**Language Model**: A system that predicts the next word

**Recurrent Neural Network**: A family of neural networks that:

- Take sequential input of any length; apply the same weights on each step
- Can optionally produce output on each step

Recurrent Neural Network ≠ Language Model

- RNNs can be used for many other things (see later)

Language Modeling is a traditional subcomponent of many NLP tasks, all those involving generating text or estimating the probability of text:

- Now everything in NLP is being rebuilt upon Language Modeling: GPT-3 is an LM!
- Language modeling can be done with different models, e.g., *n*-grams or transformers

essentially everything we do in NLP is being done by Language models

Language Model (語言模型) ≠ Language modeling (語言建模)

① Evaluation

## Evaluating Language Models

困惑度

- The standard evaluation metric for Language Models is perplexity.

$$\text{perplexity} = \prod_{t=1}^{T} \left( \frac{1}{P_{\text{LM}}(\boldsymbol{x}^{(t+1)} | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})} \right)^{1/T}$$

Normalized by number of words

Inverse probability of corpus, according to Language Model

- This is equal to the exponential of the cross-entropy loss $J(\theta)$:

$$= \prod_{t=1}^{T} \left( \frac{1}{\hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}}} \right)^{1/T} = \exp\left( \frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}} \right) = \exp(J(\theta))$$

**Lower** perplexity is better!

the probability of predicting ever word with model

$$P_{LM}(x^{(1:T)}) = \prod_{t=1}^{T} P(x^{(t+1)} | x^{(1:t)})$$

$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \log P(x^{(t+1)} \mid x^{(1:t)})$

$\text{perplexity} = e^{J(\theta)}$

Be aware, people use base e for measuring perplexity now.
perplexity will be different depending on what base you're using for things

## ② The problems of RNNs

### Vanishing gradient and exploding gradient

#### Vanishing gradient proof sketch (linear case)

- Recall: $$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)$$
- What if $\sigma$ were the identity function, $\sigma(x) = x$ ?

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = \operatorname{diag}\left(\sigma'\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right)\right) \boldsymbol{W}_h \qquad \text{(chain rule)}$$
$$= \boldsymbol{I}\, \boldsymbol{W}_h = \boldsymbol{W}_h$$

- Consider the gradient of the loss $J^{(i)}(\theta)$ on step $i$, with respect to the hidden state $\boldsymbol{h}^{(j)}$ on some previous step $j$. Let $\ell = i - j$

$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \qquad \text{(chain rule)}$$

$$= \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \prod_{j<t\leq i} \boldsymbol{W}_h = \frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{\ell} \qquad \text{(value of } \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}\text{)}$$

If $W_h$ is "small", then this term gets exponentially problematic as $\ell$ becomes large

#### Vanishing gradient proof sketch (linear case)

- What's wrong with $\boldsymbol{W}_h^{\ell}$ ?
- Consider if the eigenvalues of $\boldsymbol{W}_h$ are all less than 1: (sufficient but not necessary)

$$\lambda_1, \lambda_2, \ldots, \lambda_n < 1$$
$$\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_n \text{ (eigenvectors)}$$

- We can write $\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{\ell}$ using the eigenvectors of $\boldsymbol{W}_h$ as a basis:

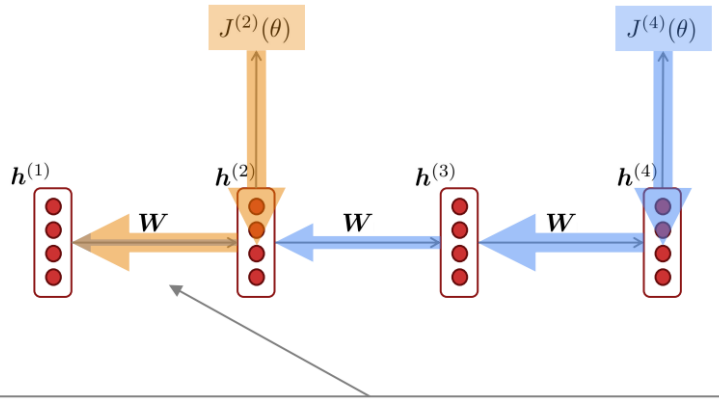
$$\frac{\partial J^{(i)}(\theta)}{\partial \boldsymbol{h}^{(i)}} \boldsymbol{W}_h^{\ell} = \sum_{i=1}^{n} c_i \lambda_i^{\ell} \boldsymbol{q}_i \approx \boldsymbol{0} \text{ (for large } \ell\text{)}$$

Approaches 0 as $\ell$ grows, so gradient vanishes

- What about nonlinear activations $\sigma$ (i.e., what we use?)
  - Pretty much the same thing, except the proof requires $\lambda_i < \gamma$ for some $\gamma$ dependent on dimensionality and $\sigma$

These function like tanh and sigmod themselves also have a gradient upper (梯度上界) bound less than 1 (their maximum derivative is 1), after each time step, the gradient not only needs to be multiplied by Wh, but also by the derivative of the activation function. This is equivalent to multiplying the gradient by a factor less than 1, exacerbating the vanishing gradient

## Why is vanishing gradient a problem?




Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are updated only with respect to near effects, not long-term effects.

