CS/DS 541: Class 20

Jacob Whitehill

seq2seq models

seq2seq models

- Seminal paper:
  - Sutskever et al. 2014

Sequence-to-sequence translation model (seq2seq)

- Suppose we want to translate from one language to another.

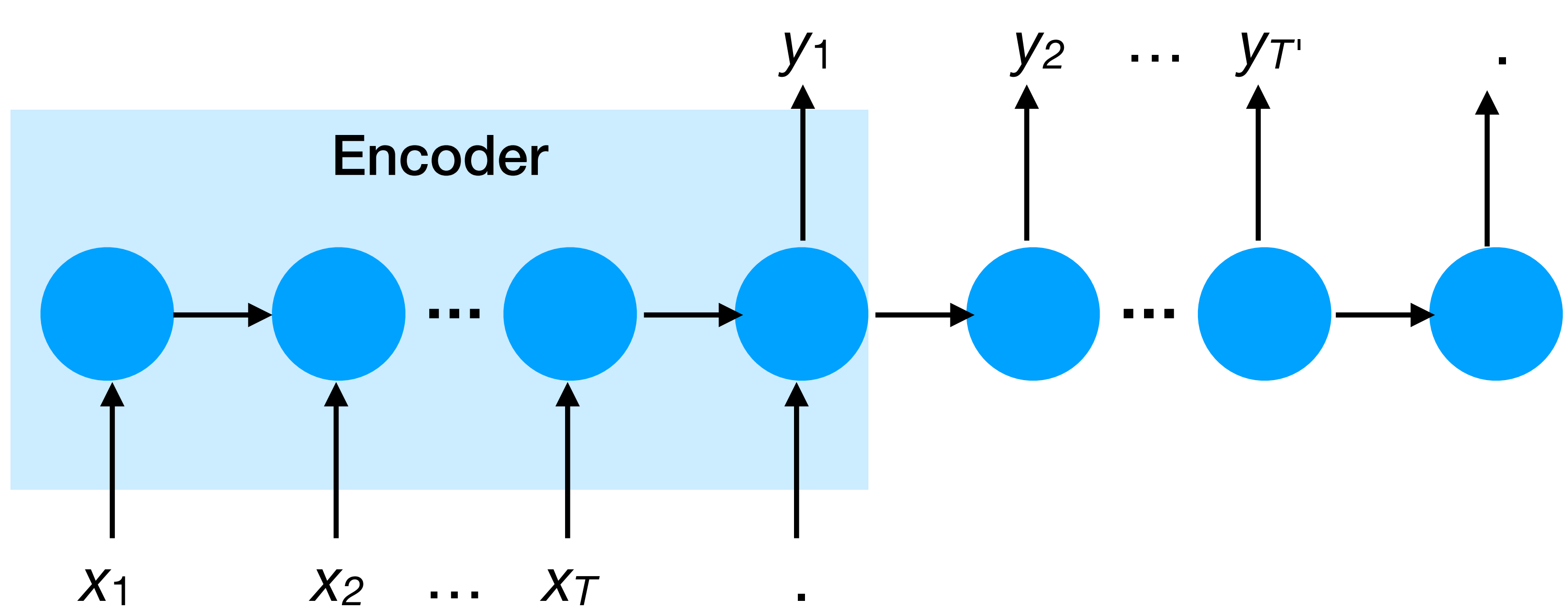
- Language 1 vocabulary: { a, b, . }.

- Language 2 vocabulary: { u, v, w, . }.

- We add to both vocabularies a “.” symbol that means **end-of-sentence** (EOS).

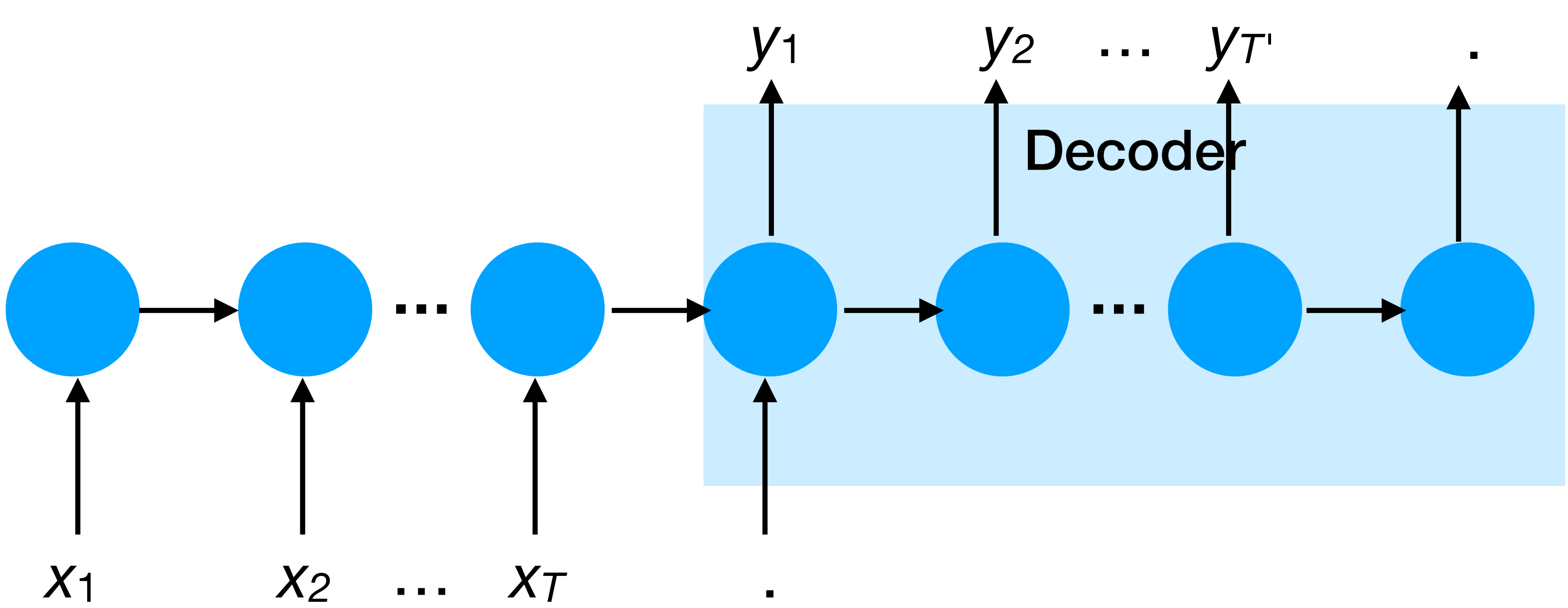
Sequence-to-sequence translation model (seq2seq)

- We construct a sequence-to-sequence model consisting of an **encoder** RNN and a decoder RNN:



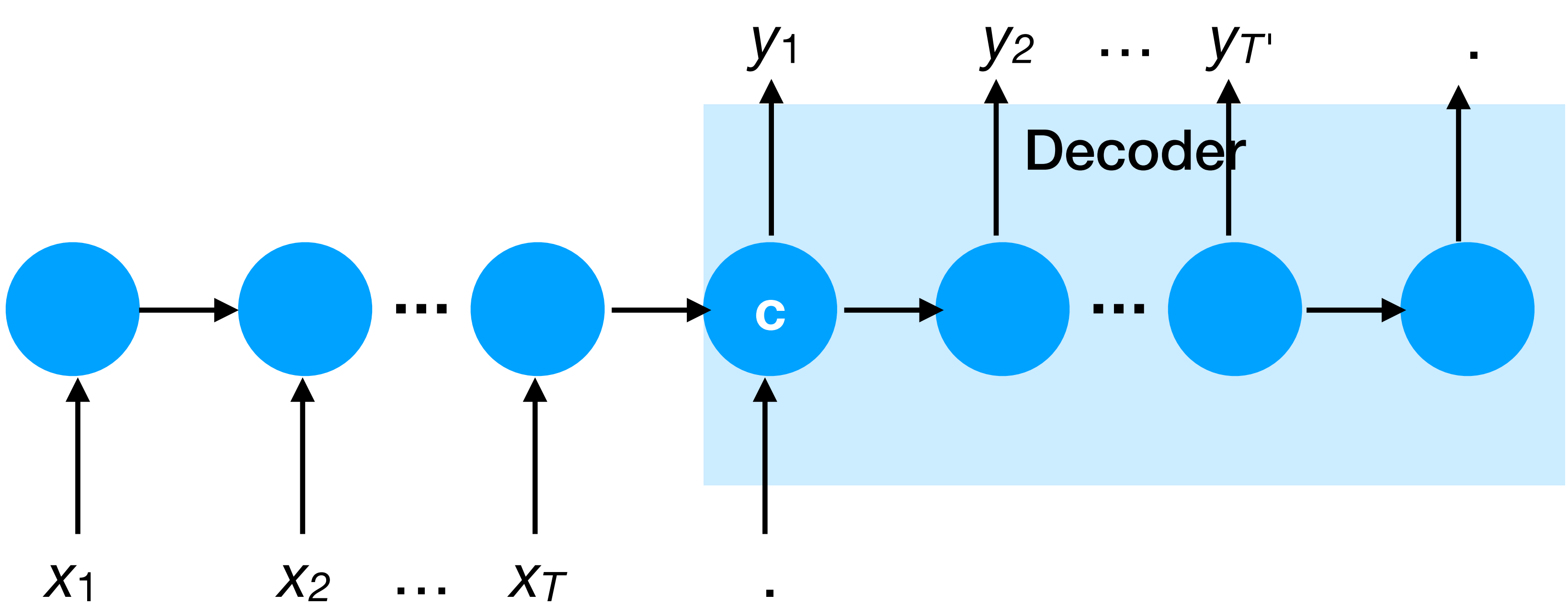
Sequence-to-sequence translation model (seq2seq)

- We construct a sequence-to-sequence model consisting of an encoder RNN and a **decoder** RNN:



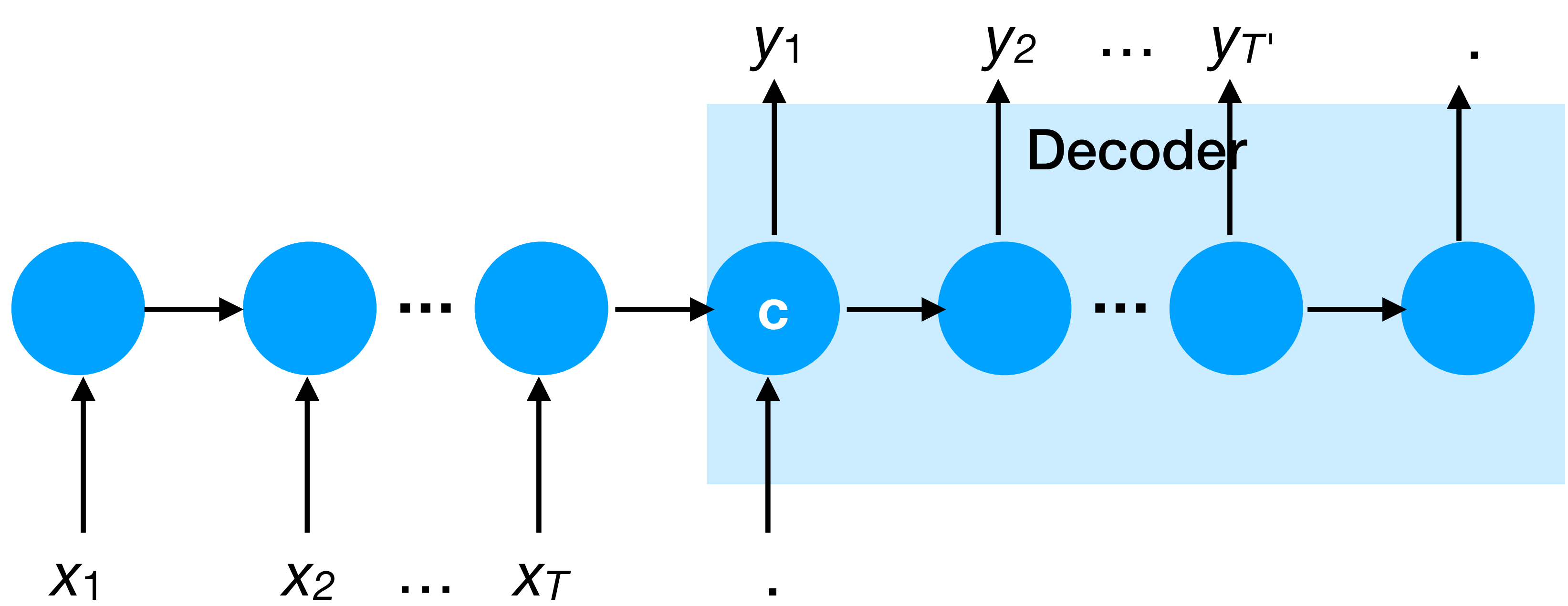
Sequence-to-sequence translation model (seq2seq)

- The encoder digests the input sequence $x_1, \ldots, X_T$ and produces a context vector **c**.
- The decoder uses the context vector to produce the translation sequence $y_1, \ldots, y_{T'}$.



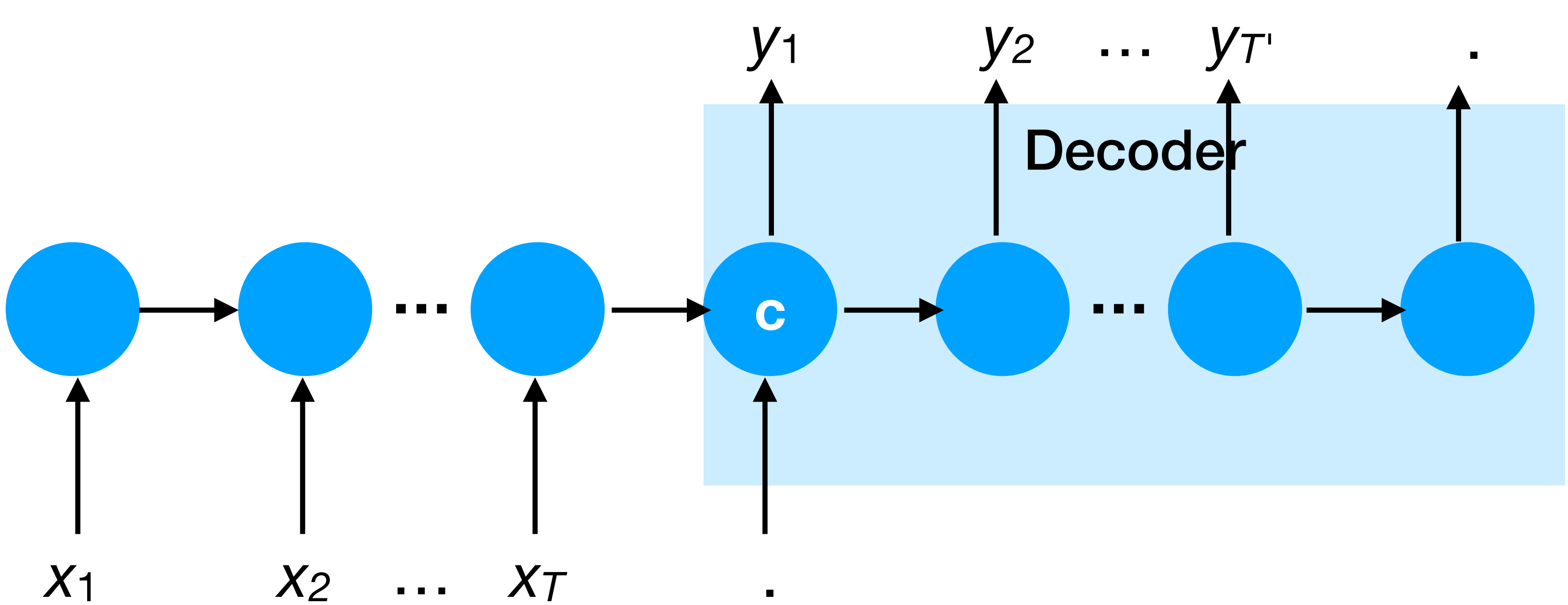
Sequence-to-sequence translation model (seq2seq)

- At each timestep t=1, …, T', the decoder tries to estimate the probability distribution $P(y_t \mid y_1, \ldots, y_{t-1}, x_1, \ldots, x_T)$.



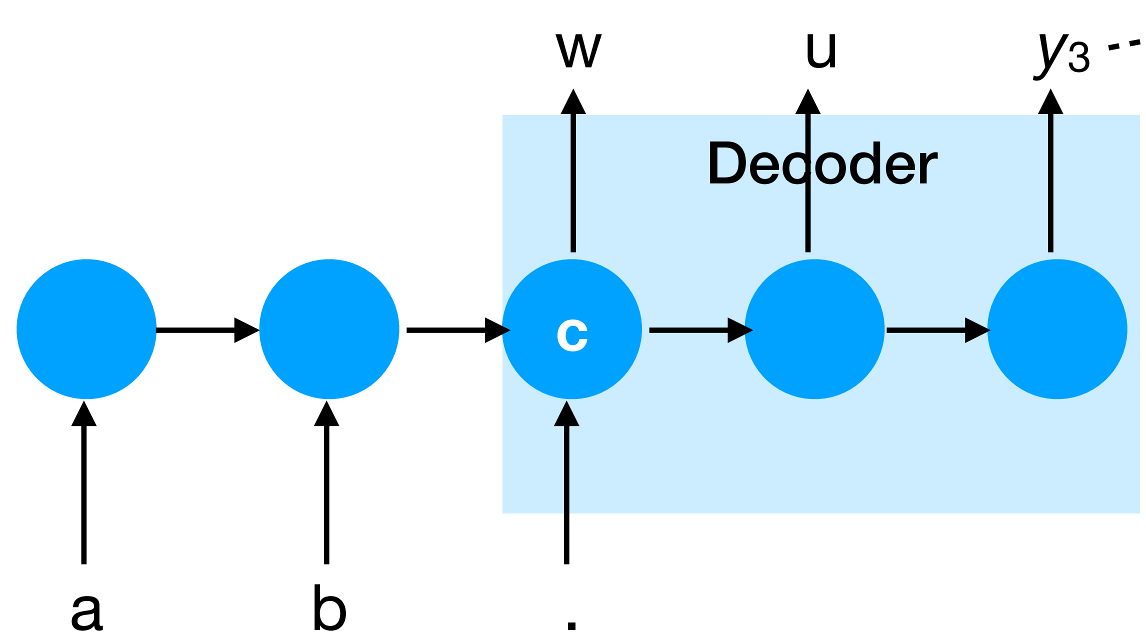
Sequence-to-sequence translation model (seq2seq)

- More precisely, the RNN produces a probability distribution conditioned on the context: $P(y_t \mid y_1, \ldots, y_{t\text{-}1}, \mathbf{c})$.



Sequence-to-sequence translation model (seq2seq)

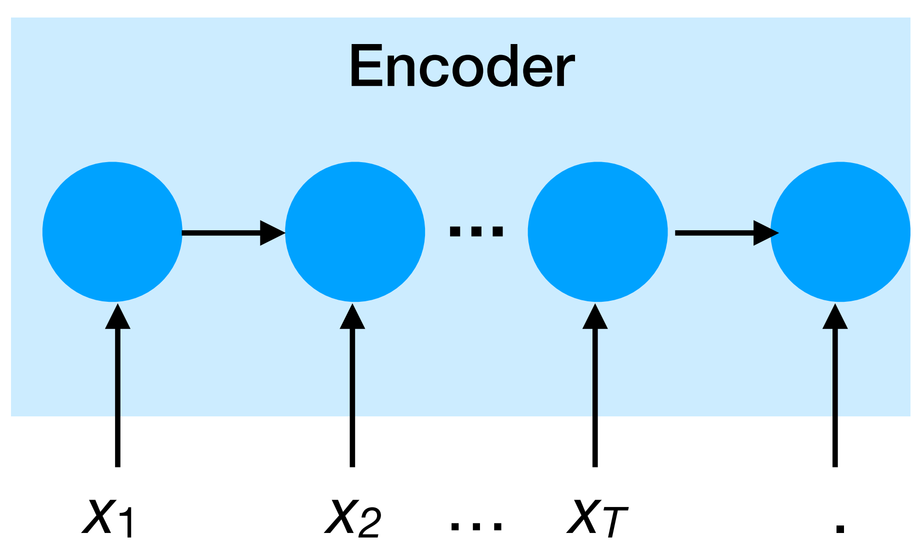
- For example, if the input is “ab” and the first two output symbols were “wu”, then the decoder estimates
  $P(y_3 \mid Y_1{=}\text{w}, Y_2{=}\text{u}, X_1{=}\text{a}, X_2{=}\text{b}, X_3{=}.) \cong$
  $P(y_3 \mid Y_1{=}\text{w}, Y_2{=}\text{u}, \mathbf{c})$



u	0.6
v	0.2
w	0.05
.	0.15

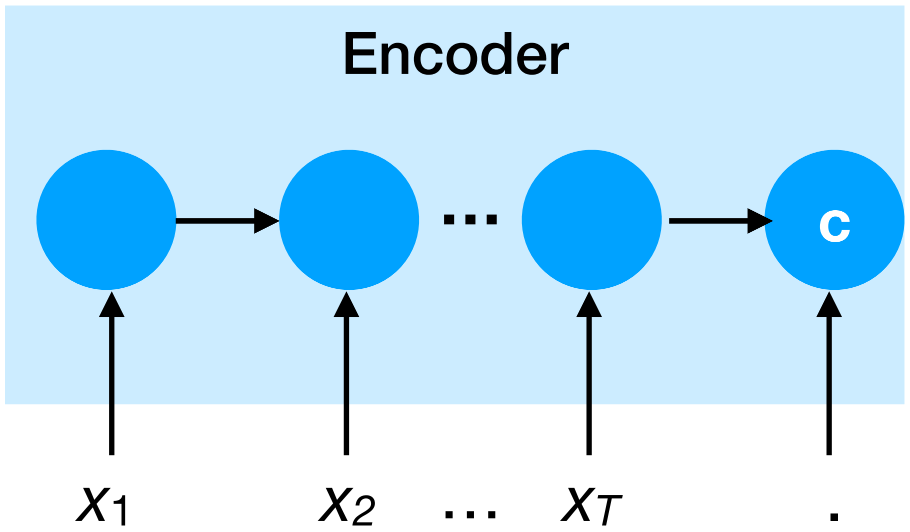
Generating a translation

- Given an input sentence, we can generate an output sentence as follows:



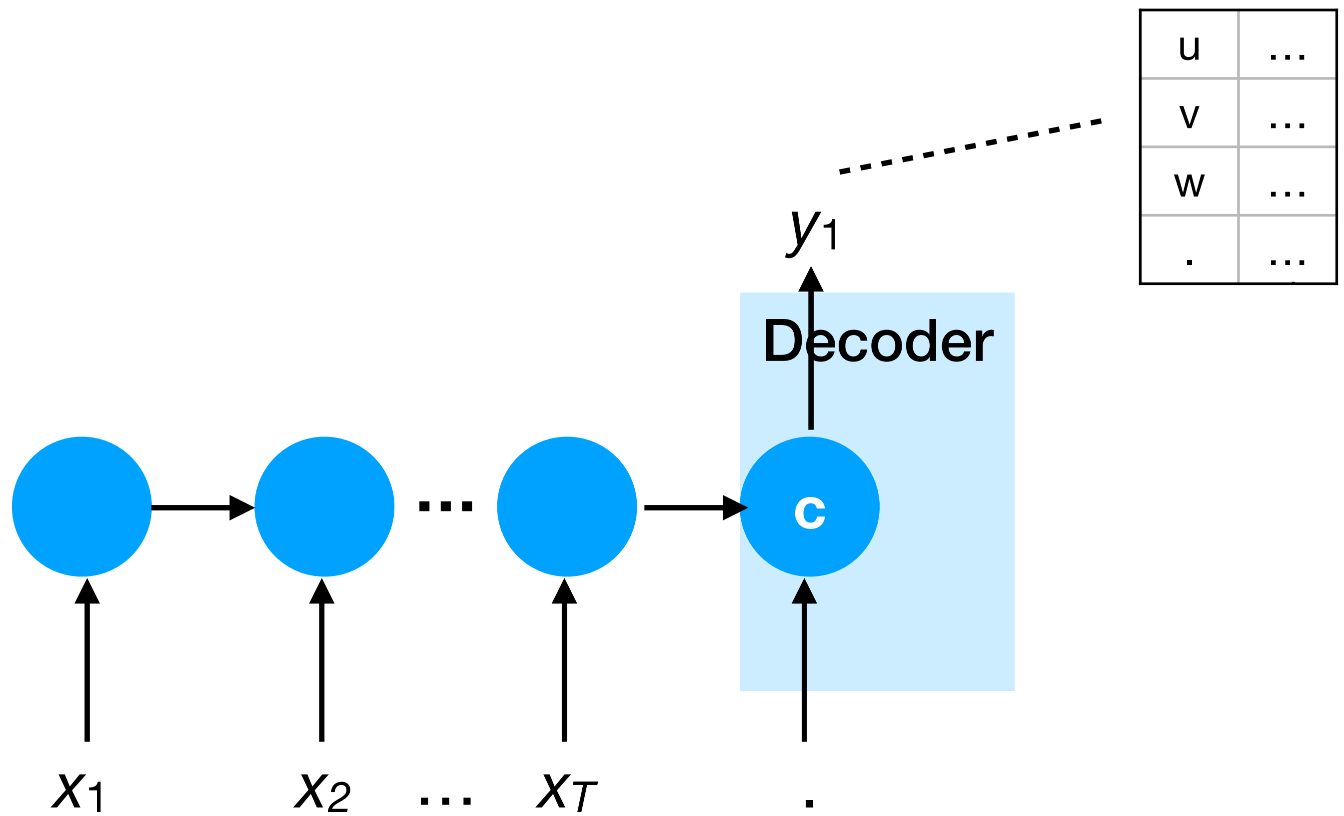
Generating a translation

1. Encode $x_1, \ldots, x_T$ into $\mathbf{c}$:



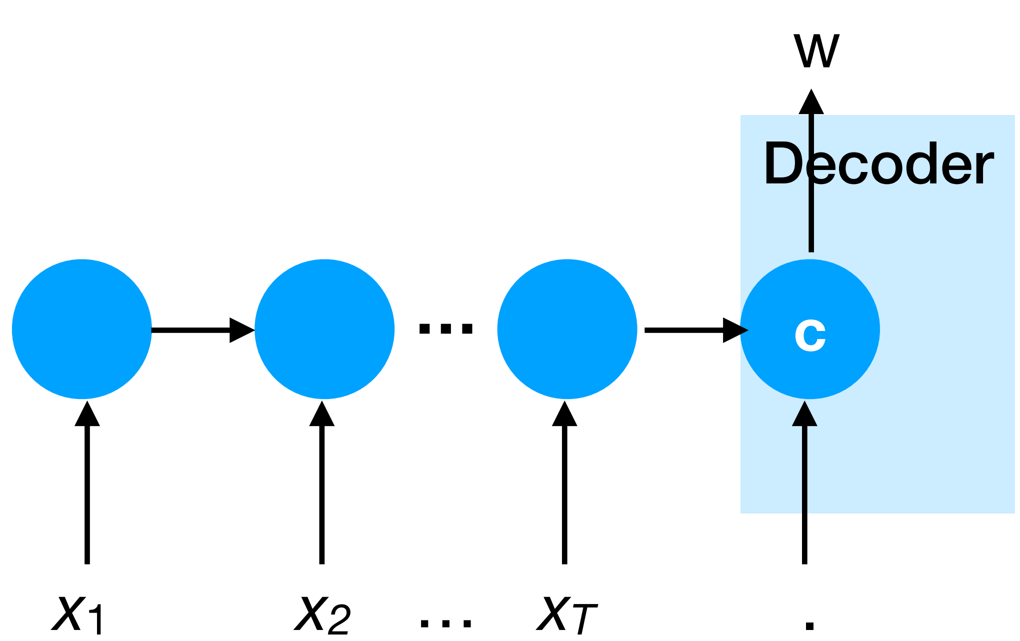
Generating a translation

2. Decode **c** for one time-step to compute $P(y_1 \mid \mathbf{c})$.



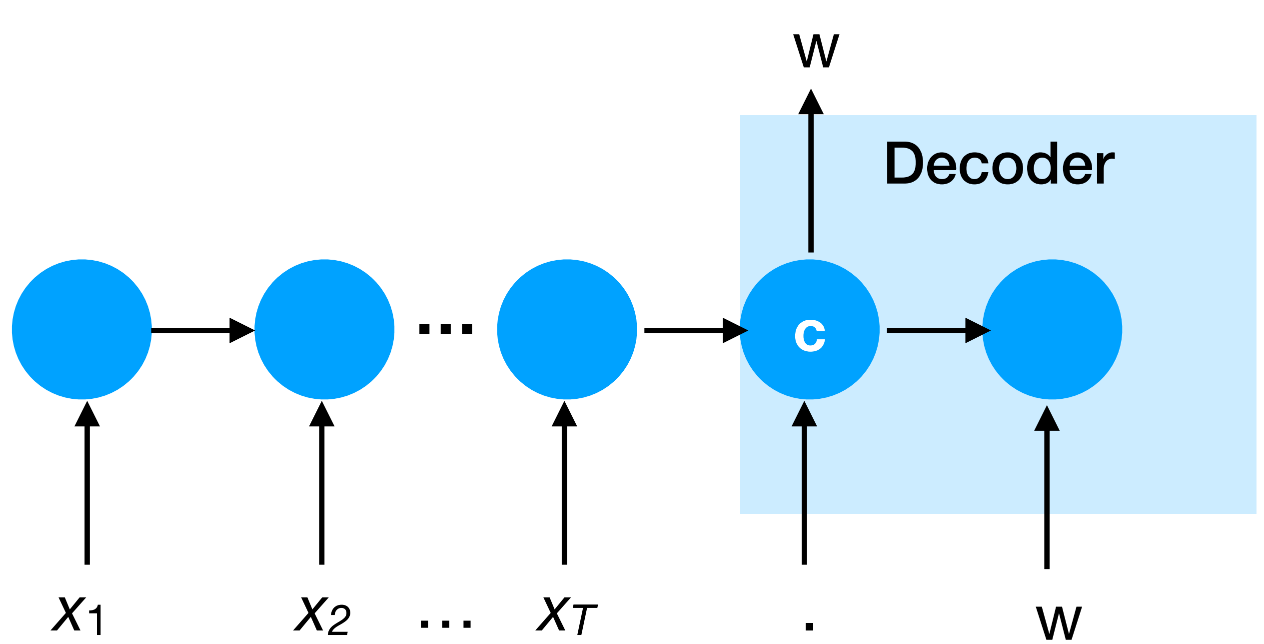
Generating a translation

3. Sample from $P(y_1 \mid \mathbf{c})$ to obtain a symbol $y_1 \in \{ u, v, w, . \}$, e.g., w.



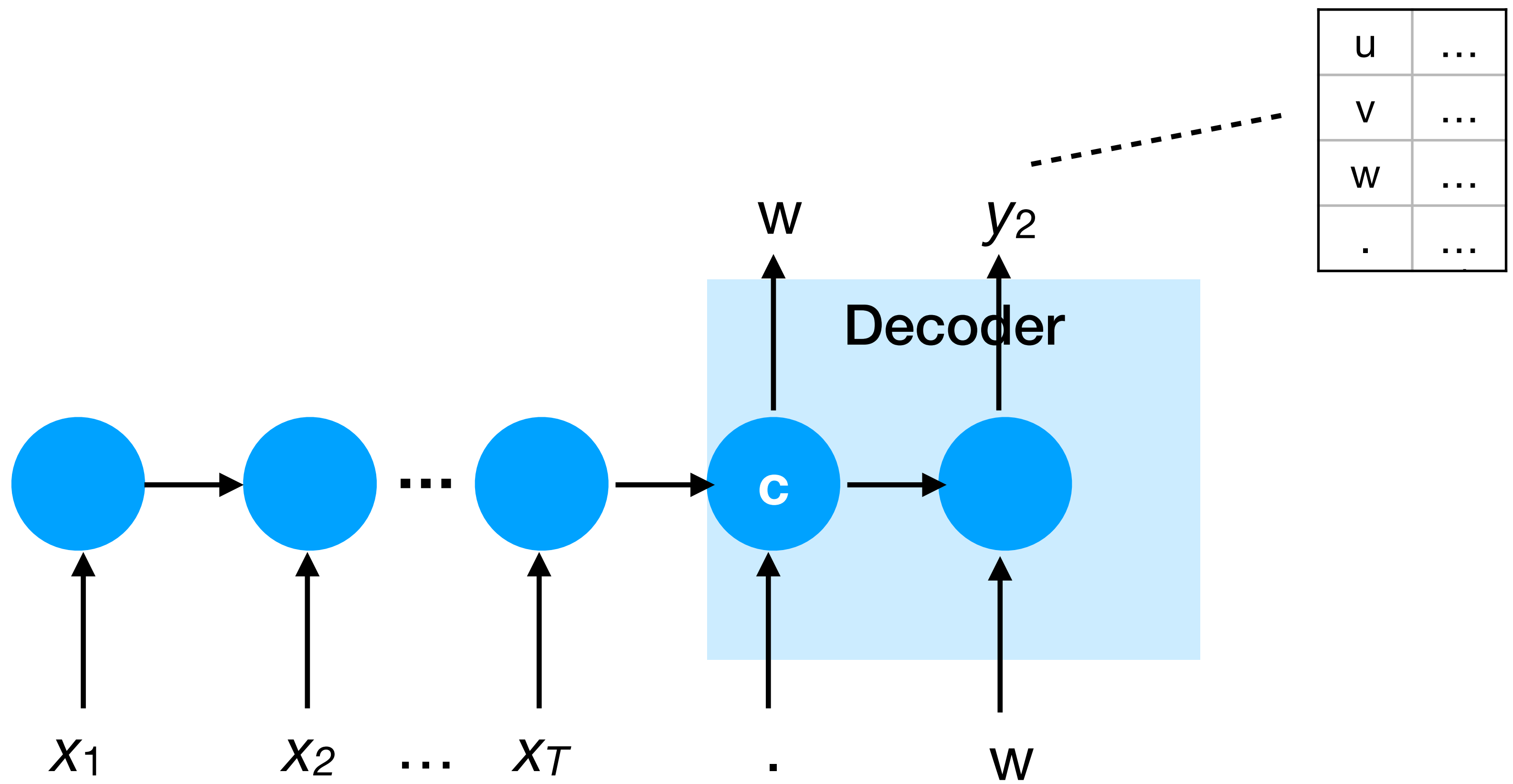
Generating a translation

4. If y_1=. then stop; else feed y_1 as input to the decoder during the next time-step.



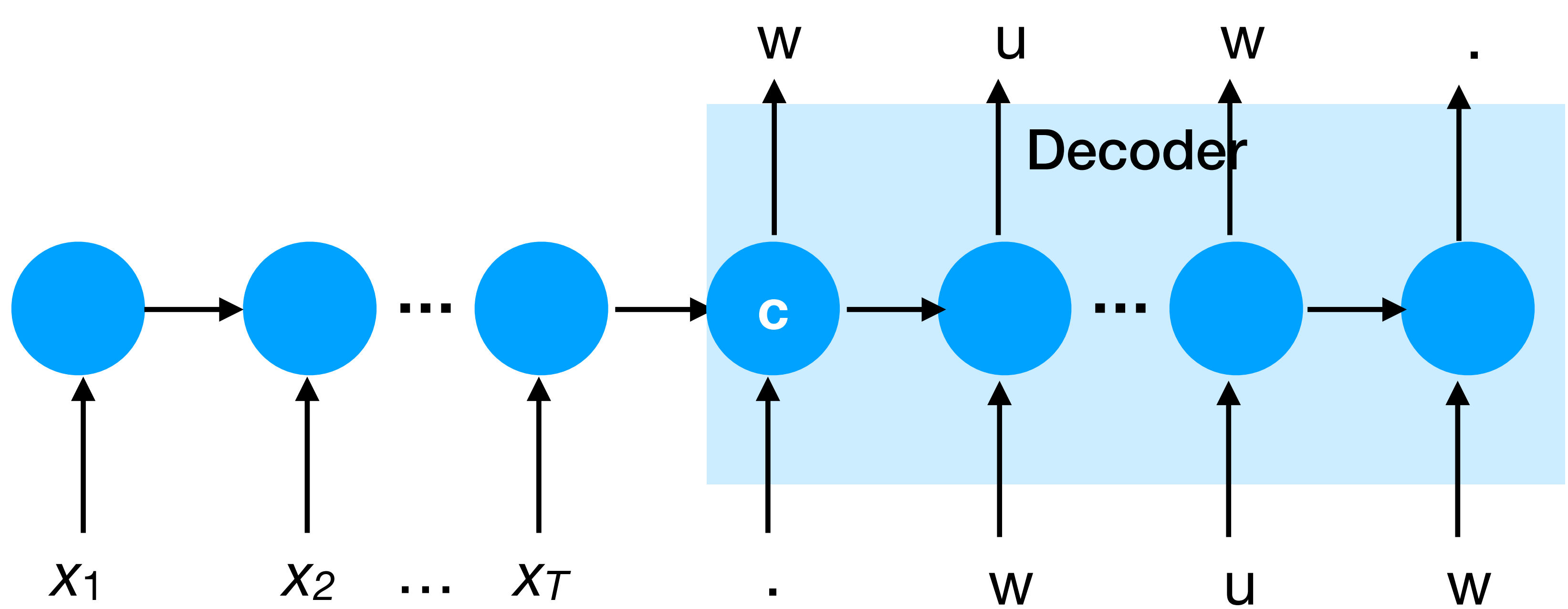
Generating a translation

