ECE 696B: Spring 2025
Trustworthy Machine Learning

Lecture 2: *N-Grams, RNNs, LSTMs, Seq2Seq, BLEU*
(Prelude to Attention models)

Instructor: Dr Ravi Tandon

Department of ECE

Lecture Outline

- N-Gram Models
- RNNs & LSTMs
- Tokenization in LLMs
- Sequence-to-Sequence Models
- BLEU Score

An Application: Machine Translation

• Machine Translation (MT): Problem of converting sentences/text from one language to another language. E.g., English to Spanish (or vice versa).

•Example: Convert from English to French:

Input (English): “The weather is beautiful today”

Output (French): “Le temps est magnifique aujourd'hui.”

•Evolution of MT models:

- N-gram models. (1980s- early 2010s)
- RNN based NMT (Neural Machine Translation) (mid 2010s)
- Attention based NMT (2015-2017)
- Transformer based NMT (2017+)
- Pre-trained models (2018+); multi-lingual, few-shot translation

N-Gram Models (1)

- How do N-gram models work ? – they are trained to learn a conditional probability distribution on outputs for a fixed input "context" length.
- Context length (or memory size) = (N-1)
- 1-gram **(Unigram)** model (memory =0)
  - $p(a)$ for all "a" in vocabulary
- 2-gram **(Bigram)** (memory = 1)
  - $p(a|b)$ for all "a", "b" in vocabulary
- 3-gram **(Trigram)** (memory =2)
  - $p(a|b,c)$ for all possible triples "a", "b", "c" in vocabulary
- Example: take a text corpus with 100000 words;
- # of possible combinations of length N = $10000^N = 10^{5N}$
- As N (context length) increases, N-gram models do not scale well…too many sequences to learn from..

N-Gram Models (2)

- Inference/Decoding: Auto-regressive decoding via N-gram models:
- Example: N=2 : $p(w_t|w_{t-1})$ → complete the t-th word based on the memory from one past word..
- Example: N=3 : $p(w_t|w_{t-1}, w_{t-2})$ → complete the t-th word based on the memory from two past word..
- Sample text (Shakespeare) generated by N-gram models..

Unigram Model	**Bigram Model**
To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have	What means, sir. I confess she? then all sorts, he is trim, captain.
Every enter now severally so, let	Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.
Hill he late speaks; or! a more to leg less first you enter	What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?
Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like	Enter Menenius, if it so many good direction found'st thou art a strong upon command of fear not a liberal largess given away, Falstaff! Exeunt

N-Gram Models (3)

- Inference/Decoding: Auto-regressive decoding via N-gram models:
- Example: N=2 : $p(w_t|w_{t-1})$ → complete the t-th word based on the memory from one past word..
- Example: N=3 : $p(w_t|w_{t-1}, w_{t-2})$ → complete the t-th word based on the memory from two past word..
- Sample text (Shakespeare) generated by N-gram models..

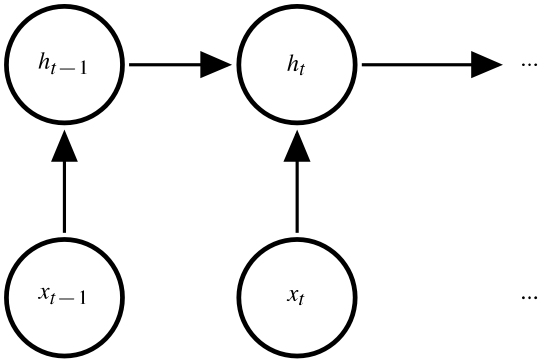
Trigram Model	**Quadrigram Model**
Sweet prince, Falstaff shall die. Harry of Monmouth's grave.	King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;
This shall forbid it should be branded, if renown made it empty.	Will you not tell me who I am?
Indeed the duke; and had a very good friend.	It cannot be but so.
Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.	Indeed the short and the long. Marry, 'tis a noble Lepidus.