## Effect of vanishing gradient on RNN-LM

- **LM task:** *When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her* _____
- To learn from this training example, the RNN-LM needs to model the dependency between *"tickets"* on the 7th step and the target word *"tickets"* at the end.
- But if the gradient is small, the model can't learn this dependency
  - So, the model is unable to predict similar long-distance dependencies at test time

In practice, a simple RNN will only condition ~7 tokens back [vague rule-of-thumb]

2

## Why is exploding gradient a problem?

- If the gradient becomes too big, then the SGD update step becomes too big:

$$\theta^{new} = \theta^{old} - \underbrace{\alpha}_{\text{learning rate}} \underbrace{\nabla_\theta J(\theta)}$$

gradient

- This can cause bad updates: we take too large a step and reach a weird and bad parameter configuration (with large loss)
  - You think you've found a hill to climb, but suddenly you're in Iowa
- In the worst case, this will result in Inf or NaN in your network (then you have to restart training from an earlier checkpoint)

However, we have solution for exploding Gradient —— Gradient clipping

and this's a accept wisdom to solve exploding gradient of all models

easy, but it do works well

## Gradient clipping: solution for exploding gradient

- **Gradient clipping**: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update

**Algorithm 1** Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
**if** $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
**end if**

- **Intuition**: take a step in the same direction, but a smaller step
- In practice, **remembering to clip gradients is important**, but exploding gradients are an easy problem to solve

So, Gradient exploding is solvable, but fixing the vanishing gradient seemed a more difficult problem.

## How to fix the vanishing gradient problem?

- The main problem is that *it's too difficult for the RNN to learn to preserve information over many timesteps*.
- In a vanilla RNN, the hidden state is constantly being rewritten

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}\right)$$

- First off next time: How about an RNN with separate memory which is added to?
  - LSTMs
- And then: Creating more direct and linear pass-through connections in model
  - Attention, residual connections, etc.

# ③ Long Short-Term Memory RNNs (LSTMs)

## Long Short-Term Memory RNNs (LSTMs)

- On step *t*, there is a hidden state $\boldsymbol{h}^{(t)}$ and a cell state $\boldsymbol{c}^{(t)}$
  - Both are vectors length *n*
  - The cell stores long-term information
  - The LSTM can read, erase, and write information from the cell
    - The cell becomes conceptually rather like RAM in a computer
- The selection of which information is erased/written/read is controlled by three corresponding gates (gates are calculated things whose values are probabilities)
  - The gates are also vectors of length *n*
  - On each timestep, each element of the gates can be open (1), closed (0), or somewhere in-between
  - The gates are dynamic: their value is computed based on the current context

## Long Short-Term Memory (LSTM)

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

$$\boldsymbol{f}^{(t)} = \sigma\left(\boldsymbol{W}_f\boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f\boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right)$$
$$\boldsymbol{i}^{(t)} = \sigma\left(\boldsymbol{W}_i\boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i\boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right)$$
$$\boldsymbol{o}^{(t)} = \sigma\left(\boldsymbol{W}_o\boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o\boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right)$$
$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c\boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c\boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right)$$
$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \tilde{\boldsymbol{c}}^{(t)}$$
$$\boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \tanh \boldsymbol{c}^{(t)}$$

All these are vectors of same length *n*

Gates are applied using element-wise (or Hadamard) product: ⊙

22

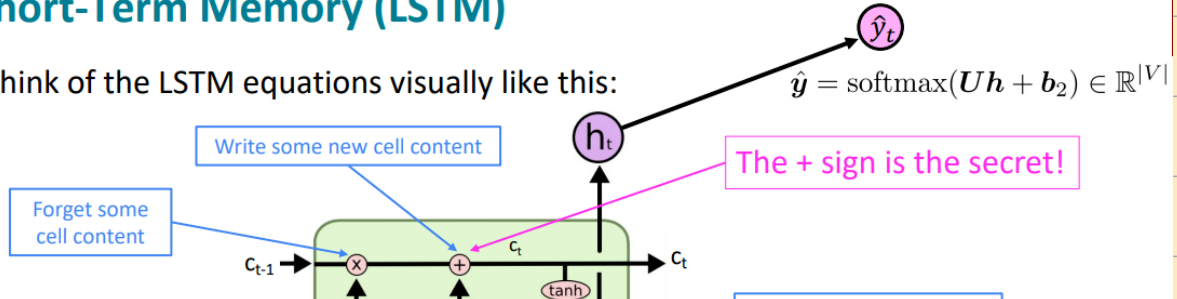
PS:

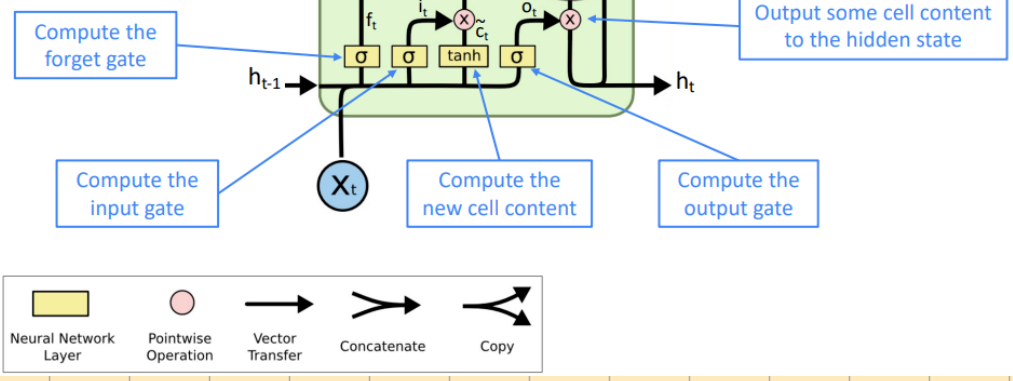
Though it's called Forget gate, but it actually remmers the last hidden state

