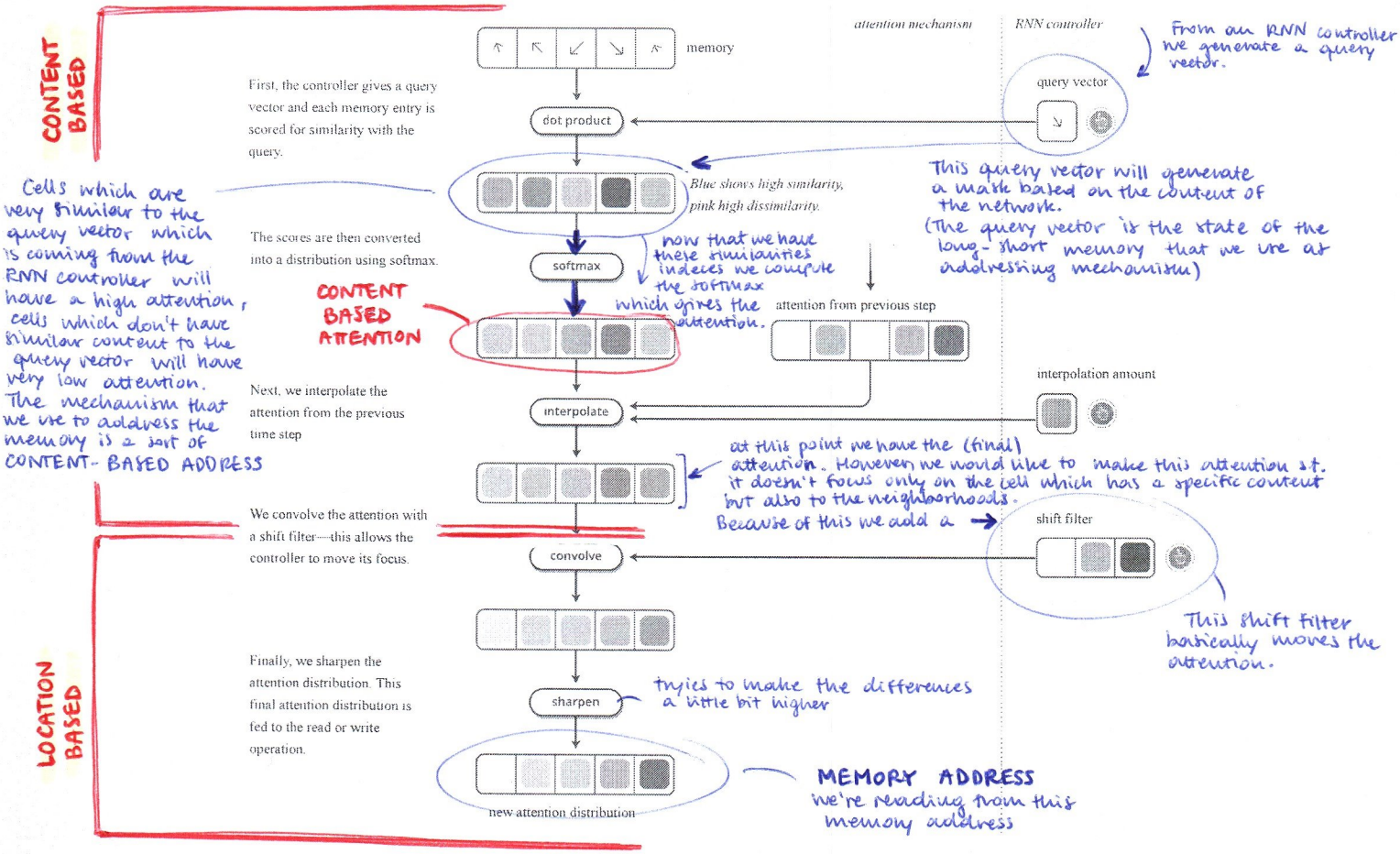
As an example, let's focus on reading. Instead of specifying a single location, the RNN outputs an "attention distribution" that describes how we spread out the amount we care about different memory positions. As such, the result of the read operation is a weighted sum.



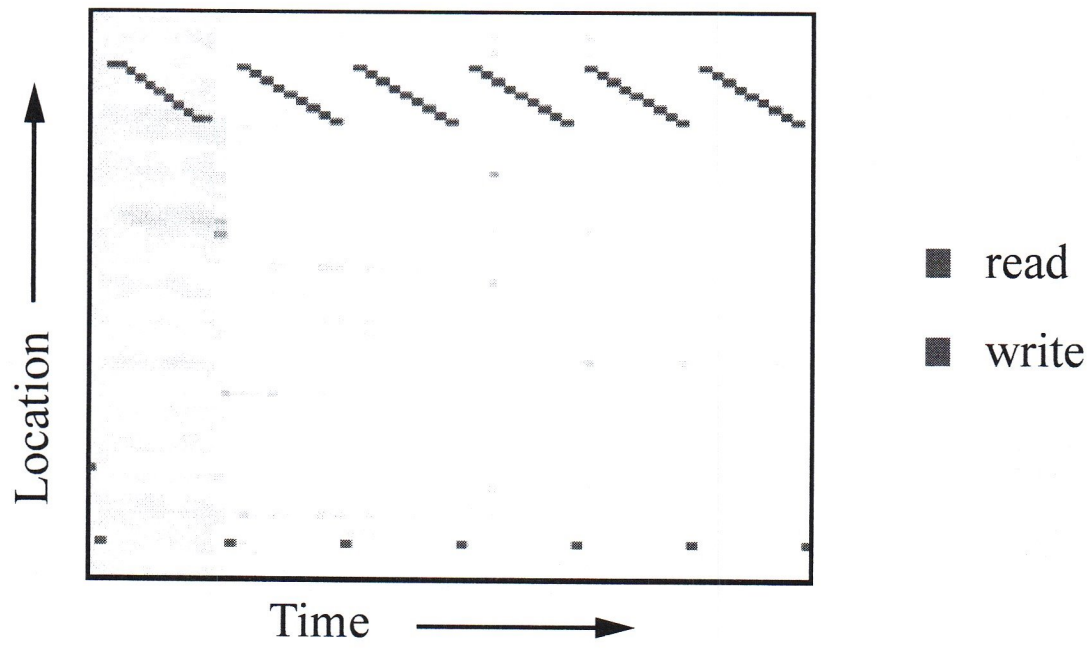
Similarly, we write everywhere at once to different extents. Again, an attention distribution describes how much we write at every location. We do this by having the new value of a position in memory be a convex combination of the old memory content and the write value, with the position between the two decided by the attention weight.



But how do NTMs decide which positions in memory to focus their attention on? They actually use a combination of two different methods: content-based attention and location-based attention. Content-based attention allows NTMs to search through their memory and focus on places that match what they're looking for, while location-based attention allows relative movement in memory, enabling the NTM to loop.



This capability to read and write allows NTMs to perform many simple algorithms, previously beyond neural networks. For example, they can learn to store a long sequence in memory, and then loop over it, repeating it back repeatedly. As they do this, we can watch where they read and write, to better understand what they're doing:



See more experiments in [3]. This figure is based on the Repeat Copy experiment.

They can also learn to mimic a lookup table, or even learn to sort numbers (although they kind of cheat)! On the other hand, they still can't do many basic things, like add or multiply numbers.

Since the original NTM paper, there have been a number of exciting papers exploring similar directions. The Neural GPU [4] overcomes the NTM's inability to add and multiply numbers. Zaremba & Sutskever [5] train NTMs using reinforcement learning instead of the differentiable read/writes used by the original. Neural Random Access Machines [6] work based on pointers. Some papers have explored differentiable data structures, like stacks and queues [7, 8]. And memory networks [9, 10] are another approach to attacking similar problems.

