

Deep Learning for NLP

The lecture
starts at 13:15

Florina Piroi

What we did last week

- Vector Semantics & Embeddings
 - Lexical and Vector Semantics
 - Words as Vectors
 - Measuring similarity & tf-idf
 - Word2Vec
- Neural Networks
 - Perceptron, units, activation functions
 - Feed forward
 - Training
- Neural Language Models

Contents

- Neural Language Models
- Recurrent Neural Networks
- LSTMs (Long Short-Term Memory Networks)

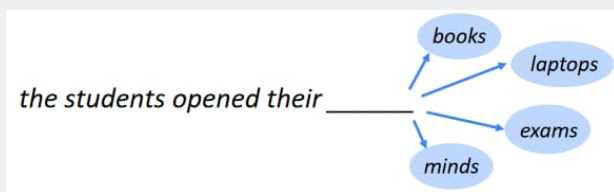
Neural Language Models

Relevant Literature

- Jurafsky & Martin, SLP, 3rd Edition: *Chapters 7, 9*
 - (including slides), references therein
- Cho, 2017, NLU with Distributional Representation, Chapters 4, 5
- Other material listed on individual slides

What is a “Language Model”?

- A model that predicts $P(W)$ or $P(w_n | w_1, w_2 \dots w_{n-1})$
- Probabilistic Language Models
 - Compare probabilities of sequences of words
 - Probability of upcoming word



A language model is a model that predicts the next word in a sequence based on the prior word context

In probabilistic language models you would compute the probability of a sentence, or a sequence of words, and starting from that, depending on your task, you would either compare probabilities of sequence of words (if you look into machine translation) or compute the probability of an upcoming word

What is a “Language Model”?

- A model that predicts $P(W)$ or $P(w_n | w_1, w_2 \dots w_{n-1})$
- Probabilistic Language Models
 - Compare probabilities of sequence of words
 - Probability of upcoming word

- How did you compute P ?

- Count and divide
- Markov Assumption

$$P(\text{the} | \text{its water is so transparent that}) = \frac{\text{Count}(\text{its water is so transparent that the})}{\text{Count}(\text{its water is so transparent that})}$$

$$P(\text{the} | \text{its water is so transparent that}) \approx P(\text{the} | \text{that})$$

$$P(\text{the} | \text{its water is so transparent that}) \approx P(\text{the} | \text{transparent that})$$

How do you compute these probabilities?

-> count the probability of any given word following a sequence of given words, and divide by the total number of possible combinations, for example. But that would give you way too many things to count, because combining words is an endless story.

So we used the simplifying Markov Assumption that we approximate the probabilities we are looking for at simpler ones, and simpler especially in a computational meaning.

What is a “Language Model”?

- A model that predicts $P(W)$ or $P(w_n | w_1, w_2 \dots w_{n-1})$
- Probabilistic Language Models
 - Compare probabilities of sequence of words
 - Probability of upcoming word
- How did you compute P ?
 - Count and divide
 - Markov Assumption
- Unigrams
- Bi-grams
- ...
- N-grams

And we using n-grams we would approximate the probability of a word in a sequence by the product of the probabilities of each words in that sentence.

Language Model: A simple example

$$P(w_i | w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

<s> I am Sam </s>

<s> Sam I am </s>

<s> I do not like green eggs and ham </s>

Symbols for the start and end
of a sentence

$$P(\text{I} | \text{<s>}) = \frac{2}{3} = .67$$

$$P(\text{Sam} | \text{<s>}) = \frac{1}{3} = .33$$

$$P(\text{am} | \text{I}) = \frac{2}{3} = .67$$

$$P(\text{</s>} | \text{Sam}) = \frac{1}{2} = 0.5$$

$$P(\text{Sam} | \text{am}) = \frac{1}{2} = .5$$

$$P(\text{do} | \text{I}) = \frac{1}{3} = .33$$

Very simple LM extracted from these three sentences.

$$P(\text{I} | \text{<s>}) = 2/3$$

$$P(\text{Sam} | \text{<s>}) = 1/3$$

$$P(\text{am} | \text{I}) = 2/3$$

$$P(\text{</s>} | \text{Sam}) = 1/2$$

$$P(\text{Sam} | \text{am}) = 1/2$$

$$P(\text{do} | \text{I}) = 1/3$$

Recall: Language Model

- A model that predicts $P(W)$ or $P(w_n | w_1, w_2 \dots w_{n-1})$
- Probabilistic Language Models
 - Compare probabilities of sequence of words
 - Probability of upcoming word

Perplexity

$$PP(W) = \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i | w_1 \dots w_{i-1})}}$$

- How did you compute P ?
 - Count and divide
 - Markov Assumption
 - Issues: zero probabilities, smoothing, interpolation
- Unigrams
 - Bi-grams
 - ...
 - N-grams

Perplexity measure: used to assess how good a language model performs.

But there were still a few issues with these LMs:

- zero probabilities which would ruin your computations, you could smoothen them out, you could use some interpolation ...
- all extra operations on top on counting n-grams.

So there are quite a few issues with the usual way of counting n-grams to obtain a language model. Can we do, under certain circumstances better?

Neural Language Model

- No smoothing
 - Longer histories (compared to the fixed N in "N-gram")
 - Generalize over contexts
 - "chases a dog" vs. "chases a cat" vs. "chases a rabbit"
 - Higher predictive accuracy!
 - Further models are based on NLMs.
- Slower to train!

Yes: We can compute Language Models using Neural Networks -> Neural Language Models.

A NLM turns out to have many advantages, in comparison:

generalize: "chases a cat" vs. "chases a dog" vs. "chases a rabbit" here's a clear pattern, but an n-gram model would not detect it if "chases a rabbit" does not occur in your data.

For many tasks, n-grams are actually still a very good choice, as NLMs are slow to train.

Neural Language Model - Definition

- Standard Feed-Forward Network
- Input: a representation of previous words (w_1, w_2, \dots)
- Output: probability distribution over possible next words.

$$P(w_n | w_1, w_2 \dots w_{n-1}) = f_{\theta}^{w_n}(w_1, w_2 \dots w_{n-1})$$

i.e.: find the function

In the end we actually want to find a function that takes as input $n-1$ words and returns a conditional probability of a next word

Neural Language Model - Input

- Standard Feed-Forward Network
- **Input:** a representation of previous words (w_1, w_2, \dots)
- Output: probability distribution over possible next words.
- N-grams used exact words! ($P(\text{"cat"})$)
- Equi-distance! (lowest prior knowledge)
- 1-of-N encoding (aka. one-hot vector)

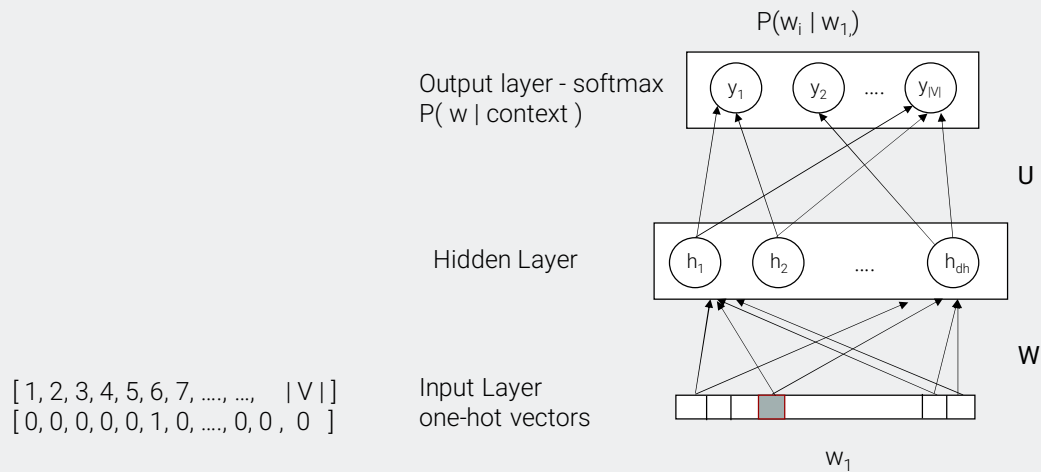
```
[ 1, 2, 3, 4, 5, 6, 7, ..., ..., | V | ]  
[ 0, 0, 0, 0, 0, 1, 0, ..., 0, 0, 0 ]
```

How do we represent the input to a NLM?

If we want to use the lowest amount of prior knowledge we will need to represent each word such that each and every word is equi-distant from the other!

Orthogonality!

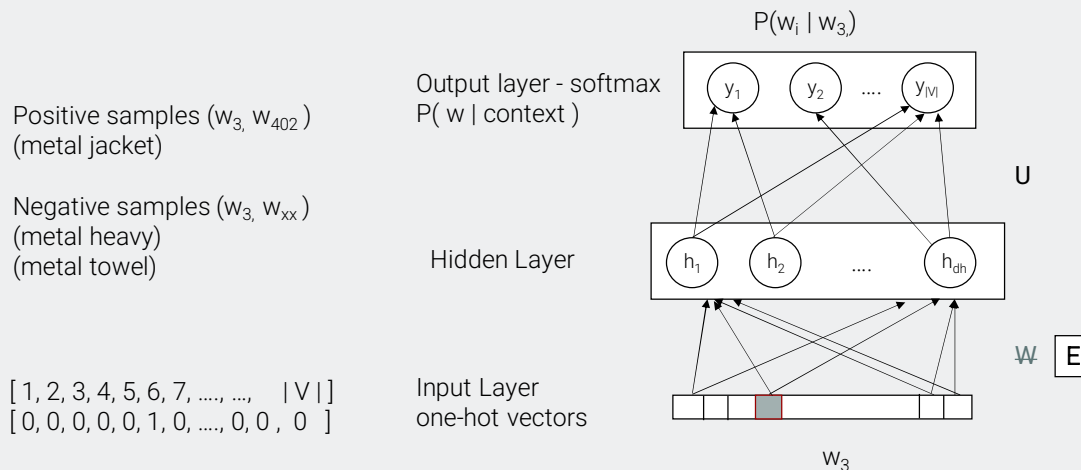
Feed Forward Net - Execution



We want to have such a FF network that is able to compute a probability distribution for the input sequence.

Gives you a probability distribution over all words in our vocabulary (output layer has $|V|$ nodes)

Feed Forward Net - Training



TU Informatics

The network is fully connected

As with any NN, we use a loss function, a gradient descent we compute, some back propagation ...

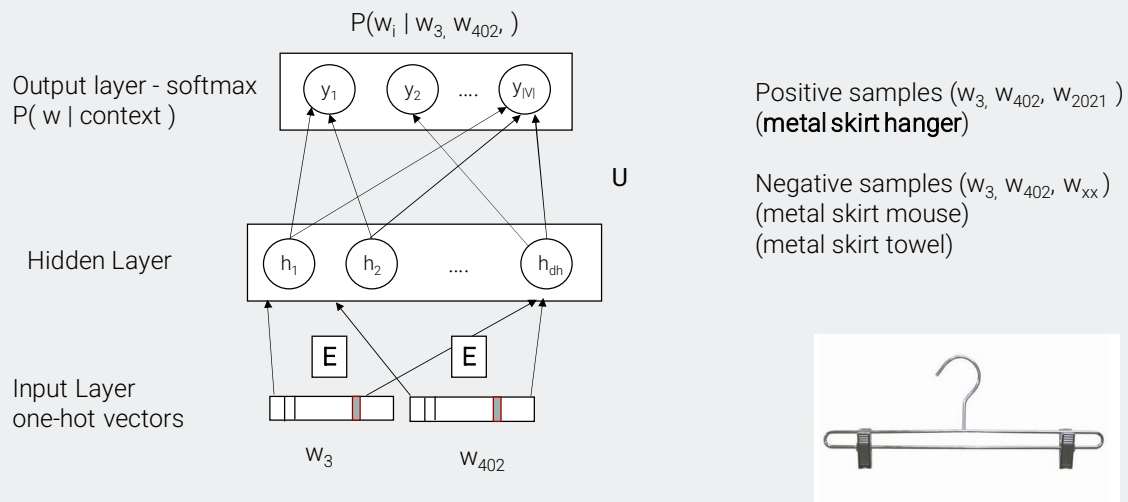
We are learning the weight matrices W and U .

If at the end of the training we throw everything away, but we keep only W , this matrix represent the word embeddings

And let's call this matrix E , for embeddings.

This is a very simple model – a language model for bigrams – that, besides being trained to predict the next word of a sequence of two words, it learns, as a side effect, the embeddings of our words.

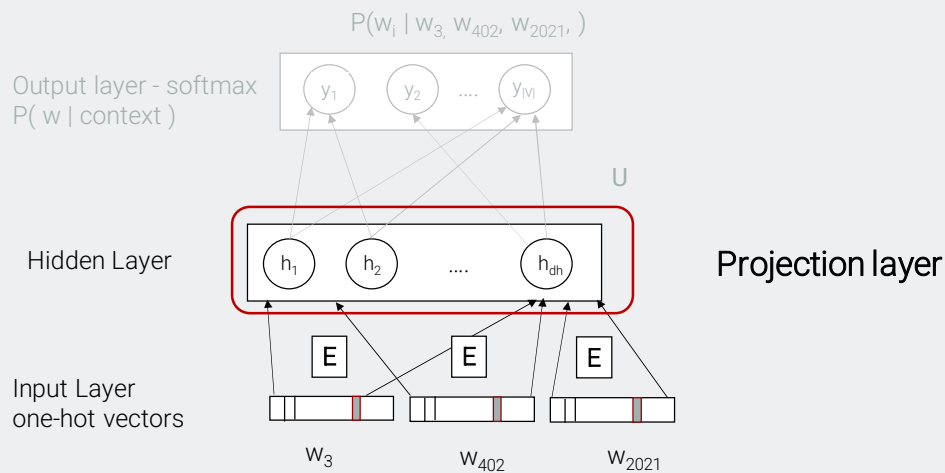
Feed Forward Net - Training



Here is how you would train a network to predict the last word in a sequence of three words.