Recurrent Neural Networks (RNNs)

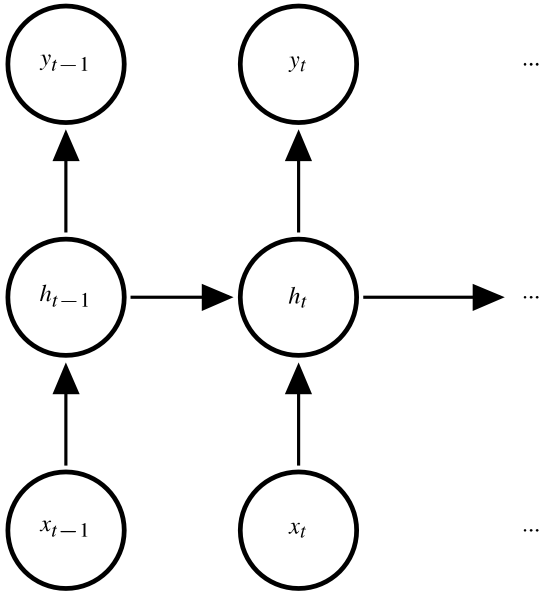
Key idea: $h_t = g_\theta(h_{t-1}, x_t)$. A *hidden state* carries longer-term context information

- RNNs use a neural network for this evolution of hidden state (but it needn't be)
- A *single, fixed* network g_θ governs transitions (cf. HMM transition matrix)

Output can be h_t



Output can be $y_t|h_t$ (cf. Markov model vs HMM)



Recurrent Neural Networks (RNNs)

Consider the following simple *character* model:

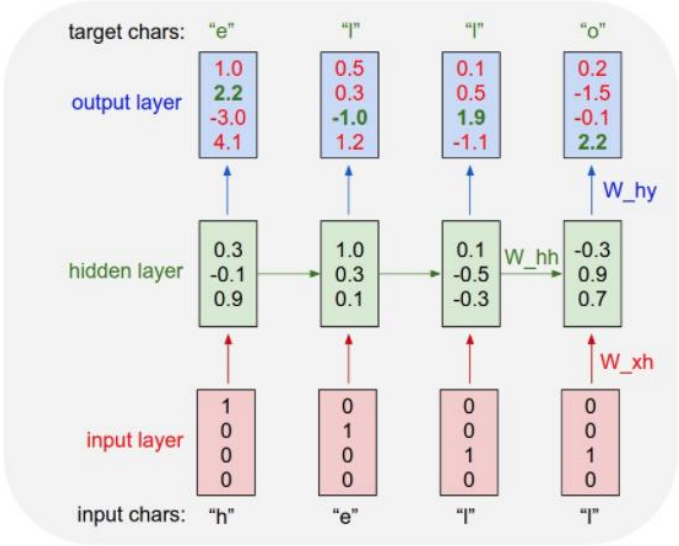
- alphabet consists of $\{h, e, l, o\}$, one-hot encoded
- hidden layers evolve as $h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t)$

... (σ is usual activation nonlinearity, here tanh)

- output $y_t = W_{hy}h_t$ (think logits ... then take softmax)

Whh, Wxh
(weights of hidden layers)

Wyh
(weights of output layers)



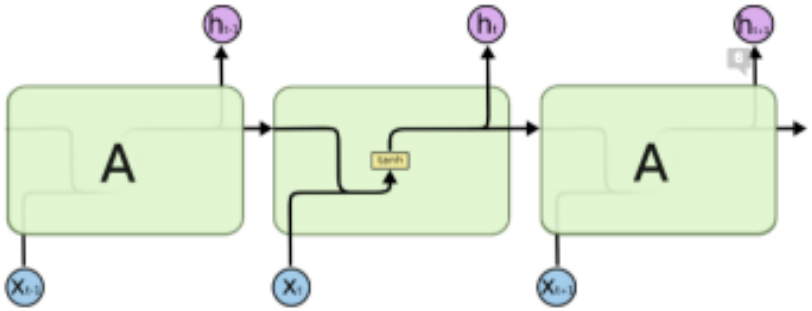
Weights shared across layers! (fewer parameters to learn)

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Intent: h_t carries longer-range context, without exponential parameters of N-gram models.