5. Compute $P(y_2 \mid y_1, \mathbf{c})$.



Generating a translation

6. Repeat the procedure until we produce a . symbol.

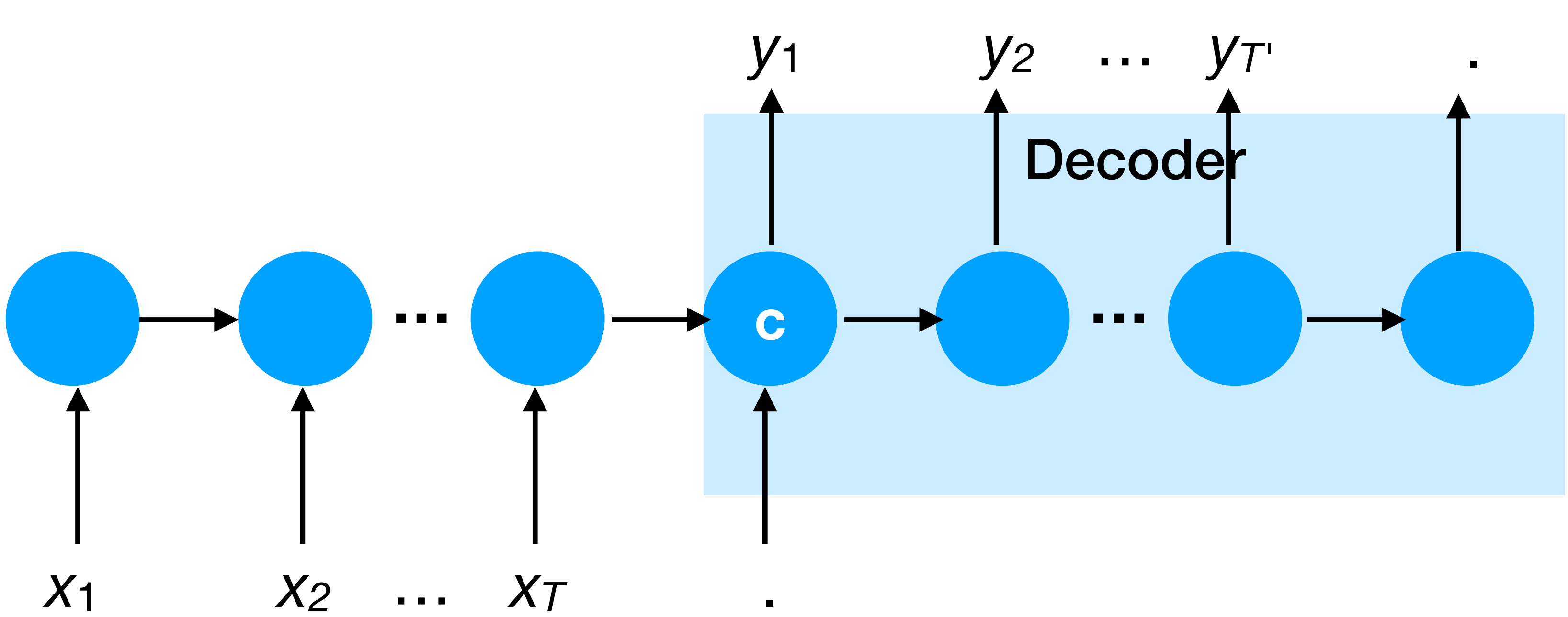


Training a seq2seq model

- We train a seq2seq model using maximum likelihood estimation (MLE) on a set of input-output pairs.

- Given any input-output pair, we want the RNN's weights θ to maximize the likelihood:

$$P(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T, \theta)$$



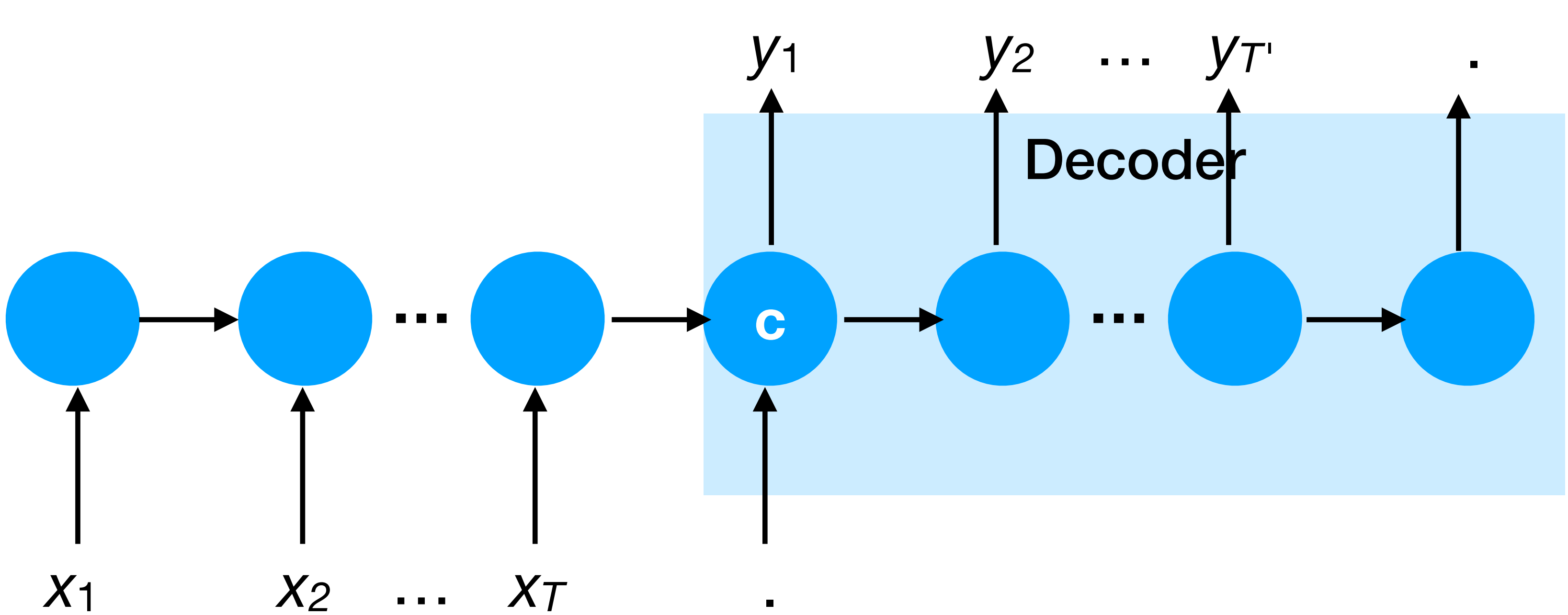
Training a seq2seq model

- We train a seq2seq model using maximum likelihood estimation (MLE) on a set of input-output pairs.