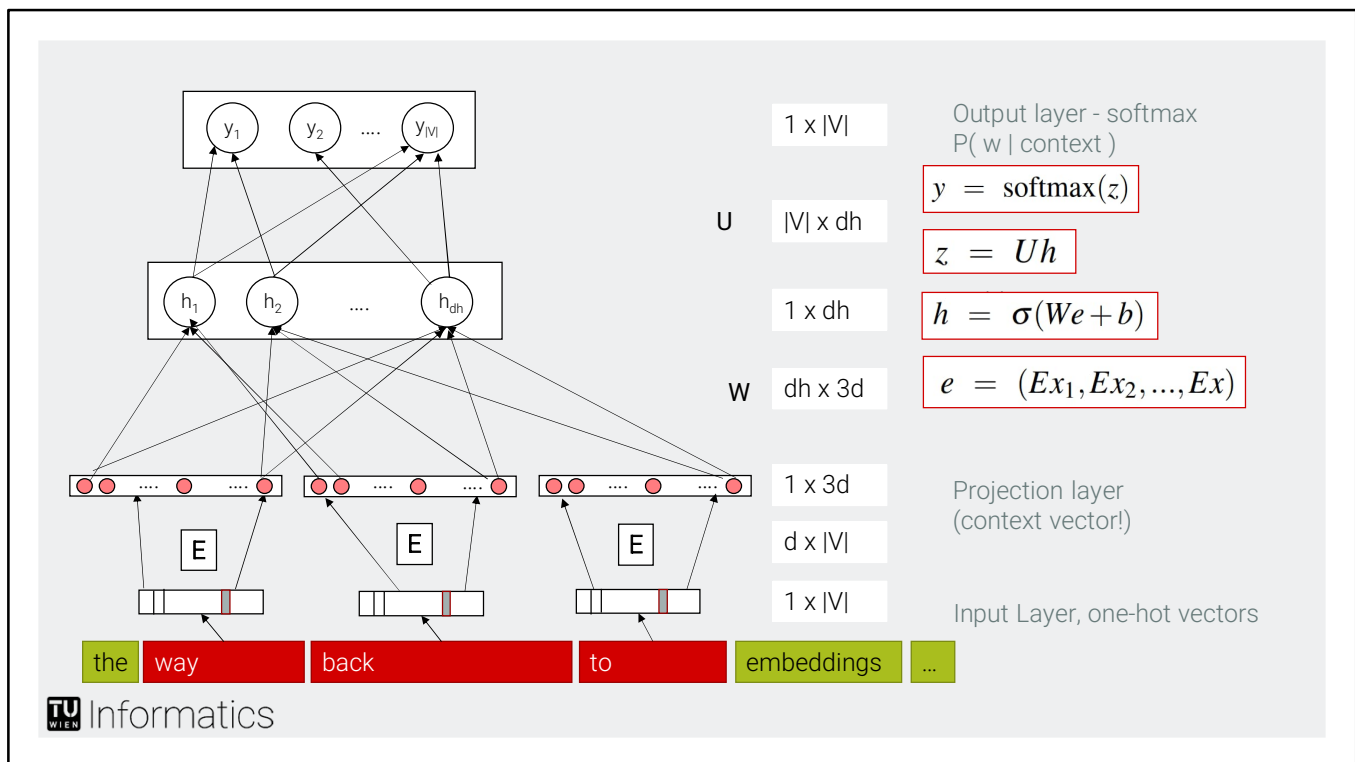
Note that the matrix E is shared between the two input vectors, so, and we also have full connections between the one-hot input vectors and the hidden layers.

Feed Forward Net - Training



Throwing away everything and keeping only E we have the word embeddings for context windows of 3 (one word ± 1)

The hidden layer which is used to extract, at the end, the single embedded vectors for the input words is called a projection layer.



How to USE existing Embeddings? (E)

Look at the forward pass of the network

1. Select three embeddings from E:

Given the three previous words, we look up their indices, create 3 one-hot vectors, and then multiply each by the embedding matrix E.

The first word in the sequence will give this part of the first hidden layer (the projection layer) Since each row of the input matrix E is just an embedding for a word, and the input is a one-hot column vector x_i for word V_i , the projection layer for input w will be $Ex_i = e_i$, the embedding for word i .

We now concatenate the three embeddings for the context words.

2. Multiply by W:

We now multiply by W (and add b - bias) and pass through the rectified linear (or other) activation function to get the hidden layer h .

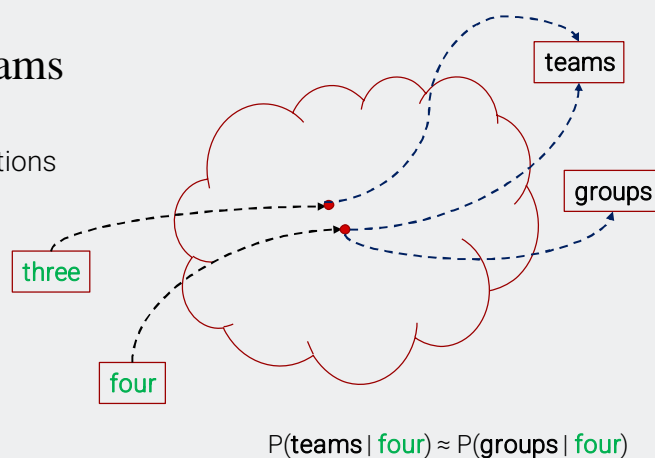
3. Multiply by U

4. Apply softmax: After the softmax, each node i in the output layer estimates the

probability $P(w_t = i \mid w_{t-1}, w_{t-2}, w_{t-3})$

Generalization to Unseen n-grams

- There are **three** teams left for the qualifications
- **four** teams have passed the first round
- **four** groups are playing in the field



How do embeddings deal with out of vocabulary occurrences. While searching for such OOV occurrences in the corpus will give you an empty set (in a classic IR-like setting, with n-gram or counts over doc-term matrixes), a NLM will be able to give you a prediction on the likelihood of such an OOV being a valid one. I.e. how can we generalize to unseen things.

Let's look at how Neural Language Models generalize to unseen n-grams, which is problem similar to the out of vocabulary one.

This is an artificially built example to help understand how this generalization happens. In this example we do bigram modelling with only these three sentences.

We will focus on only these bold-faced phrases: "three teams", "four teams" and "four group". The first word of each of these bigrams is a context word, and the neural language model is asked to compute the probability of the word following the context word.

It is important to notice that the model *must* project "three" and "four" to nearby points in the context space

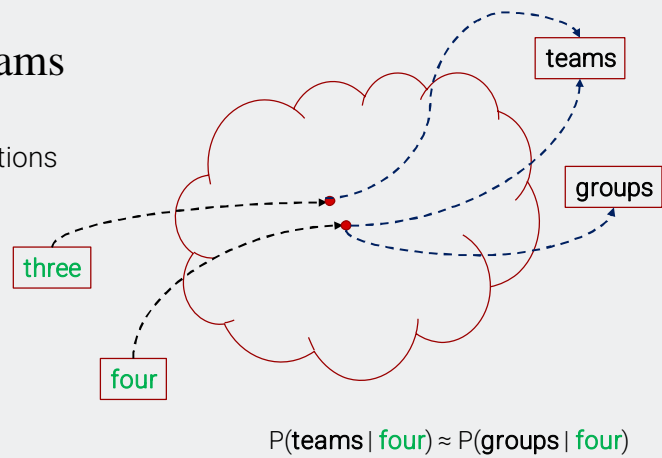
This is because the context vectors from these two words need to give a similar probability to the word "teams"

And, from these two context vectors (which are close to each other), the model assigns similar probabilities to "teams" and "groups", because they occur in the training corpus.

In other words, the target word vector for the word "teams" and "groups" will also be similar to each other, because otherwise the probability of "teams" given "four" ($p(\text{teams} \mid \text{four})$) and "groups" given "four" ($p(\text{groups} \mid \text{four})$) will be very different despite the fact that they occurred equally likely in the training corpus.

Generalization to Unseen n-grams

- There are **three** teams left for the qualifications
 - **four** teams have passed the first round
 - **four** groups are playing in the field
- Assign probability to “three groups”



Let's use the neural language model trained on this tiny training corpus to assign a probability to an *unseen* bigram “three groups”.

The neural language model will project the context word “three” to a point in the context space close to the point of “four”. From this context vector, the neural language model will have to assign a high probability to the word “groups”, because the context vector for three and the target word vector for groups well align.

Thereby, even without ever seeing the bigram “three groups”, the neural language model can assign a reasonable probability to this bigram.

Neural Language Models – In a small nutshell

- pattern recognition problems
- Data-driven
- High performance in many problems
- No domain knowledge needed
- Generalization

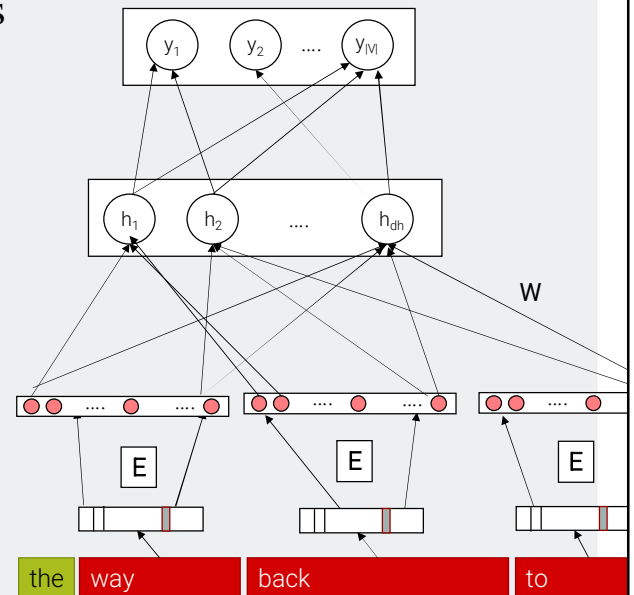
- Data-hungry (bad for small data sets)
- Cannot handle symbols very well
- Computationally high costs

Content

- Neural Language Models
- Recurrent Neural Networks
- LSTMs (Long Short-Term Memory Networks)

(Simple) Neural Language Models

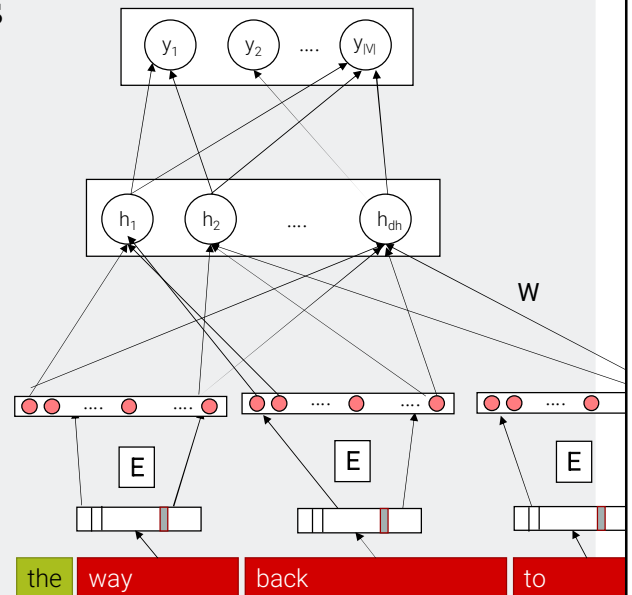
- Improvements over n-gram LM
 - No sparsity problem
 - Don't need to store all observed n-grams
- Remaining problems:
 - Fixed window is too small
 - Enlarging window enlarges W
 - Window can never be large enough!
 - (embedded) words are multiplied by completely different weights in W
(No symmetry in input processing)



Let's recall what we fixed with the NLM solution

(Simple) Neural Language Models

- How to deal with inputs of varying lengths (i.e. sequences)?
- Slide the input window
- Still, decision on one window does not influence decision on other window.
- Cannot learn systematic patterns (e.g. Constituency)



Very Importantly, the decision made for one window has no impact on later decisions anything outside the context window has no impact on the decision being made.

This is an issue since there are many overlapping windows, deciding on them being not influenced by the other overlapping window – we'd like that.

Second, the use of windows makes it difficult for networks to learn systematic patterns arising from phenomena like constituency.

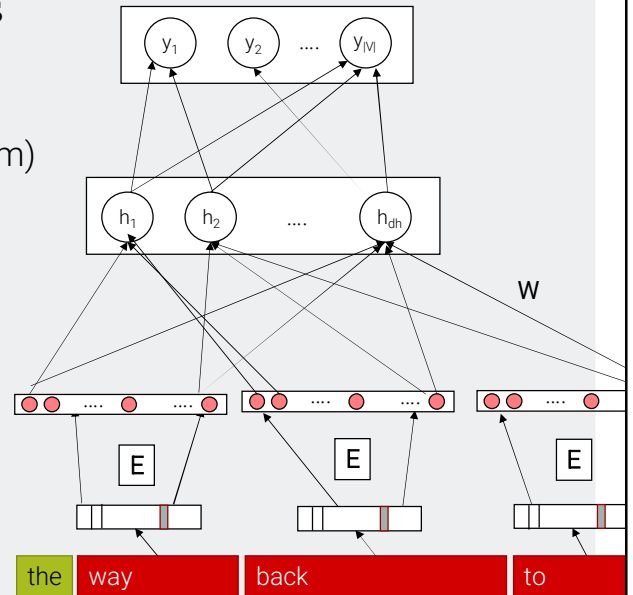
Language tasks that require access to information that can be arbitrarily distant from the point at which processing is happening.

For example, in this figure the noun phrase "the way" appears in two separate windows: once, as shown, in the first and second positions in the window, and in the preceding step where it appears in the second and third positions, thus forcing the network to learn two separate patterns for what should be a constituent.

And one answer to handling these problems are Recurrent Neural Networks

(Simple) Neural Language Models

- Language is temporal (continuous stream)
 - "Sequence that unfolds in time"
- Algorithms use this
 - Viterbi
- Previous ML approaches have access to all input, simultaneously
- How to deal with *sequences of varying lengths*?



When applied to the problem of part-of-speech tagging, the Viterbi algorithm works its way incrementally through its input a word at a time, taking into account information gleaned along the way.

The neural network introduced for LM operates on fixed-sized windows, and when you have input sequences of different length, it will slide the input window along the sequence.

Sequences – Input of **Variable Lengths**

$$x^1 = (x_1^1, x_2^1, \dots, x_{l^1}^1)$$

- Each input has a variable number of elements:
- Simplification: binary elements (0 or 1 values)
- How many 1s in this sequence? How can we implement that?
- ADD1, Recursive function
- Call it for each element of the input.

Algorithm 1 A function ADD1

```
s ← 0
function ADD1(v,s)
  if v = 0 then return s
  else return s + 1
  end if
end function
```

Algorithm 2 A function ADD1

```
s ← 0
for i ← 1, 2, ..., l do s ← ADD1(xi, s)
end for
```

A variable length input x is a sequence where each input x has a different number of elements.