Recurrent Neural Networks (RNNs)

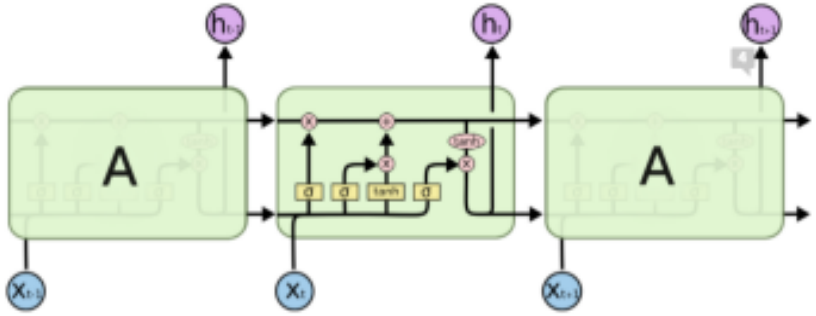
Original RNN



- RNNs keep a "hidden" state to track for memory
- Weights shared across layers! (fewer parameters to learn)
- RNNs suffer from vanishing gradients problem

$$h_t = tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

Long Short Term Memory (LSTMs)



$$
\begin{aligned}
f_t &= \sigma\left(W_{xf} x_t + W_{hf} h_{t-1} + b_f\right) \\
i_t &= \sigma\left(W_{xi} x_t + W_{hi} h_{t-1} + b_i\right) \\
\tilde{c}_t &= \tanh\left(W_{xc} x_t + W_{hc} h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo} x_t + W_{ho} h_{t-1} + b_o\right) \\
c_t &= c_{t-1} \odot f_t + \tilde{c}_t \odot i_t \\
h_t &= \tanh\left(c_t\right) \odot o_t
\end{aligned}
$$

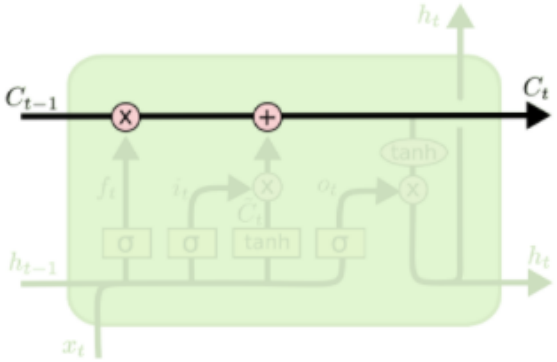
LSTMs introduce a "gating" mechanism for better incorporating memory/context

- **Forget gate (f):** which information to discard from previous cell state
- **Input gate (i):** which information to store in current cell state
- **Candidate cell state (~c):** proposed new candidate valued to be stored in cell state
- **Cell state (c) :** this is the memory of LSTM with long-term dependencies
- **Output gate (o):** controls what information to be passed from previous hidden state and the output
- **Hidden state (h):** this is the output of LSTM at current time step

LSTM Cell State

Rather than hidden state h_t, we now pass h_t and a *cell* state c_t

- This is no problem: define $\bar{h}_t$, $\begin{bmatrix} h_t \\ c_t \end{bmatrix}$, and it is still an RNN.



