Istituto
Dalle Molle
di studi
sull'intelligenza
artificiale
IDSIA
USI/SUPSI

Neural Networks - Sequence Modeling and Advanced Architectures

Machine Learning

Michael Wand

TA: Vincent Herrmann

{michael.wand,vincent.herrmann}@idsia.ch

Dalle Molle Institute for Artificial Intelligence Studies (IDSIA) USI - SUPSI

Fall Semester 2024

Contents

- Recurrent Neural Networks
- The LSTM
- Recurrent Network Setups: Modeling Aligned and Unaligned Sequences
- Attention Mechanisms
- Advanced Topic: The Transformer Architecture

USI/SUPSI
Istituto
Dalle Molle
di studi
sull'intelligenza
artificiale
IDSIA

Recurrent Neural Networks

Introduction (1)

- So far, we have looked at mapping samples *independently*.
- Now we look at *sequential* inputs where the output y can depend on more than just the immediate input:

$$y_t = f(x_t, \ldots, x_1)$$

  where t is a time parameter.
- In order to achieve this, the neural network holds a state, which changes when a new input arrives:

$$s_{t+1} = g(s_t, x_t)$$

- State provides memory, which in RNNs is implemented by feedback or recurrent connections: The state at time step t is the output at time step $t-1$.

Introduction (2)

Another way to introduce recurrent neural networks:

- So far, we have looked at mappings with *fixed-size* inputs.
- There are many situations where the input has *varying* size.
- We concentrate on the case of *sequential* input[1].
- A neural network cannot process such an input sequence all at once, but it can process it sequentially; in order to do so, it needs to retain *state* between inputs.
- State provides memory, which in RNNs is implemented by feedback or recurrent connections: The state at time step t is the output at time step $t-1$.

[1] If you want to apply a recurrent neural network to images, have a look at Stollenga et al., *Parallel Multi-Dimensional LSTM, With Application to Fast Biomedical Volumetric ImageSegmentation*, Proc. NIPS 2015

Sequence Modeling

- Sequential data occurs in a variety of situations:
  - ➔ Speech recognition and natural language processing
  - ➔ Video analysis
  - ➔ Text generation
  - ➔ DNA analysis
  - ➔ In reinforcement learning, short-term memory can be essential for determining the state of the world
- For now assume sequential data with one target value at each input timestep, i.e.

$$\mathbf{x} = (x_1, \ldots, x_T) \quad \text{with} \quad x_t \in \mathbb{R}^D$$
$$\mathbf{o} = (o_1, \ldots, o_T) \quad \text{with} \quad o_t \in \mathbb{R}^K.$$

- (On this slide, we use the symbol o for the target, in order to distinguish it from the time index t.)
- The length of the sequences can vary between samples.

Recurrent Layer

- Remember the standard feedforward fully-connected layer:

$$\mathbf{z}^{(\ell)} = f\left(\mathbf{W}^{(\ell)}\mathbf{z}^{(\ell-1)} + \mathbf{b}^{(\ell)}\right).$$

- Now define a Recurrent Layer as a building block of a neural network.
- The output at time step t depends on the input from the previous layer at time t *and* on the layer state at time $t-1$, i.e.

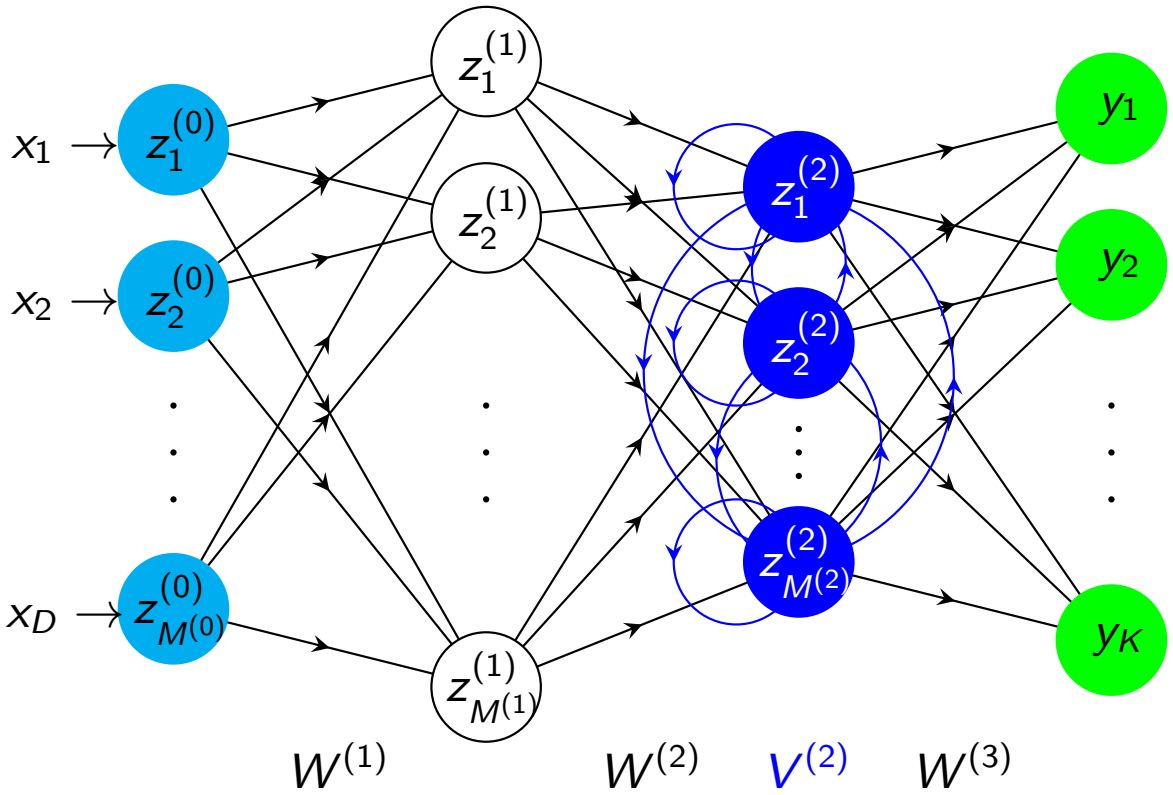
$$\mathbf{z}^{(\ell)}(t) = f\left(\mathbf{W}^{(\ell)}\mathbf{z}^{(\ell-1)}(t) + \mathbf{V}^{(\ell)}\mathbf{z}^{(\ell)}(t-1) + \mathbf{b}^{(\ell)}\right)$$

  with suitable weight matrices $\mathbf{W}^{(\ell)}$ and $\mathbf{V}^{(\ell)}$, bias $\mathbf{b}^{(\ell)}$, and nonlinearity f.
- Usually, the state $\mathbf{z}^{(\ell)}(0), \ell \in 1, \ldots, L$ is initialized to zeros.
- Storage requirement of the recurrent layer (for forward propagation): one extra set of layer activations; temporal requirement: as many computation steps as the sequence has elements.

Recurrent Network Setup

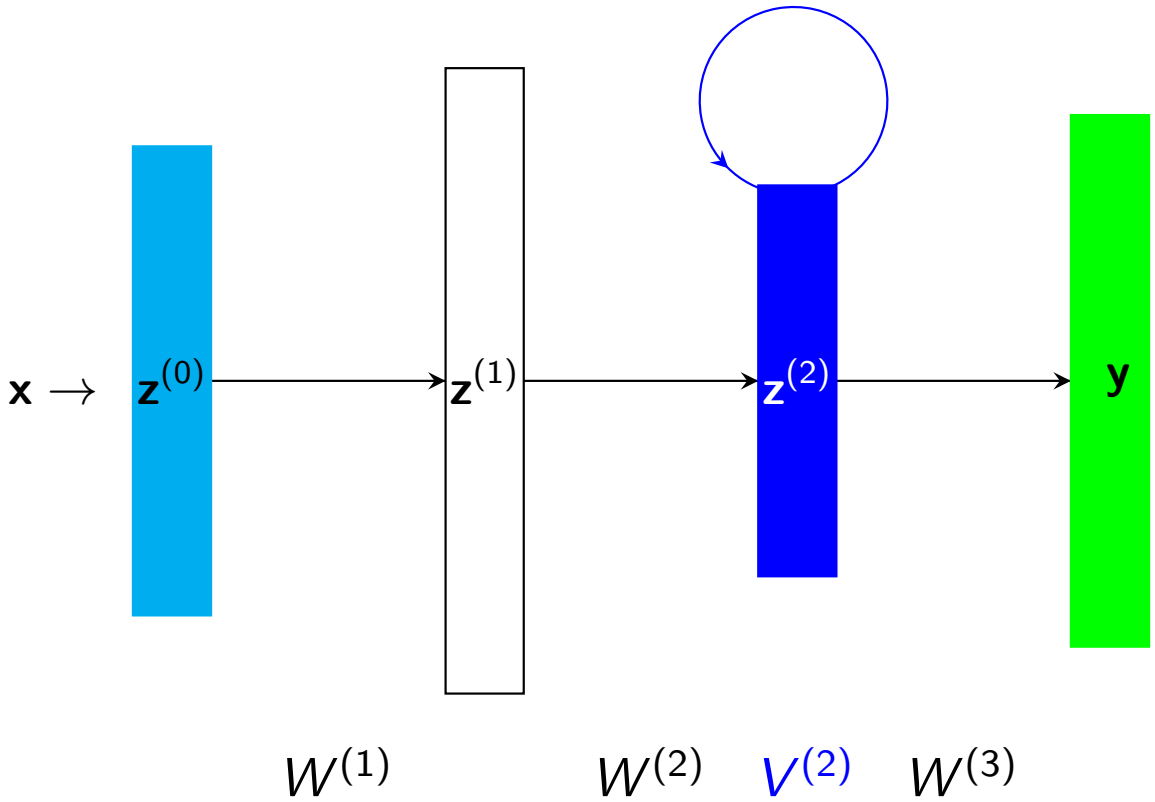
- A typical setup is to have a stack of feedforward layers (fully connected or convolutional), after which one or several recurrent layers are placed.
- The final output is usually computed by a feedforward step on the output of the last recurrent layer, followed by a standard loss function (MSE, cross-entropy).
- Depending on the task, we might consider *all* the outputs, or only the output at the last timestep. (Later on we will get to know more complex input/output setups.)

Visual: Recurrent Neural Network



The figure shows a neural network with one hidden feedforward layer and one hidden recurrent layer, followed by a feedforward output layer. Remember that the blue connections incur a delay of one time step; this makes the computation well-defined.

Visual: Recurrent Neural Network



As always, we usually reason about the network by grouping neurons to *layers*.

RNNs as Universal Approximators

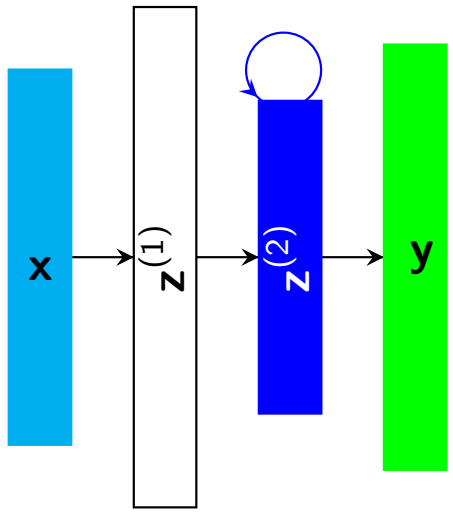
- All the models we have talked about before today can only implement static input-output mappings.
- Still, feedforward neural networks are very powerful: A sufficiently wide net can approximate any continuous function!
- Recurrent nets are *dynamical systems*: A recurrent network can implement any algorithm (given enough storage size).
- In practice, it is easier to learn some algorithms than others...
- We will now see how to extend the backpropagation algorithm which we got to know in the last lecture to deal with recurrence.