In some objective sense, many of the tasks these models can perform—such as learning how to add numbers—aren't that objectively hard. The traditional program synthesis community would eat them for lunch. But neural networks are capable of many other things, and models like the Neural Turing Machine seem to have knocked away a very profound limit on their abilities.

Code

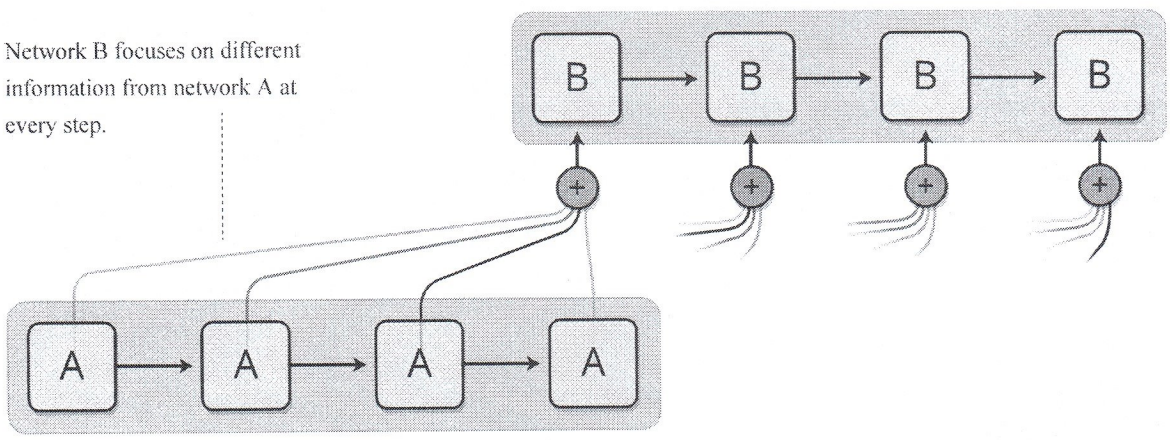
There are a number of open source implementations of these models. Open source implementations of the Neural Turing Machine include Taehoon Kim's (TensorFlow), Shawn Tan's (Theano), Fumin's (Go), Kai Sheng Tai's (Torch), and Snip's (Lasagne). Code for the Neural GPU publication was open sourced and put in the TensorFlow Models repository. Open source implementations of Memory Networks include Facebook's (Torch/Matlab), YerevaNN's (Theano), and Taehoon Kim's (TensorFlow).

Attentional Interfaces

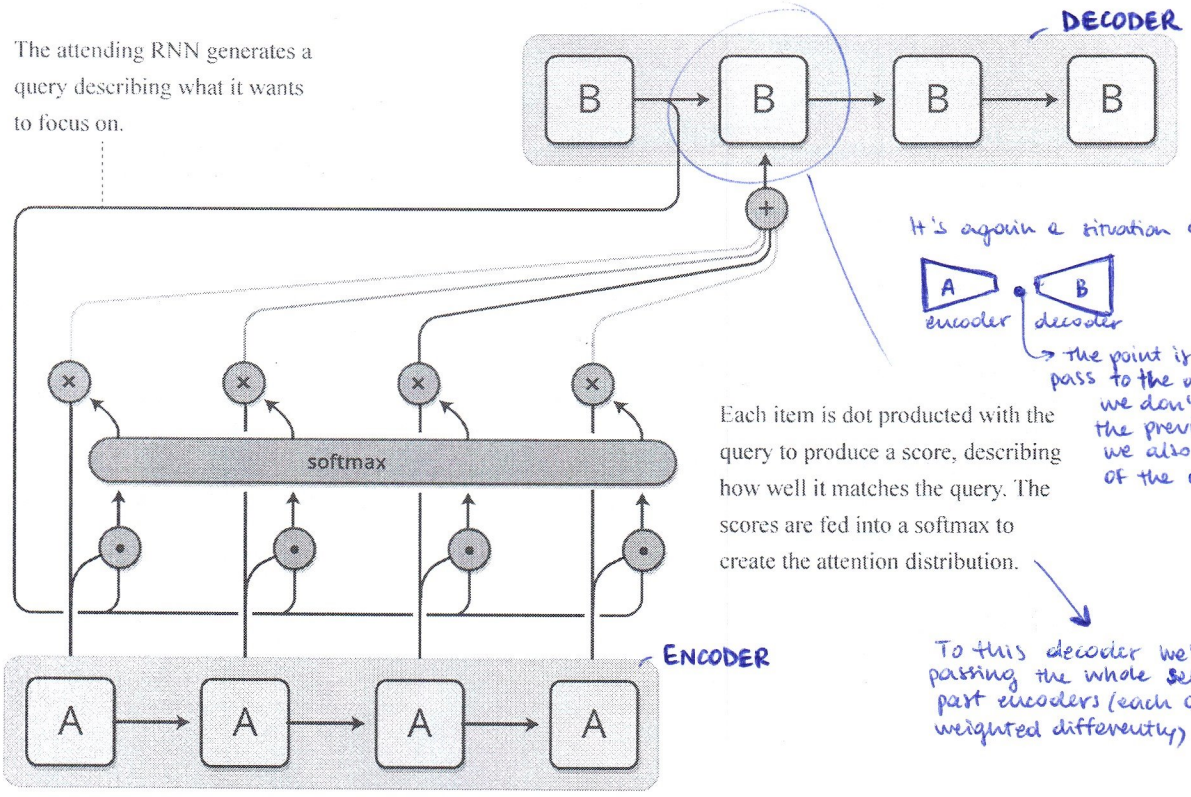
When I'm translating a sentence, I pay special attention to the word I'm presently translating. When I'm transcribing an audio recording, I listen carefully to the segment I'm actively writing down. And if you ask me to describe the room I'm sitting in, I'll glance around at the objects I'm describing as I do so.

Neural networks can achieve this same behavior using *attention*, focusing on part of a subset of the information they're given. For example, an RNN can attend over the output of another RNN. At every time step, it focuses on different positions in the other RNN.

We'd like attention to be differentiable, so that we can learn where to focus. To do this, we use the same trick Neural Turing Machines use: we focus everywhere, just to different extents.



The attention distribution is usually generated with content-based attention. The attending RNN generates a query describing what it wants to focus on. Each item is dot-producted with the query to produce a score, describing how well it matches the query. The scores are fed into a softmax to create the attention distribution.



One use of attention between RNNs is translation [11]. A traditional sequence-to-sequence model has to boil the entire input down into a single vector and then expands it back out. Attention avoids this by allowing the RNN processing the input to pass along information about each word it sees, and then for the RNN generating the output to focus on words as they become relevant.

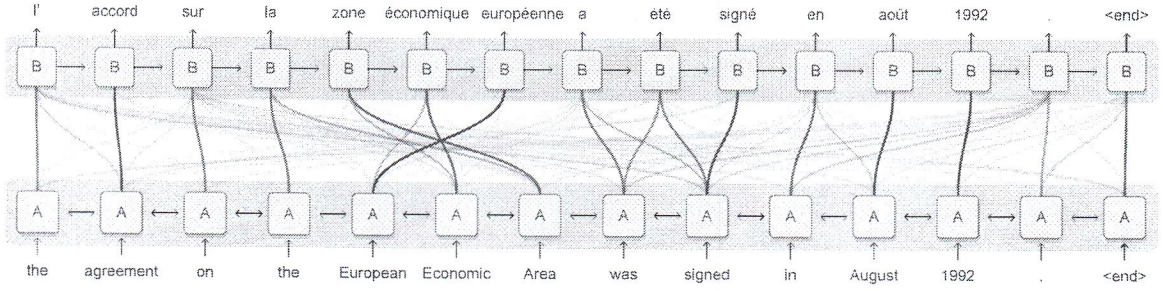


Diagram derived from Fig. 3 of Bahdanau, *et al.* 2014

This kind of attention between RNNs has a number of other applications. It can be used in voice recognition [12], allowing one RNN to process the audio and then have another RNN skim over it, focusing on relevant parts as it generates a transcript.

