# Advanced Topics in Machine Learning

Alejo Nevado-Holgado

## Lecture 12 (NLP 4) - Language models and vanilla RNNs

V 0.3 (15 Feb 2020 - final version)

# Course structure

- **Introduction:** What is NLP. Why it is hard. Why NNs work well ← Lecture 9 (NLP 1)
- **Word representation:** How to represent the meaning of individual words
  - Old technology: One-hot representations, synsets ← Lecture 9 (NLP 1)
  - Embeddings: First trick that boosted the performance of NNs in NLP ← Lecture 9 (NLP 1)
    - Word2vec: Single layer NN. CBOW and skip-gram ← Lecture 10 (NLP 2)
    - Co-occurrence matrices: Basic counts and SVD improvement ← Lecture 10 (NLP 2)
    - Glove: Combining word2vec and co-occurrence matrices idea ← Lecture 10 (NLP 2)
    - Evaluating performance of embeddings ← Lecture 10 (NLP 2)
- **Named Entity Recognition (NER):** How to find words of specific meaning within text
  - Multilayer NNs: Margin loss. Forward- and back-propagation ← Lecture 11 (NLP 3)
  - Better loss functions: margin loss, regularisation ← Lecture 11 (NLP 3)
  - Better initializations: uniform, xavier ← Lecture 11 (NLP 3)
  - Better optimizers: Adagrad, RMSprop, Adam... ← Lecture 11 (NLP 3)

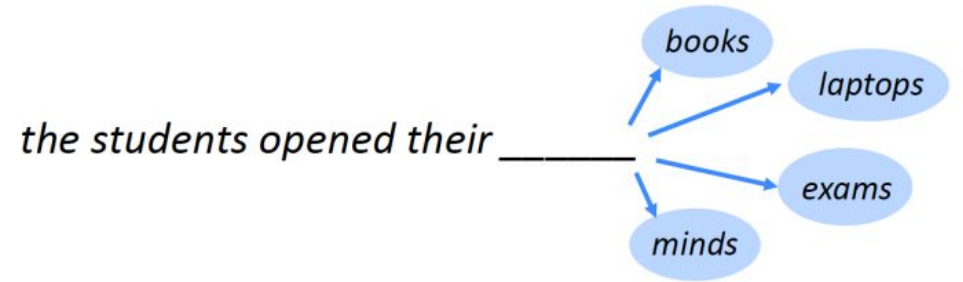
# Course structure

- **Language modelling:** How to represent the meaning of full pieces of text
  - Old technology: N-grams ← Lecture 12 (NLP 4)
  - Recursive NNs language models (RNNs) ← Lecture 12 (NLP 4)
  - Evaluating performance of language models ← Lecture 12 (NLP 4)
  - Vanishing gradients: Problem. Gradient clipping ← Lecture 13 (NLP 5)
  - Improved RNNs: LSTM, GRU ← Lecture 13 (NLP 5)
- **Machine translation:** How to translate text
  - Old technology: Georgetown–IBM experiment and ALPAC report ← Lecture 16 (NLP 6)
  - Seq2seq: Greedy decoding, encoder-decoder, beam search ← Lecture 16 (NLP 6)
  - Attention: Simple attention, transformers, reformers ← Lecture 16 (NLP 6)
  - Evaluating performance: BLEU ← Lecture 16 (NLP 6)

# Language modelling

**The problem:** Having a method to accurately represent the meaning of individual words (e.g. word embeddings), enormously helps to solve the simpler NLP tasks, but falls short in complex ones. Having a more advanced method that represents the meaning of full pieces of text (rather than separated words), helps to solve the more challenging tasks.

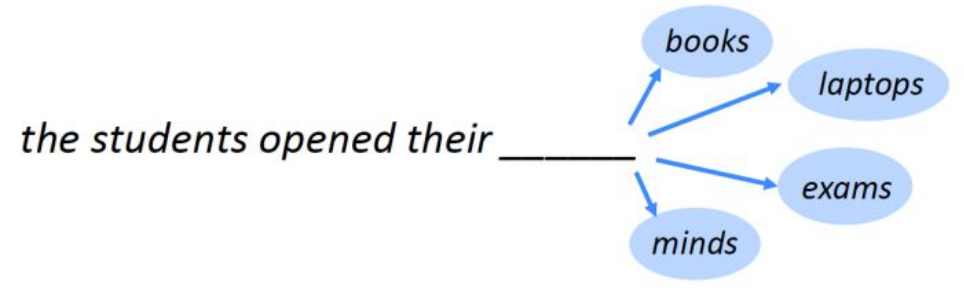
We created good word representations by predicting the presentation of the central word from the representations of a few neighbouring words (CBOW). We can create good representations of full pieces of text by predicting each word from all the preceding ones → Language models




$$P(\boldsymbol{x}^{(t+1)} | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

# Language modelling

- A language model itself is simply an algorithm that assigns probabilities to the next