Backpropagation Through Time

- Remember the way we decomposed the gradient of the error w.r.t. a weight matrix as a product of gradients across the network layers?

$$\frac{\partial E}{\partial w_{ij}^{(\ell)}} = \frac{\partial E}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{z}^{(L-1)}} \cdots \frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell)}} \frac{\partial \mathbf{z}^{(\ell)}}{\partial w_{ij}^{(\ell)}}$$

- The easiest way to explain BPTT is to imagine the recurrent network as a very deep feedforward network by *unrolling* the time.



Backpropagation Through Time

- Remember the way we decomposed the gradient of the error w.r.t. a weight matrix as a product of gradients across the network layers?

$$\frac{\partial E}{\partial w_{ij}^{(\ell)}} = \frac{\partial E}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{z}^{(L-1)}} \cdots \frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell)}} \frac{\partial \mathbf{z}^{(\ell)}}{\partial w_{ij}^{(\ell)}}$$

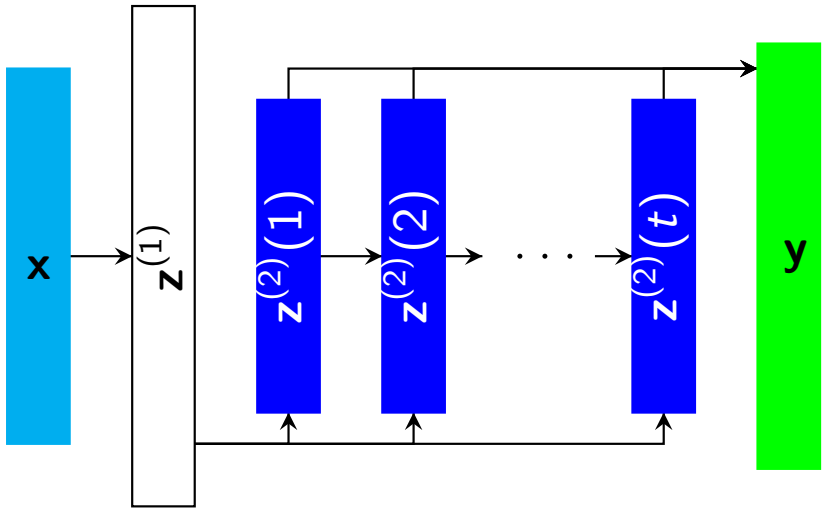
- The easiest way to explain BPTT is to imagine the recurrent network as a very deep feedforward network by *unrolling* the time.

Backpropagation Through Time

- Remember the way we decomposed the gradient of the error w.r.t. a weight matrix as a product of gradients across the network layers?

$$\frac{\partial E}{\partial w_{ij}^{(\ell)}} = \frac{\partial E}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{z}^{(L-1)}} \cdots \frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell)}} \frac{\partial \mathbf{z}^{(\ell)}}{\partial w_{ij}^{(\ell)}}$$

- The easiest way to explain BPTT is to imagine the recurrent network as a very deep feedforward network by *unrolling* the time.



Backpropagation Through Time

- We now derive the weight updates as in the feedforward case, taking into account that weights are *shared* between the time steps, and that there are *multiple* paths through the network.
- This means that the gradients must be added over timesteps, for both recurrent and feedforward layers: The error at time step t causes gradients at time steps $1, \ldots, t$.
- Also remember that (in the general case), an error signal is backpropagated from the output layer at each time step. That means that simultaneously,
  - ➔ output at time step 1 causes gradients at time step 1
  - ➔ output at time step 2 causes gradients at time steps 1 and 2
  - ➔ ...
  - ➔ output at time step t causes gradients at time steps $1, \ldots, t$.
  - ➔ That sounds complicated, but it is really just a sum of relevant gradients. We have already done all required work.
- The error at time step t is backpropagated through time.

The Vanishing Gradient Problem

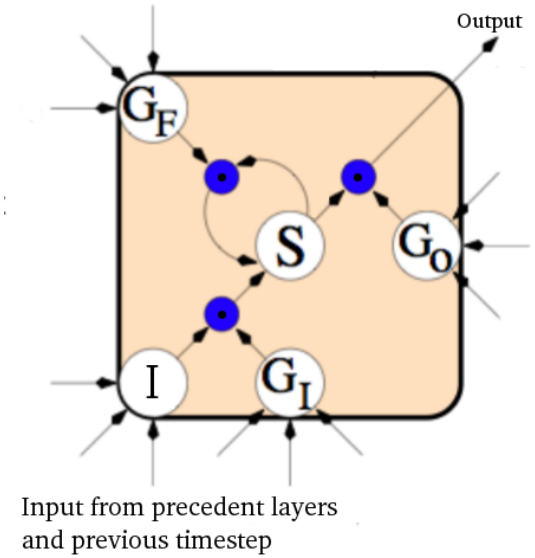
- Although RNNs can represent arbitrary sequential behavior, training suffers:
  - ➔ once the output depends on some input more than around 10 time-steps in the past, they become very difficult to train.
- Why? We apply the chain rule by *multiplying* partial gradients over time steps.
- The absolute value of these gradients shrinks exponentially (or it explodes, which is not good either).
- Thus the error gradient becomes very small: *weights cannot be adjusted to respond to events far in past*. This is the Vanishing Gradient Problem.
- Even simple tasks cannot properly be solved by the RNN if the relevant information covers a long time (example: check whether parentheses in a string are balanced).

USI/SUPSI
Istituto
Dalle Molle
di studi
sull'intelligenza
artificiale
IDSIA

Long Short Term Memory (LSTM)

LSTM Cell

- An LSTM cell can be imagined as a memory cell with a state S that is controlled by 3 gates:
  - ➔ the input gate G_i controls whether the state is updated with external input
  - ➔ the output gate G_o controls whether the state is visible to the outside
  - ➔ the forget gate G_f allows to reset the state.



- The inputs (for the state update and for all three gates) are the output of the LSTM layer of the previous timestep and the output from preceding layer of the current timestep.

References:

- Hochreiter & Schmidhuber: *Long Short-Term Memory*. Neural Computation 9, 1997.
- Gers, Schmidhuber, Cummins: *Learning to Forget: Continual Prediction with LSTM*. Neural Computation 12, 2000.

LSTM Cell

- Assume $\mathbf{z}^{(\ell-1)}(t)$ is the output of the preceding layer at the current timestep, and $\mathbf{z}^{(\ell)}(t-1)$ is the output of the LSTM layer at the previous time step.
- These are the input for the LSTM cell, whose precise behavior is as follows.
- Each gate yields a value between 0 and 1 (using sigmoid nonlinearities). *The gates are trainable*, they each have their own set of weights which connect to the input vector:

$$G_X = \sigma\left(\mathbf{W}_X\mathbf{z}^{(\ell-1)}(t) + \mathbf{V}_X\mathbf{z}^{(\ell)}(t-1) + \mathbf{b}_X\right),$$

  where X stands for any of the gates 'I', 'O', and 'F'.
- Occasionally, gates are allowed to access the cell state (peephole connections).

LSTM Cell

- The input vector is likewise processed:

$$\tilde{s}(t) = f\left(\mathbf{W}\mathbf{z}^{(\ell-1)}(t) + \mathbf{V}\mathbf{z}^{(\ell)}(t-1) + \mathbf{b}\right),$$

  where f is normally a tanh nonlinearity.

- At time t, the new state becomes

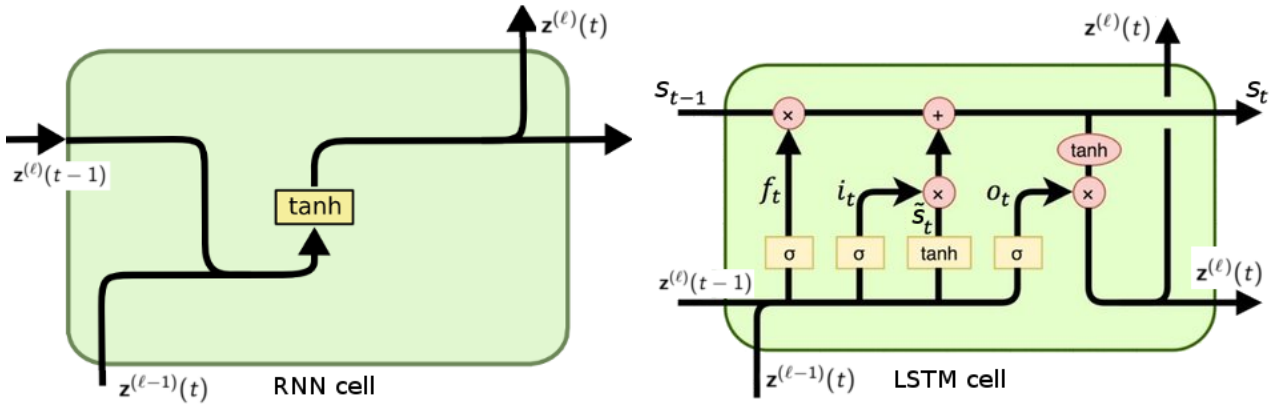
$$s(t) = G_F(t) \cdot s(t-1) + G_I(t) \cdot \tilde{s}(t).$$

- Finally, the output $z^{(\ell)}(t)$ is computed from the cell state by passing it through another nonlinearity (usually tanh), and multiplying it with the output gate:

$$z^{(\ell)}(t) = G_O(t) \cdot f(s(t)).$$

LSTM Cell

- Compare the dynamics of the LSTM and the RNN.
  - ➔ Notation: yellow blocks are fully connected layers, red are operations.
  - ➔ Time flows from left to right.
- Left panel - RNN: output from previous step $z^{(\ell)}(t-1)$ and input from current step $z^{(\ell-1)}(t)$ are concatenated, new output $z^{(\ell)}(t)$ is computed.
- Right panel - LSTM: The upper line is the *state*, can you see how it can remain unchanged over many time steps?
- f, i, o stand for the forget, input, and output gates.



Images modified after `colah.github.io/posts/2015-08-Understanding-LSTMs`

LSTM Layer

- An LSTM layer consists of a number of LSTM cells (which are all connected, i.e. the behavior of a cell $z_m^{(\ell)}$ at time t depends on *all* the other cells at time $t-1$).
- It is a neural network building block very much like a recurrent layer (just replace each recurrent neuron with an LSTM cell).
- But the behavior is very different:
  - ➔ The trainable gates control the flow of information, and also make it easy to *store* information over longer periods of time!
  - ➔ During backpropagation, this means that the error signal is likewise propagated over many time steps.
  - ➔ By contrast, in standard RNNs, the state is *completely recomputed* in every time step, which also causes the error signal to suffer degradation during backpropagation.
- The whole architecture can be trained end-to-end.
- This remarkably powerful architecture solves the vanishing gradient problem and makes it possible to solve many challeging sequence-related tasks!

USI/SUPSI
Istituto
Dalle Molle
di studi
sull'intelligenza
artificiale
IDSIA

Modeling Aligned and Unaligned Sequences

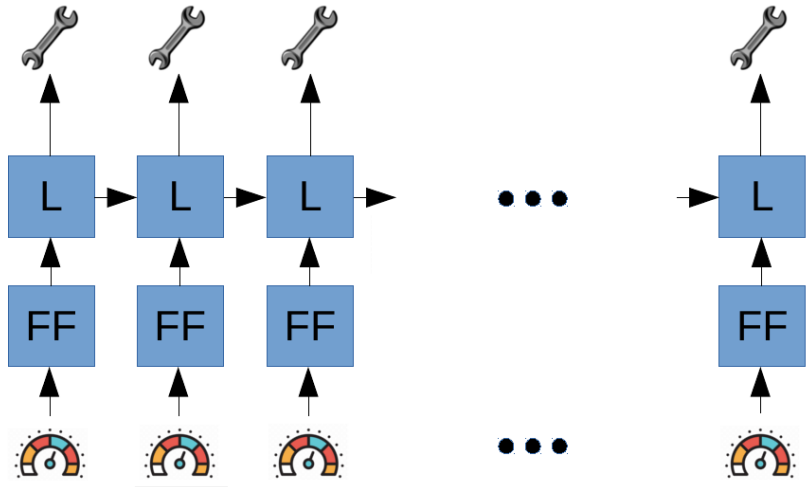
LSTM Setups

(Everything which is covered in this section technically also works for standard RNNs, but will usually not work in practice due to the vanishing gradient problem.)