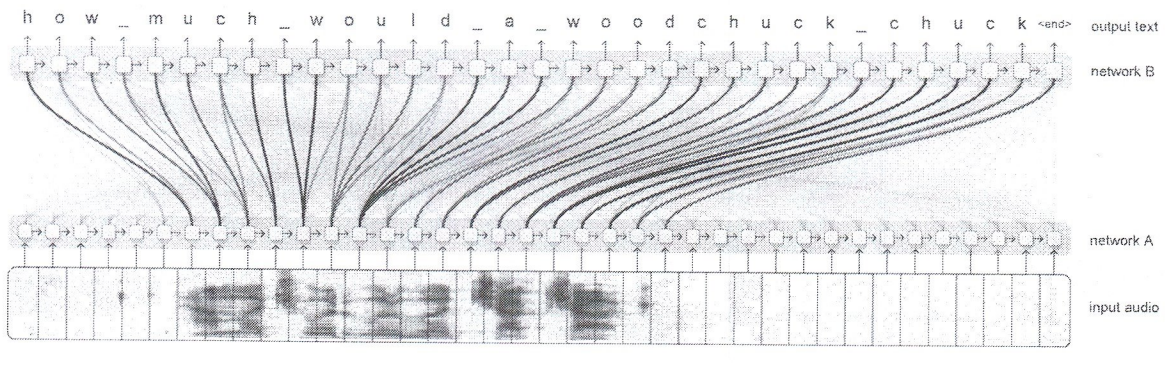


Figure derived from Chan, *et al.* 2015

Other uses of this kind of attention include parsing text [13], where it allows the model to glance at words as it generates the parse tree, and for conversational modeling [14], where it lets the model focus on previous parts of the conversation as it generates its response.

Attention can also be used on the interface between a convolutional neural network and an RNN. This allows the RNN to look at different position of an image every step. One popular use of this kind of attention is for image captioning. First, a conv net processes the image, extracting high-level features. Then an RNN runs, generating a description of the image. As it generates each word in the description, the RNN focuses on the conv net's interpretation of the relevant parts of the image. We can explicitly visualize this:

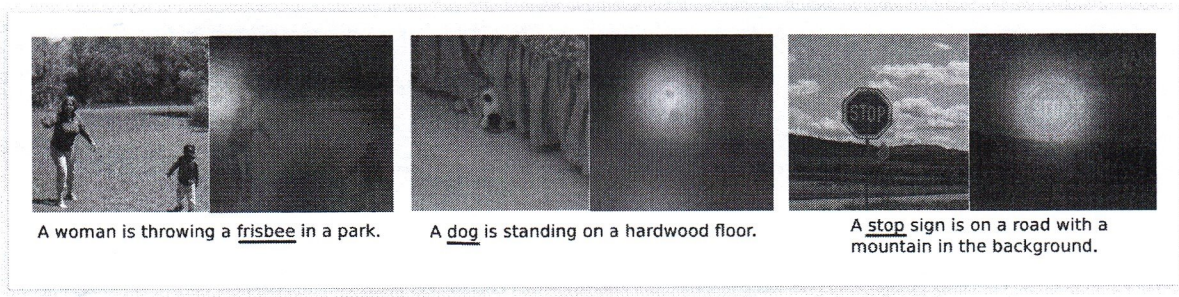


Figure from [3]

More broadly, attentional interfaces can be used whenever one wants to interface with a neural network that has a repeating structure in its output.

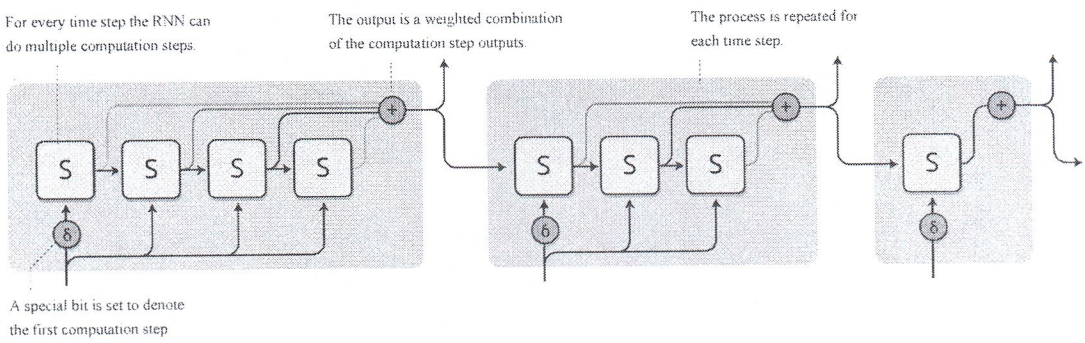
Attentional interfaces have been found to be an extremely general and powerful technique, and are becoming increasingly widespread.

Adaptive Computation Time

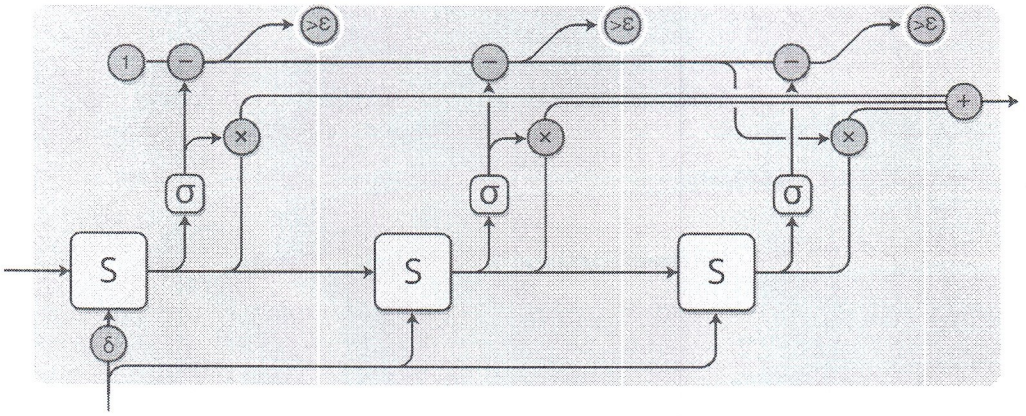
Standard RNNs do the same amount of computation for each time step. This seems unintuitive. Surely, one should think more when things are hard? It also limits RNNs to doing $O(n)$ operations for a list of length n.

Adaptive Computation Time [15] is a way for RNNs to do different amounts of computation each step. The big picture idea is simple: allow the RNN to do multiple steps of computation for each time step.