For instance, the first training example's input x_1 may consist of l_1 elements like this, while another input instance x_n may have l_n elements, with l_n different from l_1

Let's go back to very basic about dealing with these kinds of sequences. Let us assume that each element x_i is binary, meaning that it is either 0 or 1.

What would be the most natural way to write a function that returns the number of 1's in an input sequence.

Answer: first build a recursive function called ADD1, shown in Alg. 1. This function ADD1 will be called for *each element* of the input x , as in Alg. 2.

There are two important components in this implementation, which are kind of relevant for RNNs.

First, there is a *memory* s which counts or collects the number of 1's in the input

sequence x.

Second, a single function ADD1 is applied to each symbol in the sequence one at a time AND together with the memory s!

These two properties make it possible for the function ADD1 to be applied to sequences of ANY length.

Recursive Function for Natural Language Understanding

- ADD1 is hardcoded
- Parametrized recursive function
- Memory: $\mathbf{h} \in \mathbb{R}^{d_h}$
- Input x_{t-1} and memory \mathbf{h} , returns the new \mathbf{h}
- Time index!

$$h_t = f(x_t, \mathbf{h}_{t-1})$$

$$f(x_t, \mathbf{h}_{t-1}) = g(\mathbf{W}\phi(x_t) + \mathbf{U}\mathbf{h}_{t-1})$$

Algorithm 1 A function ADD1

```
s ← 0
function ADD1(v,s)
  if v = 0 then return s
  else return s + 1
end if
end function
```

Algorithm 2 A function ADD1

```
s ← 0
for i ← 1, 2, ..., l do s ← ADD1(xi, s)
end for
```

ADD1 is hard coded in the sense that this function does what it has been designed to do.

But: cannot hard code a function that understands natural language

We want: a parametric recursive function that is able to read a sequence of (linguistic) symbols and use a memory in order to understand natural languages.

To build this parametrised function, we know now that there needs to be a memory.

As is clear from Alg. 1, this recursive function takes as input both one input symbol x_t and the memory vector \mathbf{h} , and it returns the updated memory vector. It often helps to time index the memory vector as well, such that the input to this function is $\mathbf{h}_{(t-1)}$ (the memory after processing the previous symbol $x_{(t-1)}$) and we use \mathbf{h}_t to denote the memory vector returned by the function (first formula).

What happened here? Well, the big question was what *kind* of parametric form this recursive function f takes? What kind of parametric functions do we know of? We can follow the simple transformation layer we've seen in the Feed Forward NNs:

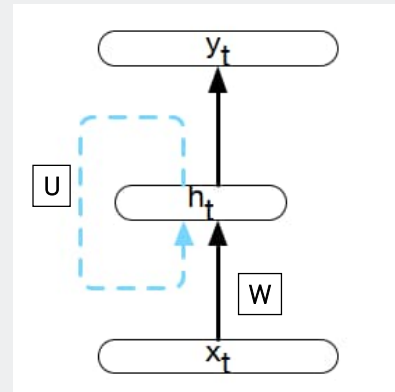
Where $\phi(x_t)$ is a function that transforms the input symbol (often discrete) into a d -dimensional real-valued vector. That would be your embedding layer. W and U are parameters of this function, and I hope you recognize here the weight matrixes. A nonlinear activation function g can be any function, let's take it to be a non-linear, element-wise function as hyperbolic tangent.

Recursive Function for Natural Language Understanding

$$\mathbf{h} \in \mathbb{R}^{d_h}$$

$$h_t = f(x_t, \mathbf{h}_{t-1})$$

$$f(x_t, \mathbf{h}_{t-1}) = g(\mathbf{W}\phi(x_t) + \mathbf{U}\mathbf{h}_{t-1})$$



Here is a simple structure of an RNN, also called an Elman Network, where the time-index is not marked, but where the recursiveness is outlined.

As with ordinary feedforward networks, an input vector representing the current input element, x_t , is multiplied by a weight matrix and then passed through an activation function to compute an activation value for a layer of hidden units.

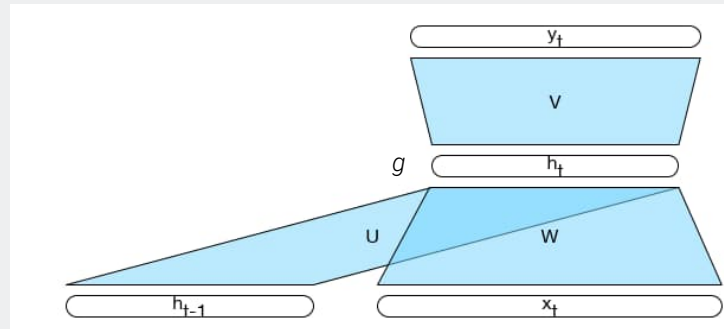
This hidden layer is, in turn, used to calculate a corresponding output, y_t . Sequences are processed by presenting one element at a time to the network. The recurrent link here augments the input to the computation at the hidden layer with the activation value of the hidden layer from the preceding point in time.

The hidden layer from the previous time step provides the memory, or context, that encodes earlier processing (or the number of 1s in the sequence) and informs the decisions to be made at later points in time. Critically, this architecture does not impose a fixed-length limit on this prior context; the context embodied in the previous hidden layer includes information extending back to the beginning of the sequence.

Recursive Neural Network – Unrolled

$$h_t = g(Uh_{t-1} + Wx_t)$$
$$y_t = f(Vh_t)$$

$$y_t = \text{softmax}(Vh_t)$$



Adding this temporal dimension may make RNNs appear to be more exotic than non-recurrent architectures. But in reality, they're not all that different. Given an input vector and the values for the hidden layer from the previous time step, we're still performing the standard feedforward calculation, which is visible in this picture.

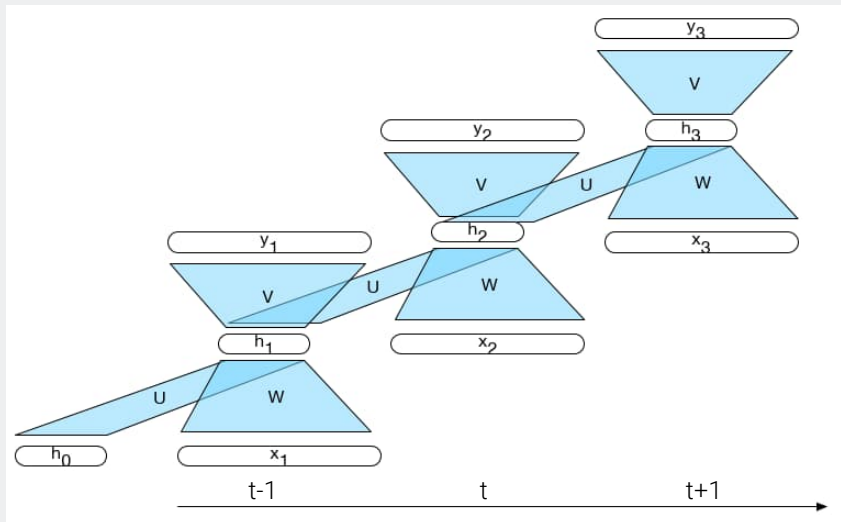
Compared with a FeedForward NN, The most significant change lies in the new set of weights, U which we've seen in the equation we derived, that connect the hidden layer from the previous time step to the current hidden layer. These weights determine how the network should make use of past context in calculating the output for the current input. As with the other weights in the network, these connections are trained via backpropagation

How do we do inferencing in such a network (provided we have had it trained)?

This is done in the usual way, by multiplying the input with the weight matrix W , and multiplying the previous step hidden layer with U , adding them, and passing them through an activation function, g . Once we have these values, we have the output vector.

And softmax gives us a normalized probability distribution over possible classes (when we apply this network to a classification problem).

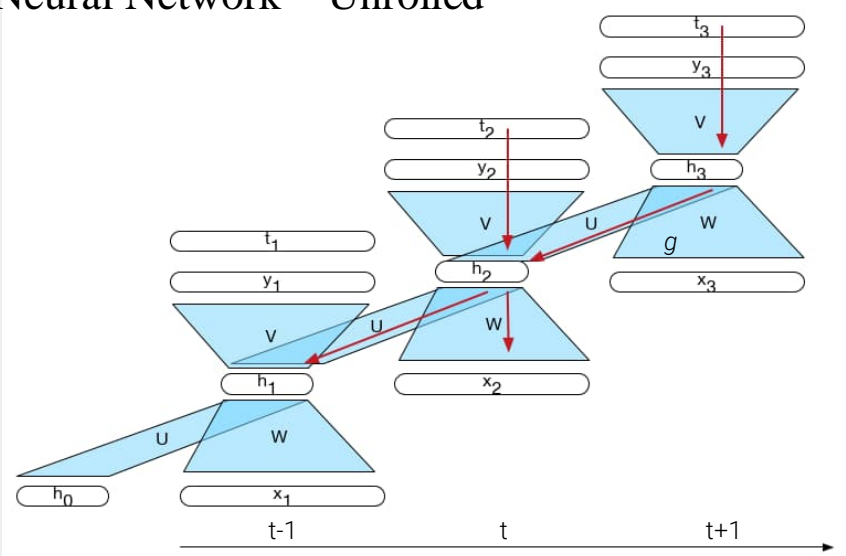
Recursive Neural Network – Unrolled



The sequential nature of the simple RNN can be evidenced by “Unrolling” the network in time.

It is important to remember that the matrixes are shared across time.

Recursive Neural Network – Unrolled



Training is done through backpropagation.

The backpropagation of errors in a simple RNN t_i vectors represent the targets for each element of the sequence from the training data. The red arrows illustrate the flow of backpropagated errors required to calculate the gradients for U , V and W at time 2. The two incoming arrows converging on h_2 signal that these errors need to be summed.

RNN – Applications

- RNN Language Models
 - (Autoregressive) generation
- Sequence labelling
- Sequence classification
- ...

RNN – Language Models

- N-gram and FF models
 - Fixed sliding window, i.e. fixed context.

$$P(w_n | w_1^{n-1})$$

- Quality of prediction largely dependent on the size of the window
- Constrained by the Markov assumption

$$P(w_n | w_1^{n-1}) \approx P(w_n | w_{n-N+1}^{n-1})$$

- Limitation is avoided in RNN!

We've seen earlier the N-gram and the FF net models, and how they process text sequences by the use of a sliding window, which has those unhappy effects for some language tasks.

In both approaches, the quality of a model is largely dependent on the size of the context and how effectively the model makes use of it. Thus, both N-gram and sliding-window neural networks are constrained by the Markov assumption embodied in the following equation.

That is, anything outside the preceding context of size N has no bearing on the computation.

This limitation is avoided when using RNNs, as they process sequences a word at a time attempting to predict the next word in a sequence by using the current word and the previous hidden state as input. This is because the hidden state embodies information from all preceding words, all the way back.

RNN – Language Models

- Limitation is avoided in RNN!

$$\begin{aligned} P(w_n | w_1^{n-1}) &= y_n \\ &= \text{softmax}(Vh_n) \end{aligned}$$

$$\begin{aligned} P(w_1^n) &= \prod_{k=1}^n P(w_k | w_1^{k-1}) \\ &= \prod_{k=1}^n y_k \end{aligned}$$

- Cross-entropy function for training

$$\begin{aligned} L_{CE}(\hat{y}, y) &= -\log \hat{y}_i \\ &= -\log \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \end{aligned}$$

- Perplexity for evaluation

$$PP(W) = \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i | w_{i-1})}}$$

As at each step the network retrieves a word embedding for the current word as input and combines it with the hidden layer from the previous step to compute a new hidden layer. This hidden layer is then used to generate an output layer which is passed through a softmax layer to generate a probability distribution over the entire vocabulary.

And the probability of an entire sequence is then just the product of the probabilities of each item in the sequence. – so note that the probability of the sequence includes now information from all prior words in the sequence

Training Phase:

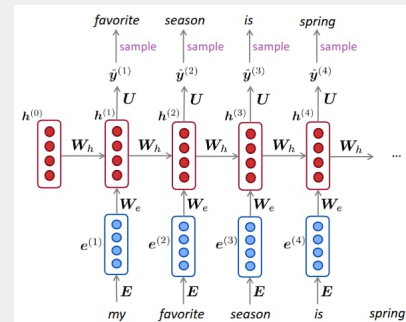
Recall that the cross-entropy loss for a single example is the negative log probability assigned to the correct class, which is the result of applying a softmax to the final output layer.

Here, the correct class i is the word that actually comes next in the data and y_i is the probability assigned to that word

The weights in the network are adjusted to minimize the cross-entropy loss over the training set via gradient descent.

RNN – Language Models

- Generate text by repeated sampling

























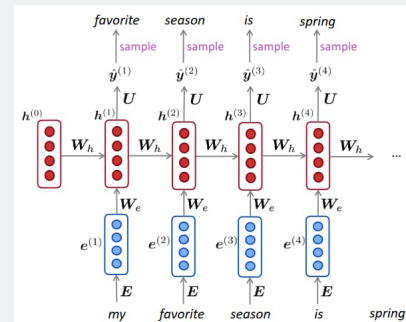
- RNN-LM trained on Obama speeches

The United States will step up to the cost of a new challenges of the American people that will share the fact that we created the problem. They were attacked and so that they have to say that all the task of the final days of war that I will not be able to get this done.

RNN – Language Models

- Generate text by repeated sampling
 - On any kind of text!
 - Character level example

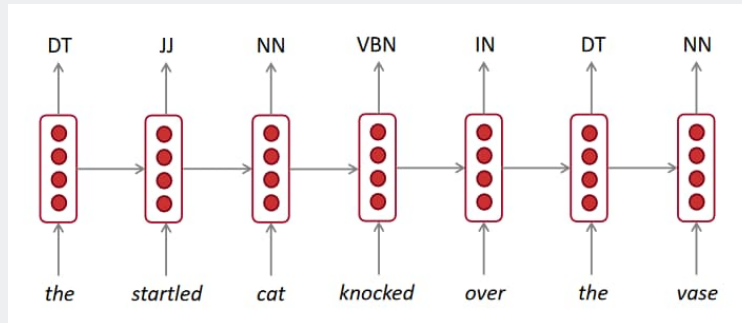
	Ghasty Pink 231 137 165		Sand Dan 201 172 143
	Power Gray 151 124 112		Grade Bat 48 94 83
	Navel Tan 199 173 140		Light Of Blast 175 150 147
	Bock Coe White 221 215 236		Grass Bat 176 99 108
	Horble Gray 178 181 196		Sindis Poop 204 205 194
	Homestar Brown 133 104 85		Dope 219 209 179
	Snader Brown 144 106 74		Testing 156 101 106
	Golder Craam 237 217 177		Stoner Blue 152 165 159
	Hurky White 232 223 215		Burbie Simp 226 181 132
	Burf Pink 223 173 179		Stanky Bean 197 162 171
	Rose Hork 230 215 198		Turdly 190 164 116



On the left you see the training data, on the right are the colours “invented” by the LM.