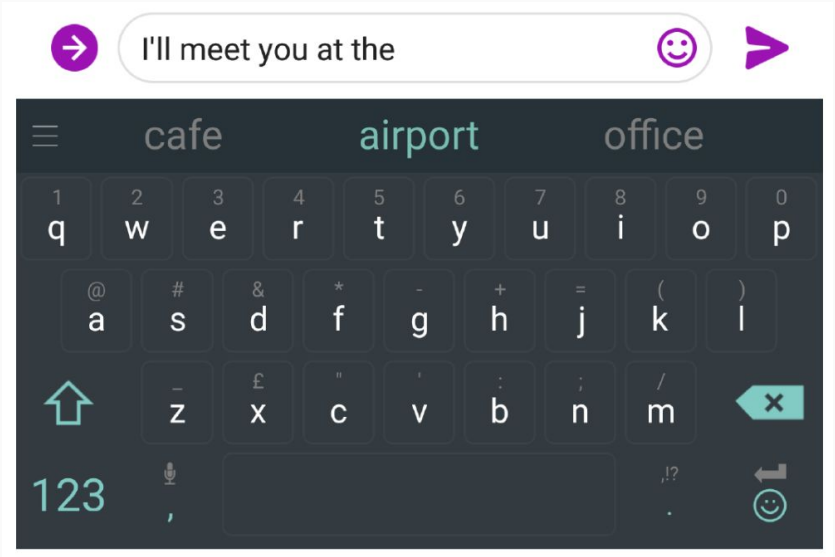


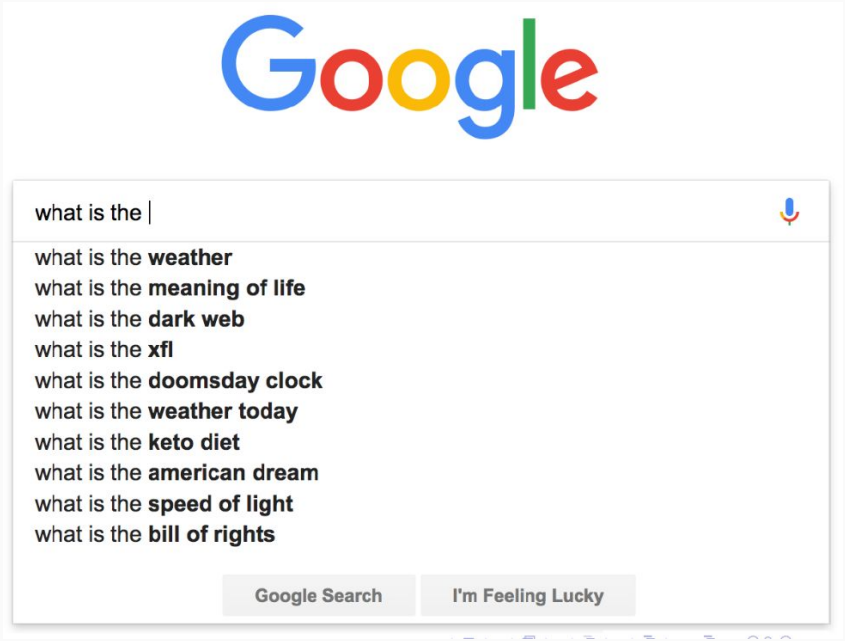

- More formally: Given a sequence of words $w^{(1)}, w^{(2)}, w^{(3)}, \ldots, w^{(t)}$ and their representations $x^{(1)}, x^{(2)}, x^{(3)}, \ldots, x^{(t)}$, what the model calculates is an approximation of:

$$P(\boldsymbol{x}^{(t+1)} | \ \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

# Language modelling

**Applications:** Predicting the next word in a piece of text has some direct applications

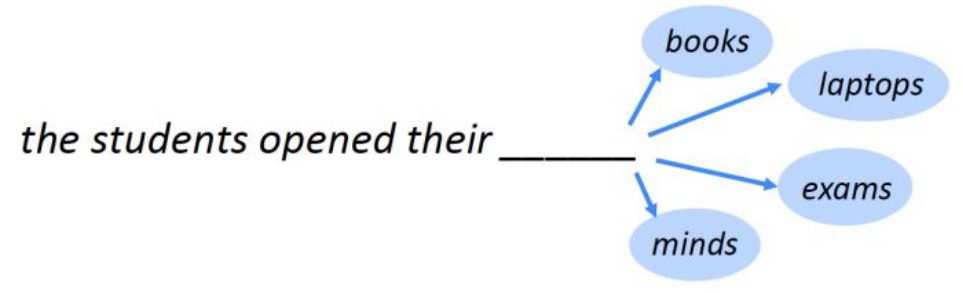


# Language modelling

**Applications:** But predicting the next word in a piece of text is not the most useful thing itself. What is more useful is (1) the numerical representation that your method builds from all preceding text in order to accurately predict the next word, and (2) the method that accurately predicts the next word.

Both the presentation (1) and the method (2), when they solve accurately the language modelling problem, they also work well in other applications: NER, translation, question answering, conference resolution...




$$P(\boldsymbol{x}^{(t+1)} | \ \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

# Old technology: N-grams

- **Idea:** Assume that the probability of the next word, depends mostly from the previous ‘n’ words, rather than from the full preceding text. Then, rather than calculating the probability of every possible imaginable meaningful sequence of words, we only need to calculate the probability of each sequence of ‘n’ words

$$P(x^{(t+1)} \mid x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(1)}) \approx P(x^{(t+1)} \mid x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(t-n)})$$

- Higher ‘n’ will theoretically be able to become more accurate… but would require very large amounts of data:
  - Unigrams: P(“the”), P(“students”), P(“opened”), P(“their”)...
  - Bigrams: P(“the students”), P(“students opened”), P(“opened their”)...
  - Trigrams: P(“the students opened”), P(“students opened their”)...
  - 4-grams: P(“the students opened their”)...

# Old technology: N-grams

- **Implementation:** For an n-gram of size ‘n’, count the number of times that each combination of ‘n’ and ‘n-1’ words appear in your corpus. Then apply the formula definition of “conditional probability”

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

$$P(x^{(t+1)}|x^{(t)}, x^{(t-1)}, \ldots, x^{(t-n)}) = \frac{count(x^{(t+1)}, x^{(t)}, x^{(t-1)}, \ldots, x^{(t-n)})}{count(x^{(t)}, x^{(t-1)}, \ldots, x^{(t-n)})}$$