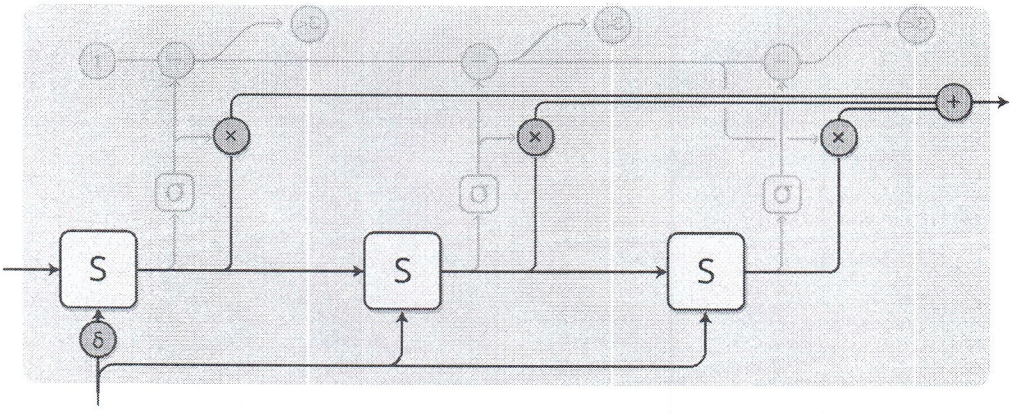
In order for the network to learn how many steps to do, we want the number of steps to be differentiable. We achieve this with the same trick we used before: instead of deciding to run for a discrete number of steps, we have an attention distribution over the number of steps to run. The output is a weighted combination of the outputs of each step.



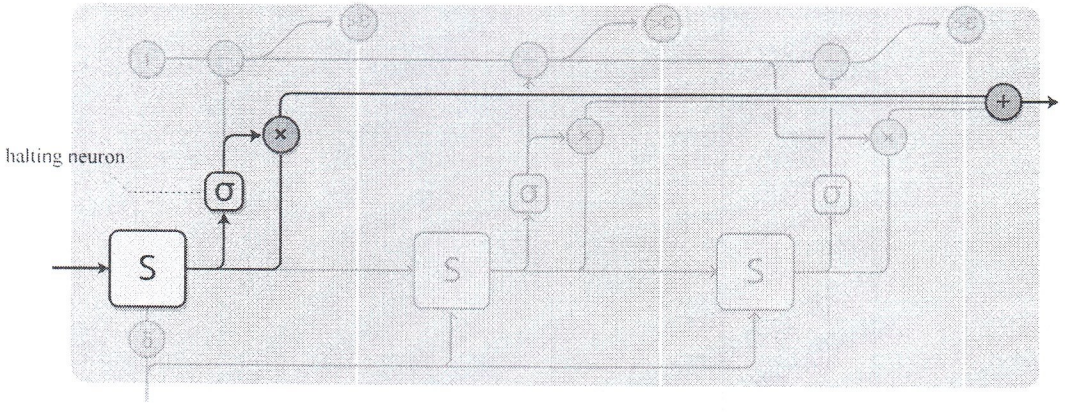
There are a few more details, which were left out in the previous diagram. Here's a complete diagram of a time step with three computation steps.



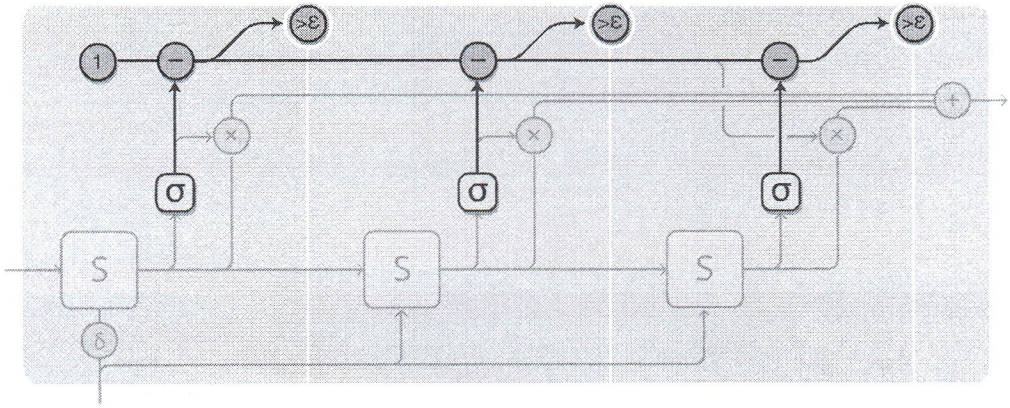
That's a bit complicated, so let's work through it step by step. At a high-level, we're still running the RNN and outputting a weighted combination of the states:



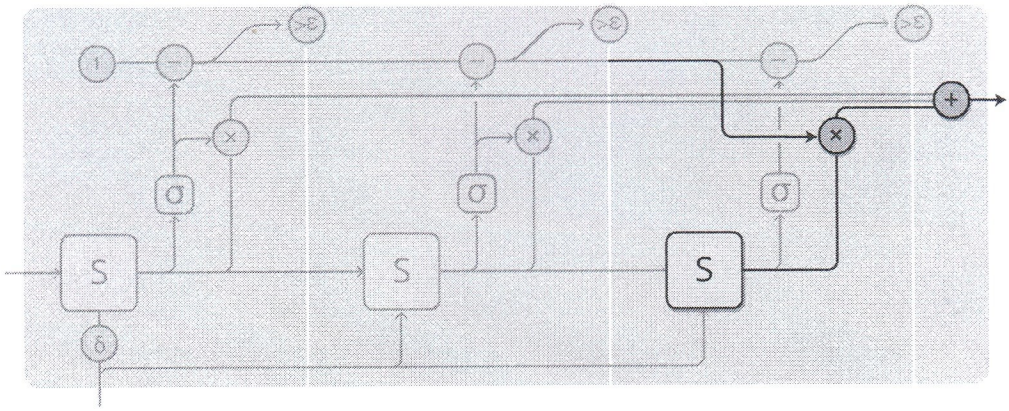
The weight for each step is determined by a "halting neuron." It's a sigmoid neuron that looks at the RNN state and gives a halting weight, which we can think of as the probability that we should stop at that step.



We have a total budget for the halting weights of 1, so we track that budget along the top. When it gets to less than epsilon, we stop.



When we stop, might have some left over halting budget because we stop when it gets to less than epsilon. What should we do with it? Technically, it's being given to future steps but we don't want to compute those, so we attribute it to the last step.



When training Adaptive Computation Time models, one adds a "ponder cost" term to the cost function. This penalizes the model for the amount of computation it uses. The bigger you make this term, the more it will trade-off performance for lowering compute time.

Adaptive Computation Time is a very new idea, but we believe that it, along with similar ideas, will be very important.

Code

The only open source implementation of Adaptive Computation Time at the moment seems to be Mark Neumann's (TensorFlow).

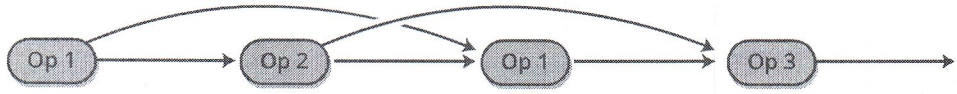
Neural Programmer

Neural nets are excellent at many tasks, but they also struggle to do some basic things like arithmetic, which are trivial in normal approaches to computing. It would be really nice to have a way to fuse neural nets with normal programming, and get the best of both worlds.

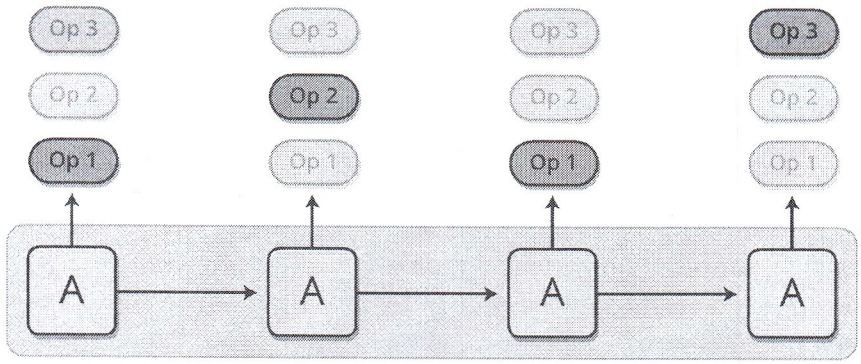
The neural programmer [16] is one approach to this. It learns to create programs in order to solve a task. In fact, it learns to generate such programs *without needing examples of correct programs*. It discovers how to produce programs as a means to the end of accomplishing some task.