RNN – Applications

- Tagging (POS, named entity recognition, IOB encoding etc.)

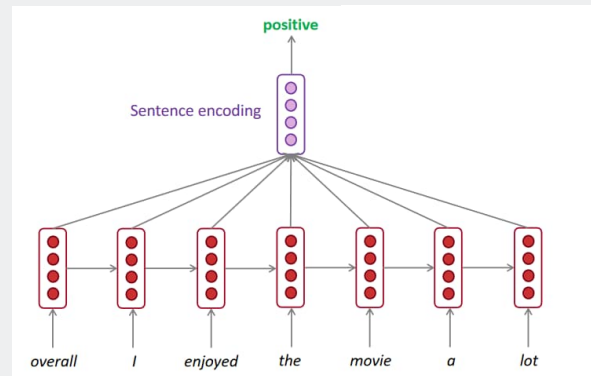
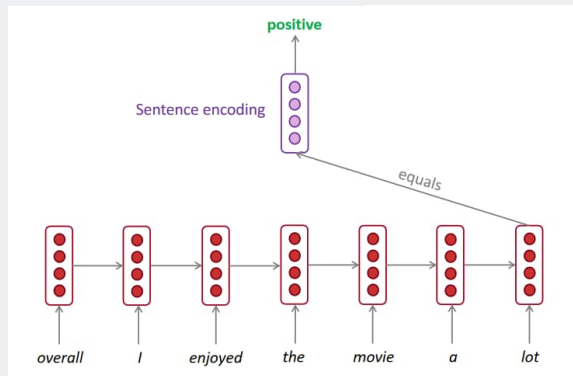


Jurafsky & Martin, SLP, 3rd Edition: Chapter 8

To understand what POS are, how to use, how many systems, etc. please look at chapter 8 in this book!

RNN – Applications

- Sentence Classification



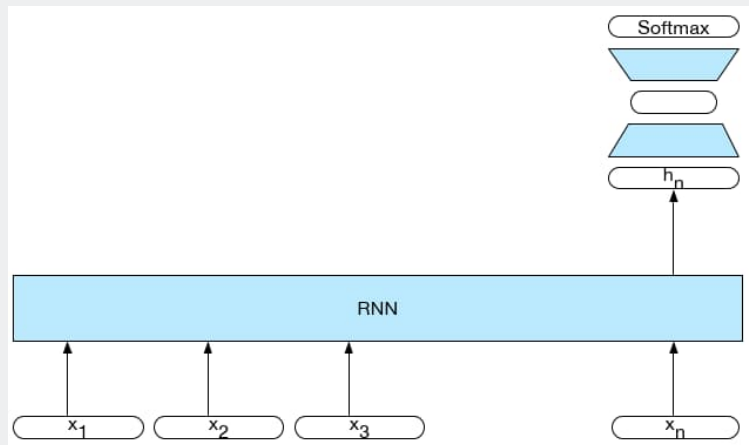
How to compute a sentence encoding?

You can use the final hidden state or ...

... take element-wise max of all the hidden state – this works usually better.

RNN – Deep Networks: Stacked and Bidirectional

- Sequence Classification
- end-to-end training



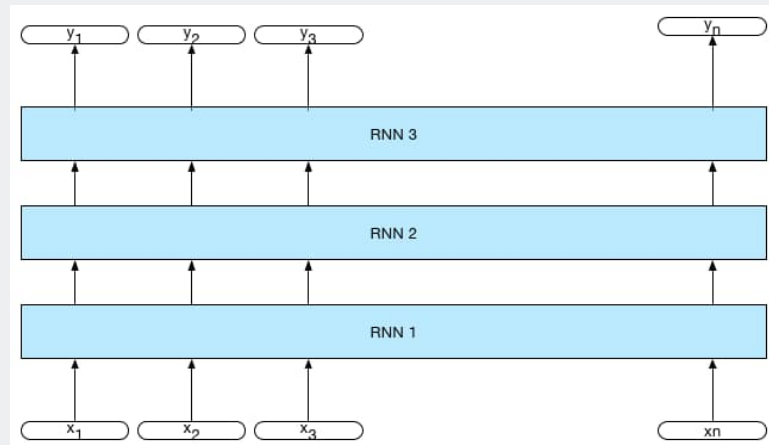
As seen already RNNs are quite flexible, can be combined with other FF networks. Like in this example where we do a sequence classification .

This combination of a simple RNN and a simple FF net is actually the first example of a Deep NL in these slides.

Training this type of network is referred to as end-to-end training

RNN – Deep Networks: Stacked

- Stacked
- Outperform single-layer
- Induce representations
- High training costs



As seen already RNNs are quite flexible, can be combined in many creative ways.

Nothing stops us from using the entire sequence of outputs from one RNN as an input sequence to another one. Stacked consist of multiple networks where the output of one layer serves as the input to a subsequent layer, as shown in Fig.

Stacked RNNs can outperform single-layer networks. One reason for this success has to do with the network's ability to induce representations at differing levels of abstraction across layers. Just as the early stages of the human visual system detect edges that are then used for finding larger regions and shapes, the initial layers of stacked networks can induce representations that serve as useful abstractions for further layers — representations that might prove difficult to induce in a single RNN.

The optimal number of stacked RNNs is specific to each application and to each training set. However, as the number of stacks is increased the training costs rise quickly.

RNN – Deep Networks: Bidirectional

- $RNN_{forward}$

$$h_t^f = RNN_{forward}(x_1^t)$$

- We have access to the entire input sequence

- $RNN_{backward}$

$$h_t^b = RNN_{backward}(x_t^n)$$

- Combine them -> Bi-RNN

$$h_t = h_t^f \oplus h_t^b$$

In a simple recurrent network, the hidden state at a given time t represents everything the network knows about the sequence up to that point in the sequence. That is, the hidden state at time t is the result of a function of the inputs from the start up through time t . We can think of this as the context of the network to the left of the current time.

Where $h_{t,f}$ corresponds to the normal hidden state at time t , and represents everything the network has gleaned from the sequence to that point.

in many applications we have access to the entire input sequence all at once. We might ask whether it is helpful to take advantage of the context to the right of the current input as well.

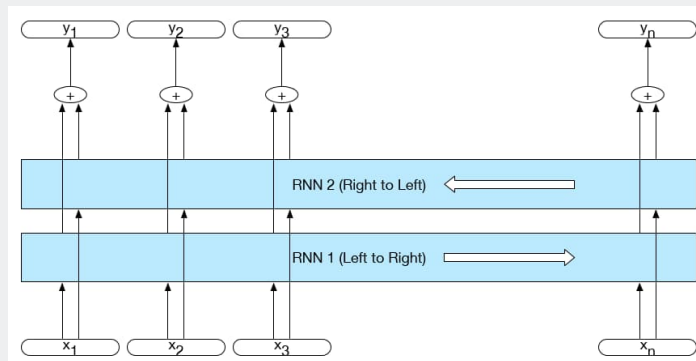
One way to recover such information is to train an RNN on an input sequence in reverse, using exactly the same kind of networks that we've been discussing. With this approach, the hidden state at time t now represents information about the sequence to the right of the current input.

A Bi-RNN consists of two independent RNNs, one where the input is processed from the start to the end, and the other from the end to the start. We then combine the outputs of the two networks into a single representation that captures both the left and right contexts of an input at each point in time.

RNN – Deep Networks: Bidirectional

- Bi-RNN combines

$$h_t = h_t^f \oplus h_t^b$$

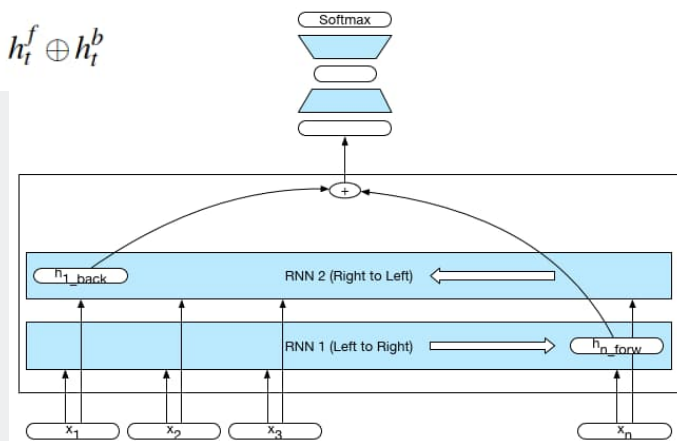


A bidirectional RNN. Separate models are trained in the forward and backward directions with the output of each model at each time point concatenated to represent the state of affairs at that point in time.

RNN – Deep Networks: Bidirectional

- Bi-RNN combines
- Sequence classification

$$h_t = h_t^f \oplus h_t^b$$



bidirectional RNN for sequence classification. The final hidden units from the forward and backward passes are combined to represent the entire sequence. This combined representation serves as input to the subsequent classifier.

Content

- Neural Language Models
- Recurrent Neural Networks
- LSTMs (Long Short-Term Memory Networks)

Mental summary.

In simple Recurrent Neural Networks sequences are processed naturally one element at a time.

- The output of a neural unit at a particular point in time is based both on the current input and value of the hidden layer from the previous time step.
- RNNs can be trained with a straightforward extension of the backpropagation algorithm, known as backpropagation through time (BPTT).
- Common language-based applications for RNNs include:
 - Probabilistic language modeling, where the model assigns a probability to a sequence, or to the next element of a sequence given the preceding words.
 - Auto-regressive generation using a trained language model.
 - Sequence labeling, where each element of a sequence is assigned a label, as with part-of-speech tagging.
 - Sequence classification, where an entire text is assigned to a category, as in spam detection, sentiment analysis or topic classification.

Long Short-Term Memory Networks

RNN Shortcomings

- Cannot use information distant from the current time
- Information encoded in the current hidden layer is local

The flights the airline was cancelling were full.

- Hidden layers and weights:
 - useful information for *current* decision
 - Update information for *future* decisions
- Vanishing gradients

In practice, it is quite difficult to train RNNs for tasks that require a network to make use of information distant from the current point of processing. Despite having access to the entire preceding sequence, the information encoded in hidden states tends to be fairly local, more relevant to the most recent parts of the input sequence and recent decisions. It is often the case, however, that distant information is critical to many language applications. To see this, consider the following example in the context of language modeling.

Assigning a high probability to *was* following *airline* is straightforward since *airline* provides a strong local context for the singular agreement. However, assigning an appropriate probability to *were* is quite difficult, not only because the plural *flights* is quite distant, but also because the intervening context involves singular constituents. Ideally, a network should be able to retain the distant information about plural flights until it is needed, while still processing the intermediate parts of the sequence correctly.

As a result, during the backward pass of training, the hidden layers are subject to repeated multiplications, as determined by the length of the sequence. A frequent result of this process is that the gradients are eventually driven to zero – the vanishing so-called vanishing gradients problem.

[→ A further serious shortcoming for simple recurrent nets is that, while training, we

need to backpropagate the error signal back through time. The loss at time t will contribute to the loss at time $t-1$ and also be part of the calculation (REPEATED multiplications). So eventually, gradient values are driven to 0 – the so called vanishing gradient problem]

RNN Shortcomings

- How to maintain relevant context over time?

The flights the airline was cancelling were full.

To address these issues, more complex network architectures have been designed to explicitly manage the task of maintaining relevant context over time. More specifically, the network needs to learn to forget information that is no longer needed and to remember information required for decisions still to come.

Recursive Function for Natural Language Understanding

- ADD1 is hardcoded
- Parametrized recursive function
- Memory: $\mathbf{h} \in \mathbb{R}^{d_h}$
- Input x_{t-1} and memory \mathbf{h} , returns the new \mathbf{h}
- Time index!

Remember this?

$$\mathbf{h}_t = f(x_t, \mathbf{h}_{t-1})$$

$$f(x_t, \mathbf{h}_{t-1}) = g(\mathbf{W}\phi(x_t) + \mathbf{U}\mathbf{h}_{t-1})$$

Algorithm 1 A function ADD1

```
s ← 0
function ADD1(v,s)
  if v = 0 then return s
  else return s + 1
  end if
end function
```

Algorithm 2 A function ADD1

```
s ← 0
for i ← 1, 2, ..., l do s ← ADD1(xi, s)
end for
```

We have here the hidden layer of a RNN playing the role of the memory

And the basic idea (Hochreiter and Schmidhuber) came up with in 1997 is to divide the context management problem (the red square) into two sub-problems:

- 1) removing information no longer needed from the context, and
- 2) adding information likely to be needed for later decision making

Long Short-Term Memory Networks (LSTMs)

- Memory (aka. context): $\mathbf{h} \in \mathbb{R}^{d_h}$
- Want: divide context management into:
 - Forgetting (old/unnecessary information)
 - memorizing (new information/context)
- If possible without hard-coding into the architecture!
- Solution:
 - add an explicit context layer
 - gates to control the forgetting/memorizing

Put these ideas on one slide:

1. We have the hidden layer of a RNN playing the role of the memory
2. divide the context management problem into two sub-problems:
removing information no longer needed from the context,
and adding information likely to be needed

The key to solving both problems is to learn how to manage this context rather than hard-coding a strategy into the architecture.

LSTMs accomplish this by first adding an explicit context layer to the architecture (in addition to the usual recurrent hidden layer), and through the use of specialized neural units that make use of gates to control the flow of information into and out of the units that comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, and previous hidden layer, and previous context layers.

Long Short

We have a sequence of inputs $x^{(t)}$, and we will compute a sequence of hidden states $h^{(t)}$ and cell states $c^{(t)}$. On timestep t :

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

New cell content: this is the new content to be written to the cell

Cell state: erase ("forget") some content from last cell state, and write ("input") some new cell content

Hidden state: read ("output") some content from the cell

Sigmoid function: all gate values are between 0 and 1

$$\begin{aligned} f^{(t)} &= \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f) \\ i^{(t)} &= \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i) \\ o^{(t)} &= \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o) \end{aligned}$$

$$\tilde{c}^{(t)} = \tanh(W_c h^{(t-1)} + U_c x^{(t)} + b_c)$$

$$c^{(t)} = f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)}$$

$$h^{(t)} = o^{(t)} \circ \tanh c^{(t)}$$

Gates are applied using element-wise product

All these are vectors of same length n

context \approx memory

TU Informatics

<https://web.stanford.edu/class/cs224n/slides/cs224n-2020-lecture07-fancy-mn.pdf>

The gates in an LSTM share a common design pattern; each consists of a feed-forward layer, followed by a sigmoid activation function, followed by a pointwise multiplication with the layer being gated. The choice of the sigmoid as the activation function arises from its tendency to push its outputs to either 0 or 1. Combining this with a pointwise multiplication has an effect similar to that of a binary mask. Values in the layer being gated that align with values near 1 in the mask are passed through nearly unchanged; values corresponding to lower values are essentially erased.