- So far, we have assumed that the training data is *aligned*: There is one output target for each input step.
- Thus, the LSTM receives one error signal per input step, which is backpropagated through time.
- Sometimes, one can force the output to have the desired length (e.g. by padding).

LSTM Setups

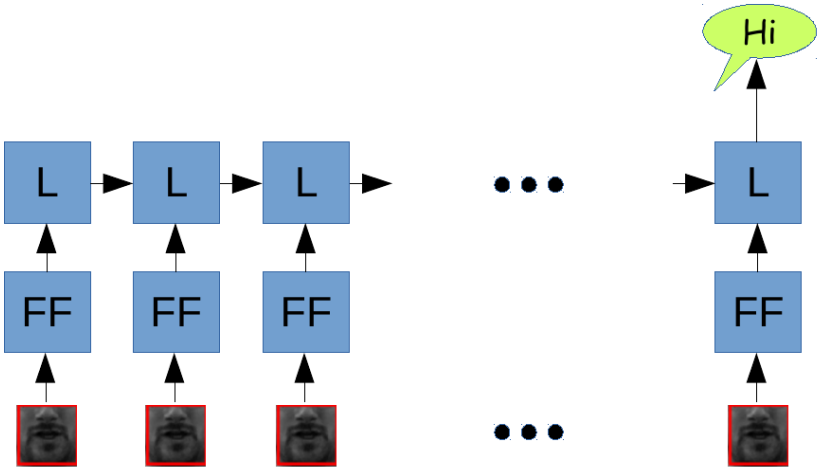
- Example – *Control task*: At each input, one wants to output a new control signal, which takes the current input and an estimate of the system state into account.



An LSTM for a control task, with aligned input and output. Input is processed by a stack of feedforward layers and a single LSTM layer.

LSTM Setups

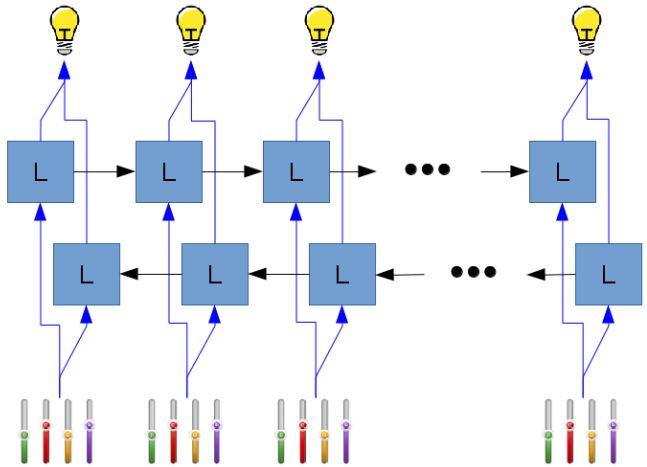
- Another simple setup: There is a *single* target per sequence.
- In such a case, one only considers the LSTM output at the end of each sequence, and likewise, errors are backpropagated only from the last element.
  - ➔ (A practical implementation would use some kind of *mask* to control error backpropagation; modern frameworks have this functionality built in.)
- Example: Word-based speech recognition (see image).



Michael's work: Word-based *Lipreading* with LSTMs. Only the output at the last step is valid. Images are processed by a stack of feedforward layers and a single LSTM layer.

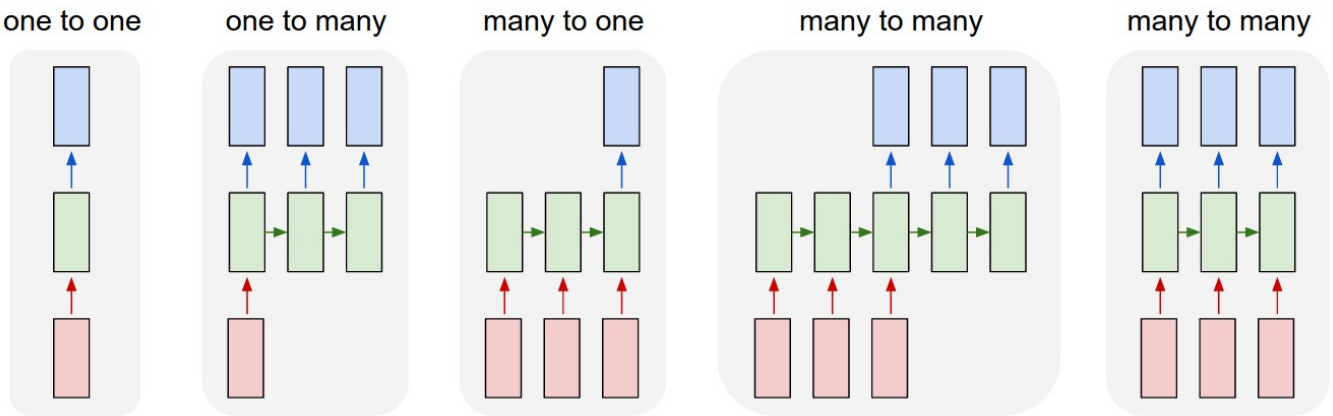
Bidirectional LSTM

- Assume that we have one output sample for each input step, but dependencies between inputs and outputs are in random order.
- Thus each output sample depends on the *entire* sequence, not just on the past frames.
- One solution: The bidirectional LSTM (layer).
- The input is fed into two (identical) parallel LSTMs, once in *forward* order, once in *backward* order.
- The output is created by stepwise concatenation of the outputs of the two LSTMs.
- Thus at each step, the output reflects the *entire* sequence.
- Easy-to-implement, standard architecture.



LSTM Setups

- The class of sequence-to-sequence tasks is however much larger.
- There are many cases in which the length of the input and output sequences do not match.
- Even if they do, inputs and outputs may not be aligned, e.g. a relevant output may appear *before* a relevant input (think about translating English to German).
- Occasionally, you may even wish to generate sequential output from a single input sample (e.g. *image description*).

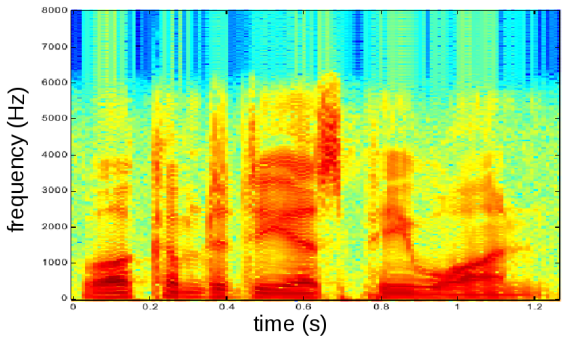


taken from karpathy.github.io

Sequence Modeling with CTC

- What if input and output sequences have different length?
- Connectionist Temporal Classification (CTC, Graves et al. 2006) was the first method to tackle this problem.
- Task Setup: A set of input sequences $\{\mathbf{x}_n\}_n$ and target sequences $\{\mathbf{t}_n\}_n$, where for each n, $\text{len}(\mathbf{t}_n) < \text{len}(\mathbf{x}_n)$.
- Assume an *orderedness* property: Features which correspond to targets appear in the same order as the targets.
- Example: In speech recognition, we have an input sequence of frequency vectors (spectrogram), which must be converted into a sequence of phones (speech sounds).

- The spectrogram usually has 100 samples per second, a phone lasts between 30 and 100 ms $\Rightarrow$ far less phones than input samples.



Sequence Modeling with CTC - Training

- The CTC solution is as follows:
  - ➔ Define a *blank* symbol (–) which is added to the set of possible output phones (as in other classification tasks, use a one-hot encoding).
  - ➔ During training, consider all paths which correspond to the given label. A path is converted into a label by removing all duplicate phones, and then removing all *blank*s.
  - ➔ For example, the following paths:

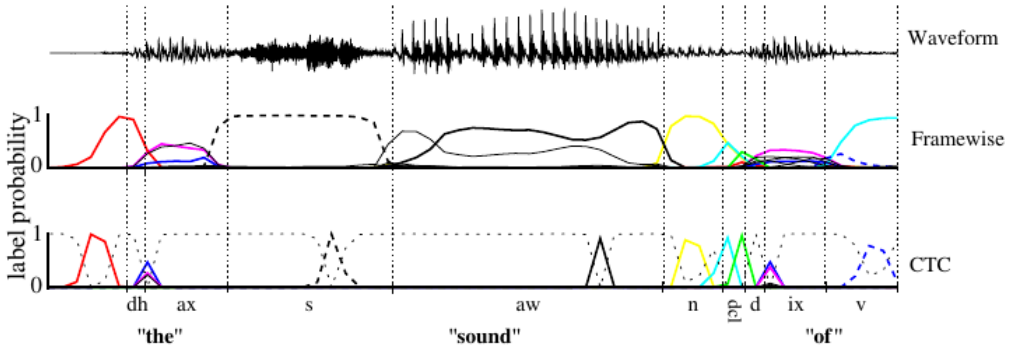
    H EH EH L OU, H – EH – – – – L L – – OU –, – – H – EH – – – L OU – –

    all correspond to the label "H EH L OU" (hello).
  - ➔ Note that blanks are necessary to output the same symbol repeatedly.
  - ➔ Obviously, this leads to a huge number of possible outputs whose errors must be computed. Fortunately, the computation can be efficiently performed by a *Dynamic Programming* approach.
  - ➔ The *Dynamic Programming* gets integrated into the CTC loss function, allowing to train the whole network end-to-end.

If you do not know Dynamic Programming, we will cover a more general version (Beam Search) in a few slides.

Sequence Modeling with CTC - Decoding

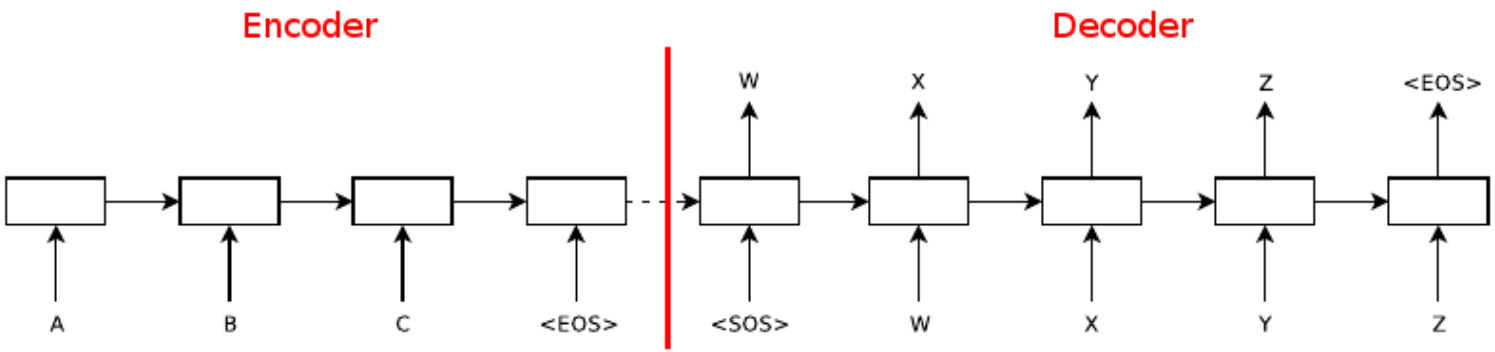
- The figure shows the output of CTC decoding on an audio clip which contains the words "THE SOUND OF".
- Note that the output is probabilistic, as for other classifiers: In each time step, one gets a vector of probabilities for the possible output symbols (phones + *blank*).
- The result is convincing: The LSTM learns to output a series of single phones, padded by *blank* symbols (dashed line).
- For recognition, we can simply take the symbol with maximal probability at each frame (greedy decoding).
- Or we can incorporate constraints (e.g. a dictionary which contains possible pronunciations) by using Beam Search.