The actual model in the paper answers questions about tables by generating SQL-like programs to query the table. However, there are a number of details here that make it a bit complicated, so let's start by imagining a slightly simpler model, which is given an arithmetic expression and generates a program to evaluate it.

The generated program is a sequence of operations. Each operation is defined to operate on the output of past operations. So an operation might be something like "add the output of the operation 2 steps ago and the output of the operation 1 step ago." It's more like a Unix pipe than a program with variables being assigned to and read from.

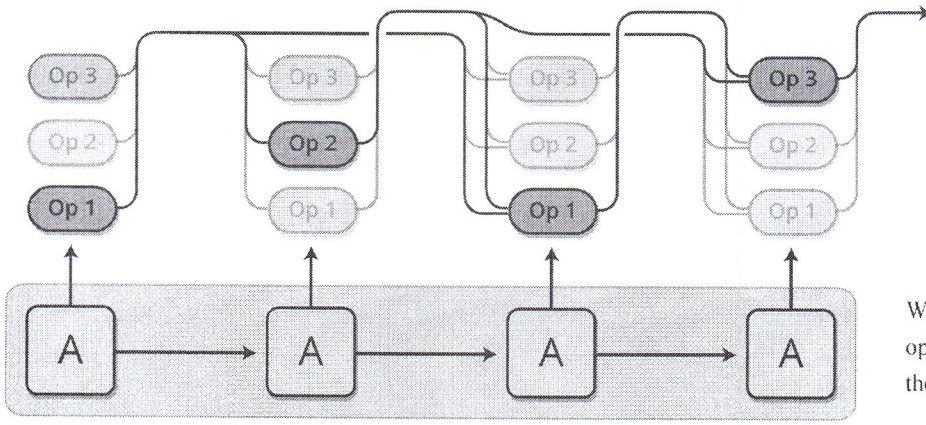


The program is generated one operation at a time by a controller RNN. At each step, the controller RNN outputs a probability distribution for what the next operation should be. For example, we might be pretty sure we want to perform addition at the first time step, then have a hard time deciding whether we should multiply or divide at the second step, and so on...



At each step the controller RNN outputs a probability distribution.

The resulting distribution over operations can now be evaluated. Instead of running a single operation at each step, we do the usual attention trick of running all of them and then average the outputs together, weighted by the probability we ran that operation.



We run all of the operations and average the outputs together.

As long as we can define derivatives through the operations, the program's output is differentiable with respect to the probabilities. We can then define a loss, and train the neural net to produce programs that give the correct answer. In this way, the Neural Programmer learns to produce programs without examples of good programs. The only supervision is the answer the program should produce.

That's the core idea of Neural Programmer, but the version in the paper answers questions about tables, rather than arithmetic expressions. There are a few additional neat tricks:

- **Multiple Types:** Many of the operations in the Neural Programmer deal with types other than scalar numbers. Some operations output selections of table columns or selections of cells. Only outputs of the same type get merged together.

- **Referencing Inputs:** The neural programmer needs to answer questions like "How many cities have a population greater than 1,000,000?" given a table of cities with a population column. To facilitate this, some operations allow the network to reference constants in the question they're answering, or the names of columns. This referencing happens by attention, in the style of pointer networks [17].

The Neural Programmer isn't the only approach to having neural networks generate programs. Another lovely approach is the Neural Programmer-Interpreter [18] which can accomplish a number of very interesting tasks, but requires supervision in the form of correct programs.

We think that this general space, of bridging the gap between more traditional programming and neural networks is extremely important. While the Neural Programmer is clearly not the final solution, we think there are a lot of important lessons to be learned from it.

Code

The more recent version of Neural Programmer for question answering has been open sourced by its authors and is available as a TensorFlow Model. There is also an implementation of the Neural Programmer-Interpreter by Ken Morishita (Keras).

The Big Picture

A human with a piece of paper is, in some sense, much smarter than a human without. A human with mathematical notation can solve problems they otherwise couldn't. Access to computers makes us capable of incredible feats that would otherwise be far beyond us.

In general, it seems like a lot of interesting forms of intelligence are an interaction between the creative heuristic intuition of humans and some more crisp and careful media, like language or equations. Sometimes, the medium is something that physically exists, and stores information for us, prevents us from making mistakes, or does computational heavy lifting. In other cases, the medium is a model in our head that we manipulate. Either way, it seems deeply fundamental to intelligence.

Recent results in machine learning have started to have this flavor, combining the intuition of neural networks with something else. One approach is what one might call "heuristic search." For example, AlphaGo [19] has a model of how Go works and explores how the game could play out guided by neural network intuition. Similarly, DeepMath [20] uses neural networks as intuition for manipulating mathematical expressions. The "augmented RNNs" we've talked about in this article are another approach, where we connect RNNs to engineered media, in order to extend their general capabilities.

Interacting with media naturally involves making a sequence of taking an action, observing, and taking more actions. This creates a major challenge: how do we learn which actions to take? That sounds like a reinforcement learning problem and we could certainly take that approach. But the reinforcement learning literature is really attacking the hardest version of this problem, and its solutions are hard to use. The wonderful thing about attention is that it gives us an easier way out of this problem by partially taking all actions to varying extents. This works because we can design media—like the NTM memory—to allow fractional actions and to be differentiable. Reinforcement learning has us take a single path, and try to learn from that. Attention takes every direction at a fork, and then merges the paths back together.

A major weaknesses of attention is that we have to take every "action" every step. This causes the computational cost to grow linearly as you do things like increase the amount of memory in a Neural Turing Machine. One thing you could imagine doing is having your attention be sparse, so that you only have to touch some memories. However, it's still challenging because you may want to do things like have your attention depend on the content of the memory, and doing that naively forces you to look at each memory. We've seen some initial attempts to attack this problem, such as [21], but it seems like there's a lot more to be done. If we could really make such sub-linear time attention work, that would be very powerful!

Augmented recurrent neural networks, and the underlying technique of attention, are incredibly exciting. We look forward to seeing what happens next!

POLITECNICO MILANO 1863

Artificial Neural Networks and Deep Learning

- Word Embedding-

How do we encode words

Matteo Matteucci, PhD (matteo.matteucci@polimi.it)
Artificial Intelligence and Robotics Laboratory
Politecnico di Milano

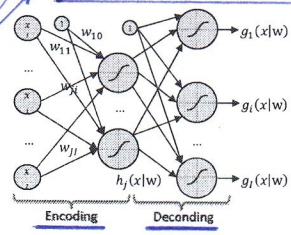
AIRLAB

AUTOENCODER:
self-supervised model where we try to map an input vector X into a hidden representation and then map this hidden representation "back", to X', where X' is the original vector X. Apparently, the autoencoder learns the identity function. However, it's a bit different:

Neural Autoencoder Recall

Network trained to output the input (i.e., to learn the identity function)

- Limited number of units in hidden layers (compressed representation)
- Constrain the representation to be sparse (sparse representation)



$$x \in \Re^I \xrightarrow{enc} h \in \Re^J \xrightarrow{dec} g \in \Re^I$$
$$J \ll I$$

$$E = \underbrace{\|g_i(x_i|w) - x_i\|^2}_{\text{Reconstruction error}} + \lambda \underbrace{\sum_j \left| h_j\left(\sum_i w_{ji}^{(1)} x_i\right)\right|}_{\text{Sparsity term}}$$

$$g_i(x_i|w) \sim x_i \qquad h_j(x_i|w) \sim 0$$

the internal representation of X is a compressed representation (it's a compression problem where we try to minimize the compression error)

not only we want to use less neurons to represent X, we want also most of them to be zero.

we try to push the inner representation into having a lot of zeros (regularization term) (in this way the vectors will probably be orthogonal, each vector contains informations that the others don't ⇒ maximize COMPRESSION)

Word Embedding Motivation

Natural language processing treats words as discrete atomic symbols

- 'cat' is encoded as Id537
- 'dog' is encoded as Id143
- ...