## Long Short-Term Memory (LSTM)

You can think of the LSTM equations visually like this:

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$






Solve Problems of RNNs

## How does LSTM solve vanishing gradients?

- The LSTM architecture makes it much easier for an RNN to preserve information over many timesteps
  - e.g., if the forget gate is set to 1 for a cell dimension and the input gate set to 0, then the information of that cell is preserved indefinitely.
  - In contrast, it's harder for a vanilla RNN to learn a recurrent weight matrix $W_h$ that preserves info in the hidden state
  - In practice, you get about 100 timesteps rather than about 7
- However, there are alternative ways of creating more direct and linear pass-through connections in models for long distance dependencies

So how?

additive update structure is the key to solving gradient vanish

$$C^{(t)} = f^{(t)} \odot C^{(t-1)} + i^{(t)} \odot \tilde{C}^{(t)}$$

Look at its back propagation

$$\frac{\partial C^{(t)}}{\partial C^{(t-1)}} = \frac{\partial}{\partial C^{(t-1)}} \left[ f^{(t)} \odot C^{(t-1)} + i^{(t)} \odot \tilde{C}^{(t)} \right]$$

$$= \begin{cases} \text{direct path} : f^{(t)} \quad (\text{multiply element-wise}) \\ i^{(t)} \odot \frac{\partial \tilde{C}^{(t)}}{\partial C^{(t-1)}} \end{cases}$$

We can find that first part ($f^{(t)}$) is an additive term that doesn't involve matrix multiplication

So, $\frac{\partial C^{(t)}}{\partial C^{(k)}} = \prod_{j=k+1}^{t} \frac{\partial C^{(j)}}{\partial C^{(j-1)}} \approx \prod_{j=k+1}^{t} f^{(j)}$

if $f_j \approx 1$

the $\prod_{j=k+1}^{t} \approx 1$

it means that gradien can flow from $C^{(t)}$ to $C^{(k)}$ hardly with reduction

other

## Is vanishing/exploding gradient just an RNN problem?

- No! It can be a problem for all neural architectures (including feed-forward and convolutional neural networks), especially very deep ones.
  - Due to chain rule / choice of nonlinearity function, gradient can become vanishingly small as it backpropagates
  - Thus, lower layers are learned very slowly (i.e., are hard to train)
- Another solution: lots of new deep feedforward/convolutional architectures add more direct connections (thus allowing the gradient to flow)

For example: 残差链接 残差网络

- Residual connections aka "ResNet"
- Also known as skip-connections
- The identity connection preserves information by default
- This makes deep networks much easier to train

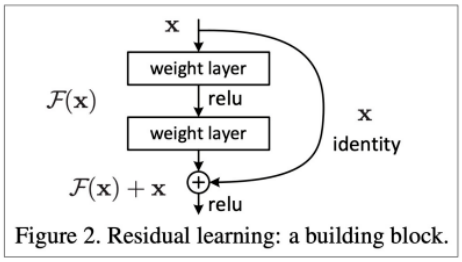


Figure 2. Residual learning: a building block.

## Is vanishing/exploding gradient just a RNN problem?

Other methods:

- Dense connections aka "DenseNet"
- Directly connect each layer to all future layers!

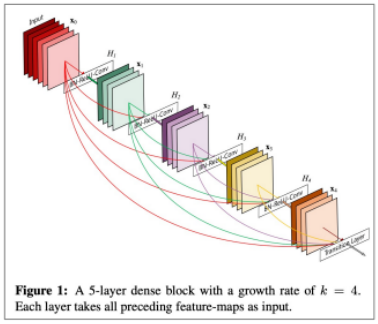
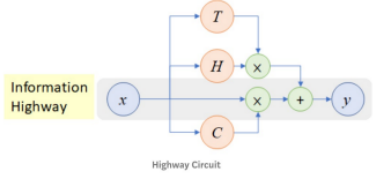


**Figure 1:** A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

- Highway connections aka "HighwayNet"
- Similar to residual connections, but the identity connection vs the transformation layer is controlled by a dynamic gate
- Inspired by LSTMs, but applied to deep feedforward/convolutional networks



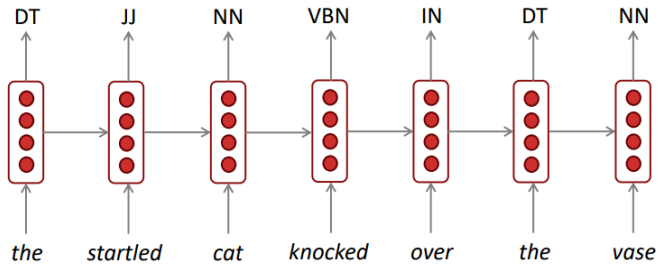

- **Conclusion**: Though vanishing/exploding gradients are a general problem, RNNs are particularly unstable due to the repeated multiplication by the same weight matrix [Bengio et al, 1994]