CTC output, from Graves et al. 2006.

Encoder-Decoder Networks

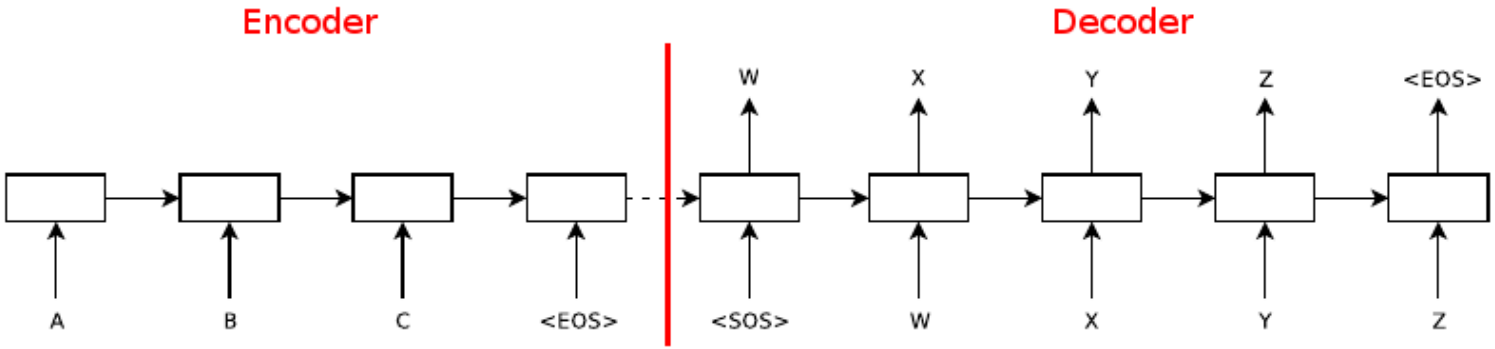
- CTC makes several assumptions: not more targets than inputs, orderedness, outputs are statistically independent (given the internal state of the network).
- We look at a simple neural network setup to deal with arbitrary sequence-to-sequence tasks. We make no assumptions about sequence lengths or orderedness.
- Idea: First read the *entire* source sequence (encoder part), then output the *entire* target sequence (decoder part).
- Special tokens (*SOS*, *EOS*) indicate the sequence start and end.



Time runs from left to right. The model reads the input tokens "ABC" and produces the output tokens "WXYZ", which are re-used as inputs in the next step. Image modified from Sutskever et al., *Sequence to Sequence Learning with Neural Networks*, NIPS 2014.

Encoder-Decoder Networks

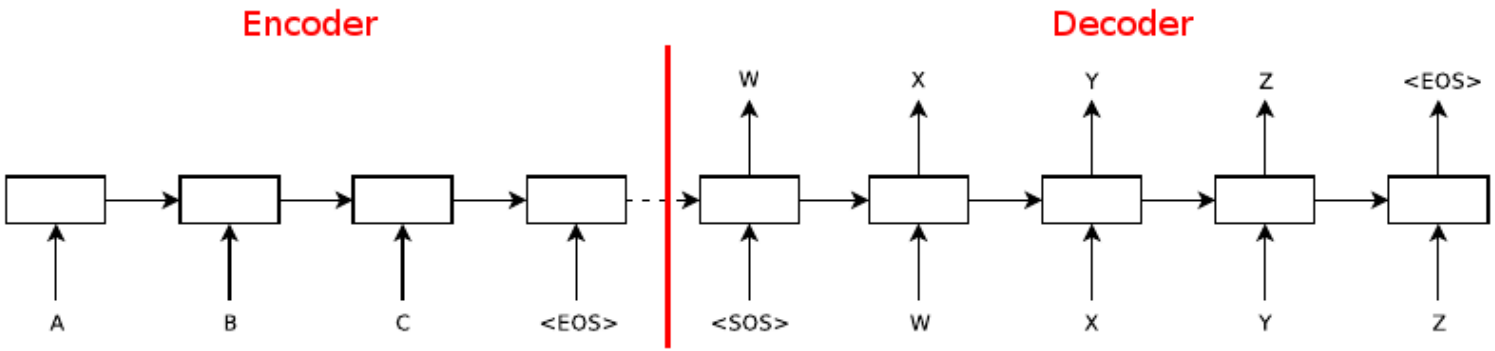
- The encoder and the decoder are usually two *separate* LSTMs.
- Information is transferred by transferring the *state* of the LSTM from encoder to decoder (and also to backpropagate the error in the same way).
- The entire input sequence is converted to a fixed-size state vector (between Encoder and Decoder).
- Architectural variations (e.g. multiple LSTM layers) are possible and are practically in use.



Time runs from left to right. The model reads the input tokens "ABC" and produces the output tokens "WXYZ", which are re-used as inputs in the next step. Image modified from Sutskever et al., *Sequence to Sequence Learning with Neural Networks*, NIPS 2014.

Encoder-Decoder Networks

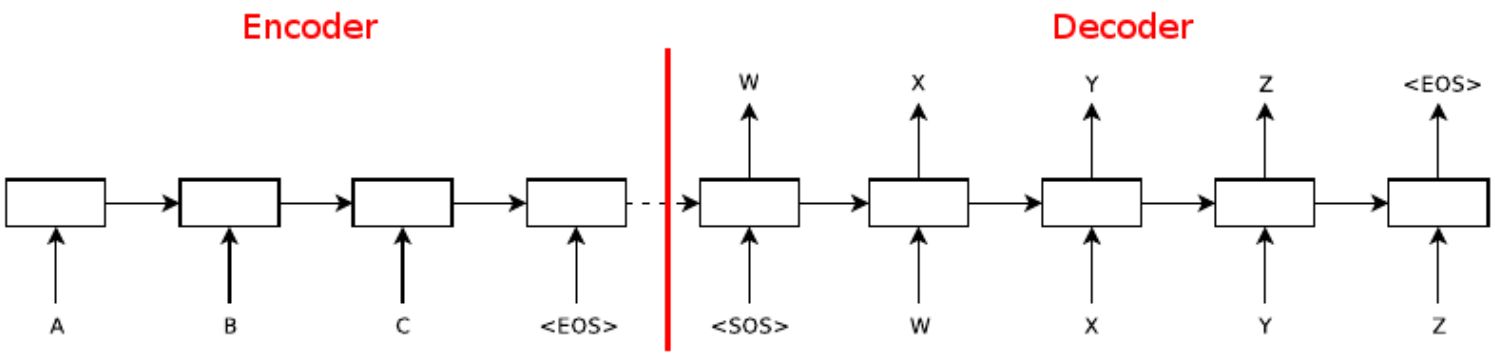
- In the decoding part, the outputs of the previous step are fed back to the network.
  - ➔ Among other advantages, this *autoregressive* setup allows to explore several hypotheses.
- During (supervised) training, one usually feeds the target symbol, even if the previous step output another hypothesized symbol (teacher forcing). One can occasionally feed hypothesized symbols to improve generalization. Training errors are backpropagated from all steps of the decoder part.



Time runs from left to right. The model reads the input tokens "ABC" and produces the output tokens "WXYZ", which are re-used as inputs in the next step. Image modified from Sutskever et al., *Sequence to Sequence Learning with Neural Networks*, NIPS 2014.

Encoder-Decoder Networks

- During testing, one uses the hypothesis from the previous step.
  - ➔ In the simplest case, we compute exactly one hypothesis sequence (greedy decoding). Since each hypothesis is fed back into the network, this means that one small error can cause a whole chain of subsequent errors.
  - ➔ One can partially mitigate this problem by using Beam Search, where several hypotheses are kept in each step.



Time runs from left to right. The model reads the input tokens "ABC" and produces the output tokens "WXYZ", which are re-used as inputs in the next step. Image modified from Sutskever et al., *Sequence to Sequence Learning with Neural Networks*, NIPS 2014.

Encoder-Decoder Networks

- As the name indicates, the *entire* source sequence must be encoded in a fixed-size vector (the state of the underlying LSTM at the EOS).
- It is not obvious that such a representation is very robust. Indeed, the authors suggest to reverse the input sentences in order to have less long-term dependencies (even though the LSTM should handle them...)
- Still, this paves the way to *embeddings*, a very important concept in contemporary neural network modeling.

Embeddings

- Embedding: Arbitrary-sized, possibly categorical inputs (like words, or sequences of words) are mapped to real-valued vectors.
- The embedding is part of a neural network which is trained end-to-end (or it can be pretrained for later use in a variety of ways).
- It turns out[2] that these representation have amazing regularities, for example, the equation

$$\phi(\text{king}) - \phi(\text{man}) + \phi(\text{woman}) \approx \phi(\text{queen})$$

  is true in *standard vector space terminology*.
- The encoder part of an encoder-decoder network computes an embedding of the input sentence.

[2] Tomas Mikolov et al, *Linguistic Regularities in Continuous Space Word Representations*. Proc. NAACL, 2013

Beam Search

- Greedy decoding may not be optimal when the elements of the output sequence are not conditionally independent.
- This is true for the Encoder-Decoder networks since output hypotheses are fed back into the network.
- It is also not true when the output is constrained: For example, in (conventional) speech recognition, one has a dictionary of possible words (and their pronunciation) and a language model which gives probabilities for word sequences. One would ideally like to incorporate these extra probabilistic constraints into the recognition result.

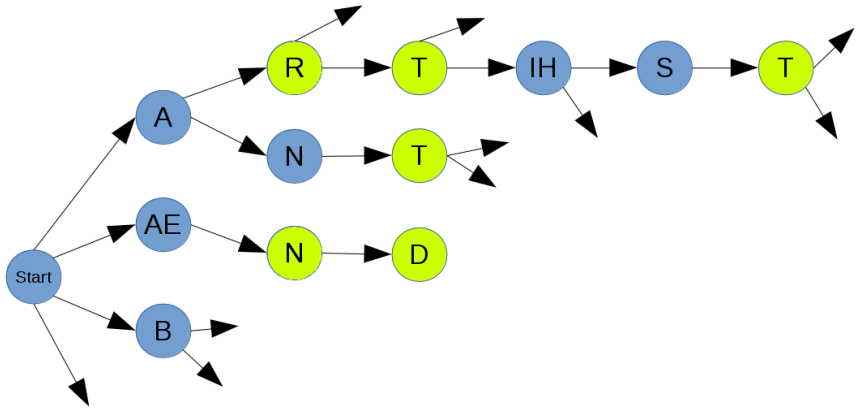
This problem occurs in a variety of tasks, and for many underlying models: Encoder-Decoder, CTC, HMM (Hidden Markov Model), . . . !

Beam Search

- *Purely theoretically*, one could consider *all* possible output sequences (up to some maximal length), compute their probabilities, and then take the one with maximum probability.
- But **no way** to compute all these probabilities, the number of computations would be enormous:
  - ➔ assume M output symbols, T time steps $\Rightarrow M^T$ possible paths
  - ➔ Speech recognition example: 26 letters, max sequence length 100 $\Rightarrow$ 26^{100} possible combinations (there are about 10^{80} atoms in the universe)
- Standard method to solve this problem: Beam Search.
  - ➔ Applicable in many cases (not only for neural networks)!
  - ➔ Occasionally the setup is a bit different, we cover the specific case of encoder-decoder network output.

Beam Search

- We use speech recognition as an example. Our goal is to get the most probable sequence of words, given a dictionary of possible pronunciations.