Audio Spectrogram
DENSE

Image pixels
DENSE

Word, context, or document vectors
SPARSE

One way is the one-hot-encoding: if "cat" is the second item in the dictionary ⇒ [0, 1, 0, ..., 0]. However if we encode words like this, we obtain very long encodings (typical dictionary are very large).

BAG OF WORDS:
we have a (long) vectors of 1/0 (or frequencies) that puts a 1 if a word appears.

Encoding Text is a Serious Thing

Performance of real-world applications (e.g., chatbot, document classifiers, information retrieval systems) depends on input encoding:

we represent a document via local features ("local" means close to the words)

Local representations

- N-grams — Language Model
- Bag-of-words
- 1-of-N coding

Continuous representations

- Latent Semantic Analysis
- Latent Dirichlet Allocation
- Distributed Representations

Determine $P(s = w_1, \ldots, w_k)$ in some domain of interest

$$P(s_k) = \prod_i^k P(w_i | w_1, \ldots, w_{i-1})$$

In traditional n-gram language models "the probability of a word depends only on the context of n−1 previous words"

$$\hat{P}(s_k) = \prod_i^k P(w_i | w_{i-n+1}, \ldots, w_{i-1})$$

Typical ML-smoothing learning process (e.g., Katz 1987):

- compute $\hat{P}(w_i | w_{i-n+1}, \ldots, w_{i-1}) = \frac{\#w_{i-n+1}, \ldots, w_{i-1}, w_i}{\#w_{i-n+1}, \ldots, w_{i-1}}$
- smooth to avoid zero probabilities

N-grams is about modelling the probability of one term given the previous terms.

evaluating $P(w_i | w_{1}, \ldots, w_{i-1})$ (considering any possible combo. of $w_1, \ldots, w_{i-1}$) is very difficult (computationally).
A possible solution is to put a limit on how many words consider (e.g. n words) ⇒ N-gram

N-gram Language Model: Curse of Dimensionality

Let's assume a 10-gram LM on a corpus of 100.000 unique words

- The model lives in a 10D hypercube where each dimension has 100.000 slots
- Model training ↔ assigning a probability to each of the 100.000^{10} slots
- *Probability mass vanishes* → more data is needed to fill the huge space
- The more data, the more unique words! → Is not going to work ...

In practice:

- Corpuses can have 10^6 unique words
- Contexts are typically limited to size 2 (trigram model), e.g., famous Katz (1987) smoothed trigram model
- With short context length a lot of information is not captured

N-gram Language Model: Word Similarity Ignorance

Problems:
- if we model each word separately we miss the property of synonimus or similarity

Let assume we observe the following similar sentences
- *Obama speaks to the media in Illinois*
- *The President addresses the press in Chicago*

With classic one-hot vector space representations

- speaks = [0 0 1 0 ... 0 0 0 0] } speaks ⊥ addresses
- addresses = [0 0 0 0 ... 0 0 1 0]
- obama = [0 0 0 0 ... 0 1 0 0] } obama ⊥ president
- president = [0 0 0 1 ... 0 0 0 0]
- illinois = [1 0 0 0 ... 0 0 0 0] } illinois ⊥ chicago
- chicago = [0 1 0 0 ... 0 0 0 0]

They seem to be unconnected since in the one-hot-encoding they're ⊥. One-hot-encoding (which is still the basis for the N-grams) loses these important connections. How can we solve this?

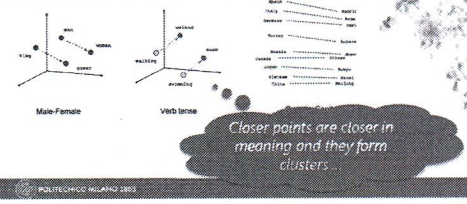
Word pairs share no similarity, and we need word similarity to generalize

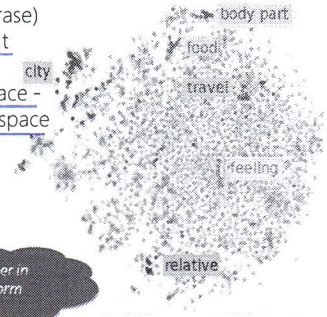
Idea:
We take these words in a sparse space of high dimensionality, transform this space in lower dense dimensionality space and see what happens.

⇒ EMBEDDING:
We take a vector in high dimension and we project it in lower dimension *

Embedding

Any technique mapping a word (or phrase) from it's original high-dimensional input space (the body of all words) to a lower-dimensional numerical vector space - so one *embeds* the word in a different space





Closer points are closer in meaning and they form clusters...

* This reminds a lot of the idea of the AUTOENCODER. In the autoencoder what we would like to do is to embed a word in such way that we can reconstruct it, here we want an embedding that preserves the fact that similar words should end-up close to each others.
⇒ CLOSEDNESS is a sort of similarity function

Word Embedding: Distributed Representation

Each unique word w in a vocabulary V (typically $\|V\| > 10^6$) is mapped to a continuous m-dimensional space (typically $100 < m < 500$)

$w \in V \xrightarrow{mapping\ C} \Re^m$

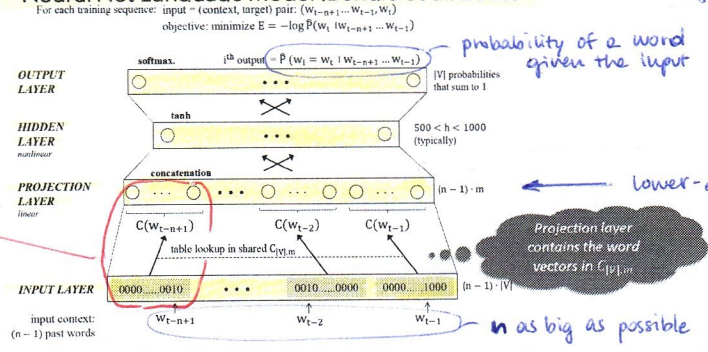
obama = [0 0 ... 0 1 0 ... 0 0] ⇒ obama = [0.12 ... -0.25]
(«one-hot» encoding: w_1 obama w_V; feature vector: f_1 f_m)

Fighting the curse of dimensionality with:
- Compression *(dimensionality reduction)*
- Smoothing *(discrete to continuous)*
- Densification *(sparse to dense)*

Similar words should end up to be close to each other in the feature space

Neural Net Language Model (Bengio et al. 2003)