# ④ Application

① 

## 3. Other RNN uses: RNNs can be used for sequence tagging

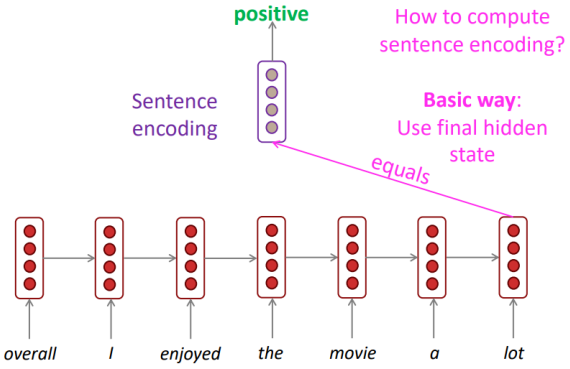
e.g., **part-of-speech tagging**, named entity recognition




② 

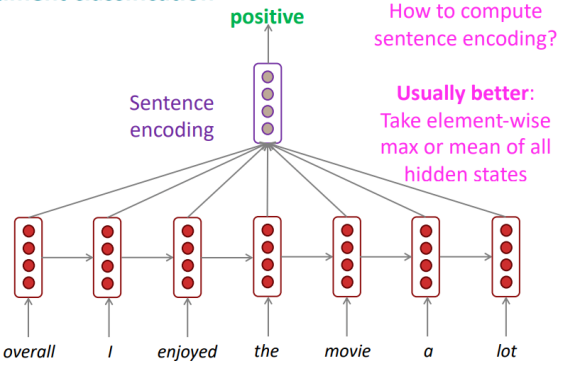
## RNNs can be used as a sentence encoder model

e.g., for **sentiment classification**




## RNNs can be used as a sentence encoder model

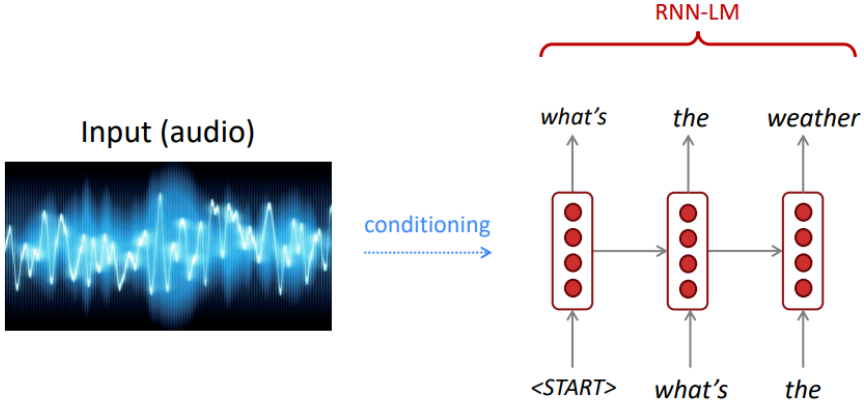
e.g., for **sentiment classification**




③ 

## RNN-LMs can be used to generate text based on other information

e.g., **speech recognition**, machine translation, summarization



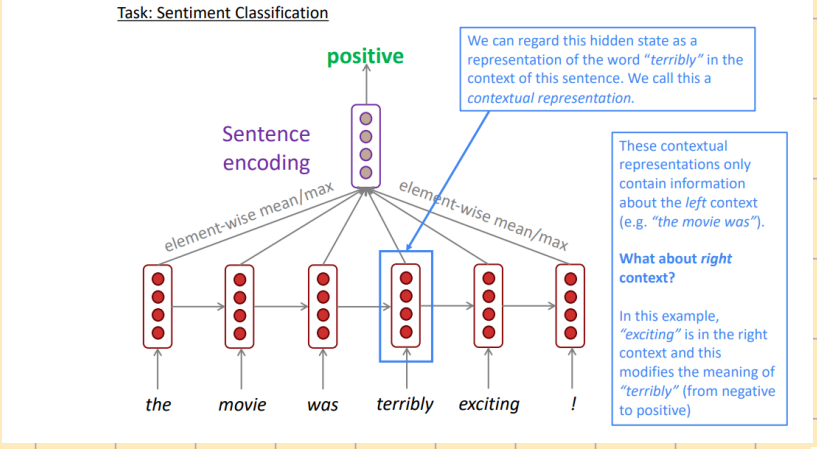

This is an example of a *conditional language model*.
We'll see Machine Translation as an example in more detail