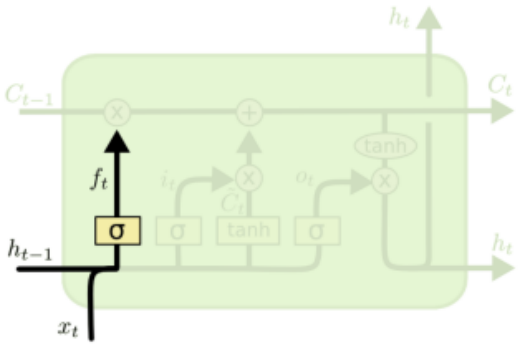
The cell state:

- provides a channel for long-range information/memory to propagate forward
- without corrupting/compromising the hidden state (which is directly output relevant)

LSTM Forget Gate

Now we must consider how the hidden state and cell state interact. First, the *forget gate*:

- Conceptually, f_t chooses to forget or pass the current cell state
- Elementwise forgetting, so it is doing so individually for each unit (the width) of c_t



$$f_t = \sigma(W_{xf} x_t + W_{hf} h_{t-1} + b_f)$$

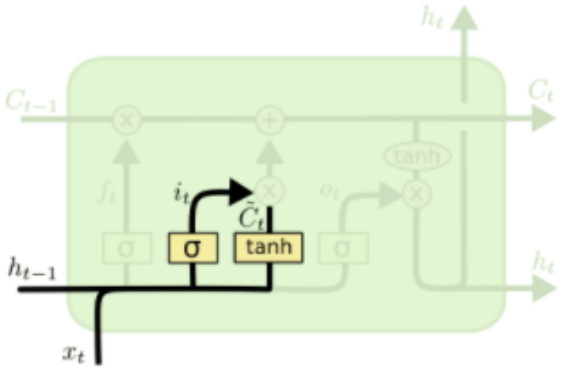
The forget gate

- can be thought of as projecting dimensions of x_t and h_{t-1}
- ... that remove or persist certain dimensions of c_t
- Convince yourself that this is a useful way to free or hold data in memory
- Note: σ must be *2* [0, 1], but can be sigmoid, tanh, etc...

LSTM Input Gate

Continuing hidden state and cell state interaction. The *input gate*:

- If f_t chooses to forget or pass the existing cell state...
- Input i_t chooses what to pass in as a new cell state
- Again elementwise...



$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$
$$\tilde{c}_t = tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

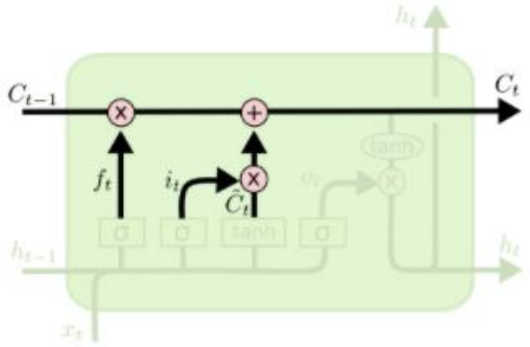
The input gate

- can be thought of as projecting dimensions of x_t and h_{t-1}
- ... that load or ignore certain dimensions of the new proposed cell state $\tilde{c}_t$
- Convince yourself that this is a useful way to load/not load data into memory
- Note: again σ must be $\in [0, 1]$, but can be sigmoid, tanh, etc...

LSTM Cell State (again)

The effects of the forget and input gates are then loaded onto the cell state c_t:

- Elementwise action of persisting/overwriting the long-term memory cell c_t



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

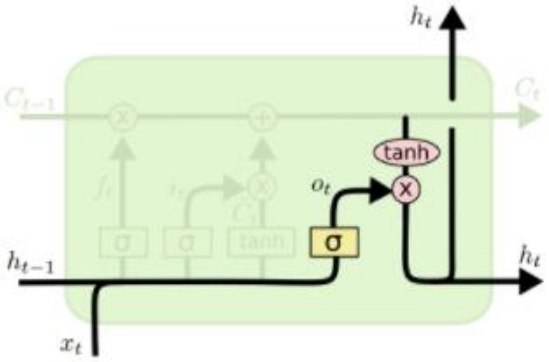
Critical to intuition:

- This is neural networks, so we hope to *learn* from data when to forget, load, etc.
- All operations here are elementwise, so many different loads/persists occur in parallel
- So far we haven't affected h_t yet...