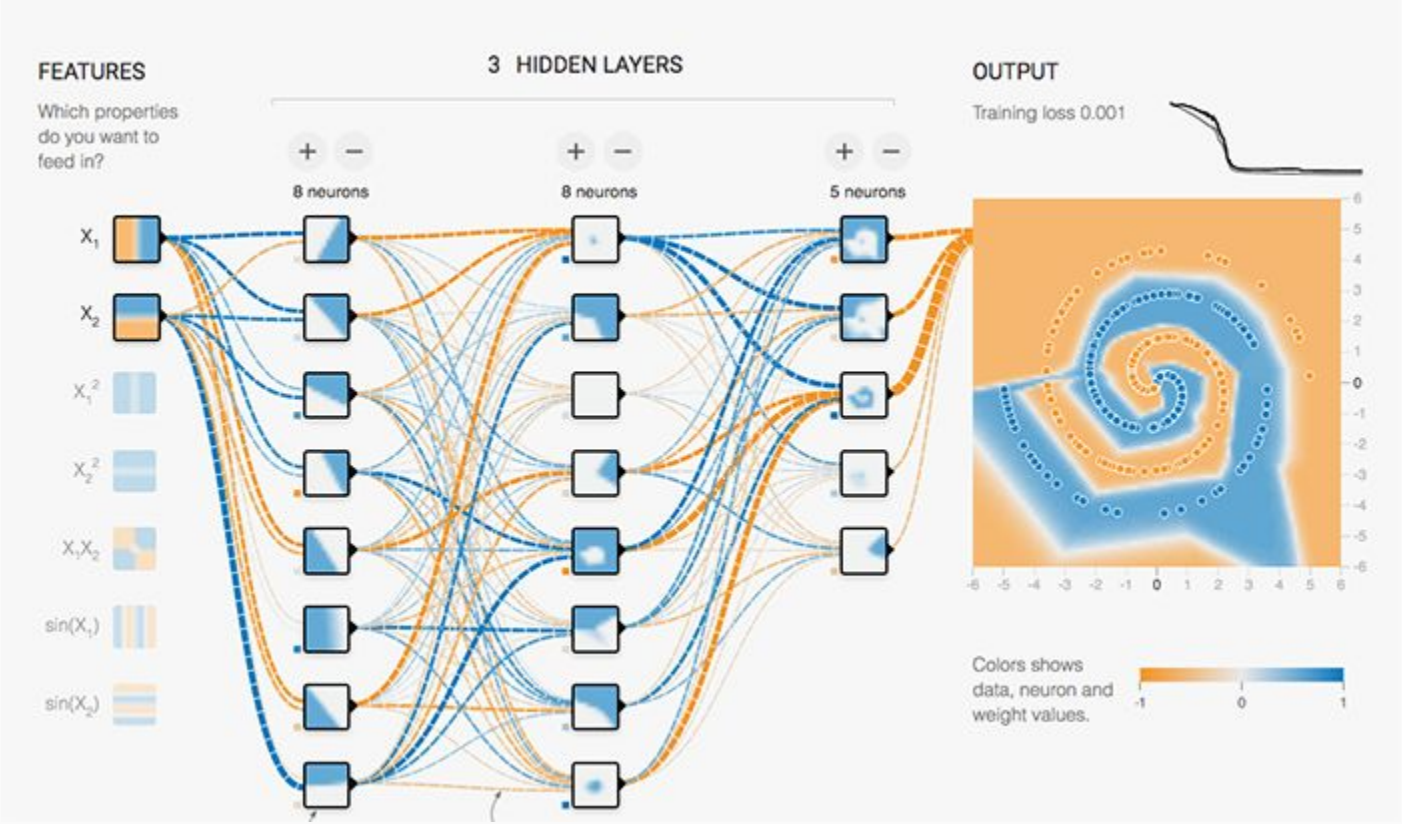
$$P(\ w \mid students\ opened\ their\ ) = \frac{count(\ students\ opened\ their\ w\ )}{count(\ students\ opened\ their\ )}$$

# Old technology: N-grams

- **Problems:** N-gram generated text is often surprisingly gramatical, but incoherent
  - 0 probabilities: P( w | students opened their ) will be 0 if ‘w’ never appears in your corpus. You can partially solve this with smoothing, which adds a small number δ to all counts
  - Undefined probabilities: P( w | students opened their ) will be ‘0/0’ if ‘students opened their’ never appears in your corpus. You can partially solve this with backoff, using a ‘n-1’-gram when the problem arises - i.e. counting ‘opened their’
  - Storage: You need to store counts for all ‘n’ and ‘n-1’ combinations. Size = $V^n + V^{n-1}$
  - Sparsity: Increasing ‘n’ to improve accuracy, makes the previous problems worse

$$P(\, w \mid students\ opened\ their\,) = \frac{count(\, students\ opened\ their\ w\,)}{count(\, students\ opened\ their\,)}$$

# Neural networks




→ Google Cloud tutorial by Kaz Sato:
https://cloud.google.com/blog/products/gcp/understanding-neural-networks-with-tensorflow-playground

# New technology: NNs

~~as the teacher started the clock~~ the students opened their _____

*discard (out of window)* *process (window)*

- Output distribution:

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

- Hidden layer:

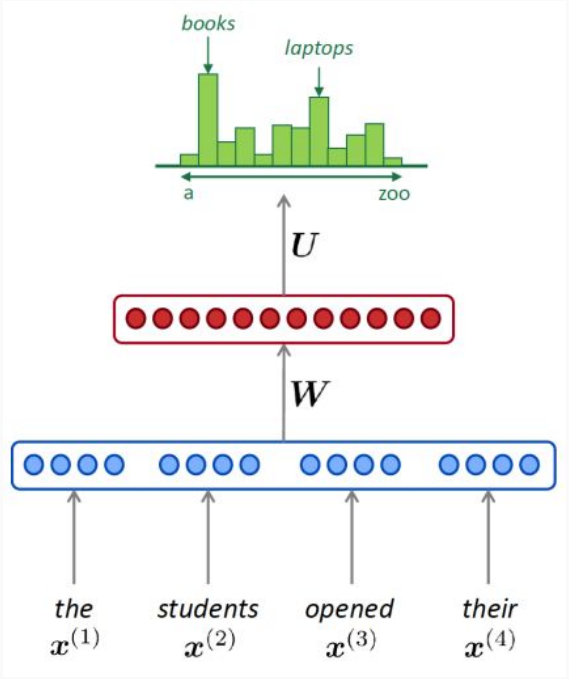
  $\mathbf{h} = f( \mathbf{W} \times \mathbf{e} + \mathbf{b}_1 ) \in \mathbb{R}^H$

- Concatenated word embeddings:

  $\mathbf{e} = [ \mathbf{e}^{(t-3)}, \mathbf{e}^{(t-2)}, \mathbf{e}^{(t-1)}, \mathbf{e}^{(t)} ] = [ \mathbf{e}^{(1)}, \mathbf{e}^{(2)}, \mathbf{e}^{(3)}, \mathbf{e}^{(4)} ] \in \mathbb{R}^{4E}$

- One-hot word vectors:

  $\mathbf{x} = [ \mathbf{x}^{(t-3)}, \mathbf{x}^{(t-2)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t)} ] = [ \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)} ] \in \mathbb{R}^{4V}$