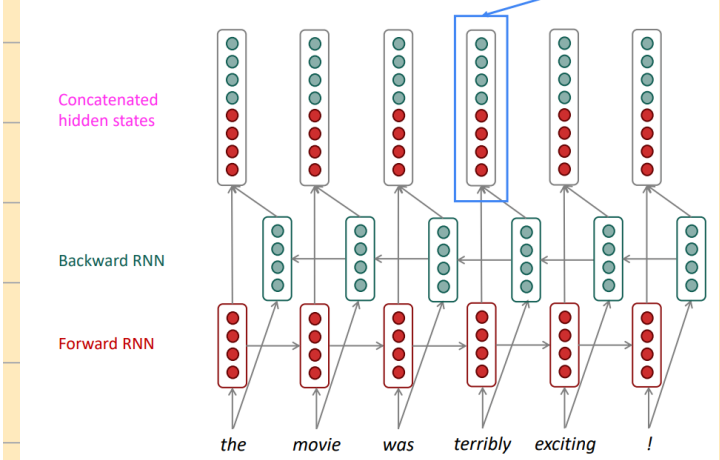
④ Bidirectional RNNs

## Bidirectional RNNs

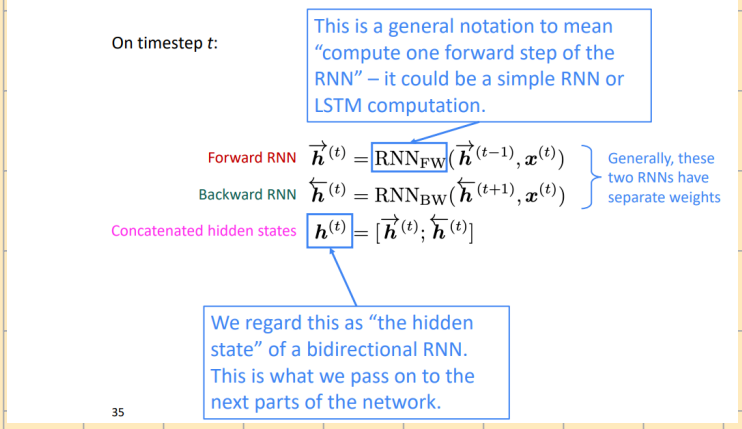
This contextual representation of "terribly" has both left and right context!

## 4. Bidirectional and Multi-layer RNNs: motivation

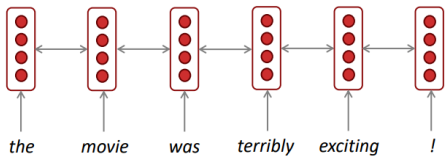



## Bidirectional RNNs




## Bidirectional RNNs: simplified diagram




The two-way arrows indicate bidirectionality and the depicted hidden states are assumed to be the concatenated forwards+backwards states

## Bidirectional RNNs

- Note: bidirectional RNNs are only applicable if you have access to the entire input sequence
  - They are **not** applicable to Language Modeling, because in LM you *only* have left context available.
- If you do have entire input sequence (e.g., any kind of encoding), bidirectionality is powerful (you should use it by default).
- For example, BERT (**Bidirectional** Encoder Representations from Transformers) is a powerful pretrained contextual representation system built on bidirectionality.
  - You will learn more about transformers, including BERT, in a couple of weeks!

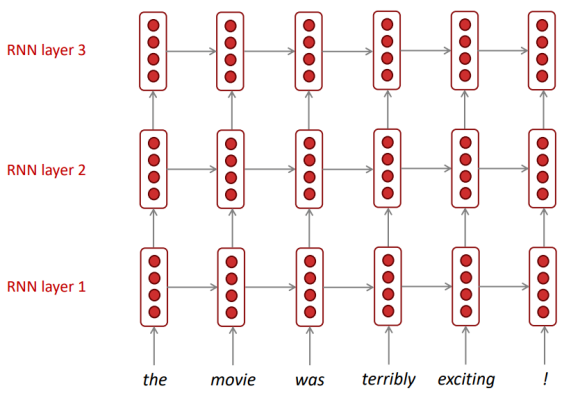
## 5 Multi-layer RNNs

## Multi-layer RNNs

- RNNs are already "deep" on one dimension (they unroll over many timesteps)
- We can also make them "deep" in another dimension by applying multiple RNNs – this is a multi-layer RNN.
- This allows the network to compute more complex representations
  - The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.
- Multi-layer RNNs are also called *stacked RNNs*.

## Multi-layer RNNs

The hidden states from RNN layer *i* are the inputs to RNN layer *i*+1



example in NLP

bottom layer encoder : learn vocabulary, Local phrases' grammer and simple semantics

middle layer encoder : learn more complex phrase structure, Clause relationship

upper layer encoder : learn to capture the main idea, tone, global contextual information of a sentence

also, it means more parameters in the model

## Multi-layer RNNs in practice

- Multi-layer or stacked RNNs allow a network to compute more complex representations – they work better than just have one layer of high-dimensional encodings!
  - The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.
- High-performing RNNs are usually multi-layer (but aren't as deep as convolutional or feed-forward networks)
- For example: In a 2017 paper, Britz et al. find that for Neural Machine Translation, 2 to 4 layers is best for the encoder RNN, and 4 layers is best for the decoder RNN
  - Often 2 layers is a lot better than 1, and 3 might be a little better than 2
  - Usually, skip-connections/dense-connections are needed to train deeper RNNs (e.g., 8 layers)
- Transformer-based networks (e.g., BERT) are usually deeper, like 12 or 24 layers.
  - You will learn about Transformers later; they have a lot of skipping-like connections

Some history

## LSTMs: real-world success

- In 2013–2015, LSTMs started achieving state-of-the-art results
  - Successful tasks include handwriting recognition, speech recognition, machine translation, parsing, and image captioning, as well as language models
  - LSTMs became the dominant approach for most NLP tasks
- Now (2019–2024), Transformers have become dominant for all tasks
  - For example, in **WMT** (a Machine Translation conference + competition):
    - In WMT 2014, there were 0 neural machine translation systems (!)
    - In WMT 2016, the summary report contains "RNN" 44 times (and these systems won)
    - In WMT 2019: "RNN" 7 times, "Transformer" 105 times

# Machine Translation (MT)

## 5. Machine Translation

**Machine Translation (MT)** is the task of translating a sentence *x* from one language (the source language) to a sentence *y* in another language (the target language).