- Given any input-output pair, we want the RNN's weights θ to maximize the likelihood:

$$
\begin{aligned}
P(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T, \theta) = & P(y_1 \mid x_1, \ldots, x_T, \theta) \\
& P(y_2 \mid y_1, x_1, \ldots, x_T, \theta) \\
& \ldots \\
& P(y_{T'} \mid y_1, \ldots, y_{T'-1}, x_1, \ldots, x_T, \theta)
\end{aligned}
$$

The probability distribution factorizes over *t*.



Finding the most likely translations

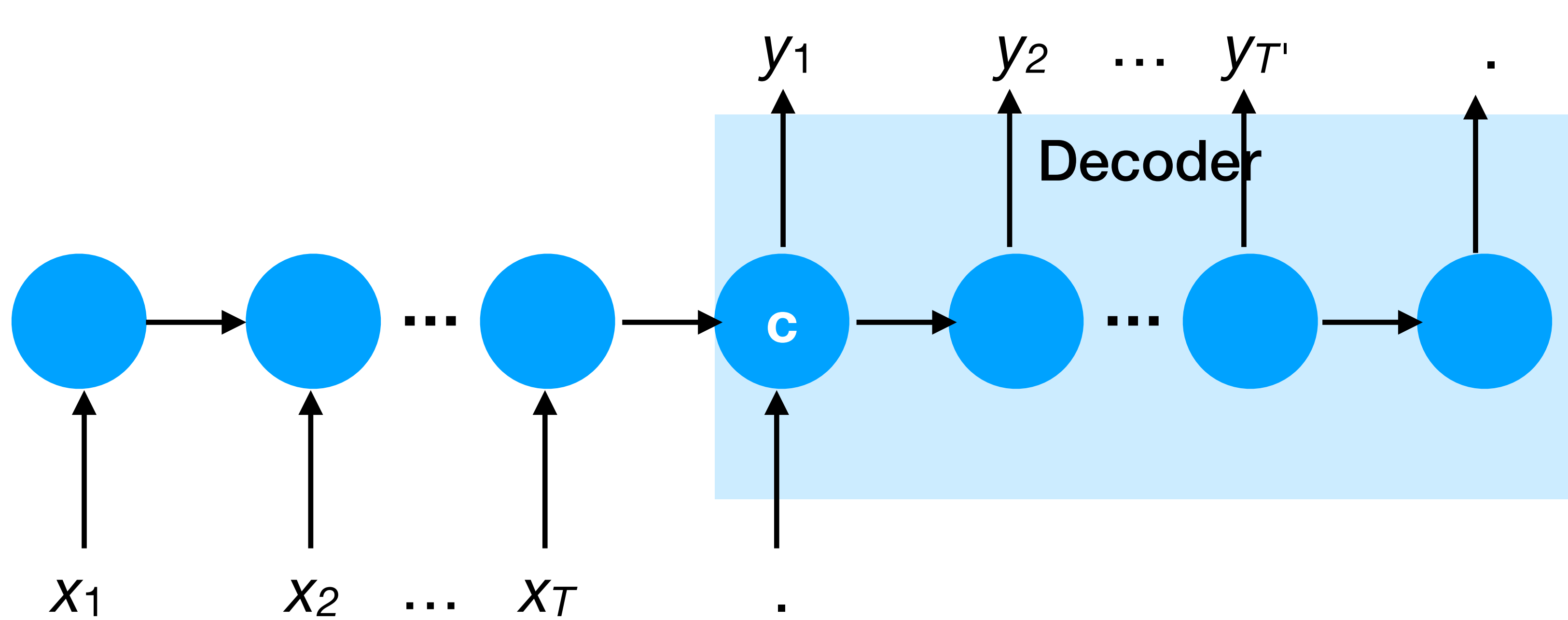
- At test time, we want to find the *most probable* output sentence given any input sentence, i.e.:

$$\mathrm{argmax}_{(y_1,\ldots,y_{T'})} P(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T, \theta)$$

- How can we search over all possible (y_1, …, $y_{T'}$) (for fixed T')?

Finding the most likely translations

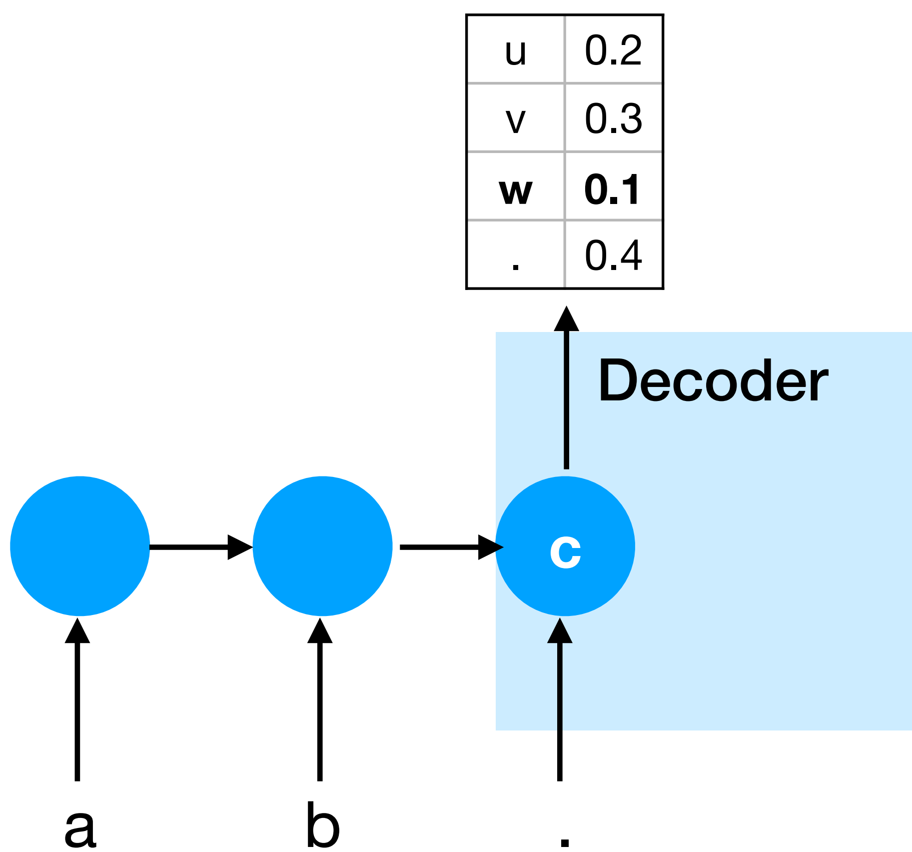
- Since $P(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T)$ factorizes over t, we can pass each candidate sequence into our RNN, obtain the probability of each symbol y_t, and then multiply the probabilities together.



Finding the most likely translations

- Suppose x_1=a and x_2=b. Then for y_1=w and y_2=v, we have:

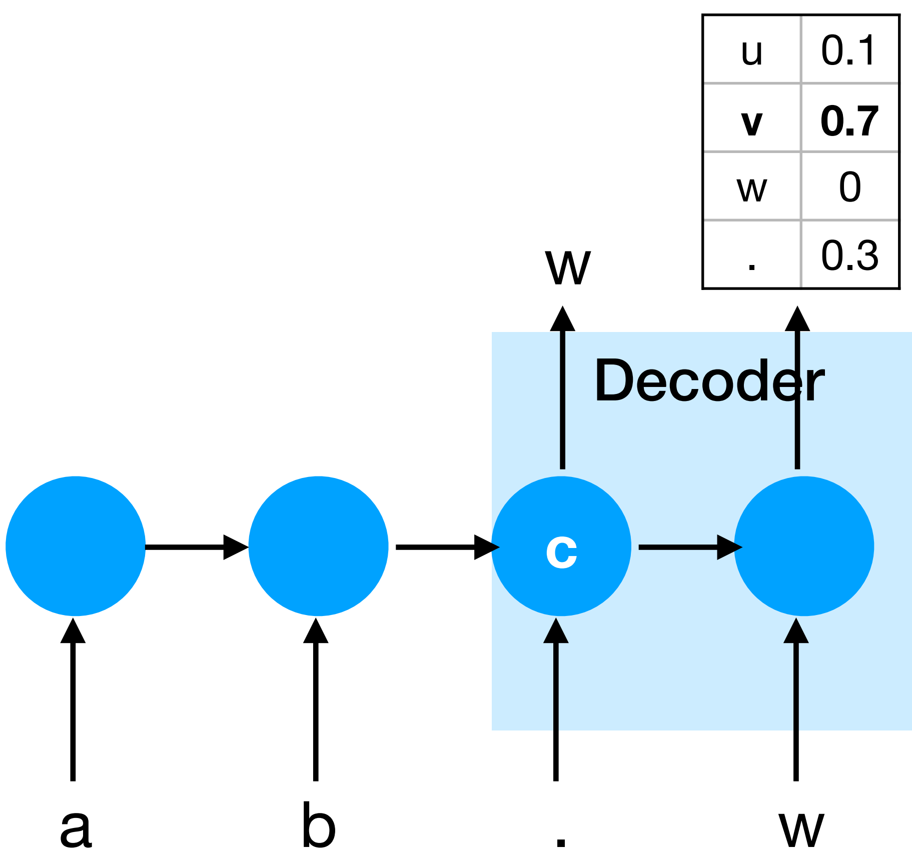
***P*(w, v | a, b) = 0.1**



Finding the most likely translations

- Suppose x_1=a and x_2=b. Then for y_1=w and y_2=v, we have:

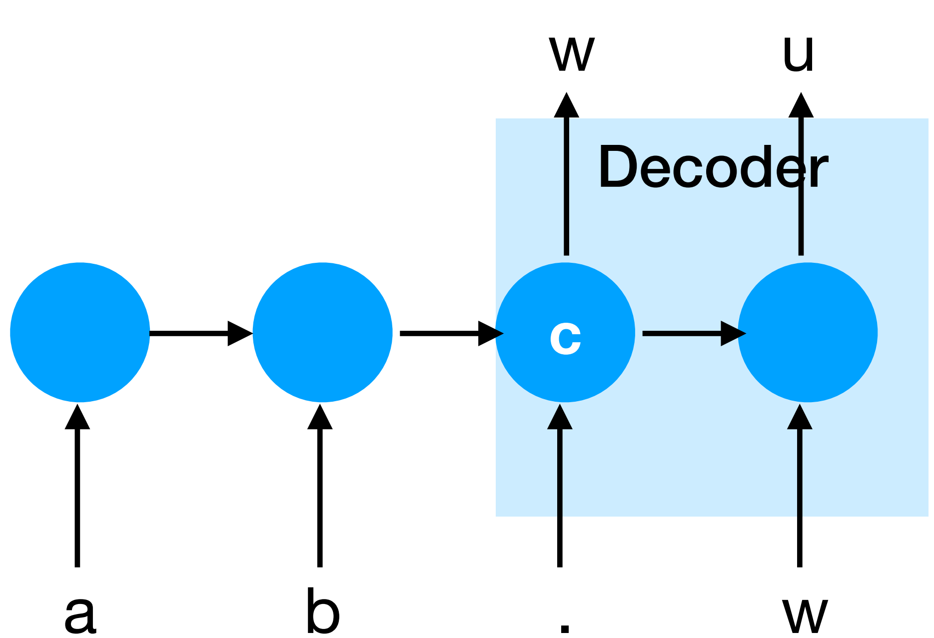
***P*(w, v | a, b) = 0.1 * 0.7**



Finding the most likely translations

- Suppose x_1=a and x_2=b. Then for y_1=w and y_2=v, we have:

$$P(\text{w, v} \mid \text{a, b}) = 0.1 \quad * \quad 0.7 \quad = \quad 0.07$$



Finding the most likely translations

- Unfortunately, for large T', there are exponentially many different probabilities $P(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T)$ we would need to compute.