LSTM Output Gate

Continuing hidden state and cell state interaction, but now to h_t. The *output gate*:

- If f_t chooses to forget or pass, and i_t chooses what to pass...
- o_t chooses when to write out the cell c_t to h_t.



$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$
$$h_t = tanh(c_t) \odot o_t$$

Same as before: the output gate is a useful way to send data onto h_t

Note the key and complementary differences here between h_t and c_t;

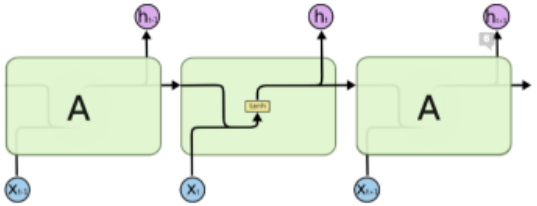
- h_t is either the output or parameterizes the output $y_t|h_t$.
- h_t thus has short-term or more immediately relevant data
- c_t can persist over long-range periods and needn't (directly) drive output (o_t)

RNNs vs LSTMs

We have built up the structure of a standard LSTM

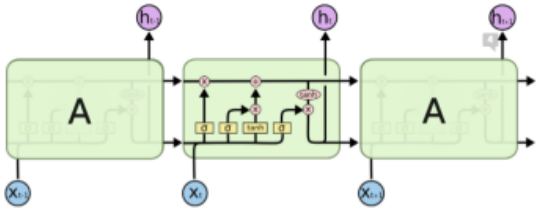
- there are many minor variants
- but all share the basic forget/input/output and cell/hidden components
- thankfully, neural network libraries abstract all these blocks and parameters away
- The key reminder: like a CNN, this is just a (highly engineered) neural network $g_{\checkmark}$

Original RNN



$$h_t = tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

Full LSTM



$$\begin{aligned} f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\ i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\ \tilde{c}_t &= tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\ o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\ c_t &= c_{t-1} \odot f_t + \tilde{c}_t \odot i_t \\ h_t &= tanh(c_t) \odot o_t \end{aligned}$$

Tokenization in LLMs

- *What is Tokenization: Tokenization is the process of breaking down text into smaller units ("tokens") for processing by language models.*

- Types of tokens for language models:
    1. Words: whole words treated as "tokens"
    2. **Sub-words: pieces of words (prefix, suffix etc) can be tokens (most commonly used)**
    3. Characters: individual letters or symbols

- **Word-based Tokenization:**
    - Example: "I love AI." → ["I", "love", "AI"]
    - Pros: Simplicity.
    - Cons: Large vocabulary; doesn't handle rare/misspelled words well.

- **Subword Tokenization (e.g., Byte-Pair-Encoding (BPE)):**
    - Example: "running" → ["run", "ning"]
    - Pros: Compact vocabularies, effective for unseen words.
    - Cons: More computational complexity.

- **Character-level Tokenization:**
    - Example: "AI" → ["A", "I"]
    - Pros: Handles rare words naturally.
    - Cons: Long sequences, harder for models to learn dependencies.

Tokenization in LLMs

- Each token in Vocabulary is assigned a high-dimensional embedding (Embedding dim = D)

Model	V (vocabulary size)	D (embedding dim)	Tokenizer
GPT-2 (S, M, L)	50257	768, 1024, 1280	BPE
GPT-3	50257	12,288	BPE
BERT	30522	768	WordPiece
T5		32000	SentencePiece
LLaMA	32000	4096	SentencePiece

V = # of tokens in the vocabulary

D = Size of the embedding vector for each token

E (embedding matrix) = V x D matrix (can think of it as a look-up table)

Byte-Pair Encoding (BPE): *Vocabulary Creation*

- Goal: given a large text corpus, create a vocabulary of words/sub-words of a desired size which can handle hopefully unseen words. We also need to learn "merge rules", i.e., if/how to combine adjacent tokens.