History

## The early history of MT: 1950s

- Machine translation research began in the early 1950s on machines less powerful than high school calculators (before term "A.I." coined!)
- Concurrent with foundational work on automata, formal languages,

probabilities, and information theory

- MT heavily funded by military, but basically just simple rule-based systems doing word substitution
- Human language is more complicated than that, and varies more across languages!
- Little understanding of natural language syntax, semantics, pragmatics
- Problem soon appeared intractable

## 1990s–2010s: Statistical Machine Translation

- SMT was a huge research field
- The best systems were extremely complex
  - Hundreds of important details
- Systems had many separately-designed subcomponents
  - Lots of feature engineering
    - Need to design features to capture particular language phenomena
  - Required compiling and maintaining extra resources
    - Like tables of equivalent phrases
  - Lots of human effort to maintain
    - Repeated effort for each language pair!

2014: Neural Machine Translation (NMT)

## NMT: the first big success story of NLP Deep Learning

Neural Machine Translation went from a fringe research attempt in **2014** to the leading standard method in **2016**

## 6. What is Neural Machine Translation?

- Neural Machine Translation (NMT) is a way to do Machine Translation with a *single end-to-end neural network*
- The neural network architecture is called a sequence-to-sequence model (aka seq2seq) and it involves *two* RNNs

What's SMT

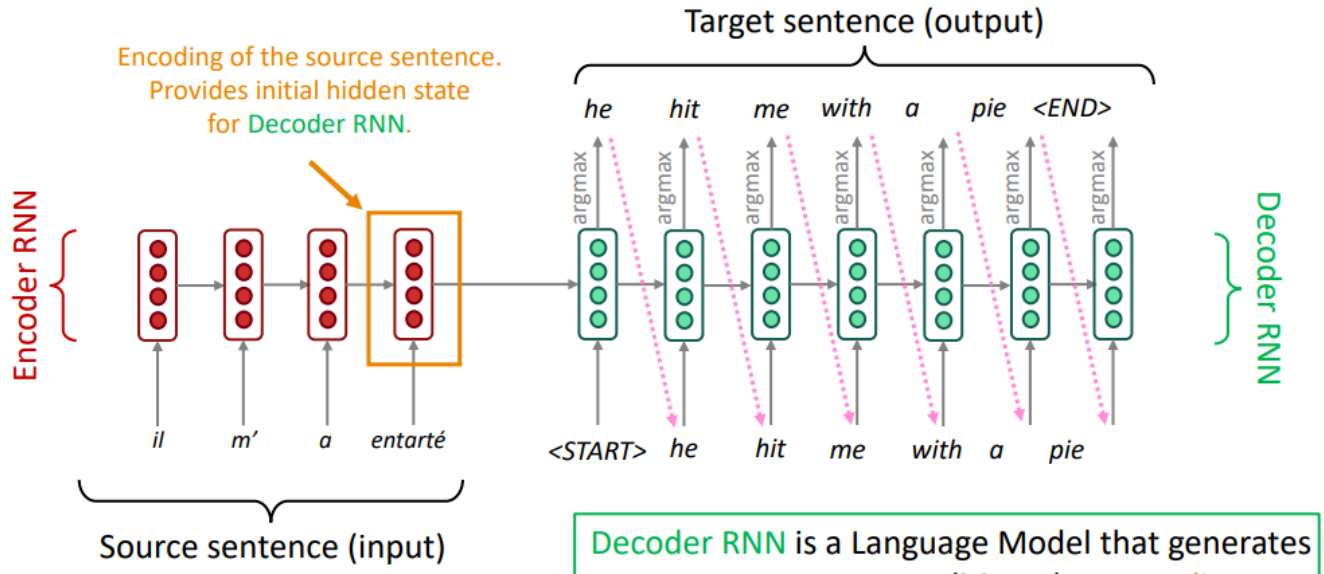
Core idea: learn a probabilistic model from data

Use Bayes Rule to break $\text{argmax}_y P(y|x)$ down into 2 components to be leaned separately

$$= \text{argmax}_y P(x|y) P(y)$$

## Neural Machine Translation (NMT)

### The sequence-to-sequence model




Decoder RNN is a Language Model that generates target sentence, *conditioned on encoding*.