- Heuristic: perform a greedy **beam search** to keep track of the top-K most likely translations $y_1, \ldots, y_{T'}$.

Beam search

Beam search

1. At each output timestep *t*, keep track of top-*K* most likely translations, where *K* is the **beam width**:

$$\{(y_1, \ldots, y_t)^{(1)},\ \ldots,\ (y_1, \ldots, y_t)^{(K)}\}$$

2. For each of our *K* candidates, we can compute:

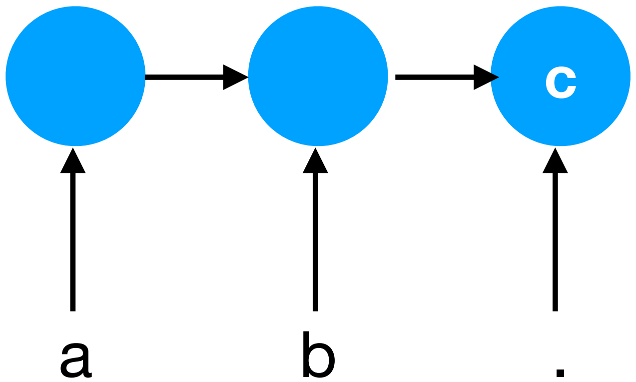
$$P(y_{t+1} \mid y_1, \ldots, y_t, x_1, \ldots, x_T)$$

3. If the output vocabulary has *N* words, then this results in *N***K* possible sequences of length *t*+1.

4. From these *N***K* choices, we select the top-*K* most likely translations of length *t*+1.

Example

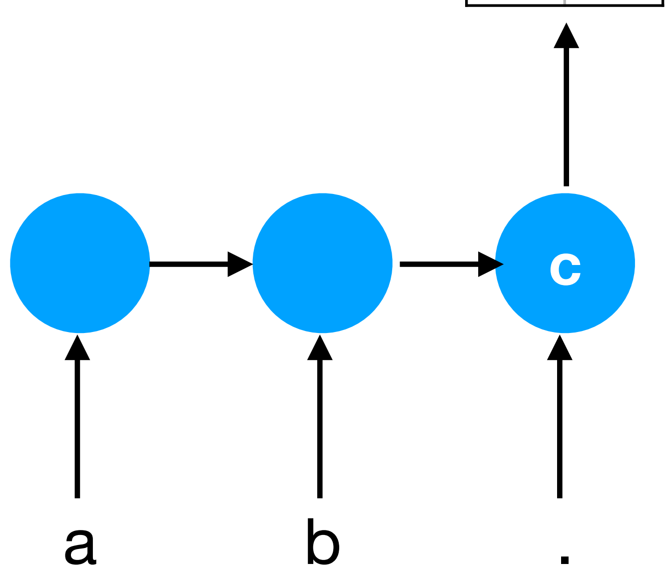
- Let input vocabulary={a, b, .} and output vocabulary={u, v, w, .}.
- Let beam width *K*=2.



Example

- Beam at t=0: {}
- At t=1, pick top-K most likely possible symbols:

u	0.3
v	0.5
w	0.1
.	0.1



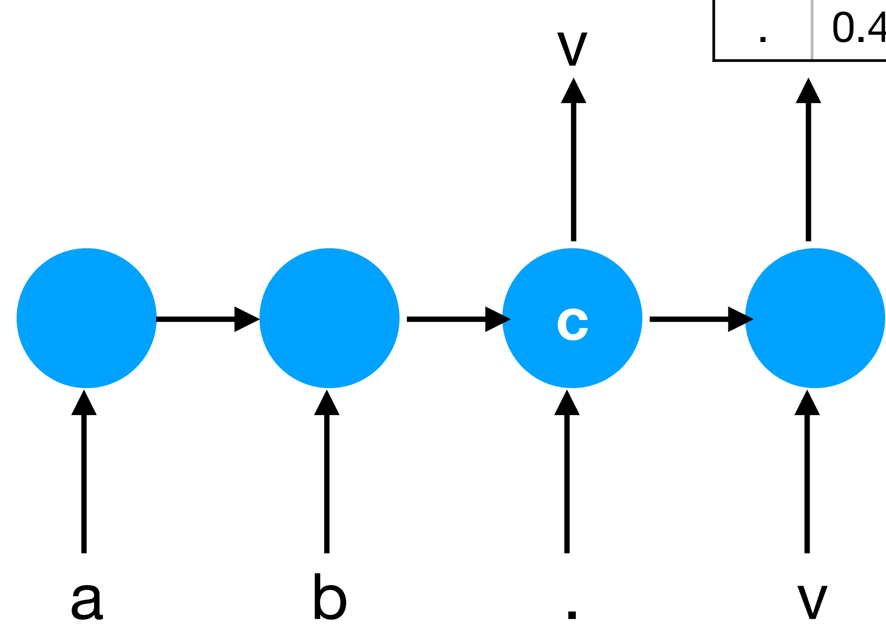
Example

- Beam at t=1: { $\overset{0.5}{(y_1=\text{v})}$, $\overset{0.3}{(y_1=\text{u})}$ }

- At t=2, compute $P(y_2 \mid y_1, x_1=\text{a}, x_2=\text{b})$ for each (y_1) in the beam:

$P(y_2 \mid y_1=\text{v}, x_1=\text{a}, x_2=\text{b})$

u	0.2
v	0.3
w	0.1
.	0.4



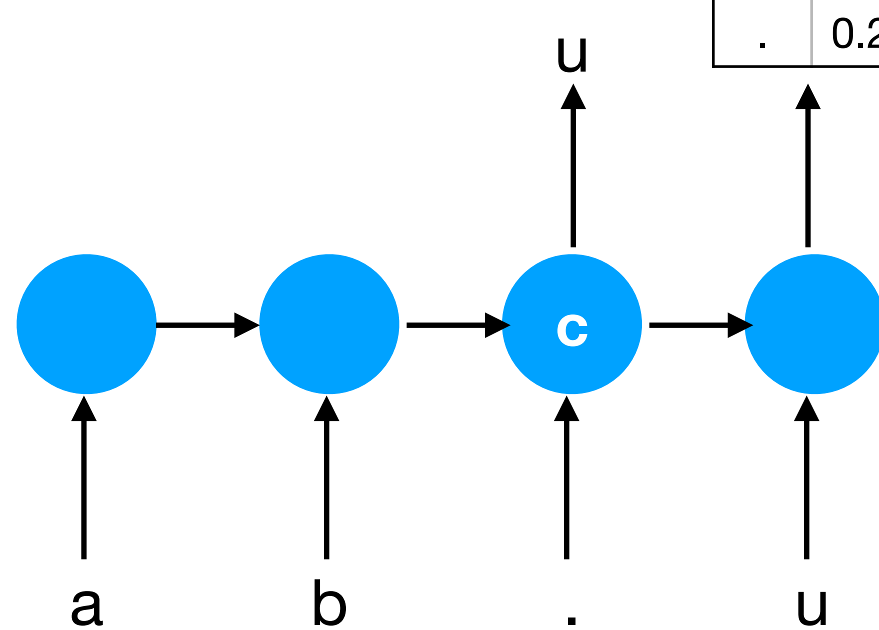
Example

- Beam at t=1: { $\overset{0.5}{(y_1=\text{v})}$, $\overset{0.3}{(y_1=\text{u})}$ }

- At t=2, compute $P(y_2 \mid y_1, x_1=\text{a}, x_2=\text{b})$ for each (y_1) in the beam:

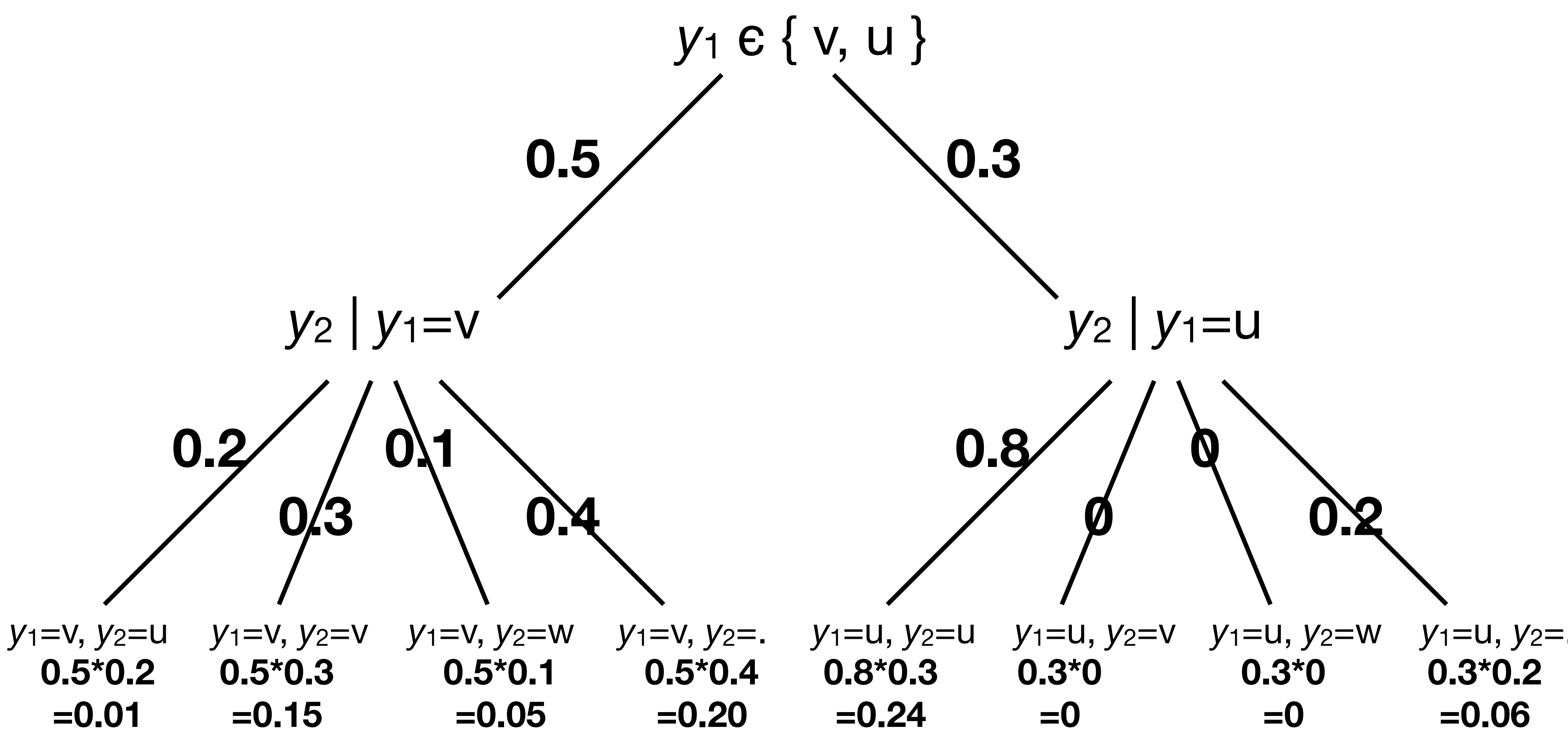
$P(y_2 \mid y_1=\text{u}, x_1=\text{a}, x_2=\text{b})$