- **Step 1 (initialize tokens)**
  - Corpus: "low lower lowest" Tokens: ["l o w _", "l o w e r _", "l o w e s t _"]
- **Step 2 (count adjacent pair frequencies)**
  - Pair Frequencies: ("l", "o") → 3, ("o", "w") → 3, ("w", "_") → 3, ("w", "e") → 2, ("e", "r") → 2, ("e", "s") → 1, ("s", "t") → 1
- **Step 3 (Merge most frequent pair)**
  - Tokens after merging: ["lo w _", "lo w e r _", "lo w e s t _"]
- **(Keep repeating Steps 2 and 3.. compute new pair frequencies & merge again..)**
  - Pair Frequencies: ("lo", "w") → 3, ("w", "_") → 3, ("w", "e") → 2, ("e", "r") → 2, ("e", "s") → 1, ("s", "t") → 1
- Final Vocabulary:["l", "o", "w", "lo", "low", "low_", "e", "r", "s", "t", "_"]
- Final Merge Rules:
  1. ("l", "o") → "lo" (rule learned in 1st iteration)
  2. ("lo", "w") → "low" (rule learned in 2nd iteration)
  3. ("low", "_") → "low_" (rule learned in 3rd iteration)..and so on

Byte-Pair Encoding (BPE): *Inference Stage*

- We are provided a Vocabulary of tokens: ["l", "o", "w", "lo", "low", "er", "low_", "er_", "_"]
- We are provided the Merge Rules:
    1. ("l", "o") → "lo"
    2. ("lo", "w") → "low"
    3. ("low", "_") → "low_"
    4. ("e", "r") → "er"
    5. ("er", "_") → "er_"

Goal: given some text, we wish to convert it to a sequence of tokens via BPE

Example: Input is "lower"--> Initial Tokens: ["l", "o", "w", "e", "r", "_"] *Next apply merge rules* ***in order***

Merge ("l", "o"): Apply Rule 1: ("l", "o") → "lo" Updated Tokens: ["lo", "w", "e", "r", "_"]

Merge ("lo", "w"): Apply Rule 2: ("lo", "w") → "low" Updated Tokens: ["low", "e", "r", "_"]

Merge ("low", "_"): Apply Rule 3: ("low", "_") → "low_" Updated Tokens: ["low_", "e", "r", "_"]

Merge ("e", "r"): Apply Rule 4: ("e", "r") → "er" Updated Tokens: ["low_", "er", "_"

Merge ("er", "_"): Apply Rule 5: ("er", "_") → "er_" Updated Tokens: ["low_", "er_"]

- Final Tokens: ["low_", "er_"]
- Vocabulary Mapping: "low_" → 6 "er_" → 7
- Token IDs: [6, 7]

“Seq2seq” Paper (2014)