Note: This diagram shows **test time** behavior: decoder output is fed in ·····→ as next step's input

Encoder RNN produces an encoding of the source sentence.

what'more about ↑ ?

encoder: $h_t = RNN(x_t, h_{t-1})$

the last hidden state $h_T$ is considered to contain summary information of the entire input. And this $h_T$ is the Context Vector, recorded as $c$

decoder: what does it generate? a varible length sequence

$$\begin{cases} S_0 = c & S_0 \text{ is the initial hidden state} \\ S_t = RNN(y_{t-1}, S_{t-1}) & y_{t-1} \text{ is what does decoder generate at last timestep} \\ P(y_t | y_{<t}, c) = Softmax(W \cdot S_t + b) \end{cases}$$

In practice, people often use LSTM instead of RNNs

## Sequence-to-sequence is versatile!

- The general notion here is an encoder-decoder model
  - One neural network takes input and produces a neural representation
  - Another network produces output based on that neural representation
  - If the input and output are sequences, we call it a seq2seq model

- Sequence-to-sequence is useful for *more than just MT*
- Many NLP tasks can be phrased as sequence-to-sequence:
  - Summarization (long text → short text)
  - Dialogue (previous utterances → next utterance)
  - Parsing (input text → output parse as sequence)
  - Code generation (natural language → Python code)

## Neural Machine Translation (NMT)

- The sequence-to-sequence model is an example of a **Conditional Language Model**
  - **Language Model** because the decoder is predicting the next word of the target sentence *y*
  - **Conditional** because its predictions are *also* conditioned on the source sentence *x*

- NMT directly calculates $P(y|x)$:

$$P(y|x) = P(y_1|x)\, P(y_2|y_1, x)\, P(y_3|y_1, y_2, x) \ldots \underbrace{P(y_T|y_1, \ldots, y_{T-1}, x)}$$

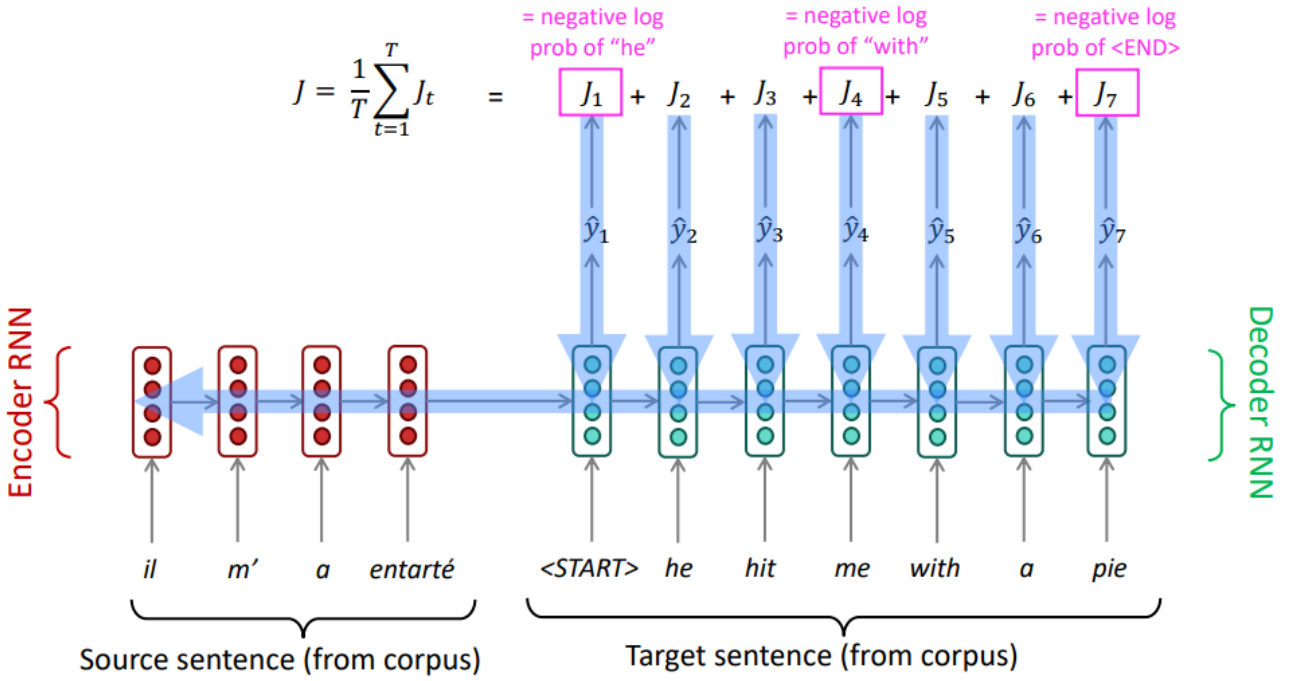
Probability of next target word, given target words so far and source sentence *x*

- **Question:** How to train an NMT system?
- **(Easy) Answer:** Get a big parallel corpus…
  - But there is now exciting work on "unsupervised NMT", data augmentation, etc.

x is the context vector generated by encoder

Be aware! CLM itself is not a specific model architecture but a modeling concept

## Training a Neural Machine Translation system




Seq2seq is optimized as a **single system.** Backpropagation operates "*end-to-end*".