There are several ways to describe the LSTM computation steps:

One task is compute the actual information we need to extract from the previous hidden state and current inputs — the same basic computation we've been using for all our recurrent networks. (1st purple text box). Note that the weight matrixes are switched from what we have seen until now – but the idea is the same – multiply the input with weights (W in our previous slides, on this slide U), apply activation function, take output multiply with another set of weights (here W but in our previous slides U). And bias.

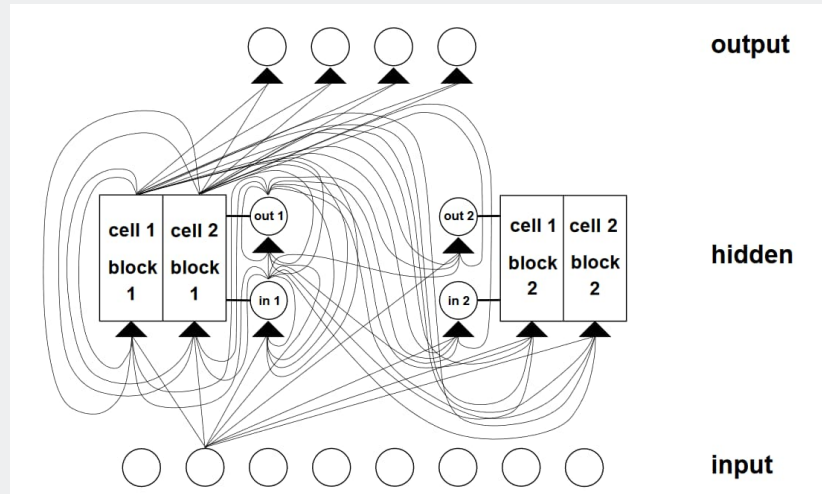
Then we need to decide what to forget and what to keep from the new information (2nd purple text box)

We have, therefore a “forget” gate and an “input” gate. The forget gate will delete information from the context that is no longer needed. It will compute a weighted sum of the previous state’s hidden layer and the current input and passes that through a sigmoid.

The “input” gate, or the “add” gate selects the information to add to the current context.

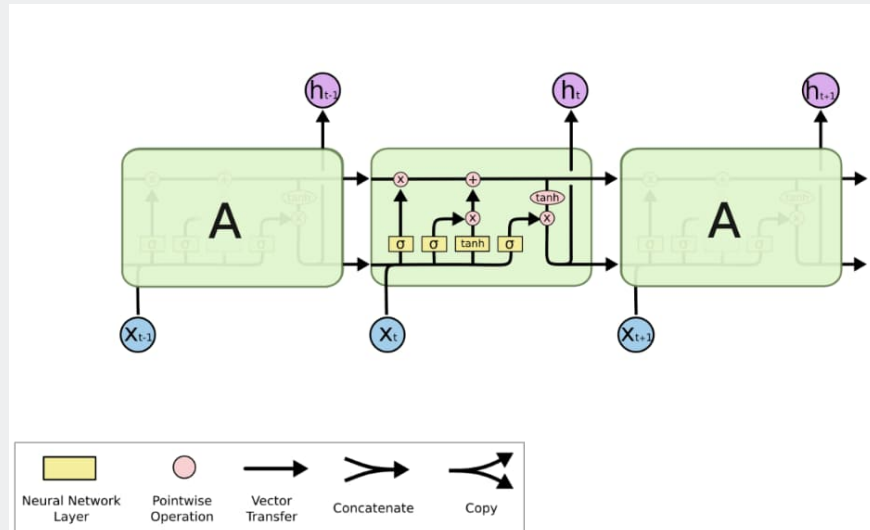
The final gate we’ll use is the output gate which is used to decide what information is required for the current hidden state (as opposed to what information needs to be preserved for future decisions).

Long Short-Term Memory Networks (LSTMs)



Original description of these ideas – original paper.

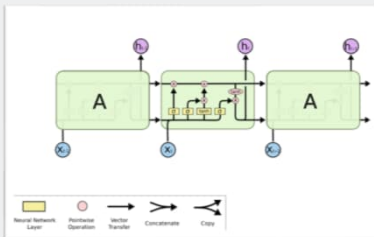
Long Short-Term Memory Networks (LSTMs)



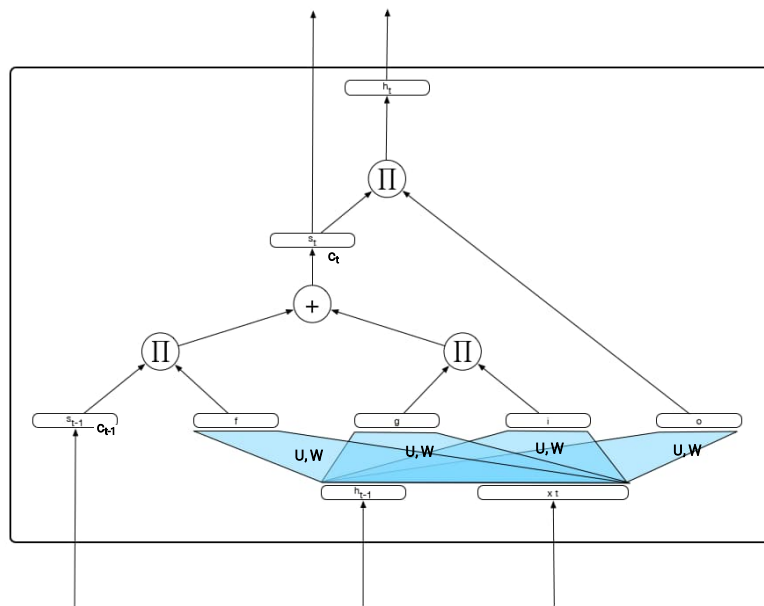
But you might be more used to these pictures

LSTMs

- Forgetting (unnecessary info)
- Memorizing (new information)
- Learning 8 weight matrixes!



TU
M Informatics



<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

So more complex network architectures have been designed to explicitly manage the task of maintaining relevant context over time. More specifically, the network needs to learn to forget information that is no longer needed and to remember information required for decisions still to come.

Recap: the LSTM architecture proposed by Hochreiter and Schmidhuber in 1997: On step t there is a hidden state h_t and a cell state c_t . Both are vectors of length n . With these two components, the cell stores long-term information.

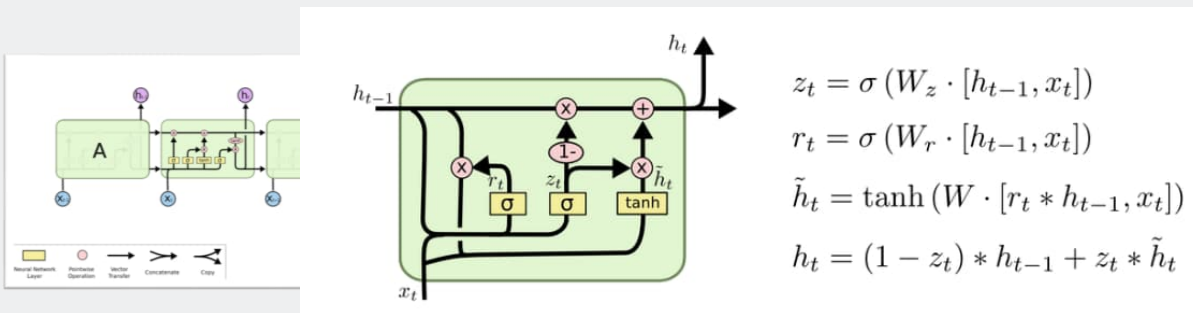
The LSTM can erase, write, and read information from the cell. The selection of which info to erase/read/write is controlled by three corresponding gates – also of length n . The sigmoid function that these gates use ensure that the gate components are either open (1s), closed (0s), or somewhere in between. Importantly, the gates are dynamic (remember that we didn't want to hard-code them) – their value is computed based on the current context.

LSTMs introduce a considerable number of additional parameters to our recurrent networks. We now have 8 sets of weights to learn (i.e., the U and W for each of the 4 gates within each unit), whereas with simple recurrent units we only had 2. Training these additional parameters imposes a much significantly

higher training cost.

Gated Recurrent Unit

- Uses only two gates: “reset”, r , and “update”, z
- Collapse “forget” and “input” gates into the “update” gate z



Gated Recurrent Units (GRUs) (Cho et al., 2014) ease this burden by dispensing with the use of a separate context vector, and by reducing the number of gates to 2 — a reset gate, r and an update gate, z .

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. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.

As with LSTMs, the use of the sigmoid in the design of these gates results in a binary-like mask that either blocks information with values near zero or allows information to pass through unchanged with values near one. The purpose of the reset gate is to decide which aspects of the previous hidden state are relevant to the current context and what can be ignored. This is accomplished by performing an element-wise multiplication of r with the value of the previous hidden state (first two equations).

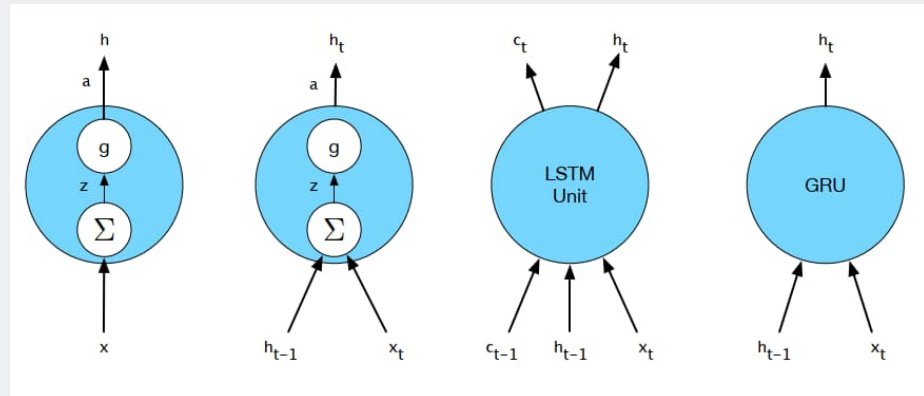
We then use this masked value in computing an intermediate representation for the new hidden state at time t . (3rd equation)

The job of the update gate z is to determine which aspects of this new state will be used directly in the new hidden state and which aspects of the previous state need to be

preserved for future use. This is accomplished by using the values in z to interpolate between the old hidden state and the new one. (last equation).

Neural Units

- Complexity encapsulated in basic processing units



The neural units used in LSTMs and GRUs are obviously much more complex than those used in basic feedforward networks. Fortunately, this complexity is encapsulated within the basic processing units, allowing us to maintain modularity and to easily experiment with different architectures. Figure illustrates the inputs and outputs associated with each kind of unit.

At the far left, (a) is the basic feedforward unit where a single set of weights and a single activation function determine its output, and when arranged in a layer there are no connections among the units in the layer. Next, (b) represents the unit in a simple recurrent network. Now there are two inputs and an additional set of weights to go with it. However, there is still a single activation function and output.

The increased complexity of the LSTM (c) and GRU (d) units on the right is encapsulated within the units themselves. The only additional external complexity for the LSTM over the basic recurrent unit (b) is the presence of the additional context vector as an input and output. The GRU units have the same input and output architecture as the simple recurrent unit.

It is this modularity that is key to the power and widespread applicability of LSTM and GRU units. LSTM and GRU units can be substituted into any of the network architectures. And, as with simple RNNs, multi-layered networks making use of gated units can be unrolled into deep feedforward networks and trained in the usual fashion with backpropagation.

Content

- Neural Language Models
- Recurrent Neural Networks
- LSTMs (Long Short-Term Memory Networks)

- **Encoder-Decoder**
- **Attention**

- Very active research area – not all details are included

Machine Translation

(sequence-to-sequence processing)

Now, I will introduce the encoder-decoder architectures, through the help of an actual Natural Language Processing and Understanding Application: Machine Translation

Sequence-to-Sequence aka. Encoder-decoder Models

- Neural Machine Translation
- **Source** sentence X in source language
- **Target** sentence Y in target language
- Translation: function application:
- More than one correct translation

$$X = (x_1, x_2, \dots, x_{T_x})$$

$$Y = (y_1, y_2, \dots, y_{T_y})$$

$$f : V_x^+ \rightarrow C_{|V_y|-1}^+$$

$$C_k = \left\{ (t_0, \dots, t_k) \in \mathbb{R}^{k+1} \mid \sum_{i=1}^k t_i = 1 \text{ and } t_i \geq 0 \text{ for all } i \right\}$$

$$P(Y|X)$$

Machine translation with statistical methods.

Let's first think of what it means to translate one sentence X in a source language to an equivalent sentence Y in a target language which is different from the source language. A process of translation is a function that takes as input the source sentence X and returns a correct translation Y , and it is clear that there may be more than one correct translations.

Where V_x is a source vocabulary, and V_x^+ is a set of all possible source sentences of any length $T_x > 0$. V_y is a target vocabulary, and C_k is a standard k -simplex.

In short, this set C_k contains all possible settings for distributions of $k+1$ possible outcomes.

This means that the translation function f returns a probability distribution $P(Y|X)$ over all possible translations of length $T_y > 1$.

Neural Machine Translation

- Conditional language modelling!

$$X = (x_1, x_2, \dots, x_{T_x})$$

$$Y = (y_1, y_2, \dots, y_{T_y})$$

$$f: V_x^+ \rightarrow C_{|V_y|-1}^+$$

$$P(Y|X) = \prod_{t=1}^{T_y} \underbrace{P(y_t | y_1, \dots, y_{t-1}, \underbrace{X}_{\text{conditional}})}_{\text{language modelling}}$$

$$C_k = \left\{ (t_0, \dots, t_k) \in \mathbb{R}^{k+1} \mid \sum_{i=1}^k t_i = 1 \text{ and } t_i \geq 0 \text{ for all } i \right\}$$

- Use what we learned to compute these!
 - N-grams
 - Embeddings
 - ...