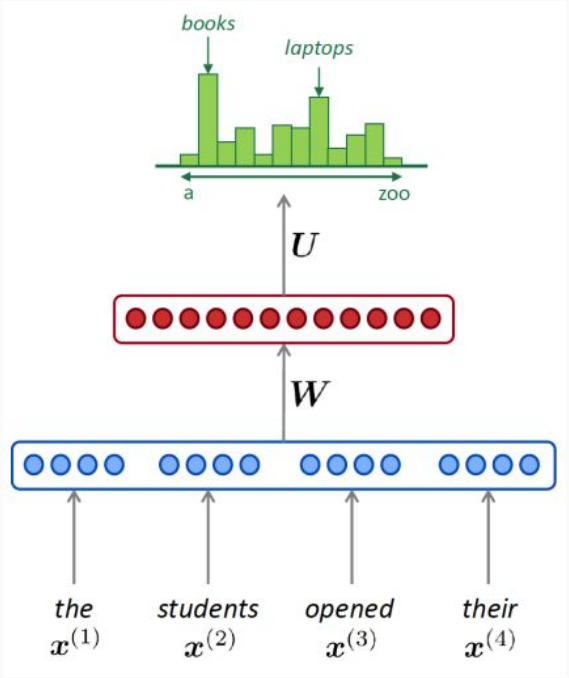


# New technology: NNs

~~as the teacher started the clock~~ the students opened their _____

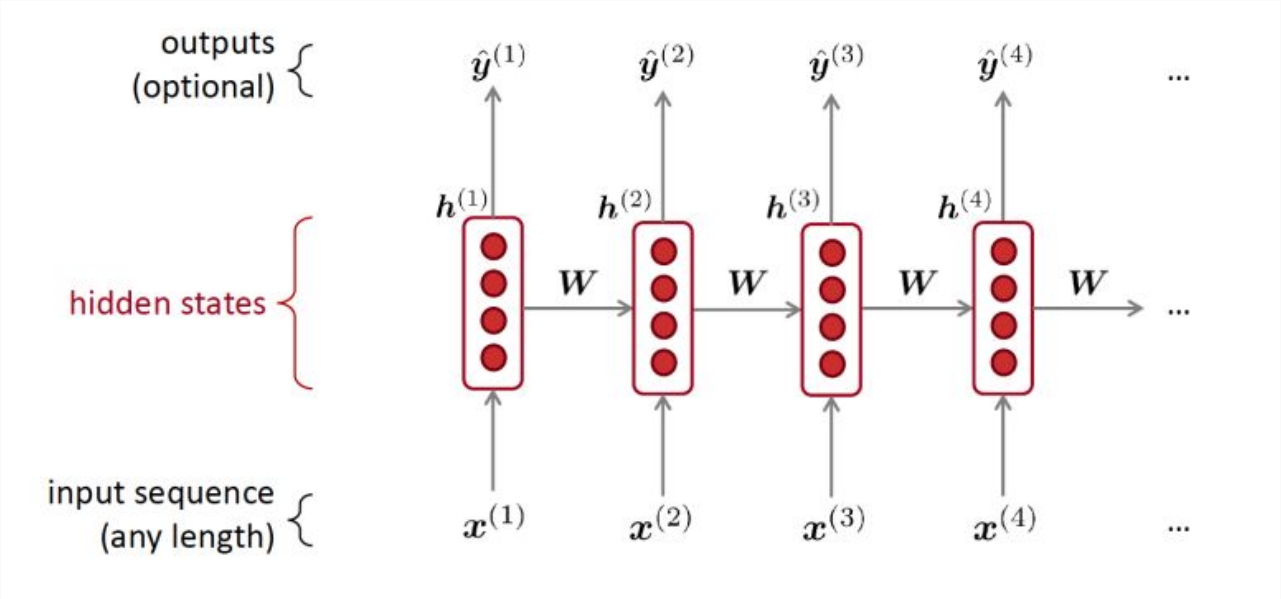
*discard (out of window)* *process (window)*

- ➢ Advantages of window-NNs over n-grams:
  - No sparsity problem
  - Don't need to store all combinations of n words
- ➢ Remaining problems:
  - Fixed window width is too small… and we can never make it large enough!
  - Enlarging window increases the size of W
  - No symmetry in how words are processed: Each word (e.g. $x^{(1)}$ and $x^{(2)}$) are multiplied by completely different weights in W

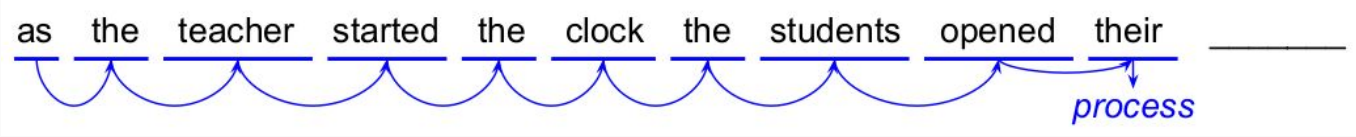


# Recursive NNs (RNNs)

- **Idea:** Recursively process one word at a time rather than all words within a window → No window limit, no increase in size of W, processing symmetry (all words are processed in the same manner)



# Recursive NNs (RNNs)




- Output distribution:

  $\mathbf{y}^{(t)} = \text{softmax}(\mathbf{U} \times \mathbf{h}^{(t)} + \mathbf{b}_2) \in \mathbb{R}^V$

- Hidden layer:

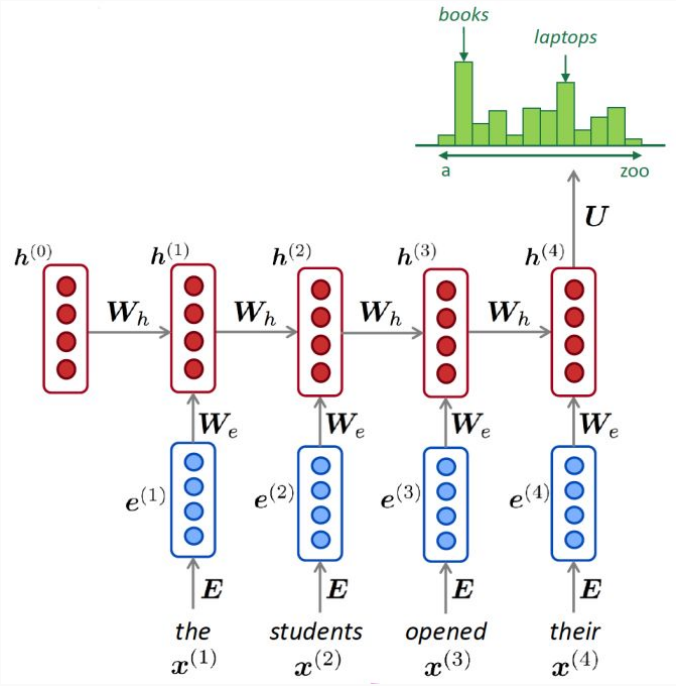
  $\mathbf{h}^{(t)} = f(\mathbf{W}_h \times \mathbf{h}^{(t-1)} + \mathbf{W}_e \times \mathbf{e}^{(t)} + \mathbf{b}_1) \in \mathbb{R}^H$