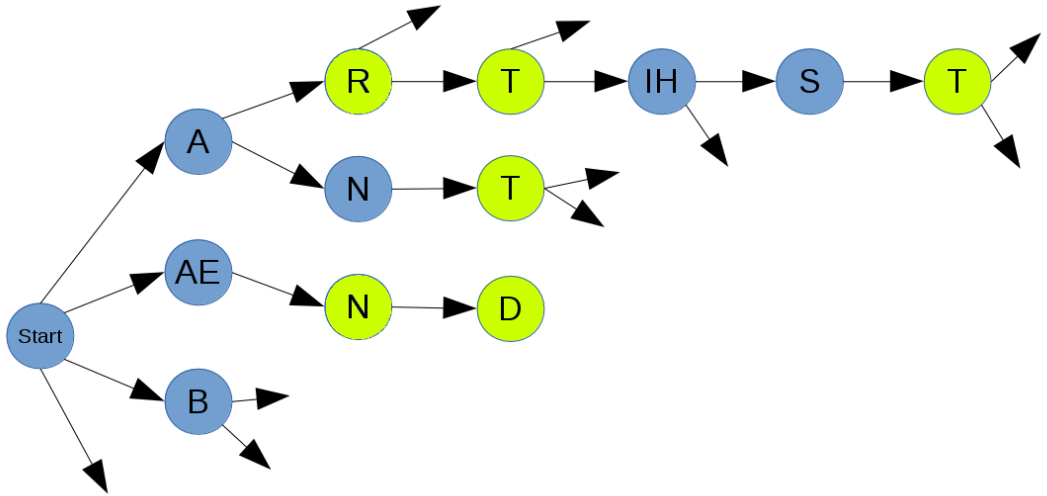
- We construct a prefix tree of phone sequences corresponding to all possible words.
- Beam Search is a *time-synchronous* search method: We completely process one time frame before moving to the next one.
- Accumulated probabilities of *prefixes* of words (hypotheses) are saved in the nodes of the prefix tree.

Beam Search

Assume you have a sequence of softmax outputs. Derive most probable sequence of words. This is the step-by-step algorithm:

- Initialize single hypothesis with probability 1.0 for the start node of the prefix tree.
- At each step:
  - ➔ Take all current hypotheses
  - ➔ Propagate each of them to all possible successor nodes by multiplying with corresponding probability from softmax outputs at current timestep
  - ➔ This requires performing the decoding several (many) times, feeding different hypotheses.
  - ➔ Repeat for each timestep
  - ➔ Result: Hypothesis with maximal probability of all *final* nodes of the tree.

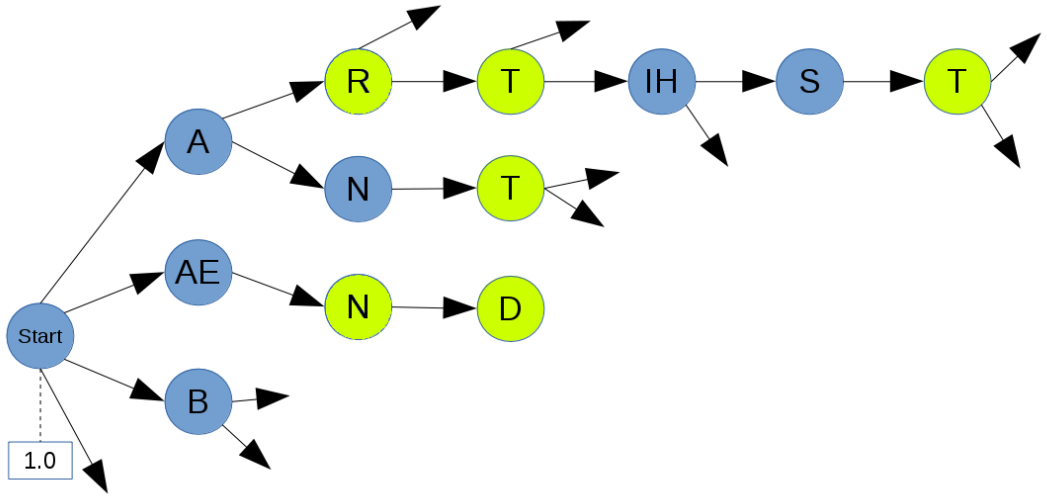
Beam Search



A part of a prefix tree. Green nodes are final, they correspond to words. Can you see which ones?

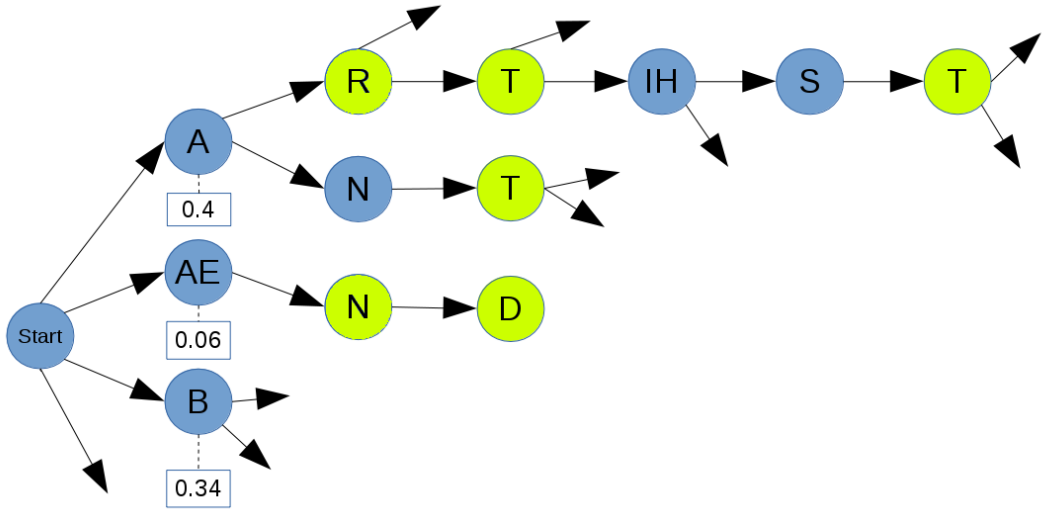
ARE, ART, ARTIST, ANT, AN, AND, ...

Beam Search



Initialization with a probability of 1.0, since there is just one start node. (You may also skip the start node and have several disconnected trees, one for each possible phone which may start a word. It is an implementation detail.)

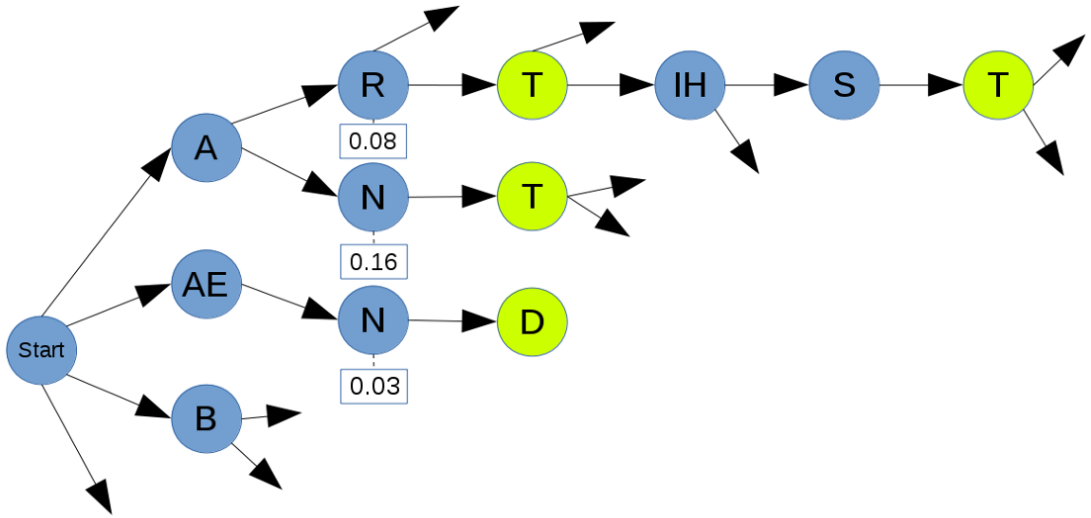
Beam Search



First propagation step! We need to consider the output phone probabilities; if we have an encoder-decoder network, they are computed by the final softmax layer of the decoder.

Phone	A	AE	B	...
Prob.	0.4	0.06	0.34	...

Beam Search



Next propagation step!

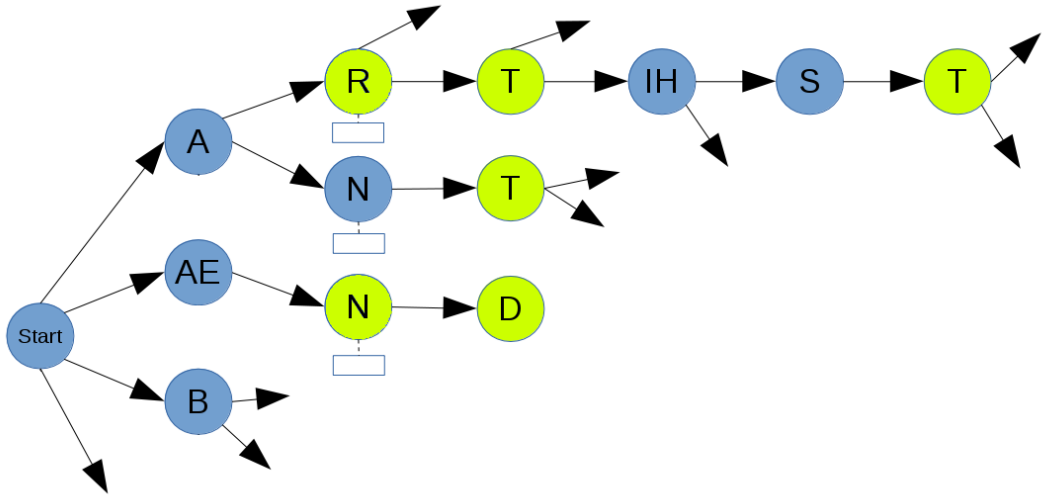
History	A	A	AE	...
Phone	R	N	N	...
Prob.	0.2	0.4	0.5	...

Probabilities are *accumulated* along the tree.

Beam Search

- In practical implementations, one usually computes in the logarithmic domain.
- Note that in the case of an encoder-decoder network, we always have a transition to the next node (exception: final node).
- In other setups, you may have transitions to the *same* node (for example, if you have CTC output, a *blank* symbol causes transition to the same node).
- In this case, if we have several ways to get to a node (e.g. you can get to the first 'A' by having *blank* + A or A + *blank*), you usually retain the hypothesis with maximum probability.

Beam Search



- A hypothesis in a final node can be propagated to the same node, using the probability of the EOS symbol (once). Then this hypothesis is not changed any more.
- The search is finished when all frames of the softmax output sequence are exhausted, then take the hypothesis with *maximum probability* of all *final* nodes as result!

Beam Search

We have skipped the following details:

- In the specific case of speech recognition, you want to recognize a *sequence* of words, so you propagate from each final node to the start node of the tree. You may have to incorporate linguistic probabilities, and you get several hypotheses per node (with different word histories).
- This means that you also have to propagate several hypotheses from a node.
- Occasionally, one merges tails of words in the tree (then it is not a tree any more), mostly due to speed considerations.
- In general, this style of search is easy to implement, but if you want it to be fast, there are a lot of details to consider...

Beam Search

- In practical cases, it is still necessary to limit the number of hypotheses during the search.
- This can be done by simply
  - ➔ retaining a fixed number of most probable ones
  - ➔ or by retaining a fixed number of hypotheses per node
  - ➔ or by other heuristic/adaptive criteria
  - ➔ or by all of the above.
- Imagine that you search for solutions with a flashlight: You look in the limited space which is lit by the *beam* of your lamp. The position of the beam depends on where you currently are, it could also become wider or smaller depending on the circumstances.