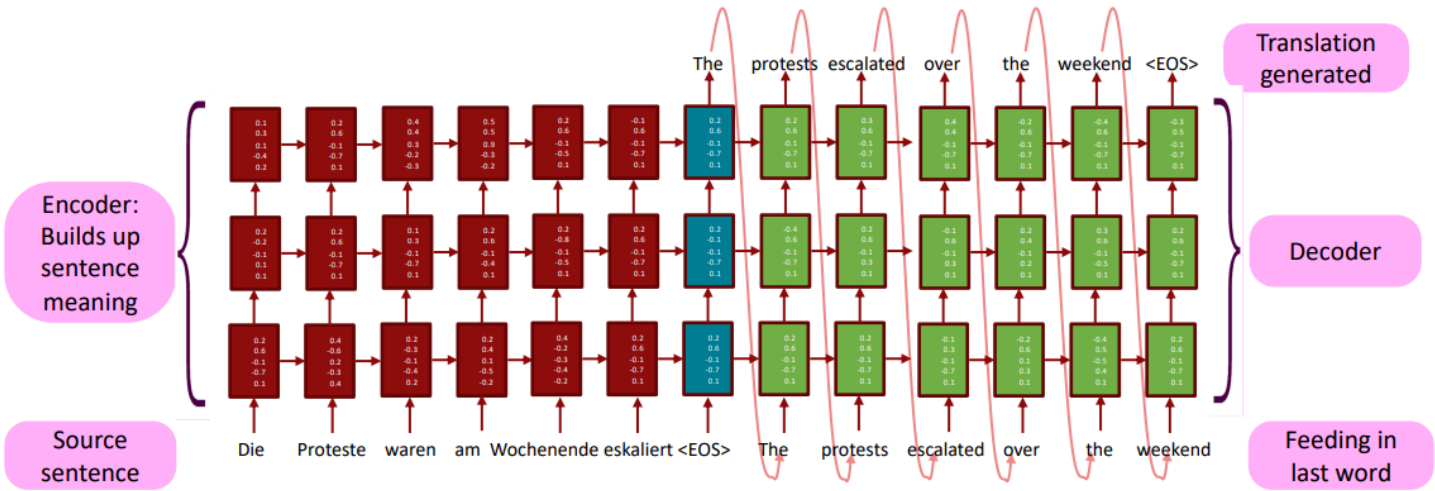
end-to-end

NTM is that it achieves differentiable differentation (可微分的可微分化) of the entire system - all operations from the controller to the memory - through a differentiable attention mechanism

## Multi-layer deep encoder-decoder machine translation net

[Sutskever et al. 2014; Luong et al. 2015]

The hidden states from RNN layer *i* are the inputs to RNN layer *i*+1




MT Evaluation

## How do we evaluate Machine Translation?

Commonest way: **BLEU** (**B**ilingual **E**valuation **U**nderstudy)

You'll see BLEU in detail in Assignment 3!

- BLEU compares the machine-written translation to one or several human-written translation(s), and computes a similarity score based on:
  - Geometric mean of *n*-gram precision (usually for 1, 2, 3 and 4-grams)
  - Plus a penalty for too-short system translations
- BLEU is useful but imperfect
  - There are many valid ways to translate a sentence
  - Therefore, a good translation can get a poor BLEU score because it has low *n*-gram overlap with the human translation ☹

history: until BLEU, the only way is human being to look at these translations. And say how the translation it is. What's more, man human evaluation is still useful because many automatic measure have lot biases and problems.

→ But people still want to have a halfway decent automatic method of translation Because we'd like to iterate quickly on evaluations

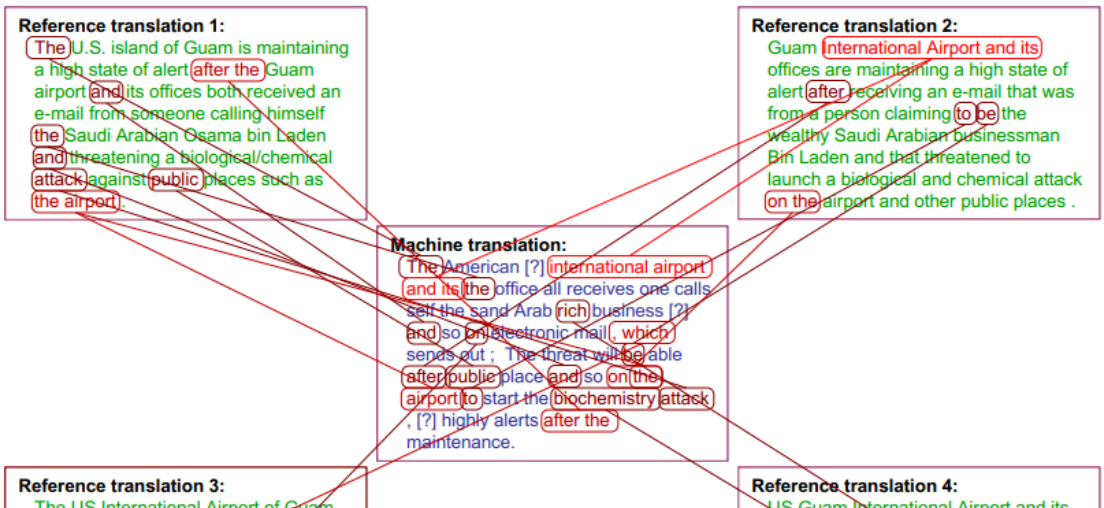
Why imperfect, because instead of image recognization, we have many different ways to translate a sentence.

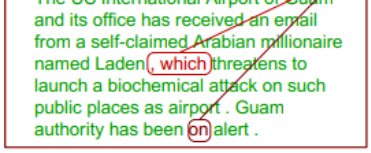
Intro: the more overlap, the score higher

→ otherwise it could leave out everything difficult and translate the easy part

# BLEU score against 4 reference translations

But commonly now there is only one references and so the results are more "in expectation"






[Papineni et al. 2002]