- One word embedding at a time:

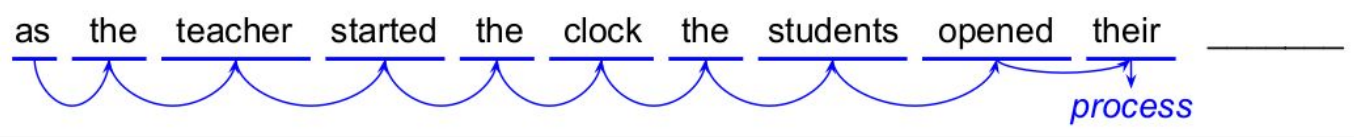
  $\mathbf{e}^{(t)} = \mathbf{E} \times \mathbf{x}^{(t)} \in \mathbb{R}^E$

- One one-hot word vector at a time:

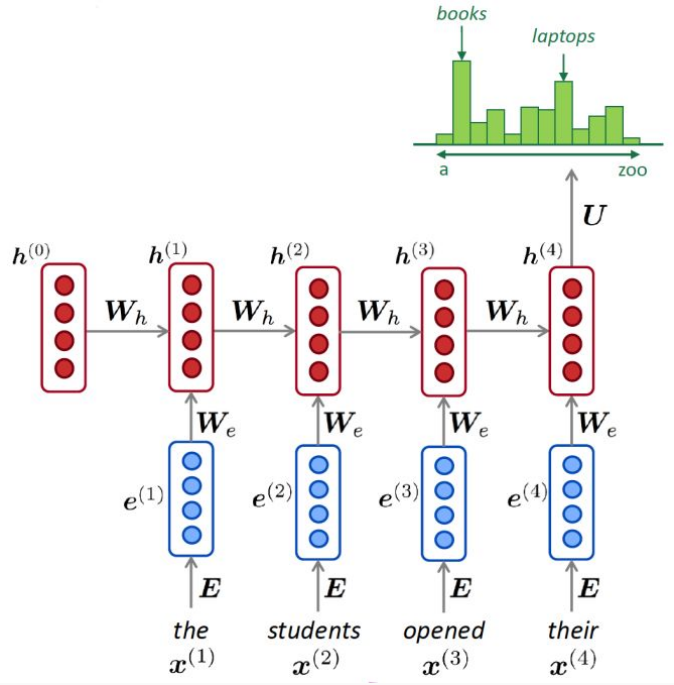
  $\mathbf{x}^{(t)} \in \mathbb{R}^V$



# Recursive NNs (RNNs)




- ➢ Advantages of RNNs over window-NNs:
  - No window limit, can process any number of words
  - No increase in size of W when processing more words
  - Processing symmetry, all words are processed in the same manner
- ➢ Remaining problems:
  - Recurrent computation is slow
  - In practice, RNNs have problems using information from many words back (e.g. $x^{(t-100)}$) → short memory span



# Learning in word2vec word model

- Word2vec very successfully implemented word embeddings using this context-meaning idea.
  - We start with a very large corpus of text (e.g. all of Wikipedia)
  - Every word is represented by a vector in $\mathbb{R}^n$ space (n~200 dimentions)
  - You have a model (e.g. a NN) that tries to predict the vector of a word (i.e. the central word) given the vectors of the words around it (i.e. its context). In probability terms, the NN models the probability P( $w_c$ | $w_{c-3}$, $w_{c-2}$, $w_{c-1}$, $w_{c+1}$, $w_{c+2}$, $w_{c+3}$ )
  - Go through each central word - context pair in the corpus
  - In each iteration, modify the NN and vectors a little bit for words with similar contexts to have similar vectors
  - Repeat last 2 steps many times

# Learning in RNNs language model

- Word2vec very successfully implemented word embeddings using this context-meaning idea.
  - We start with a very large corpus of text (e.g. all of Wikipedia)
  - Every word is represented by a vector in $\mathbb{R}^n$ space (n~200 dimentions)
  - You have a model (~~e.g. a NN~~ i.e. a RNN) that tries to predict the vector of a word (~~i.e. the central word~~ i.e. the next word) given the vectors of the words ~~around it (i.e. its context)~~ before it. In probability terms, the NN models the probability ~~$P( w_c \mid w_{c-3}, w_{c-2}, w_{c-1}, w_{c+1}, w_{c+2}, w_{c+3} )$~~ $P( w_{t+1} \mid w_t, w_{t-1}, w_{t-2}, w_{t-3}, \dots )$
  - Go through each central word - context pair in the corpus
  - In each iteration, modify the NN and vectors a little bit for words with similar contexts to have similar vectors → We now know we do this with backpropagation
  - Repeat last 2 steps many times

# Recursive NNs (RNNs)

- With backpropagation we want our RNN to gradually learn to approximate the probability P( $w_{t+1}$ | $w_t$, $w_{t-1}$, $w_{t-2}$, $w_{t-3}$, … )
- But backpropagation needs an error function that represents numerically how far our RNN is from perfectly approximating P( $w_{t+1}$ | $w_t$, $w_{t-1}$, $w_{t-2}$, $w_{t-3}$, … )
- Error functions used before:
  - CBOW (word2vec): Negative Sampling[arXiv:1310.4546] (simplified version of Negative Contrastive Sampling[arXiv:1206.6426])

$$J = log\ sig(\ \underset{E}{[y_{centre}]} \cdot \underset{E}{[e_{context}]}\ ) + \sum_{i \sim P_{noise}}^{K} log\ sig(\ \underset{E}{[y_i]} \cdot \underset{E}{[e_{context}]}\ )$$

$\underset{V}{[something]} \longrightarrow$ 'something' is a vector of V elements

$\underset{V}{[something]}(v) \longrightarrow$ element 'v' of vector $\underset{V}{[something]}$

$E \longrightarrow$ number of embedding dimensions

$\underset{E}{[e_{something}]} \longrightarrow$ embedding of 'something'