Rewrite this conditional probability according to what we've discussed previously on Language Models – and this formula (in the red square) should be quite familiar to you. Which means, that we can use any of the previous techniques for statistical MT?

Neural Machine Translation

- Conditional language modelling!

$$X = (x_1, x_2, \dots, x_{T_x})$$

$$Y = (y_1, y_2, \dots, y_{T_y})$$

$$P(Y|X) = \prod_{t=1}^{T_y} \underbrace{P(y_t | y_1, \dots, y_{t-1}, \underbrace{X}_{\text{conditional}})}_{\text{language modelling}}$$

- Training:
 - Maximizing the log-likelihood cost function for a given training set

$$-\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_y} \log p(y_t^n | y_{<t}^n, X^n) \\ \{(X^1, Y^1), (X^2, Y^2), \dots, (X^N, Y^N)\}$$

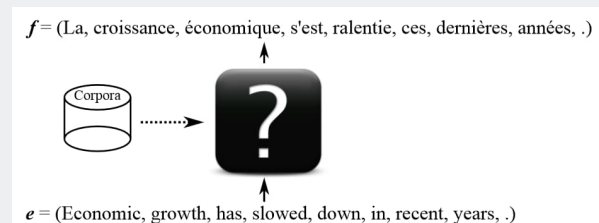
Training can be trivially done by maximizing the log-likelihood or equivalently minimizing the negative log-likelihood for a given training set of N pairs.

Neural Machine Translation

- The big picture:

- 1) Assign probabilities to sentences
- 2) Handle variable length sequences (RNNs)
- 3) Train with costs functions & gradient descent

? Training data
? Evaluating MT



A big picture on this process translation is shown in the figure. More specifically, building a statistical machine translation model simply consists of these steps :

1. Assign a probability to a sentence.
2. Handle variable-length sequences with recurrent neural networks
3. Compute the gradient of an empirical cost wrt. the parameters of a neural network, and minimize this cost function with some version of gradient descent appropriate to the concrete task.

Of course, simply knowing all these does not get you a working neural network that translates from one language to another. We will look at how to build such a neural network in the next slides.

Two issues:

- (1) where do we get training data?
- (2) how do we evaluate machine translation systems?

Training Data for Machine Translation

- Sequence-to-sequence
- Sentence pairs (source_language, target_language)
- *parallel-corpus*
 - where to get it?
- International news agencies (AFP)
- Books published in multiple languages
- Ebay/Amazon/... (product descriptions)

Copyright issues!

It is machine translation, and from the description in the previous section is a sentence-to-sentence translation task. We approach this problem by building a model that takes as input a source sentence X and computes the probability $P(Y|X)$ of a target sentence Y , equivalently a translation. In order for this model to translate, we must train it with a training set of pairs of a source sentence and its correct translation.

1st problem: where to get such a *parallel corpus*

But we most probably will run in copyright issues.

Training Data for Machine Translation

- Sequence-to-sequence
- Sentence pairs (source_language, target_language)
- *parallel-corpus*
 - where to get it?
- proceedings from the Canadian parliament (Brown et al, 1990)
 - French – English, curated (professional translators)
- EU parliament - more than 20 languages

Fortunately, it turned out that there are a number of legitimate sources where we can get documents translated in more than one languages, often very faithfully to their content. These sources are parliamentary proceedings of bilingual, or multilingual countries.

Brown et al. used the proceedings from the Canadian parliament, which are by law kept in both French and English. All of these proceedings are digitally available and called Hansards. (the latest version is to be found under this link)

Similarly, the European parliament used to provided the parliamentary proceedings more than official languages Unfortunately, the European parliament decided to stop translating its proceedings into all 23 official languages on 21 Nov 2011 as an effort toward budget cut.

These proceedings-based parallel corpora have two distinct advantages. First, in many cases, the sentences in those corpora are well-formed, and their translations are done by professionals, meaning the quality of the corpora is guaranteed. Second, surprisingly, the topics discussed in those proceedings are

quite diverse.

Training Data for Machine Translation

- translated subtitle of the TED talks, (WIT, <https://wit3.fbk.eu/>)
 - 104 languages
- Russian-English: Yandex (<https://translate.yandex.ru/corpus?lang=en>)
- SWRC English-Korean multilingual corpus: 60,000 sentence pairs
- <https://github.com/jungyeul/korean-parallel-corpora> (~94K sentence pairs)
- Crawl the internet for pairs of pages
 - Wikipedia
- Common Crawl Parallel Corpus (Smith et. Al, 2013)
 - <http://www.statmt.org/wmt13/training-parallel-commoncrawl.tgz>

So, then, where can we get all data for all these non-European languages? There are a number of resources you can use, here are a few of them:

You can continue with other languages by searching the internet

This is just not large enough. One way to avoid this or mitigate this problem is to automatically mine parallel corpora from the Internet.

There have been quite some work in this direction as a way to increase the size of parallel corpora. The idea is to build an algorithm that crawls the Internet and find a pair of corresponding pages in two different languages. One of the largest pre-processed corpus of multiple languages from the Internet is the Common Crawl Parallel Corpus created by Smith et al. available at <http://www.statmt.org/wmt13/training-parallel-commoncrawl.tgz>

Evaluating Machine Translation

- There may be many correct translations for one sentence
 - It is a guide to action that ensures that the military will forever heed Party commands.
 - It is the guiding principle which guarantees the military forces always being under the command of the Party.
 - It is the practical guide for the army always to heed the directions of the party.
- Quality is not success or failure

A big question follows: how do we evaluate this model? In the case of classification, evaluation is quite straightforward. All we need to do is to classify held-out test examples with a trained classifier and see how many examples were correctly classified. This is however not true in the case of translation.

For instance, the following three sentences are the translations made by a human translator given a single Chinese sentence

They all clearly differ from each other, although they are the translations of a single source sentence.

Evaluating Machine Translation

- Quality is not success or failure:
 - French: "J'aime un llama, qui est un animal mignon qui vit en Amérique du Sud"
 - "I like a llama which is a cute animal living in South America". 100
 - "I like a llama, a cute animal that lives in South America". 90
 - "I like a llama from South America"? 50
 - "I do not like a llama which is an animal from South America"?
- We want automated evaluation!

One possible English translation of this French sentence is "I like a llama which is a cute animal living in South America".

Let's give this translation a score 100 (success).

Another translation of the French sentence above is "I like a llama, a cute animal that lives in South America".

With omitted "qui est" from the original sentence, but the whole meaning has well been captured. Let us give this translation a slightly lower score of 90.

Then, how about "I like a llama from South America"? This is certainly not a correct translation, but except for the part about a llama being cute, this sentence does communicate most of what the original French sentence tried to communicate. Maybe, we can give this translation a score of 50.

How about "I do not like a llama which is an animal from South America"? This translation correctly describes the characteristics of llama exactly as described in the source sentence. However this translation incorrectly states that I do not like

a llama, when I like a llama according to the original French sentence. What kind of score would you give this translation?

We cannot look at thousands of validation or test sentence pairs to tell how well a machine translation model does. Even if we somehow did it for a single model, in order to compare this translation model against others, we must do it for every single machine translation model under comparison. We must have an automatic evaluation metric in order to efficiently test and compare different machine translation models.

Evaluating Machine Translation – BLEU score

- geometric mean of the modified N-gram precision scores multiplied by brevity penalty.

- N-gram precision:

$$p_n = \frac{\sum_{S \in C} \sum_{\text{ngram} \in S} \hat{c}(\text{ngram})}{\sum_{S \in C} \sum_{\text{ngram} \in S} c(\text{ngram})}$$

$$\hat{c}(\text{ngram}) = \min(c(\text{ngram}), c_{\text{ref}}(\text{ngram})).$$

- Geometric mean

$$P_1^4 = \exp \left(\frac{1}{4} \sum_{n=1}^4 \log p_n \right)$$

- But: “cute animal that lives” P = 1
- Brevity Penalty (BP)

$$\text{BP} = \begin{cases} 1 & , \text{ if } l \geq r \\ \exp \left(1 - \frac{r}{l} \right) & , \text{ if } l < r \end{cases}$$

One of the most widely used automatic evaluation metric for assessing the quality of translations is BLEU, dating from 2002

Where C is a corpus of all the sentences/translations, and S is a set of all unique n-grams in one sentence in C. c(ngram) is the count of the n-gram, and $\hat{c}(\text{ngram})$ is:

$C_{\text{ref}}(\text{ngram})$ is the count of the n-gram in reference sentences.

The modified n-gram precision measures the ratio between the number of n-grams in the translation and the number of those n-grams actually occurred in a reference (ground-truth) translation.

If there is no n-gram from the translation in the reference, this modified precision will be zero.

It is common to use the geometric average of modified 1-, 2-, 3- and 4-gram precisions, which is computed by this formula.

HOWEVER, if we just use this formula one can get a high average modified precision by making as short a translation as possible. Brevity Penalty will deal with this.

In order to avoid this behaviour, BLEU penalizes the geometric average of the modified n-gram precisions by the ratio of the lengths between the reference and translation I . This is done by first computing a brevity penalty.

If the translation is longer than the reference, it uses the geometric average of the modified n-gram precisions as it is. Otherwise, it will penalize it by multiplying the average precision with a scalar less than 1.

Evaluating Machine Translation – BLEU score

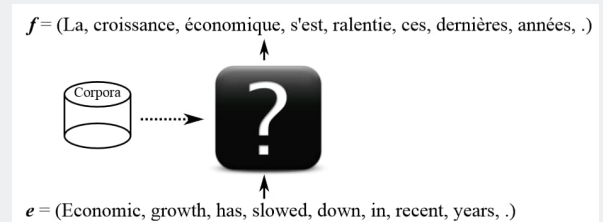
- The BLEU was shown to correlate well with human judgements
- But not perfect automatic evaluation metric
- METEOR (M. Denkowski and A. Lavie, 2014)
- TER (Translation Edit Rate, M. Snover, 2006)

Neural Machine Translation

- The big picture:

- 1) Assign probabilities to sentences
- 2) Handle variable length sequences (RNNs)
- 3) Train with costs functions & gradient descent

- ✓ Training data
- ✓ Evaluating MT



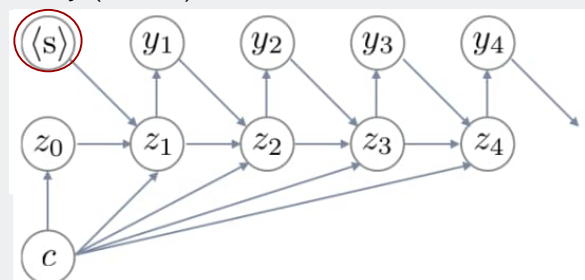
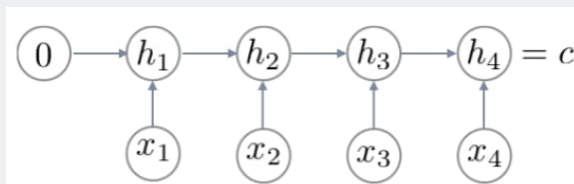
Neural Machine Translation: Encoder-Decoder Model

• Input: $Y = (y_1, \dots, y_{t-1})$ $X = (x_1, \dots, x_{T_x})$

- Start with X , how to handle it?
 - Variable-length sequence (RNN)
 - No explicit output/target \rightarrow only the summary (vector)
 - RNN \sim *encoder*

$$P(Y|X) = \prod_{t=1}^{T_y} P(y_t | y_1, \dots, y_{t-1}, \underbrace{X}_{\text{conditional}})$$

language modelling



TU Informatics

Lets see how to build a model to handle translations.

We need to model each conditional distribution inside the product as a function. This function will take as input all the previous words in the target sentence Y (Y_1, \dots, y_n) and the whole source sentence X

Given these inputs the function will compute the probabilities of all the words in the target vocabulary V_y .

The approach here has been proposed multiple times, independently, in the last two decades.

Let us start by tackling how to handle the source sentence $X = (x_1, \dots, x_{T_x})$. Since this is a variable-length sequence, we can readily use a recurrent neural network. However, unlike the previous examples, there is no explicit target/output in this case. All we need is a (vector) summary of the source sentence.

RNN can be an LSTM, a GRU, etc.

After reading the whole sentence up to x_{T_x} , the last memory state h_{T_x} of the encoder summarizes the whole source sentence into a single vector, c

We now need to design a decoder, again, using a recurrent neural network. The decoder is really nothing else but a language model, except that it is conditioned on the source sentence X . What this means is that we can build a recurrent neural network language model feeding also the context vector at each time step.

Although I have used c as if it is a separate variable this is simply a shorthand notation of the last memory state of the encoder which is a function of the whole source sentence. What does this mean? It means that we can compute the gradient of the empirical cost function with respect to all the parameters of both the encoder and decoder and maximize the cost function using stochastic gradient descent, just like any other neural network we have learned so far

Note this sentence margin maker, we'll come back to this in a minute.

Neural Machine Translation: Encoder-Decoder Model

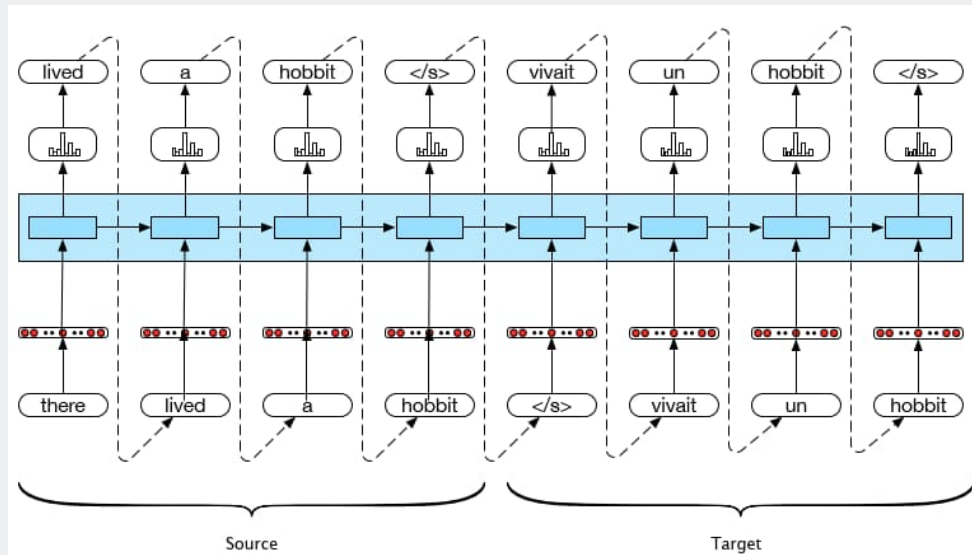
- Task: automatically translate from one language to another
- Source language/sentence/sequence
- Target language/sentence/sequence
- Parallel Corpus or **bitexts**
- Language Models & Autoregressive Generation extended to Machine Translation
 - End-of-sentence marker between **bitexts** (source</s>target)
 - Use them as training data (RNN-based LM)
 - Predict next word in the sentence

Fix some vocabulary we have been using in the last slides:

To extend language models and autoregressive generation to machine translation, we'll first add an end-of-sentence marker at the end of each bitext's source sentence and then simply concatenate the corresponding target to it. These concatenated source-target pairs can now serve as training data for a combined language model. Training proceeds as with any RNN-based language model. The network is trained autoregressively to predict the next word in a set of sequences comprised of the concatenated source-target bitexts,

Neural Machine Translation: Encoder-Decoder Model

Simple RNN,
LSTM, GRU, ...