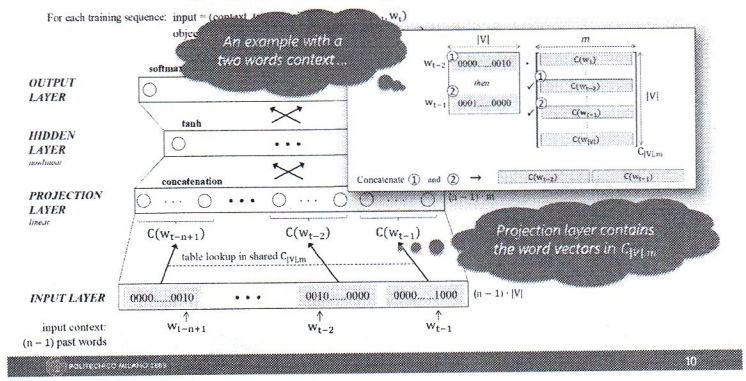
→ The goal was to solve the N-gram problem



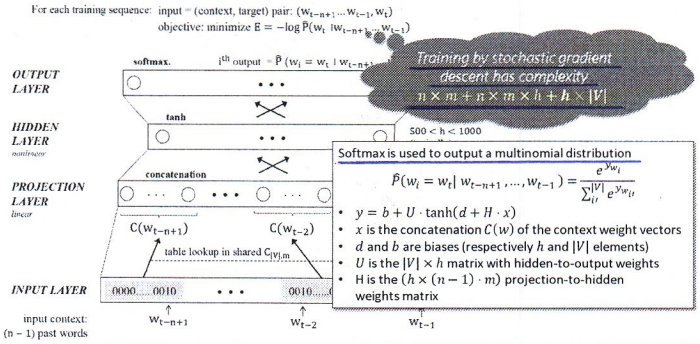
ENCODING MATRIX (C) which encodes each one-hot-word into a real-valued vector (this step is the PROJECTION)

After this we do a standard classifier which predict which is the most likely word given the input (using also a hidden layer).

Neural Net Language Model (Bengio et al. 2003)



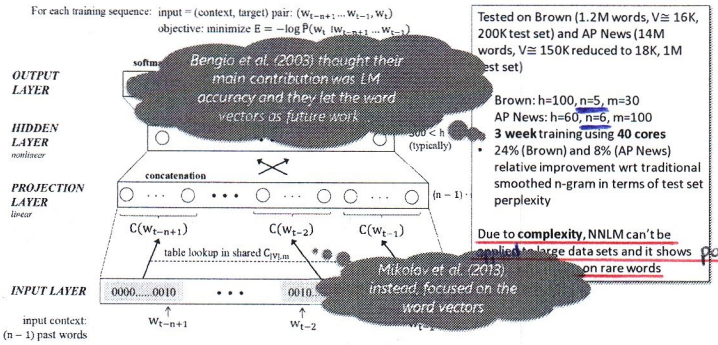
Neural Net Language Model (Bengio et al. 2003)



POLITECNICO MILANO 1863 11

$n \times m$ (encoding) + $n \times m \times h$ (hidden layer) + $h \times |V|$ (softmax)

Neural Net Language Model (Bengio et al. 2003)



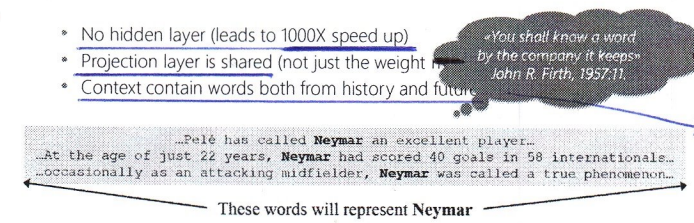
POLITECNICO MILANO 1863 12

Google's word2vec (Mikolov et al. 2013a)

Based on the idea of projecting from one-hot to a smaller dense space, we want to learn a language model

Idea: achieve better performance allowing a simpler (shallower) model to be trained on much larger amounts of data

- No hidden layer (leads to 1000X speed up)
- Projection layer is shared (not just the weight matrix)
- Context contain words both from history and future



instead of trying to predict the next word, we remove one word and we try to predict the missing word from the surrounding ones

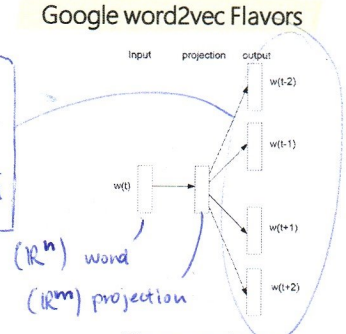
In this way we interpret the projection on the smaller-dim space as the extraction of the word's features (this projection is based on the surrounding: we want to find a representation of the surround. words which is enough to predict the missing word. The encoding of a word is the feature extraction of the text surrounding).

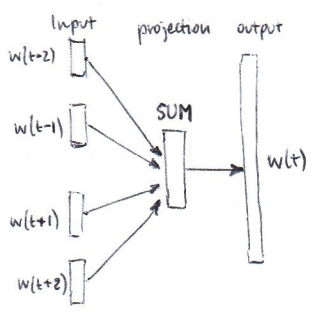
The meaning of a word is written in the words that surround it.

POLITECNICO MILANO 1863 13

Google word2vec Flavors

1. We take one word (from one-hot-encoding) and we project it. From the projection we try to predict all the surrounding words.



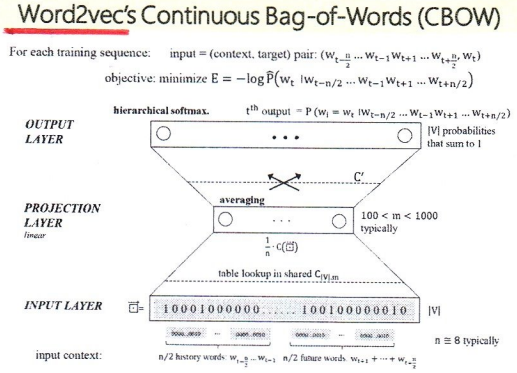


1. and 2. are one the "other way around" of the other

POLITECNICO MILANO 1863 14

Word2vec's Continuous Bag-of-Words (CBOW)

It takes n/2 words before and n/2 words after. It projects everything and it averages the projections. Then, through a softmax it predicts the more likely word.



POLITECNICO MILANO 1863 15

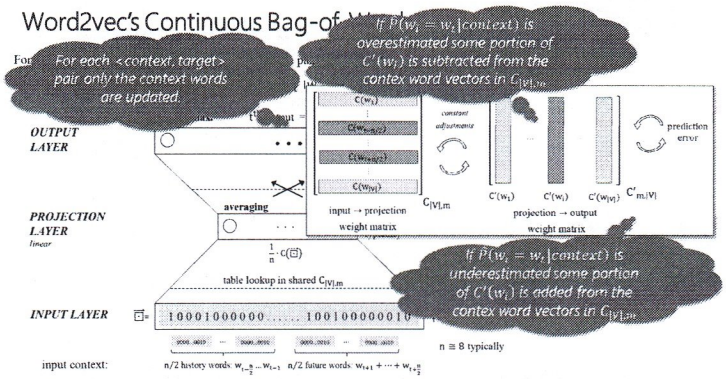
Two words surrounded by the same words, more likely will mean the same thing

2. We take the one-hot-encoding of the words around one term, we project and sum them. This is the encoding which allows to predict the word.

→ when we think about word embedding the key is the **CONTEXT**

technical details on how to improve the speed of training

Word2vec's Continuous Bag-of-W...