$\underset{E}{[y_{something}]} \longrightarrow$ output of CBOW NN when input is $\underset{E}{[e_{something}]}$

# Recursive NNs (RNNs)

- More error functions used before:
  - Glove (hybrid of NN and co-occurrence matrix): Error function based on 3 heuristic considerations

$$J = \sum_{i,j=1}^{V} ramp\Big( \underset{V,V}{[C]}(i,j) \Big) \Big( \underset{E}{[e_i]} \cdot \underset{E}{[e_j]} + b_i + b_j - log\Big( \underset{V,V}{[C]}(i,j) \Big) \Big)^2$$

$V \longrightarrow$ size of vocabulary
$b_i \longrightarrow$ arbitrary constant for word 'i'
$ramp(\ ...\ ) \longrightarrow$ almost a ramp function

$\underset{R,C}{[something]} \longrightarrow$ 'something' is a matrix of R rows and C columns
$\underset{R,C}{[something]}(r,c) \longrightarrow$ element in row 'r' and column 'c' of $\underset{R,C}{[something]}$

$\underset{V,V}{[C]} \longrightarrow$ co-occurrence count matrix
$\underset{V,V}{[C]}(i,j) \longrightarrow$ count of how many times word 'i' is near word 'j'

  - NER: Margin loss

$$J = max\big( 0,\ 1 + s_{NEG} - s_{POS} \big)$$

$s_{NEG} \longrightarrow$ output of NER NN when input is $\underset{5E}{[x_{NEG}]}$
$s_{POS} \longrightarrow$ output of NER NN when input is $\underset{5E}{[x_{POS}]}$

# Recursive NNs (RNNs)

- New error function:
  - Cross-entropy

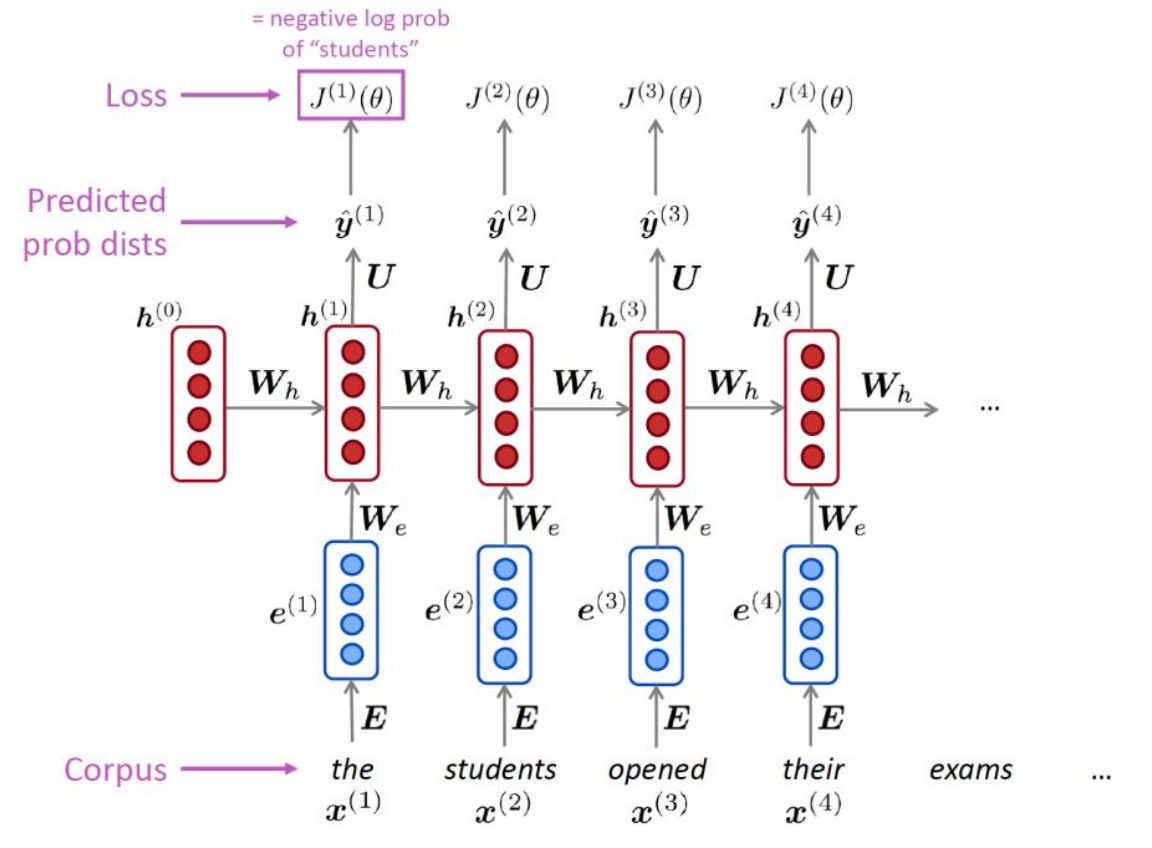
$$J = -\underset{V}{[x_{t+1}]} \cdot log(\underset{V}{[y_t]}) = -log(\underset{V}{[y_t]}(i))$$

$\underset{V}{[x_{t+1}]} \rightarrow$ one hot vector of the real next word (word number 't+1')