In the following we will abstract away from the type of network architecture, and we will only specify the input/output on which the computation is based.

To translate a source text using the trained model, we run it through the network performing forward inference to generate hidden states until we get to the end of the source. Then we begin autoregressive generation, asking for a word in the context of the hidden layer from the end of the source input as well as the end-of-sentence marker. Subsequent words are conditioned on the previous hidden state and the embedding for the last word generated.

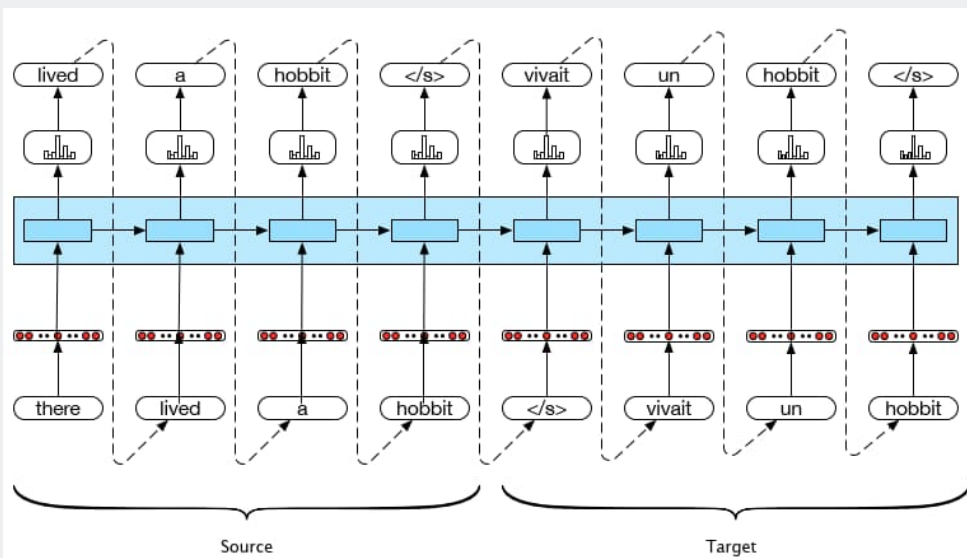
Early efforts using this clever approach demonstrated surprisingly good results on standard datasets and led to a series of innovations that were the basis for networks I'll show next.

Encoder-Decoders

(aka. Sequence-to-sequence Models)

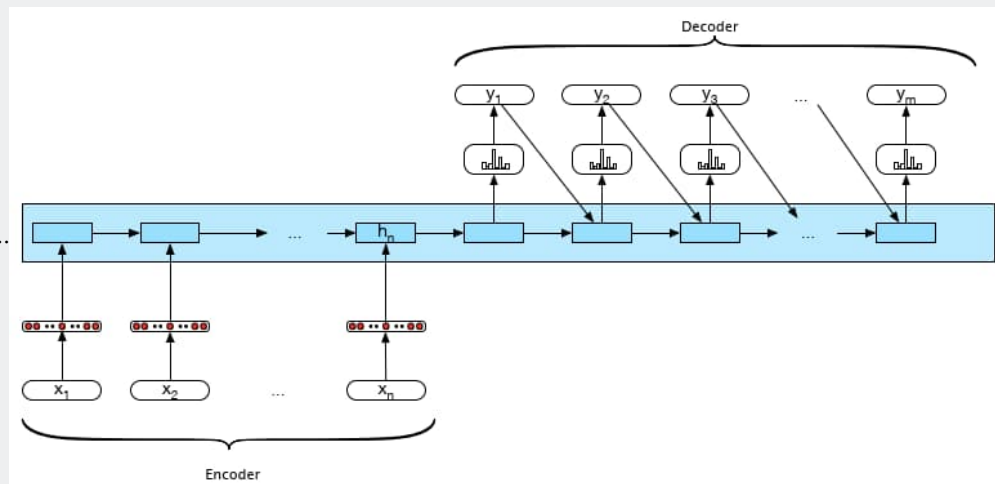
Neural Machine Translation: Encoder-Decoder Model

Simple RNN,
LSTM, GRU, ...



Neural Machine Translation: Encoder-Decoder Model

Simple RNN,
LSTM, GRU, ..



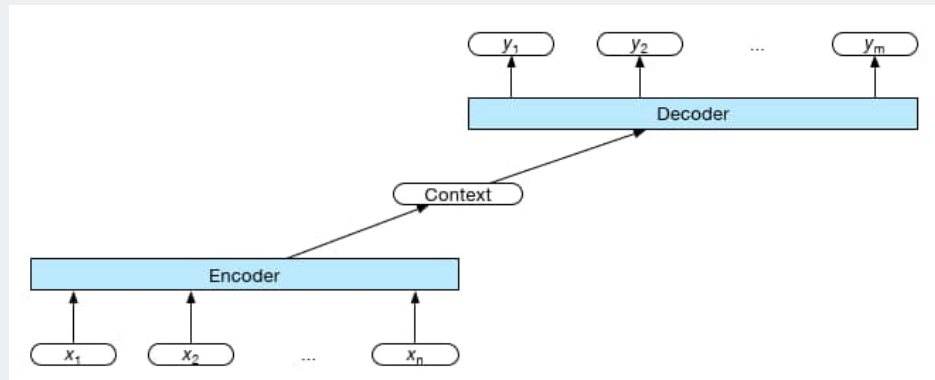
Let's abstract a bit further, away from the specifics of machine translation and illustrates a basic encoder-decoder architecture.

The elements of the network on the left process the input sequence and comprise the encoder, the entire purpose of which is to generate a contextualized representation of the input. In this network, this representation is embodied in the final hidden state of the encoder, h_n , which in turn feeds into the first hidden state of the decoder. The decoder network on the right takes this state and autoregressively generates a sequence of outputs.

This architecture is consistent with the original application of Neur Models to MT. But there are a few design choices that are less than optimal. Among the major ones are that the encoder and the decoder are assumed to have the same internal structure (RNNs in this case), that the final state of the encoder is the only context available to the decoder, and finally that this context is only available to the decoder as its initial hidden state.

Neural Machine Translation: Encoder-Decoder Model

- Three main components:
 - Encoder
 - Context vector
 - decoder



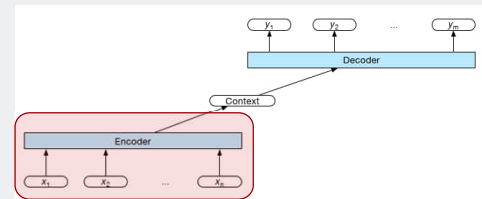
Let's abstract a bit further, away from the specifics of machine translation and illustrates a basic encoder-decoder architecture.

- 1 An encoder that accepts an input sequence, $x_{1:n}$, and generates a corresponding sequence of contextualized representations, $h_{1:n}$.
2. A context vector, c , which is a function of $h_{1:n}$, and conveys the essence of the input to the decoder.
3. And a decoder, which accepts c as input and generates an arbitrary length sequence of hidden states $h_{1:m}$, from which a corresponding sequence of output states $y_{1:m}$, can be obtained.

Which options do we have for each of these components

Encoder

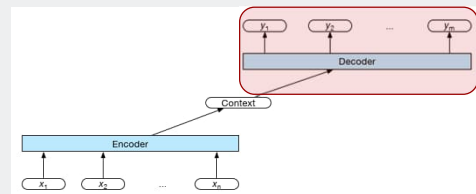
- Simple RNNs, LSTM, GRU
- Stacked
- Bi-LSTMs are the norm



Simple RNNs, LSTMs, GRUs, convolutional networks, as well as transformer networks (discussed later in this chapter), can all be employed as encoders. For simplicity, our figures show only a single network layer for the encoder, however, stacked architectures are the norm, where the output states from the top layer of the stack are taken as the final representation. A widely used encoder design makes use of stacked Bi-LSTMs where the hidden states from top layers from the forward and backward passes are concatenated as described last week to provide the contextualized representations for each time step.

Decoder

- Autoregressive generation
- Until </s> is generated
- LSTM, GRU



$$c = h_n^e$$

$$h_0^d = c$$

$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d)$$

$$z_t = f(h_t^d)$$

$$y_t = \text{softmax}(z_t)$$

For the decoder, autoregressive generation is used to produce an output sequence, an element at a time, until an end-of-sequence marker is generated. This incremental process is guided by the context provided by the encoder as well as any items generated for earlier states by the decoder.

Again, a typical approach is to use an LSTM or GRU-based RNN where the context consists of the final hidden state of the encoder, and is used to initialize the first hidden state of the decoder.

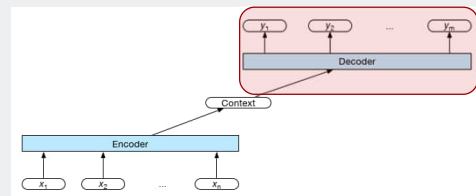
Generation proceeds as described earlier where each hidden state is conditioned on the previous hidden state and output generated in the previous state.

(To help keep things straight, we'll use the superscripts e and d where needed to distinguish the hidden states of the encoder and the decoder.)

Recall, that g is a stand-in for some flavor of RNN and \hat{y}_{t-1} is the embedding for the output sampled from the softmax at the previous step.

Decoder

- Context available only once.
- How to choose, from the output space the right "next" decoded sequence element?
 - Large search space!
 - Algorithm: Beam Search



$$c = h_n^e$$

$$h_0^d = c$$

$$h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d, c) \quad h_t^d = g(\hat{y}_{t-1}, h_{t-1}^d)$$

$$z_t = f(h_t^d)$$

$$y_t = \text{softmax}(\hat{y}_{t-1}, z_t, c) \quad y_t = \text{softmax}(z_t)$$

$$\hat{y} = \text{argmax} P(y_i | y_{<i})$$

A weakness of this approach is that the context vector, c , is only directly available at the beginning of the process and its influence will wane as the output sequence is generated.

A solution is to make the context vector c available at each step in the decoding process by adding it as a parameter to the computation of the current hidden state.

A common approach to the calculation of the output layer y is to base it solely on this newly computed hidden state. While this cleanly separates the underlying recurrence from the output generation task, it makes it difficult to keep track of what has already been generated and what hasn't.

An alternative approach is to condition the output on both the newly generated hidden state, the output generated at the previous state, and the encoder context.

Finally, as shown earlier, the output y at each time consists of a softmax

computation over the set of possible outputs (the vocabulary in the case of language models). What one does with this distribution is task-dependent, but it is critical since the recurrence depends on choosing a particular output, \hat{y} , from the softmax to condition the next step in decoding. We've already seen several of the possible options for this. For neural generation, where we are trying to generate novel outputs, we can simply sample from the softmax distribution.

However, for applications like MT where we're looking for a specific output sequence, random sampling isn't appropriate and would likely lead to some strange output.

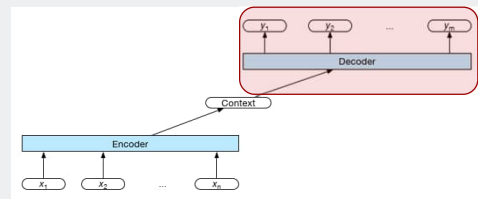
An alternative is to choose the most likely output at each time step by taking the argmax over the softmax output.

This is easy to implement but as we've seen several times with sequence labeling, independently choosing the argmax over a sequence is not a reliable way of arriving at a good output since it doesn't guarantee that the individual choices being made make sense together and combine into a coherent whole.

In sequence labelling we've used a Viterbi-style dynamic programming search, but here, this cannot be applied.

Beam Search

- Large search space
- Alternative: heuristic method, systematic exploration
- By controlling the exponential growth of the search space
- How: combine breadth first with a heuristic filter
 - Score the options
 - Prune the search space



Decoding with argmax (finding the sentence in the space of target sequence with the max likelihood to be related to the input sequence) has the problem that the search space is very large.

An alternative is to use a heuristic state-space search, and systematically explore the space of possible outputs. Through a technique called Beam Search, which I will not explain now because of time.

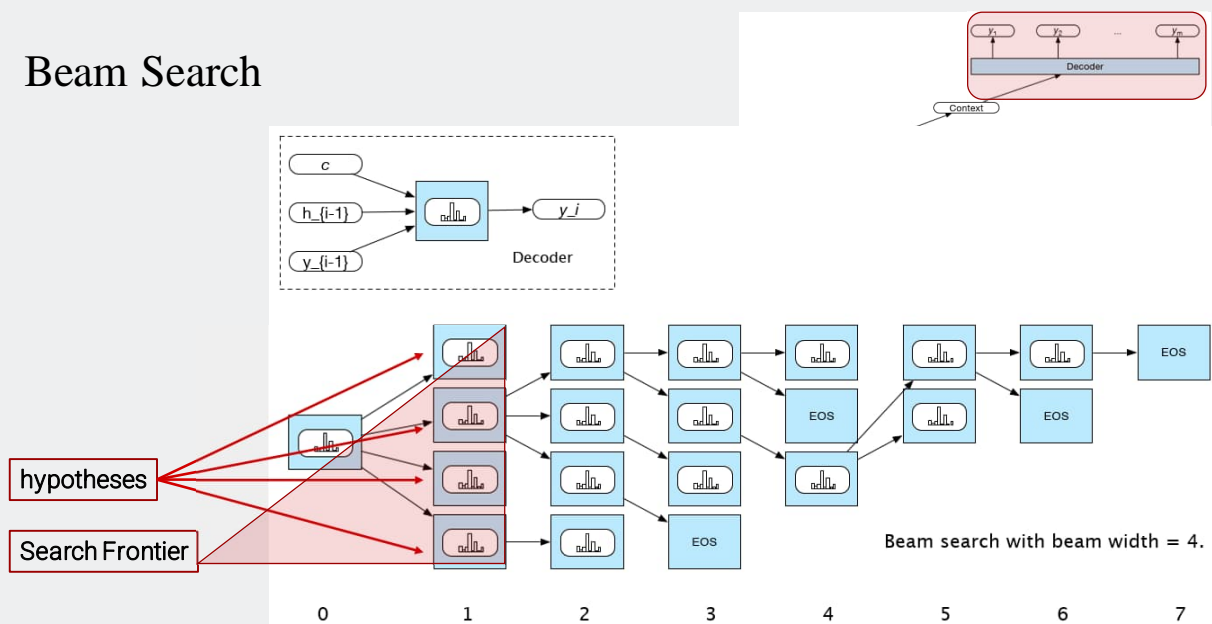
The key to such an approach is controlling the exponential growth of the search space. To accomplish this, we'll use a technique called beam search.