Word2vec facts

Word2vec shows significant improvements w.r.t. the NNML

- Complexity is $n \times m + m \times log|V|$ (Mikolov et al. 2013a)
- On Google news 6B words training corpus, with $|V| \sim 10^6$
  - CBOW with m=1000 took 2 days to train on 140 cores
  - Skip-gram with m=1000 took 2.5 days on 125 cores
  - NNLM (Bengio et al. 2003) took 14 days on 180 cores, for m=100 only!
- word2vec training speed ≅ 100K-5M words/s
- Best NNLM: 12.3% overall accuracy vs. Word2vec (with Skip-gram): 53.3%

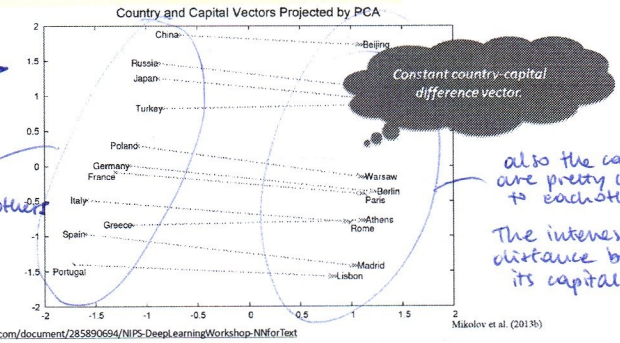
In this context the distanced are preserved: e.g. capital and country are close

Capital-Country	Past tense	Superlative	Male-Female	Opposite
Athens: **Greece**	walking: **walked**	easy: **easiest**	brother: **sister**	ethical: **unethical**

Adapted from Mikolov et al. (2013a)

Regularities in word2vec Embedding Space

Visual representation of which word is close to which:



countries are all pretty close to eachother

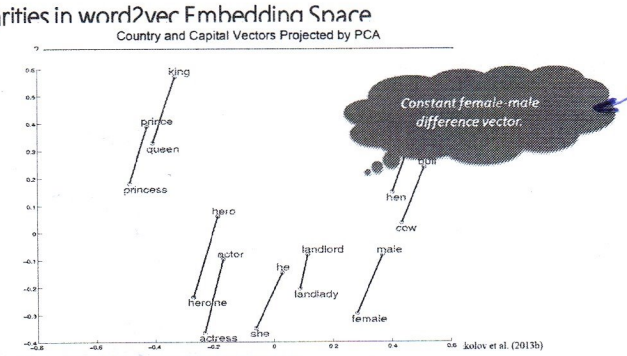
also the capitals are pretty close to eachothers

The interesting thing is the constant distance between a country and its capital.

Picture taken from: https://www.scribd.com/document/285890694/NIPS-DeepLearningWorkshop-NNforText

Mikolov et al. (2013b)

Regularities in word2vec Embedding Space



There are also other cases with constant distances

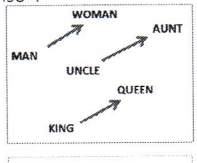
Picture taken from: https://www.scribd.com/document/285890694/NIPS-DeepLearningWorkshop-NNforText

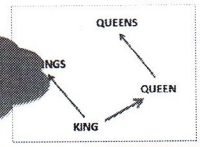
kolov et al. (2013b)

Regularities in word2vec Embedding Space

Vector operations are supported make «intuitive sense»:

- $w_{king} - w_{man} + w_{woman} \cong w_{queen}$
- $w_{paris} - w_{france} + w_{italy} \cong w_{rome}$
- $w_{windows} - w_{microsoft} + w_{google} \cong w_{android}$
- $w_{einstein} - w_{scientist} + w_{painter} \cong w_{picasso}$
- $w_{his} - w_{he} + w_{she} \cong w_{her}$
- $w_{cu} - w_{copper} + w_{gold} \cong w_{...}$
- ...





If we take from "king" the set of words which represents "man" and we add the set of words which represents "woman" we obtain "queen".

«You shall know a word by the company it keeps» John R. Firth, 1957:11.

Picture taken from: https://www.scribd.com/document/285890694/NIPS-DeepLearningWorkshop-NNforText

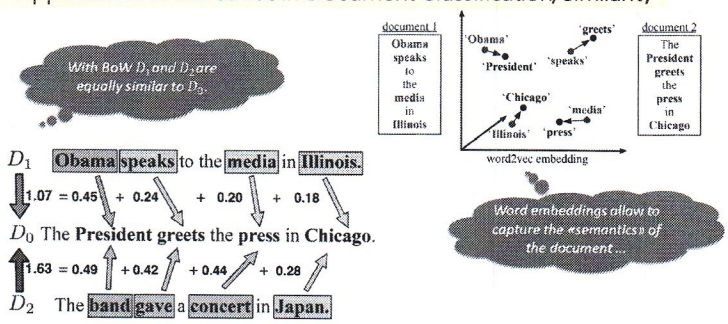
Applications of word2vec in Information Retrieval

Query: "restaurants in mountain view that are not very good"
Phrases: "restaurants in (mountain view) that are (not very good)"
Vectors: "restaurants+in+(mountain view)+that+are+(not very good)"

Expression	Nearest tokens
Czech + currency	koruna, Czech crown, Polish zloty, CTK
Vietnam + capital	Hanoi, Ho Chi Minh City, Viet Nam, Vietnamese
German + airlines	airline Lufthansa, carrier Lufthansa, flag carrier Lufthansa
Russian + river	Moscow, Volga River, upriver, Russia
French + actress	Juliette Binoche, Vanessa Paradis, Charlotte Gainsbourg

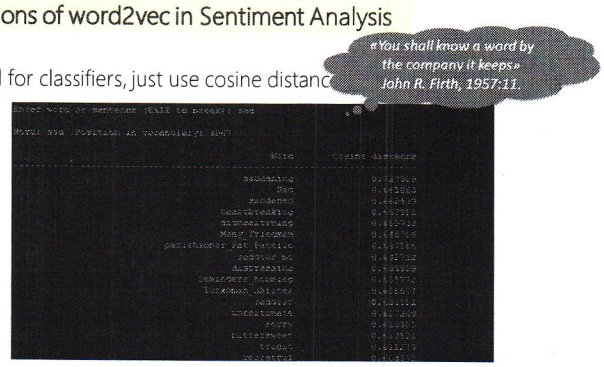
(Simple and efficient, but will not work for long sentences or documents)

Applications of word2vec in Document Classification/Similarity



Applications of word2vec in Sentiment Analysis

No need for classifiers, just use cosine distanc



just for us to know

GloVe: Global Vectors for Word Representation (Pennington et al. 2014)

GloVe makes explicit what word2vec does implicitly

- Encodes meaning as vector offsets in an embedding space
- Meaning is encoded by ratios of co-occurrece probabilities

Probability and Ratio	k = solid	k = gas	k = water	k = fashion
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}		

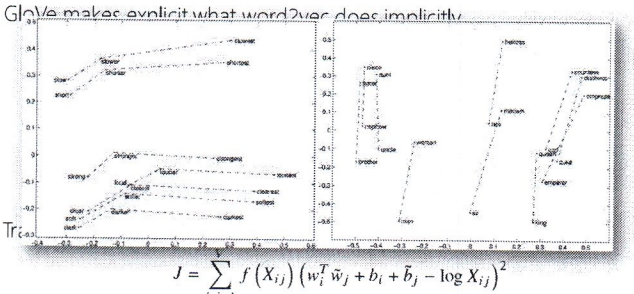
Refer to Pennington et al paper for details on this loss function...

Trained by weighted least squares

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

GloVe: Global Vectors for Word Representation (Pennington et al. 2014)

GloVe makes explicit what word2vec does implicitly



Tr

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

Nearest Neighbours with GloVe

What are the closest words to the target word *frog*:

1. *Frog*
2. Frogs
3. Toad
4. Litoria
5. Leptodactylidae
6. Rana
7. Lizard
8. Eleutherodactylus

3. litoria

4. leptodactylidae

5. rana

7. eleutherodactylus