Sequence to Sequence Learning with Neural Networks

Ilya Sutskever
Google
ilyasu@google.com

Oriol Vinyals
Google
vinyals@google.com

Quoc V. Le
Google
qvl@google.com

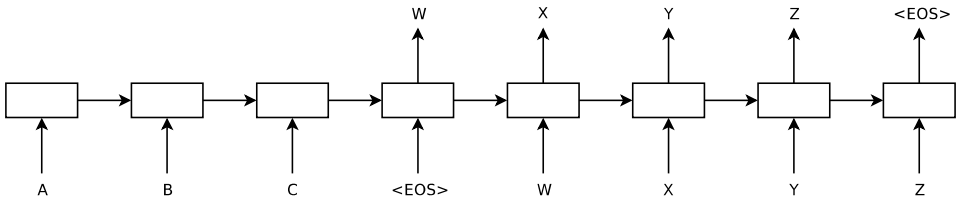
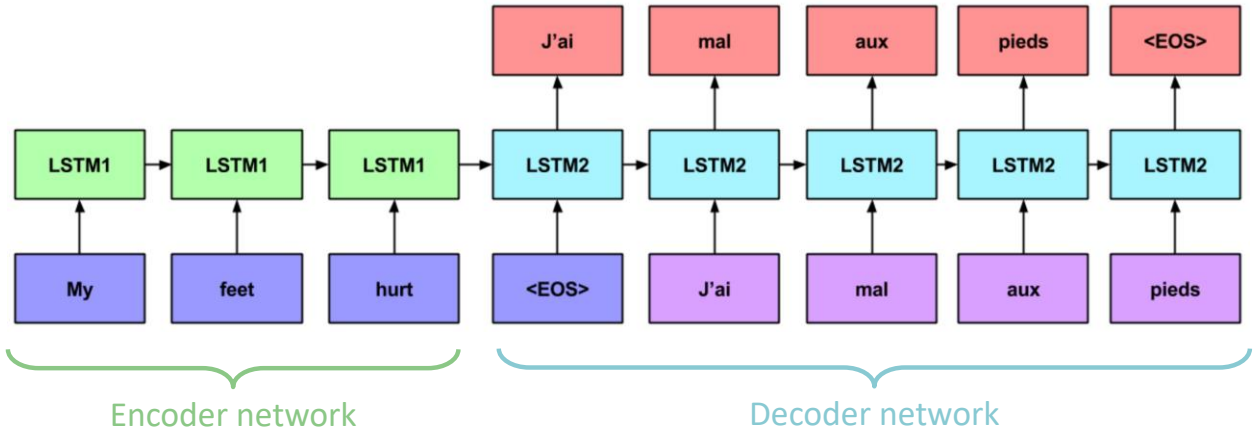


Figure 1: Our model reads an input sentence “ABC” and produces “WXYZ” as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

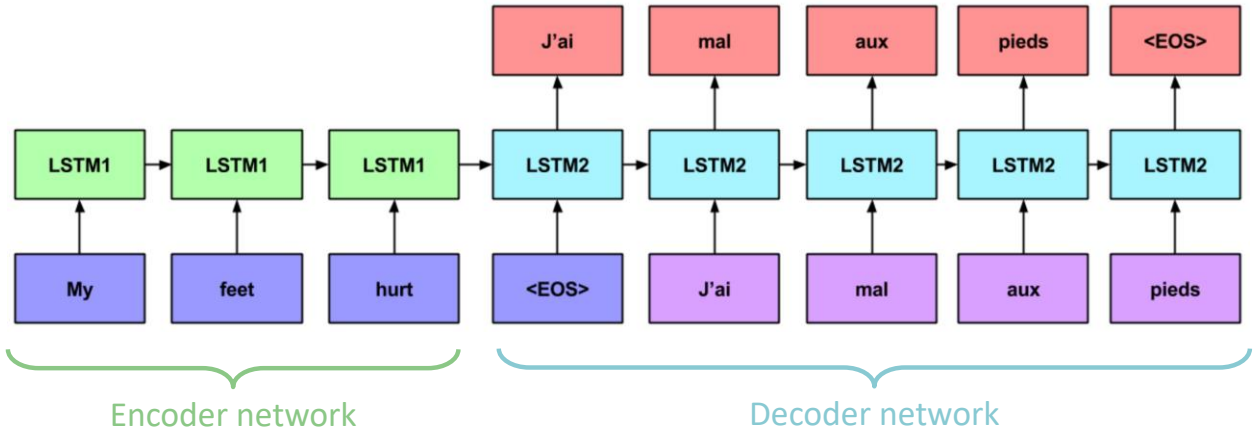
- Introduced a new approach for sequence-to-sequence translation
- **Proposed an Encoder-Decoder architecture**
- Encoder: takes an input sequence and compute a “hidden” representation of input
- Decoder: Use hidden representation to sequentially decode/produce output.