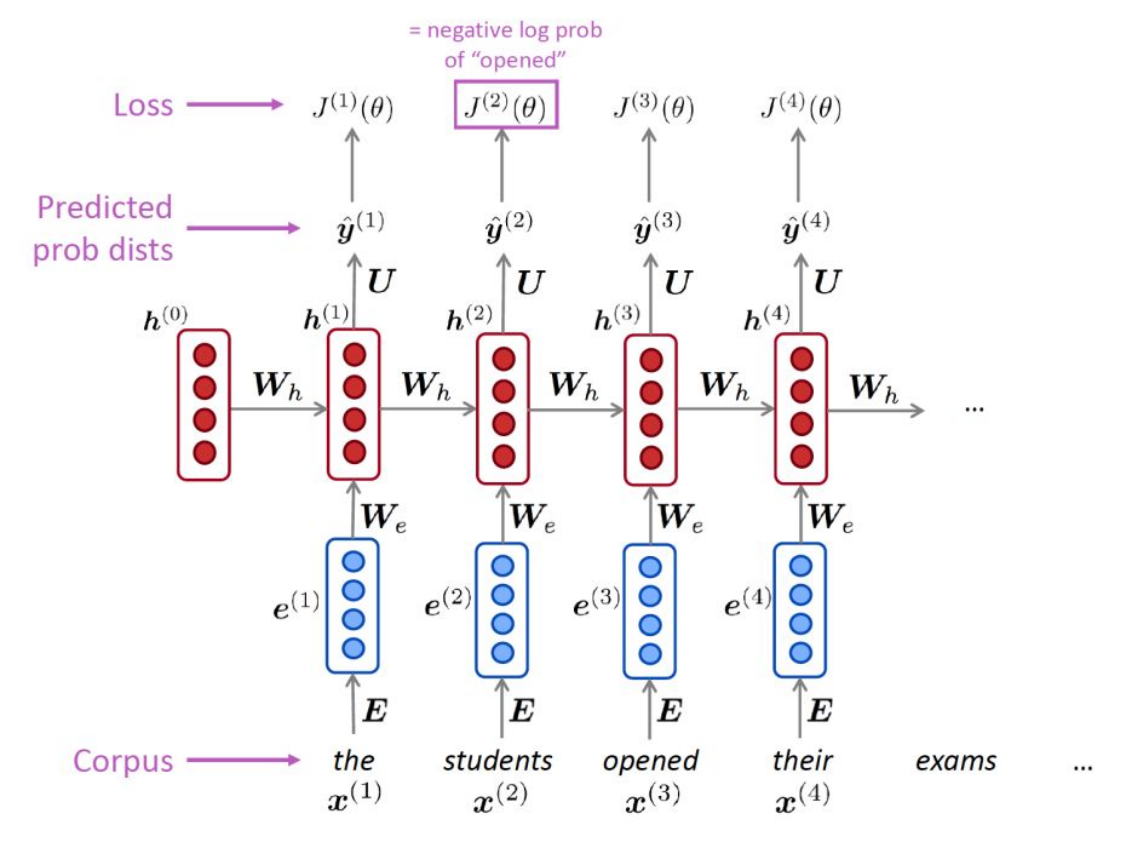
$\underset{V}{[y_t]} \rightarrow$ output of the neural network after reading up to word 't'

$i \rightarrow$ the only position of $\underset{V}{[x_{t+1}]}$ different than 0 ( $\underset{V}{[x_{t+1}]}(k) = 1 \iff k = i$ )

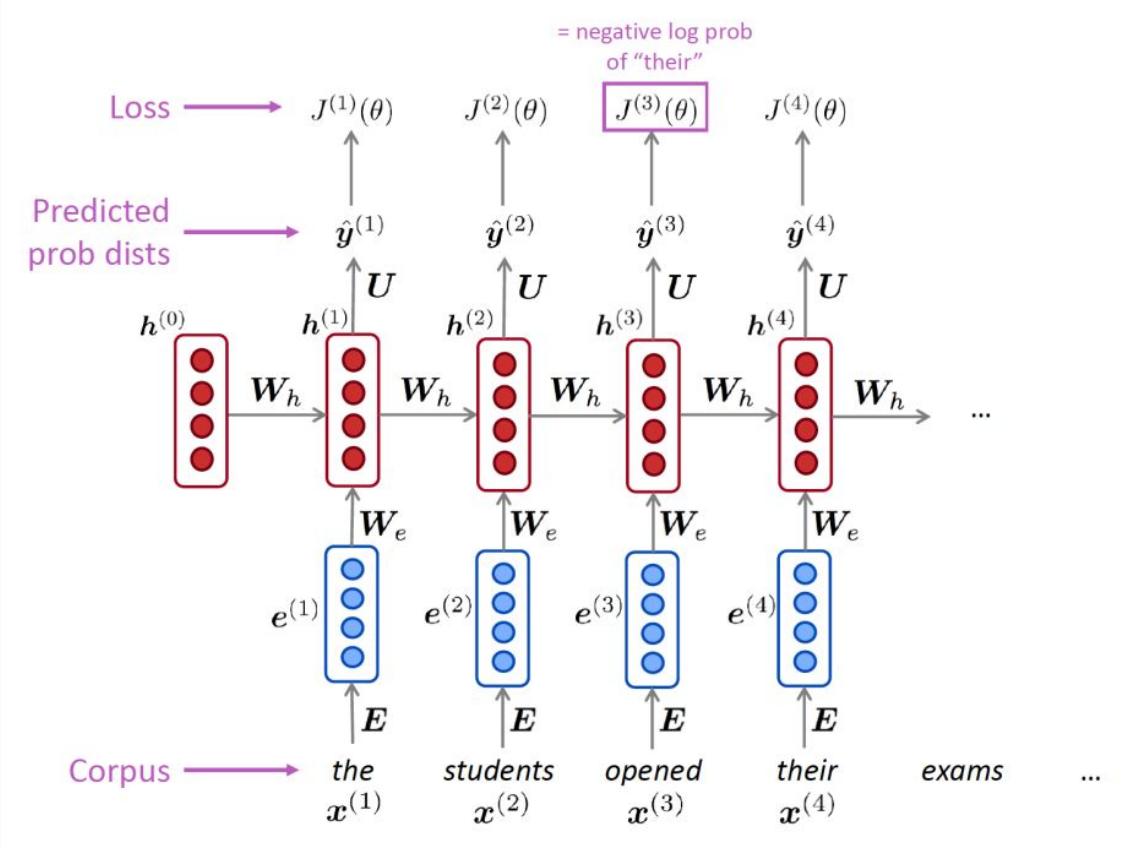
# Recursive NNs (RNNs)