Beam search operates by combining a breadth-first-search strategy with a heuristic filter that scores each option and prunes the search space to stay within a fixed-size memory footprint, called the beam width.

At the first step of decoding, we select the B-best options from the softmax output y , where B is the size of the beam. Each option is scored with its

corresponding probability from the softmax output of the decoder. These initial outputs constitute the search frontier. We'll refer to the sequence of partial outputs generated along these search paths as hypotheses.

Beam Search



At the first step of decoding, we select the B-best options from the softmax output y , where B is the size of the beam. Each option is scored with its corresponding probability from the softmax output of the decoder. These initial outputs constitute the search frontier. We'll refer to the sequence of partial outputs generated along these search paths as hypotheses.

At subsequent steps, each hypothesis on the frontier is extended incrementally by being passed to distinct decoders, which again generate a softmax over the entire vocabulary. To provide the necessary inputs for the decoders, each hypothesis must include not only the words generated thus far but also the context vector, and the hidden state from the previous step.

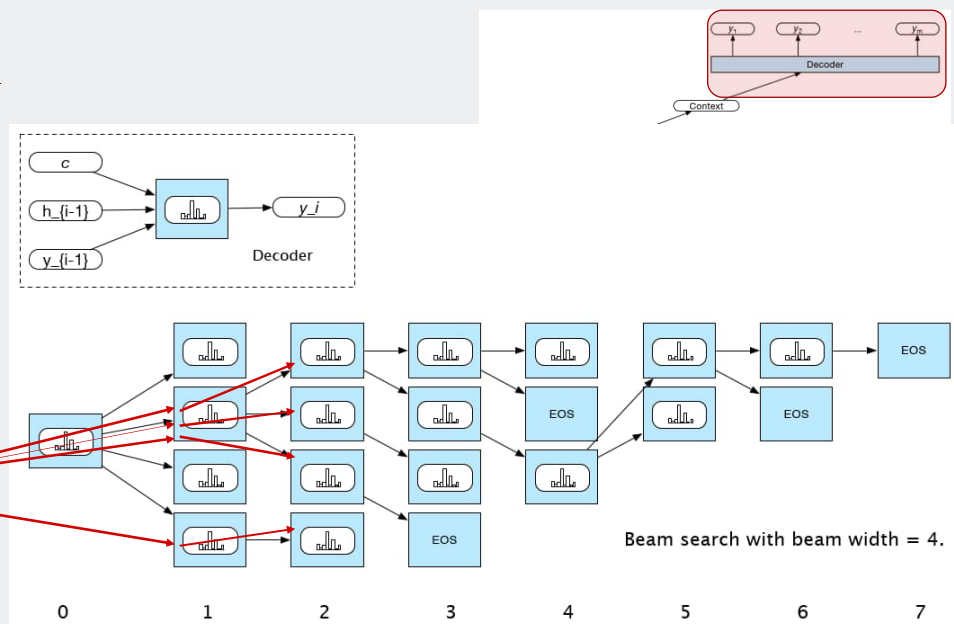
Beam Search

Scoring:

$$P(y_i | y_{<i})$$

hypotheses

Search Frontier



New hypotheses representing every possible extension to the current ones are generated and added to the frontier.

Each of these new hypotheses is scored using $P(y_i | y_{<i})$, which is the product of the probability of current word choice multiplied by the probability of the path that led to it.

To control the exponential growth of the frontier, it is pruned to contain only the top B hypotheses.

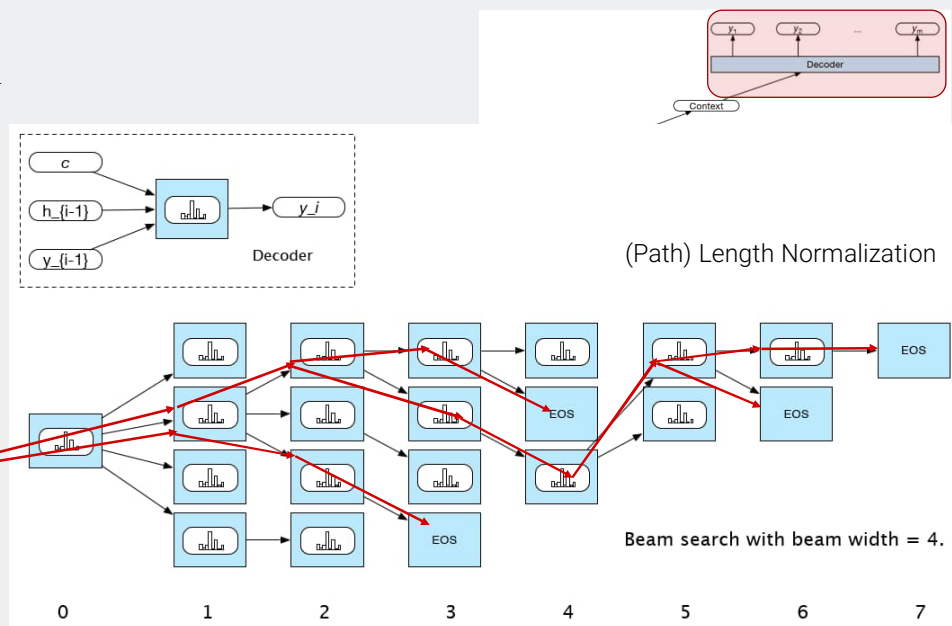
Beam Search

Scoring:

$$P(y_i | y_{<i})$$

hypotheses

Search Frontier



This process continues until a `<\s>` is generated indicating that a complete candidate output has been found. At this point, the completed hypothesis is removed from the frontier and the size of the beam is reduced by one. The search continues until the beam has been reduced to 0. Leaving us with B hypotheses to consider.

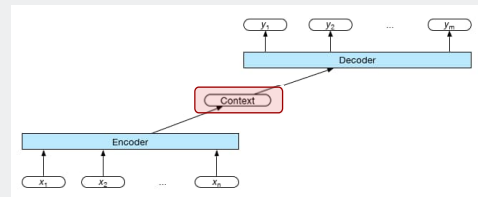
One final complication arises from the fact that the completed hypotheses may have different lengths. Unfortunately, due to the probabilistic nature of our scoring scheme, longer hypotheses will naturally look worse than shorter ones just based on their length. This was not an issue during the earlier steps of decoding; due to the breadth-first nature of beam search all the hypotheses being compared had the same length. The usual solution to this is to apply some form of length normalization to each of the hypotheses. With normalization, we have B hypotheses and can select the best one, or we can pass all or a subset of them on to a downstream application with their respective scores.

Context

- Context available only once.
- Function of the hidden encoder states

$$c = f(h_1^n)$$

- Variable number of hidden states!
- Bi-RNNs (end states of forward & backward passes, separate or concatenated)
- Average over encoder hidden states



We've defined the context vector c as a function of the hidden states of the encoder, that is, $c = f(h_{1:n})$. Unfortunately, the number of hidden states varies with the size of the input, making it difficult to just use them directly as a context for the decode.

The basic approach described earlier avoids this issue since c is just the final hidden state of the encoder. Which is simple and reduces the context to a fixed length vector.

However, this final hidden state is inevitably more focused on the latter parts of input sequence, rather than the input as whole.

One solution to this problem is to use Bi-RNNs, where the context can be a function of the end state of both the forward and backward passes. One could concatenate the final states of the forward and backward passes.

An alternative is to simply sum or average the encoder hidden states to produce a context vector. Unfortunately, this approach loses useful information about

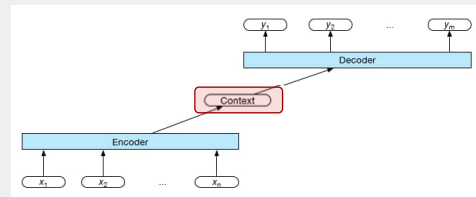
each of the individual encoder states that might prove useful in decoding.

Which has been a motivating factor to develop the mechanism called “attention”.

Attention

Attention

- Take all encoder context
- Dynamically update during decoding.
- Function of the hidden encoder states
- Condition the decoding on the dynamic context
 - Relevance of **encoder** hidden states to the current decoder state
 - Use softmax to normalize these scores
 - Vector of weights



$$h_i^d = g(\hat{y}_{i-1}, h_{i-1}^d, c_i)$$

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d \cdot h_j^e$$

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d W_s h_j^e$$

$$\alpha_{ij} = \text{softmax}(\text{score}(h_{i-1}^d, h_j^e) \quad \forall j \in e)$$

$$c_i = \sum_j \alpha_{ij} h_j^e$$

Additional information:

To overcome the deficiencies of these simple approaches to context, we'll need a mechanism that can take the entire encoder context into account, that dynamically updates during the course of decoding, and that can be embodied in a fixed-size attention vector. Taken together, we'll refer such an approach as an "attention mechanism".

Our first step is to replace the static context vector with one that is dynamically derived from the encoder hidden states, at each point during decoding. This context vector, c_i (where i is the decoding state), is generated anew with each decoding step i and takes all of the encoder hidden states into account in its derivation.

We then make this context available during decoding (see first formula) by conditioning the computation of the current decoder state on it, along with the prior hidden state and the previous output generated by the decoder.

The first step in computing c_i is to compute a vector of scores that capture the

relevance of each encoder hidden state to the decoder state captured in $h^d_{(i-1)}$ (see second formula). That is, at each state i during decoding we'll compute $\text{score}(h^d_{(i-1)}, h^e_j)$ for each encoder state j .

For now, let's assume that this score provides us with a measure of how similar the decoder hidden state is to each encoder hidden state. Use the dot-product between vectors.

The result of the dot product is a scalar that reflects the degree of similarity between the two vectors. And the vector of scores over all the encoder hidden states gives us the relevance of each encoder state to the current step of the decoder.

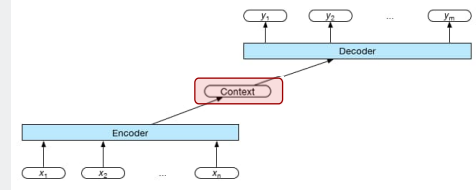
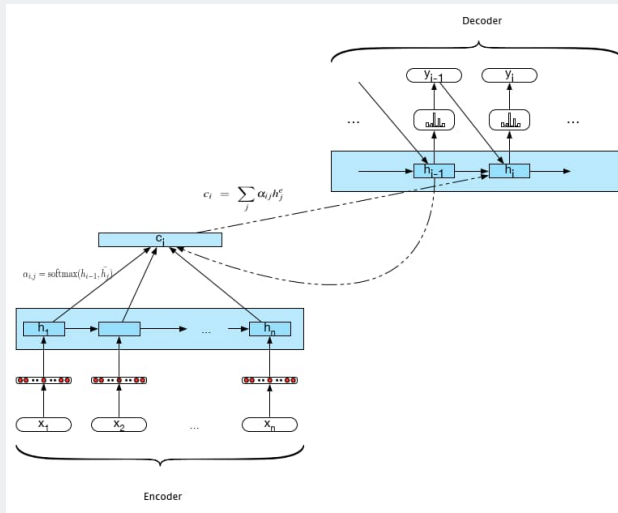
While the simple dot product can be effective, it is a static measure that does not facilitate adaptation during the course of training to fit the characteristics of given applications. A more robust similarity score can be obtained by parameterizing the score with its own set of weights, W_s

To make use of these scores, we'll next normalize them with a softmax to create a vector of weights, α_j , that tells us the proportional relevance of each encoder hidden state j to the current decoder state, i .

Finally, given the distribution in α , we can compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states.

With this, we finally have a fixed-length context vector that takes into account information from the entire encoder state that is dynamically update to reflect the needs of the decoder at each step of decoding.

Attention



$$h_i^d = g(\hat{y}_{i-1}, h_{i-1}^d, c_i)$$

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d \cdot h_j^e$$

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d W_s h_j^e$$

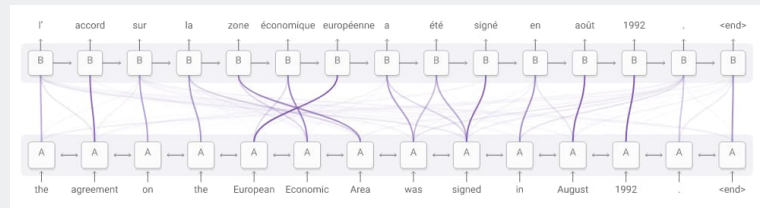
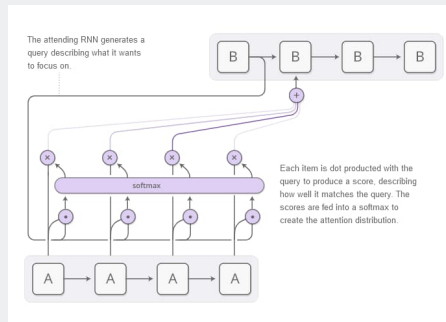
$$\alpha_{ij} = \text{softmax}(\text{score}(h_{i-1}^d, h_j^e) \forall j \in e)$$

$$c_i = \sum_j \alpha_{ij} h_j^e$$

Attention

<https://distill.pub/2016/augmented-rnns/>

Live tool to observe which words in the input sequence affect which part of the output sequence



Content

- Sequence-to-sequence (Encoder-Decoder)
- Attention

"Attention is All You Need" <https://arxiv.org/pdf/1706.03762.pdf>
<https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html>
<https://www.analyticsvidhya.com/blog/2019/06/understanding-transformers-nlp-state-of-the-art-models/>
<https://mlexplained.com/2017/12/29/attention-is-all-you-need-explained/>