“Seq2seq” Paper: *Encoder*



- Given an input sequence X: first tokenize it, say we obtain (x1, x2,..,xT)
- ***Encoder:*** *sequentially process the input and compute hidden representations*
  - *h1 = Enc(x1, h0)*
  - *h2 = Enc(x2, h1)*
  - *h3 = Enc(x3, h2)…and so on till one reaches end: hT = Enc(xT, h(T-1)).*
- *v = hT is the fixed representation of the entire input sequence*
- *Decoder:* $p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_1, \ldots, y_{t-1})$

"Seq2seq" Paper: *Decoder*



- *v = hT is the fixed representation of the entire input sequence*
- ***Decoder:***
  - *hdec(1) = Dec(V, 0) → Softmax(Whdec(1)+ b(1)) (over vocab) → decode y1*
  - *hdec(2) = Dec(V, hdec(1)) → Softmax(Whdec(2) + b(2)) → decode y2*
  - *hdec(3) = Dec(V, hdec(2)) → Softmax(Whdec(3) + b(3)) → decode y2*
- *Auto-regressive decoding as before.*
- *Each "Dec(..)" is a trainable LSTM block ("deep" LSTMs were used)*

“Seq2seq”: Datasets & Performance Metrics

- Used WMT 14 English-to-French Translation Dataset (WMT = workshop on Machine translation)

- Training Data: 36 million English-French sentence pairs

- Test Data: 4.5 million Eng-Fr sentence pairs

- Performance Metric: **BLEU Score (Bilingual Evaluation Understudy)**
  - BLEU measures the closeness of a machine translation to human translation
  - Takes set of MT generated sequences and reference sequences
  - Measures the n-gram overlap between these two sets of sequences
  - BLEU Scores range from 0 (no overlap) to 100 (highest overlap)

BLEU Score (Bilingual Evaluation Understudy)

How Does It Work?

- **Step 1:** Tokenization: Split text into words or subwords.
- **Step 2:** n-Gram Precision: Calculate precision for n-grams:

  Precision(n) = (Count of Matching n-grams) / (Count of Total n-grams in MT)
- **Step 3:** Brevity Penalty (BP): Penalizes short translations:

  BP = 1, if c > r or exp(1 - r/c), if c ≤ r, where c: MT length, r: reference length.
- **Step 4:** Final BLEU Score: Combine precision and BP:

  BLEU = BP × exp(Σ (w_n × log Precision_n)) (Default w_n = 1/N).

Example: Reference: **The cat is on the mat.** MT: **The cat is on mat.**

- 1-gram precision: 5/5 = 1.0
- 2-gram precision: 3/4 = 0.75
- Brevity Penalty: exp(1 - 6/5)
- BLEU = 0.82 × exp(0.5 × (log(1.0) + log(0.75))) ≈ 0.71.

“Seq2seq” Paper: Results

Method	test BLEU score (ntst14)
Bahdanau et al. [2]	28.45
Baseline System [29]	33.30
Single forward LSTM, beam size 12	26.17
Single reversed LSTM, beam size 12	30.59
Ensemble of 5 reversed LSTMs, beam size 1	33.00
Ensemble of 2 reversed LSTMs, beam size 12	33.27
Ensemble of 5 reversed LSTMs, beam size 2	34.50
Ensemble of 5 reversed LSTMs, beam size 12	**34.81**

- Key Takeaway(s) from Seq2Seq paper
  - •Introduced a new approach for sequence to sequence translation
  - •Proposed an Encoder-Decoder architecture
  - •Impressive results on language translation tasks
  - • Shortcomings: reliance on a single hidden representation of the input
  - • Harder to deal with longer sentences.
- What next? “Attention is all you need paper”