Beam Search

- Problem: Beam search may miss optimal results. For example, your speech recognizer might output the most optimal hypothesis *DOCK BUYS MAN*.
- Only at the *end* of the phrase, it may be clear (maybe even from the external language model) that *DOG BITES MAN* was much more probable.
- Thus you should not be too restrictive when running this algorithm (keep the "beam width" wide enough).
- Still, in practice beam search works very well, it is used in many contexts and with many recognition backends.

USI/SUPSI
Istituto
Dalle Molle
di studi
sull'intelligenza
artificiale
IDSIA

Attention Mechanisms

Introducting Attention

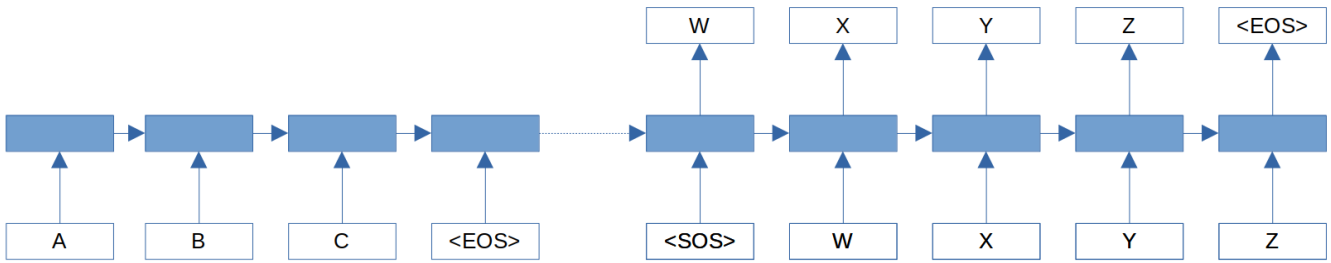
- Remember one of the reasons why we introduced recurrent networks?
  - ➔ A feedforward network cannot process variable-length input.
- One *can* process *parts* of the input.
- Attention: The neural network selectively *focuses* on parts of the input (often after some processing) which are important for the current output step.
- The current focus can be computed in a variety of ways.
- The idea dates back to Jürgen Schmidhuber's work in the 1990's[3], the modern formulation was presented by Bahdanau and colleagues[4].

[3] Schmidhuber/Huber, *Learning to Generate Artificial Fovea Trajectories for Target Detection*, International Journal of Neural Systems 2(1&2), 1991

[4] Bahdanau et al., *Neural Machine Translation by Jointly Learning to Align and Translate*. ICLR 2015

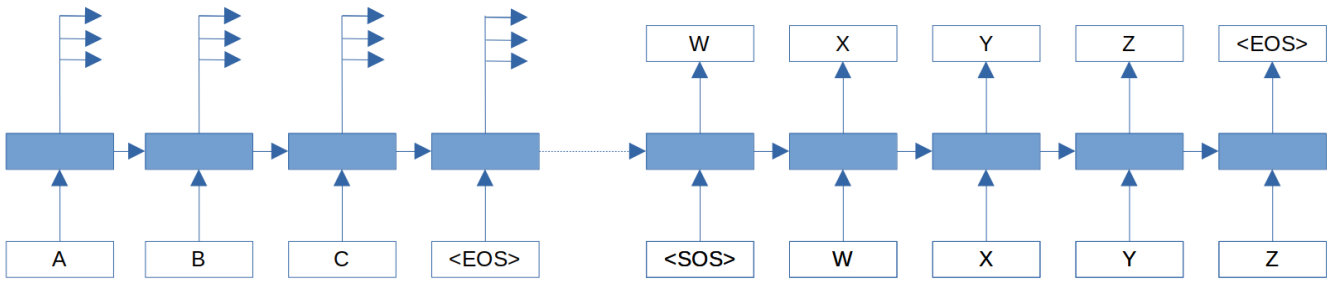
Basic Attention Mechanism

- Reconsider the encoder/decoder architecture. In the example, the input consists of the tokens A, B, C.



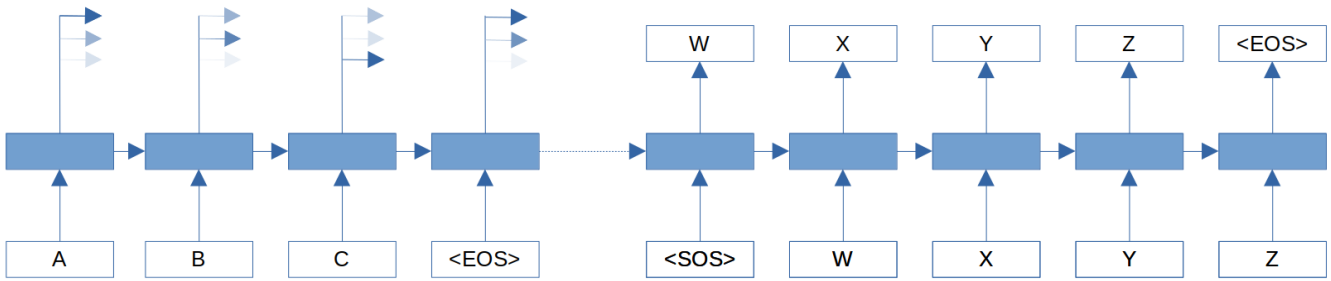
Basic Attention Mechanism

- Reconsider the encoder/decoder architecture. In the example, the input consists of the tokens A, B, C.
- How can we feed information from the encoder to the decoder?



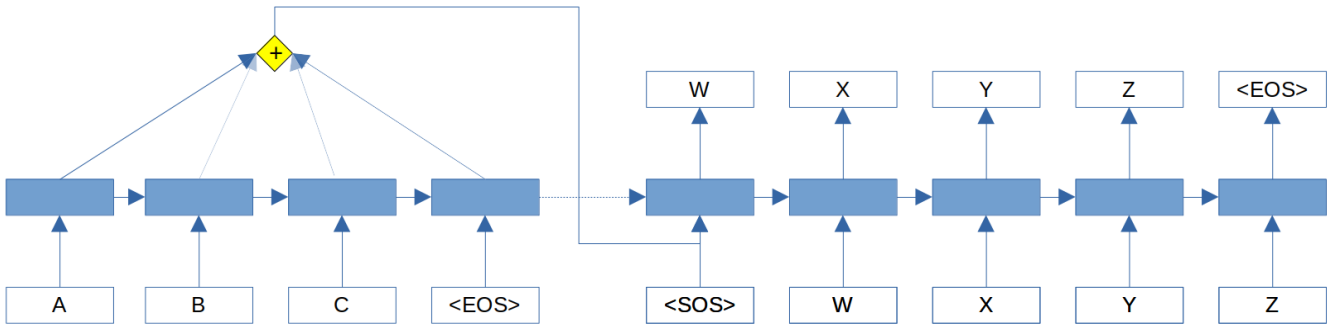
Basic Attention Mechanism

- Reconsider the encoder/decoder architecture. In the example, the input consists of the tokens A, B, C.
- How can we feed information from the encoder to the decoder?
- We assume that the relevance of each input sample varies: for each created output token, we would like to focus on different input tokens.
- The alignment between input and output samples need not be monotonic.



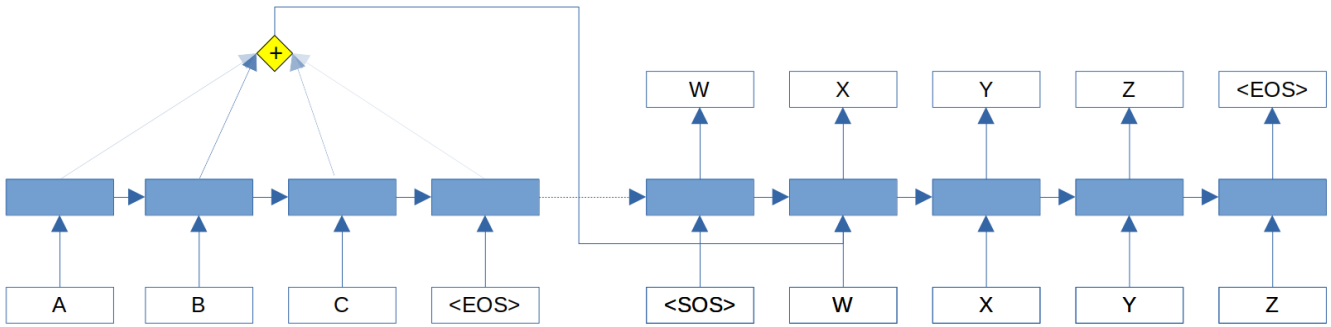
Basic Attention Mechanism

- Idea: Consider the encoded input (a fixed-size vector for each input sample).
- Compute a *weighted sum* of the encoded input vectors. The weights depend on the input data *and* on the decoding process (we will soon see how).
- This annotation vector is recomputed and fed into the decoder in each timestep.
- The decoder attends to specific areas of the input, namely to those areas with large weights.



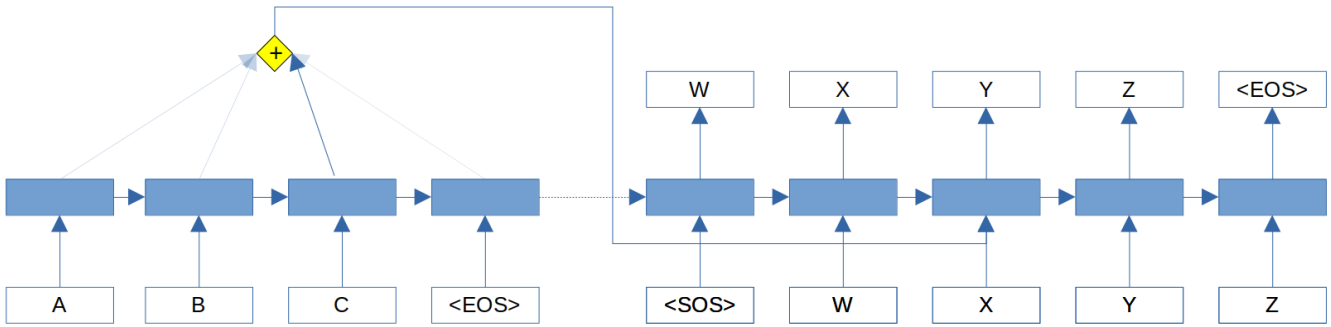
Basic Attention Mechanism

- Idea: Consider the encoded input, which is represented by a fixed-size vector for each input sample. The encoded input is also called annotation.
- Compute a *weighted sum* of the encoded input vectors. The weights depend on the input data *and* on the decoding process (we will soon see how).
- This context vector is recomputed and fed into the decoder in each timestep.
- The decoder attends to specific areas of the input, namely to those areas with large weights.



Basic Attention Mechanism

- Idea: Consider the encoded input (a fixed-size vector for each input sample).
- Compute a *weighted sum* of the encoded input vectors. The weights depend on the input data *and* on the decoding process (we will soon see how).
- This annotation vector is recomputed and fed into the decoder in each timestep.
- The decoder attends to specific areas of the input, namely to those areas with large weights.



Bahdanau Attention

- How to compute the attention weights? Note that they are different from the weights which connect neurons in the NN, because they depend on the data!
- Original idea ("Bahdanau Attention"): predict the weights from the *state* of the decoder, using a feedforward neural network.
  - ➔ For decoding step i, estimate the alignment of the decoder state s_{i-1} and each annotation h_j with a feedforward NN:

    $$e_{ij} = a(s_{i-1}, h_j)$$

    where a is a neural network.
  - ➔ Compute the attention weights α_{ij} from the alignment with a softmax function, i.e. $\vec{\alpha}_i = \text{softmax}(\vec{e}_i)$.
  - ➔ Compute the context vector as a weighted sum of the annotations: $c_i = \sum_j \alpha_{ij} h_{ij}$.
- The network a is trained jointly with the rest of the architecture.