u	0.8
v	0
w	0
.	0.2



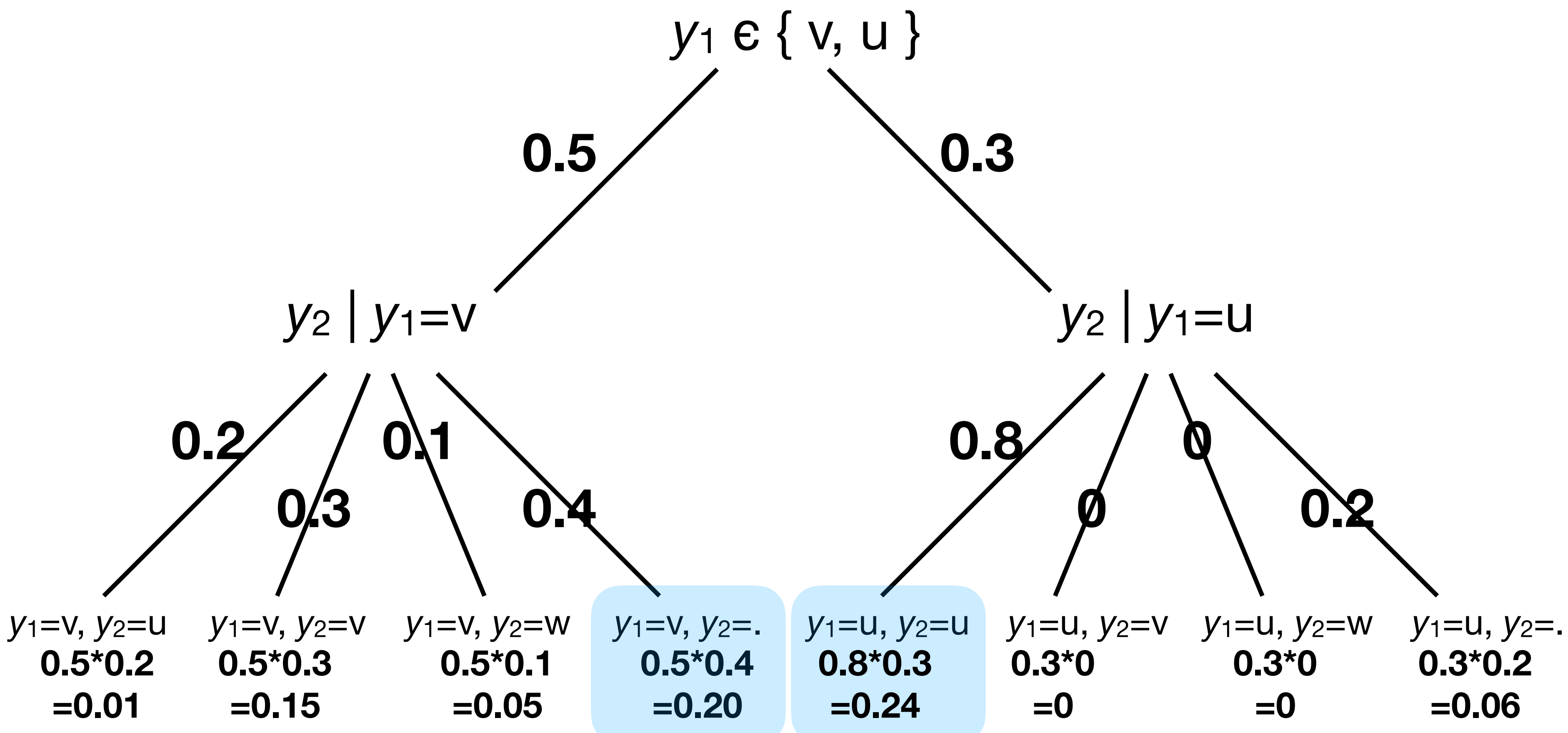
Example

- This results in a total of N*K=4*2=8 possible sequences of length 2:



Example

- We pick the top-K most likely sequences as our next beam.
- Beam at t=2: { (y_1=v, y_2=.), (y_1=u, y_2=u) }



Word embeddings

Paper discussion:

“Distributed Representations of Words and Phrases and their Compositionality” (Mikolov et al. 2013)

Presented by Jacob Whitehill

Background

- For a variety of natural language processing (NLP) tasks, it is important to represent each word in a vocabulary.
- Tasks:
  - Understanding the sentiment of natural text
  - Translating from one language to another
  - Question & answering tasks

Background

- One of the simplest and most common ways to represent each word in a vocabulary *V* is to assign each word a number, e.g.:
  - A = 1
  - a = 2
  - aa = 3
  - aal = 4
  - aalii = 5
  - aam = 6
  - Aani = 7
  - aardvark = 8
  - aardwolf = 9
  - …

Background

- One of the simplest and most common ways to represent each word in a vocabulary *V* is to assign each word a number, e.g.:
  - A = 1
  - a = 2
  - aa = 3
  - aal = 4
  - aalii = 5
  - aam = 6
  - Aani = 7
  - aardvark = 8
  - aardwolf = 9
  - …
- We can then construct a one-hot vector of length |*V*|:
  - A = [1, 0, 0, …, 0]
  - a = [0, 1, 0, …, 0]
  - aa = [0, 0, 1, 0, … 0]
  - …

Background

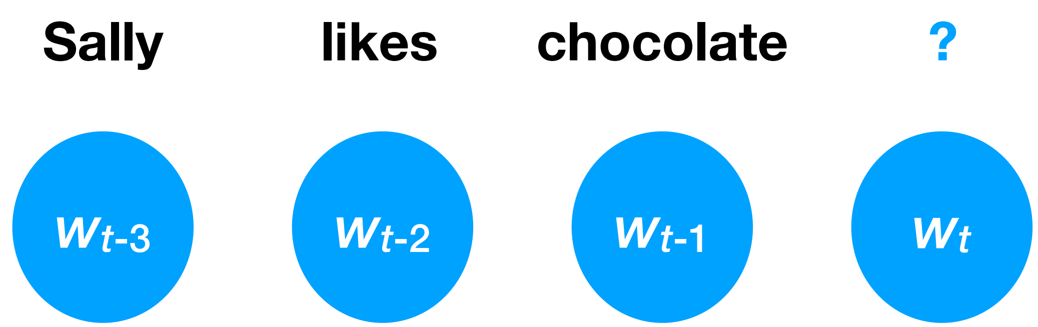
- However, this one-hot representation does not allow any generalization across words.
- For application domains with relatively small amounts of training data (e.g., speech processing in a specialized context), this is wasteful and may decrease the accuracy of downstream systems.
- Is there a way to use DL to learn an **efficient** ($\ll |V|$) and **expressive** vector for each word?

Key contributions

- In the word2vec paper by Mikolov et al. 2013, the authors explored how to train a NN to map each word in a large vocabulary *V linearly* to a continuous (real-valued) vector.
- They explored two training strategies:
  - Hierarchical softmax
  - Negative sampling
- They found that their approach yielded excellent performance on word analogy tasks.
- The learned embedding model also exhibited useful compositionality.

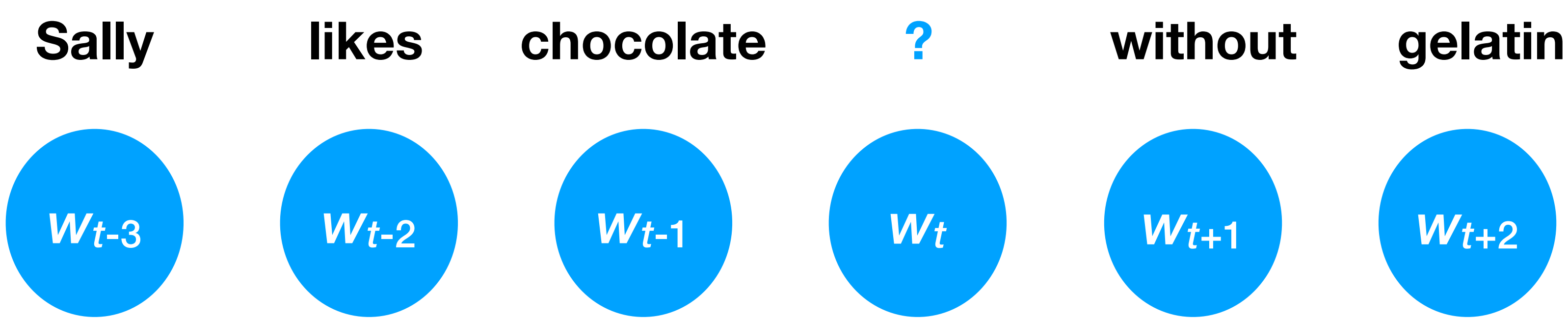
Skip-gram model

- In traditional NLP models, we often predict the t^{th} of a sentence using the previous n words (t-1, t-2, …, t-N), e.g.:



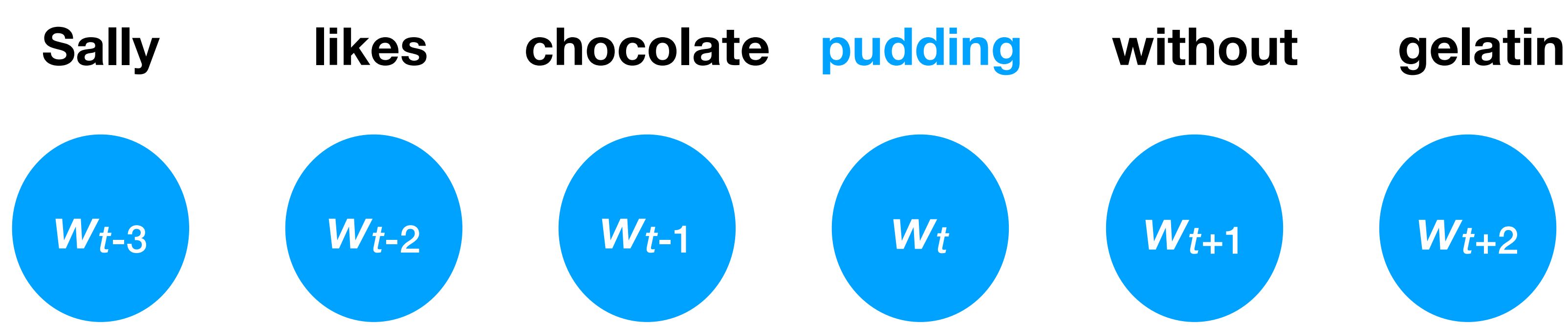
Skip-gram model

- However, recent work has found that skip-gram models often work better because they include both forward and backward context, e.g.:



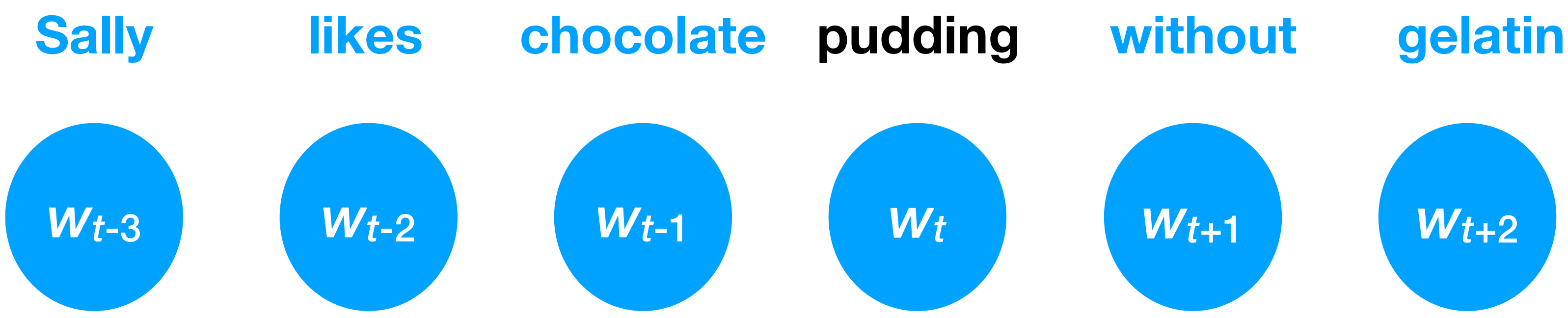
Skip-gram model

- However, recent work has found that skip-gram models often work better because they include both forward and backward context, e.g.:



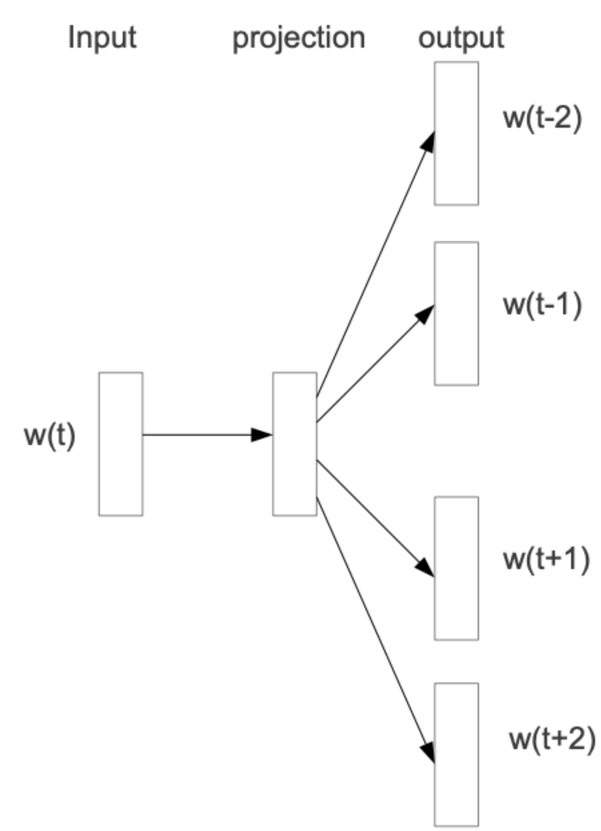
Skip-gram model

- One variant of skip-gram models is to predict the surrounding context from the word itself (conceptually: the previous example but backwards).



Skip-gram model

- To do so, they project each word *w* into an embedding space.



Word embedding model

- Their goal, given a large dataset of sentences over vocabulary *V*, is to maximize the probability of the contexts (length *c*) given the "center" words w_t:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t)$$

NN architecture

- The embedding of each word is actually just *linear*:
  - Multiply a one-hot vector (for each input word) by a (|*V*| x *K*)-dimensional matrix.
  - Due to sparsity, this corresponds to just a vector look-up.

Word embedding model

- What is the probability that word w_O is within the context of w_I?

$$p(w_O|w_I) = \frac{\exp\left({v'_{w_O}}^\top v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'_w}^\top v_{w_I}\right)}$$

- Each word is given an “input” embedding vector v_w and “output” embedding vector v'_w.

Word embedding model

- What is the probability that word w_O is within the context of w_I?

$$p(w_O|w_I) = \frac{\exp\left({v'_{w_O}}^\top v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'_w}^\top v_{w_I}\right)}$$

- If the inner product between these two vectors is relatively high (compared to other pairs of words in V), then the probability of their co-occurrence is high.

Softmax

- The problem with this softmax is that *V* is very large, and computing each probability is too slow.

$$p(w_O|w_I) = \frac{\exp\left({v'_{w_O}}^\top v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'_w}^\top v_{w_I}\right)}$$

- Instead, the authors borrow a technique called hierarchical softmax.

Hierarchical softmax

- Ultimately, all we care about is computing a probability distribution over *w* such that $P(w \mid w_I)$ sums to 1.

- Here's a clever way to achieve this without needing to normalize by summing over all words in *V*.

$$p(w|w_I) = \prod_{j=1}^{L(w)-1} \sigma\left(\llbracket n(w, j+1) = \mathrm{ch}(n(w,j)) \rrbracket \cdot {v'_{n(w,j)}}^\top v_{w_I}\right)$$

- We can think of this as navigating down a "tree", where each possible *w* corresponds to one leaf node.

Negative sampling

- Alternatively, they also explore how simply training on a set of “negative” contexts (i.e., not found in the dataset) can encourage the model to learn useful embeddings:

$$\log \sigma({v'_{w_O}}^\top v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[\log \sigma(-{v'_{w_i}}^\top v_{w_I}) \right]$$

- In this loss function, the first term represents words+contexts that are in the dataset, and the second term represents words+contexts not in the dataset.

Evaluation

- Analogical reasoning — complete the fourth word in each of the following:

Newspapers			
New York	New York Times	Baltimore	Baltimore Sun
San Jose	San Jose Mercury News	Cincinnati	Cincinnati Enquirer
NHL Teams			
Boston	Boston Bruins	Montreal	Montreal Canadiens
Phoenix	Phoenix Coyotes	Nashville	Nashville Predators
NBA Teams			
Detroit	Detroit Pistons	Toronto	Toronto Raptors
Oakland	Golden State Warriors	Memphis	Memphis Grizzlies
Airlines			
Austria	Austrian Airlines	Spain	Spainair
Belgium	Brussels Airlines	Greece	Aegean Airlines
Company executives			
Steve Ballmer	Microsoft	Larry Page	Google
Samuel J. Palmisano	IBM	Werner Vogels	Amazon

Key results

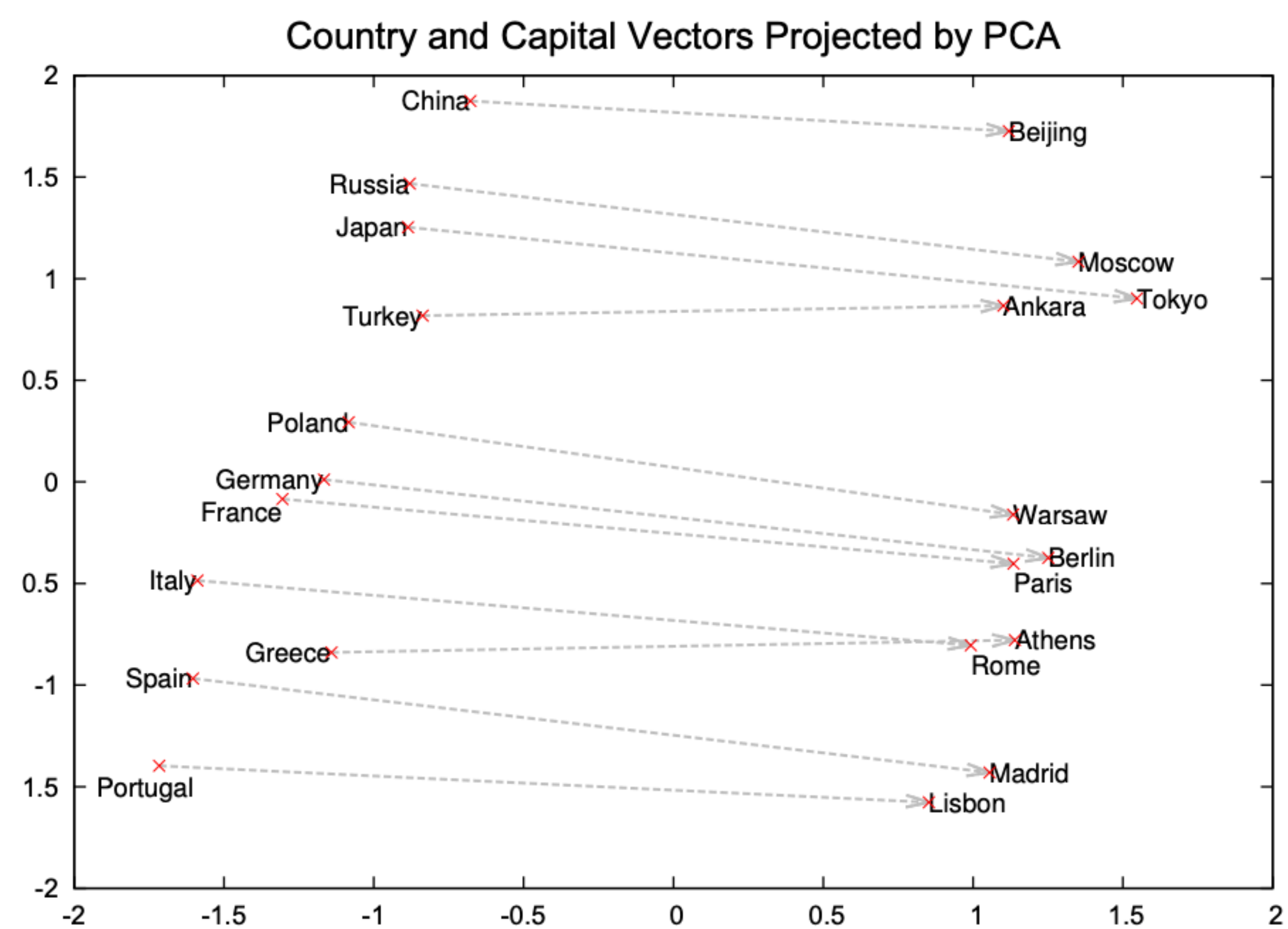
- Analogical reasoning comparison between hierarchical softmax (HS) and negative sampling (NCE):

Method	Dimensionality	No subsampling [%]	10^{-5} subsampling [%]
NEG-5	300	24	27
NEG-15	300	27	42
HS-Huffman	300	19	**47**

Compositionality

- One of the coolest parts of their work is the implicit compositionally that emerged between the words in the embedding space.

- For instance, the word “capital” seemed to have a consistent direction (as a vector) between each country and capital city:



Conclusion

- This paper was one of the first to show how a neural network can automatically compute continuous-valued low-dimensional embeddings for large vocabularies of words so that:
    - Training is fast (only 1 day for billions of sentences).
    - The set of vectors are representationally expressive (with implicit compositionality).
- word2vec now serves as the “go-to” feature representation for words in many NLP tasks.