Bahdanau Attention

- The first application of attention was to Machine Translation.
- The figure compares the quality of translations for the attention-based ("search") system and for the classical encoder-decoder ("enc") system (BLEU score, higher is better), for two different training setups.
- In particular for long sentences, the attention system is substantially better.

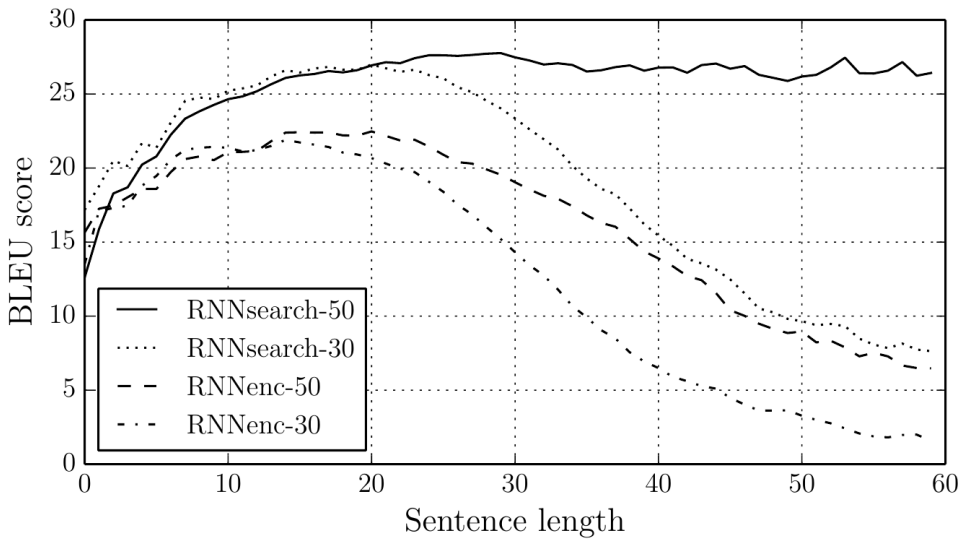
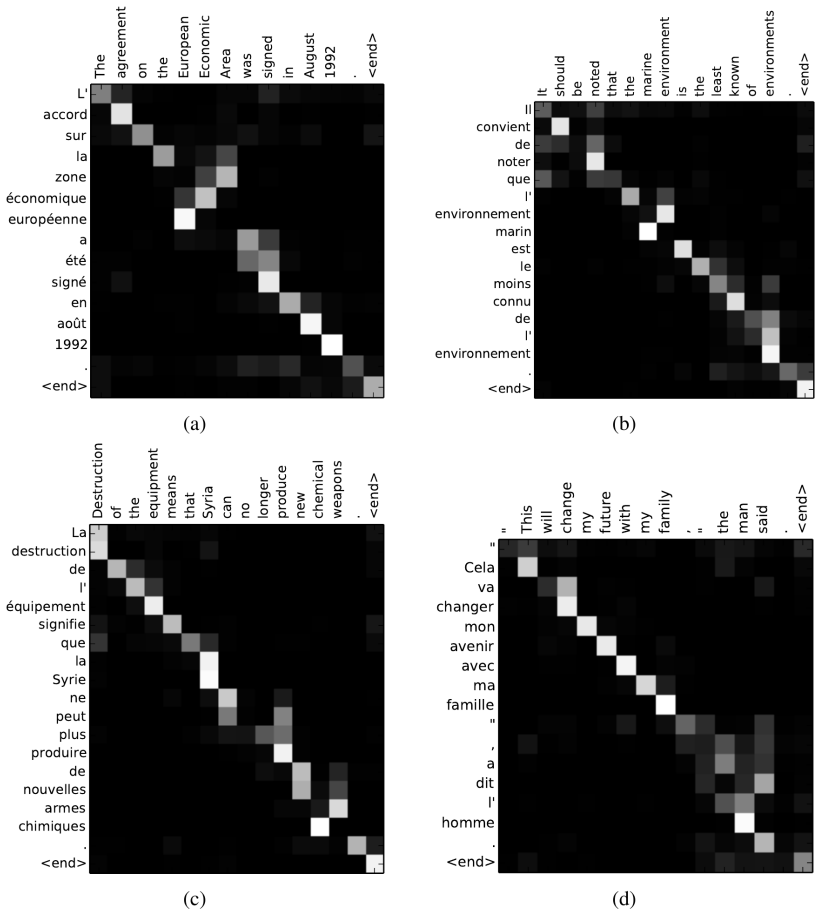


Figure 2: The BLEU scores of the generated translations on the test set with respect to the lengths of the sentences. The results are on the full test set which includes sentences having unknown words to the models.

Source: Bahdanau et al, *Neural Machine Translation by jointly learning to align and translate*

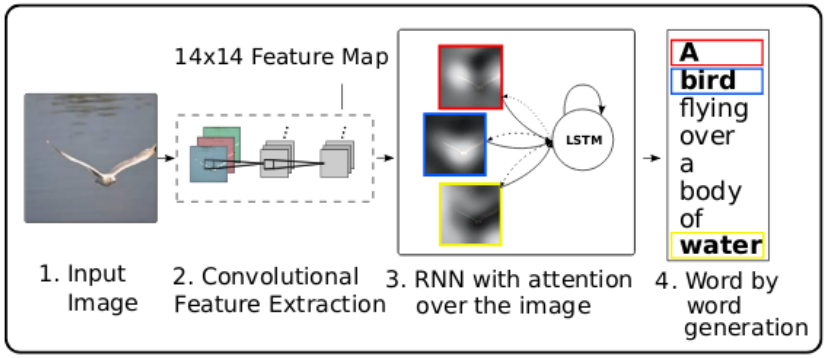
Bahdanau Attention



Source: Bahdanau et al, *Neural Machine Translation by jointly learning to align and translate*

Image Captioning with Visual Attention

- The Attention concept is powerful and can be used for many different tasks.
- Often, some interpretation of the attention weights is possible.
- Example: Create descriptions of images.
- Several styles of attention ("soft" vs "hard" attention).
- Note that the input is *not* sequential (but an image).



From Xu et al., *Show, Attend and Tell: Neural Image CaptionGeneration with Visual Attention*

Image Captioning with Visual Attention

Figure 2. Attention over time. As the model generates each word, its attention changes to reflect the relevant parts of the image. "soft" (top row) vs "hard" (bottom row) attention. (Note that both models generated the same captions in this example.)

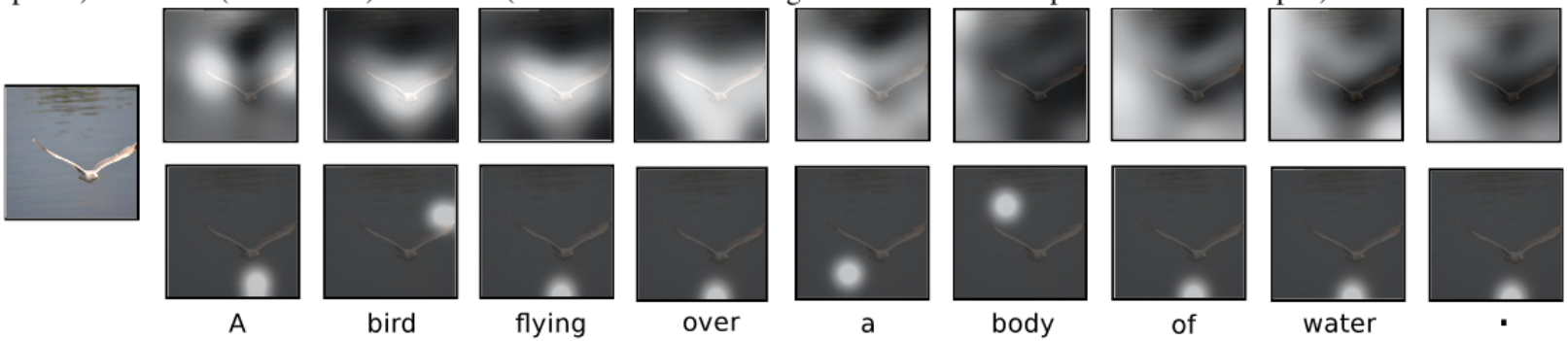
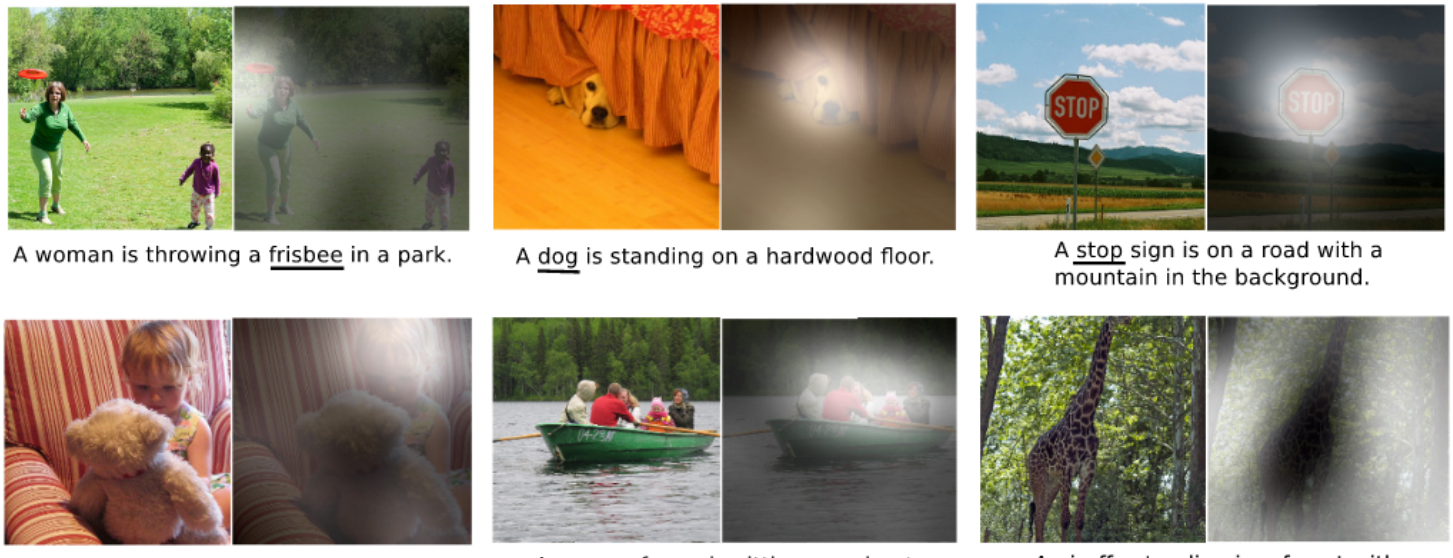


Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)



More Attention Mechanisms

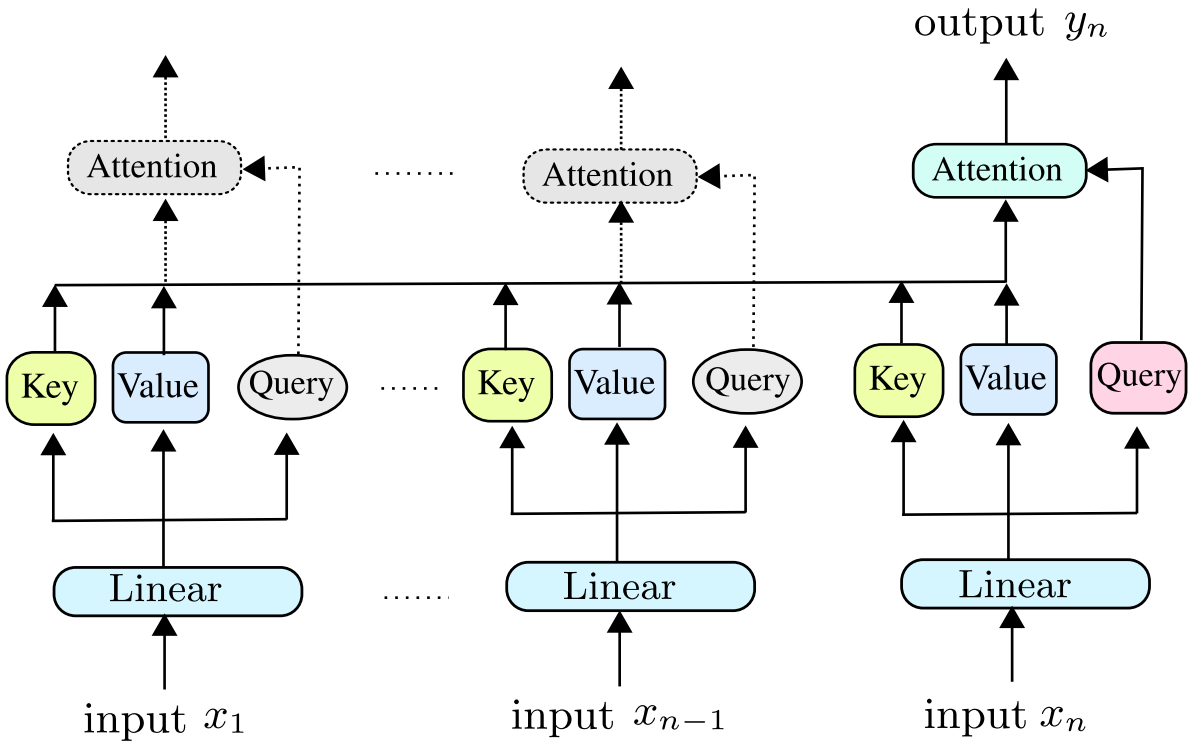
- The attention mechanism can be varied in many ways.
- Major simplification: the scalar product between the decoder state and the annotation can be used as similarity score (dot-attention). The extra subnetwork which computes the attention is often not necessary.
- A very flexible approach divides the annotation vectors into keys and values, and derives queries from the decoder state.
- Between each key vector and the query, a *similarity score* is computed, and these scores (after normalization) are used to compute the weighted average of the values.
- In Bahdanau's classical method, we used the *annotation* simultaneously as key and value, and the similarity score is computed by a neural network.
- Multi-head attention: Carry out multiple attention operations using separate parameters, and concatenate their results.

Advanced topic: The Transformer Architecture

Self-Attention

- Can we build a general purpose sequence processing layer based on attention?
- Basic idea: Take an N-step input sequence.
  - ➔ Input of each step transformed in 3 ways: key, value, and query vectors. This transformation is learned during training.
  - ➔ For any step n:
    - ➔ Match key and query over *all* steps (dot product style).
    - ➔ Compute softmax to obtain normalized weights
    - ➔ Compute output at step n as weighted sum over values, with normalized softmax weights.
  - ➔ This concept is known as self-attention.
  - ➔ The keys and values form the memory of the layer.
  - ➔ The memory grows with the size of the input sequence.
- Note that no recurrence is involved - can entirely be computed in parallel.

Self-Attention



Self-attention in detail

- Consider a sequence of vectors $(x_1, \ldots, x_N)$:
- At step n, input $x_n \in \mathbb{R}^D$ is first transformed into *three* vectors:

$$q_n, k_n, v_n = Qx_n, Kx_n, Vx_n \qquad Q, K \in \mathbb{R}^{d_{\text{key}} \times D}, V \in \mathbb{R}^{d_{\text{value}} \times D}$$

  Q, K, V are the trainable parameters of the layer.
- Compute alignments for position n by matching query q_n to all keys:

$$\alpha_{nm} = \text{softmax}\left(\left\{\frac{q_n \cdot k_m}{\sqrt{d_{\text{key}}}}\right\}_m\right)$$

  (the rescaling avoids too high variance in the alignments).
- Output y_n is computed as the weighted sum of values, where the weights are given by the alignment:

$$y_n = \sum_m \alpha_{nm} v_m$$

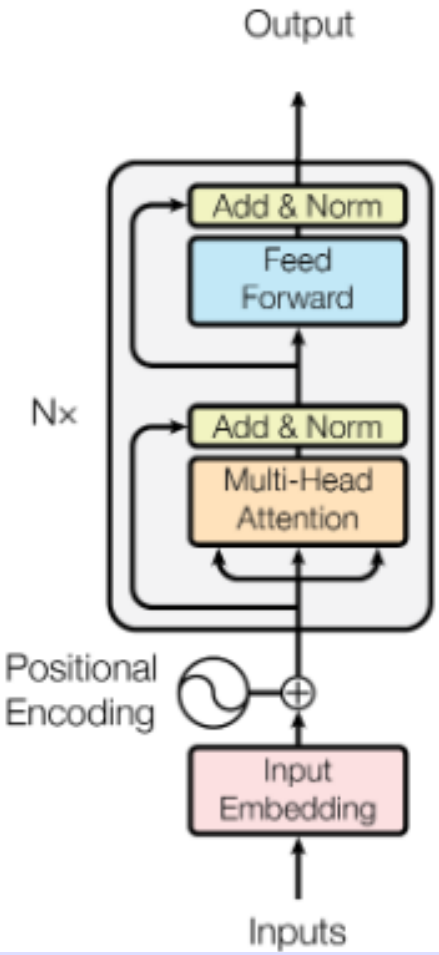
Transformer

- The Transformer (Vaswani et al. 2017) is a model with multiple layers. Each Transformer layer contains:
  - ➔ one self-attention layer
  - ➔ one feed-forward layer.

  with residual connection and layer normalization (methods to facilitate training) between each component.
- Made self-attention very popular. Latest large advancement in neural network architecture (2017).
- You will be likely using a Transformer layer instead of self-attention layer alone.
- Frequently uses multiple independent attention heads.

Transformer

- Graphical overview of transformer with multiple transformer layers.
- Each input frame is encoded into one output frame.
- One element of the system has not yet been covered: positional encoding.
- Required because unlike the RNN, in the transformer architecture we have no implicit information about the order of frames!



Positional encoding

- Positional information (frame indices) needs to be provided explicitly!
  - ➔ Can we simply count positions (1,2,...) and add these values as a component to the input?
  - ➔ Problems: bad normalization, does not generalize well
  - ➔ Divide count by maximum value (e.g. 1,2,3,4 $\rightarrow$ 0.25,0.5,0.75,1.0)
  - ➔ Problem: Does not work well for different sequence lengths
  - ➔ Idea: count in *binary* numbers, then the numeric range is between 0 and 1 (-1 to 1 can be arranged)
  - ➔ This means that we add multiple components to the input vector (e.g. maximum sequence length 1024 $\rightarrow$ 10 components)
  - ➔ Still problematic: the representation is not continuous, positions cannot be interpolated (e.g. the middle between (0,0,0) and (1,0,0) is (0,1,0)).

Positional encoding

- Idea: use the sine/cosine wave at different frequencies as a continuous way to encode positions!
- Sinusoidal positional encoding: Position i is represented by a vector e_i of dimension D where each component is computed, for $0 \leq k < \frac{D}{2}$:

$$e_{i,2k} = \sin(i/10000^{2k/D})$$
$$e_{i,2k+1} = \cos(i/10000^{2k/D})$$

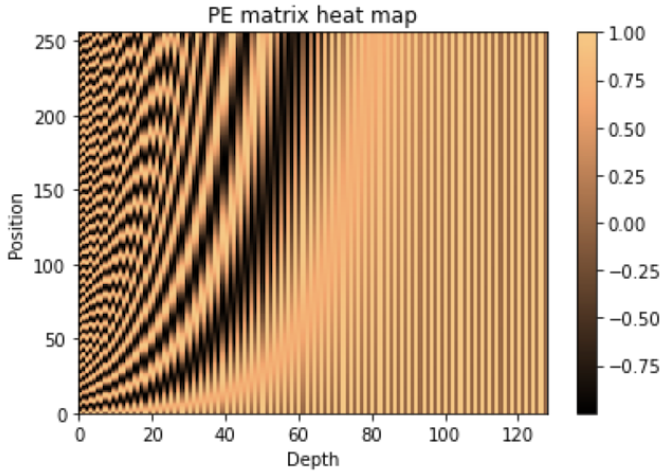


Image source: `https://towardsdatascience.com/master-positional-encoding-part-i-63c05d90a0c3`

Positional encoding

- Idea: use the sine/cosine wave at different frequencies as a continuous way to encode positions!
- Sinusoidal positional encoding: Position i is represented by a vector e_i of dimension D where each component is computed, for $0 \leq k < \frac{D}{2}$:

$$e_{i,2k} = \sin(i/10000^{2k/D})$$
$$e_{i,2k+1} = \cos(i/10000^{2k/D})$$

- Choice of 10000: a "large" number.
- Standard approach: D is the input dimensionality, and the positional encoding is *added* to the input
- Studying and improving positional encoding is still a hot research topic!

Transformer for Nonaligned Sequences

- What if the input samples and output targets are not aligned?
- Same idea as with the LSTM: Use an *encoder*, a *decoder*, and (multi-head) attention to connect these two modules.
- Decoder works in autoregressive mode.
- Self-attention in the decoder attends only to *past* tokens (i.e. tokens which have already been output).

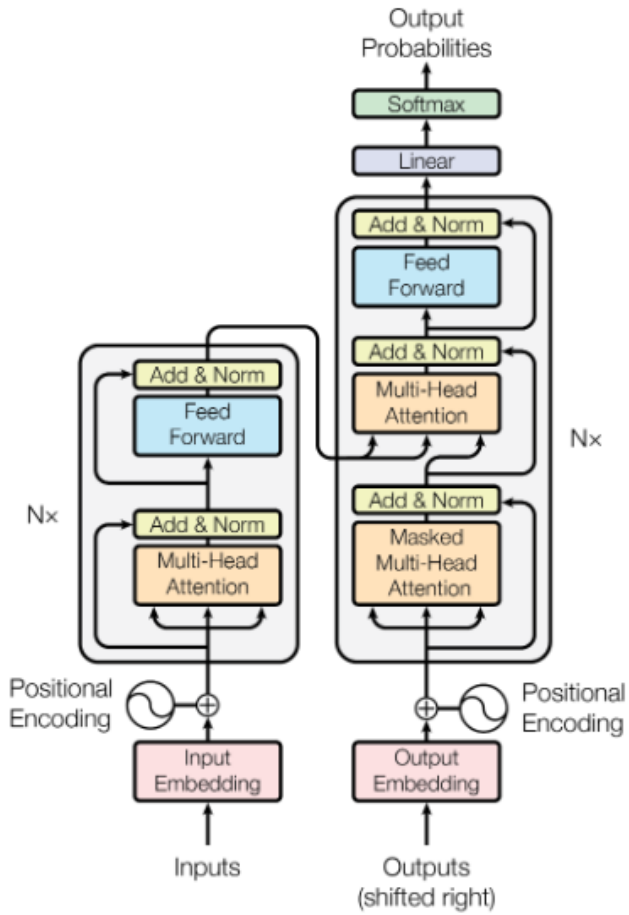


Figure 1: The Transformer - model architecture.

Image from Vaswani et al. (2017), *Attention is all you need*

Summary and Conclusion

- You have learned about two fundamental NN paradigms: *recurrence* and *attention*.
- Recurrence is a straightforward way to deal with sequences, but causes long *paths* between inputs and outputs, leading to *Vanishing Gradients*.
- The LSTM elegantly solves the vanishing gradient problem.
- The encoder-attention-decoder architecture likewise elegantly solves the problem of matching inputs and outputs, and creates short paths between inputs and outputs.
- We have also gotten to know the *Transformer*, as of now the most recent groundbreaking development in NN architecture.

Finally, remember *Beam Search* as a very common algorithm for searching through sequences; it appears in many forms and has applications also beyond neural networks.