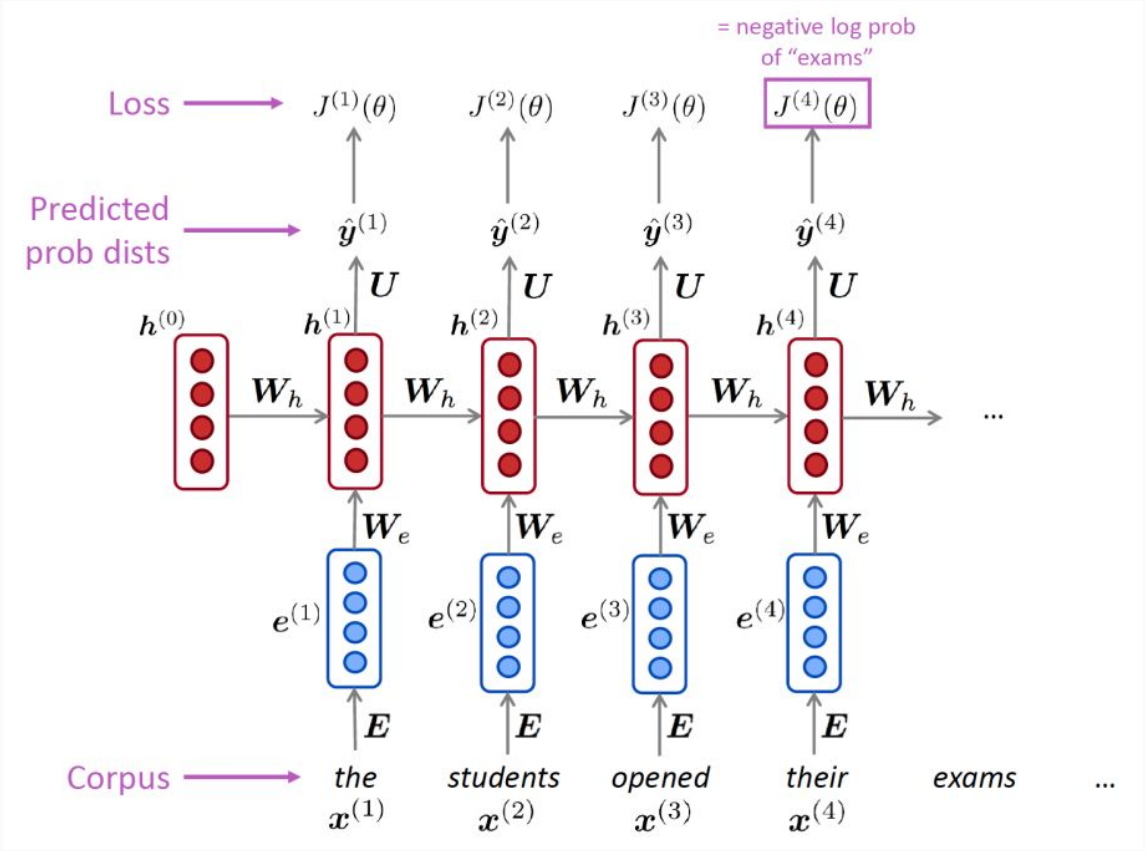
# Recursive NNs (RNNs)



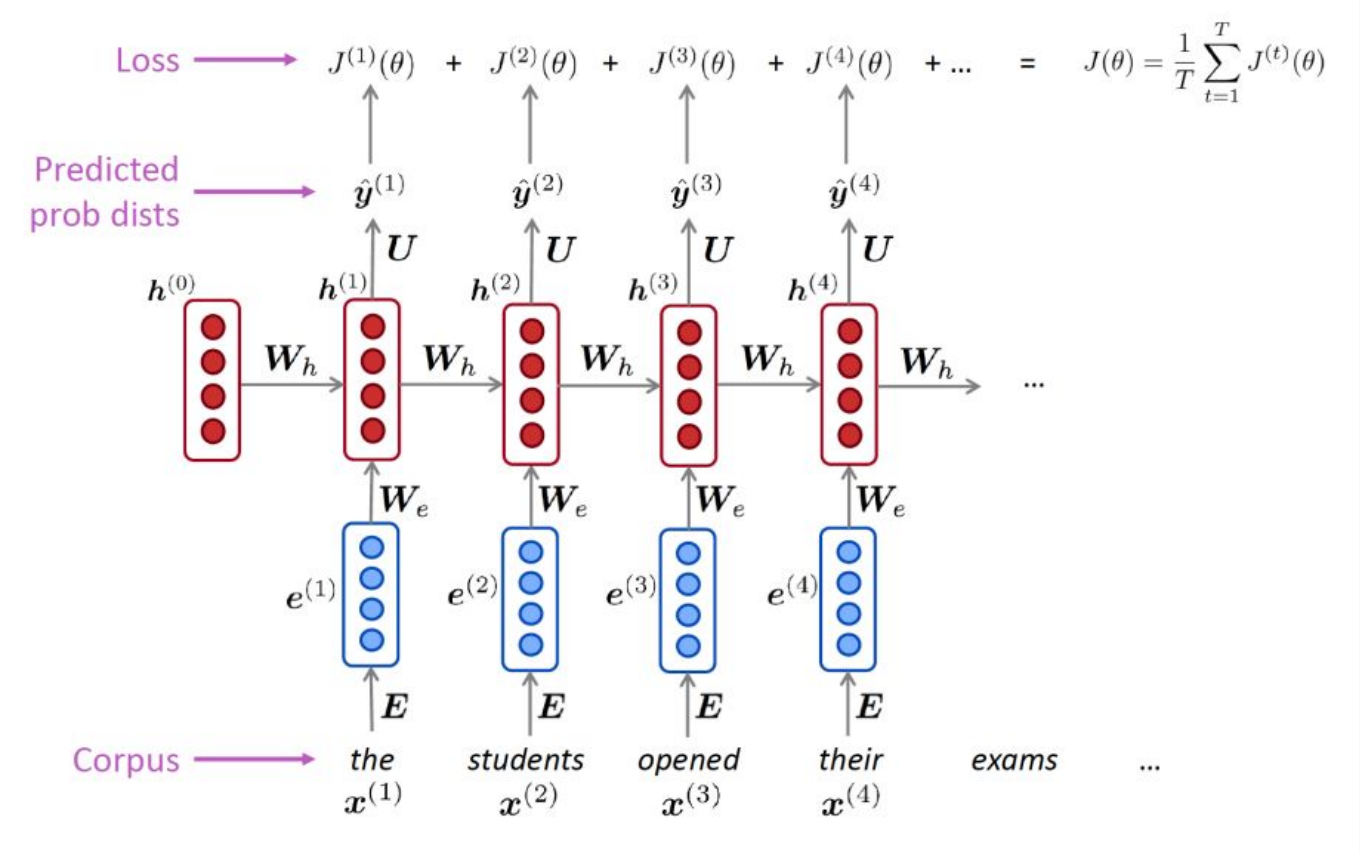
# Recursive NNs (RNNs)



# Recursive NNs (RNNs)

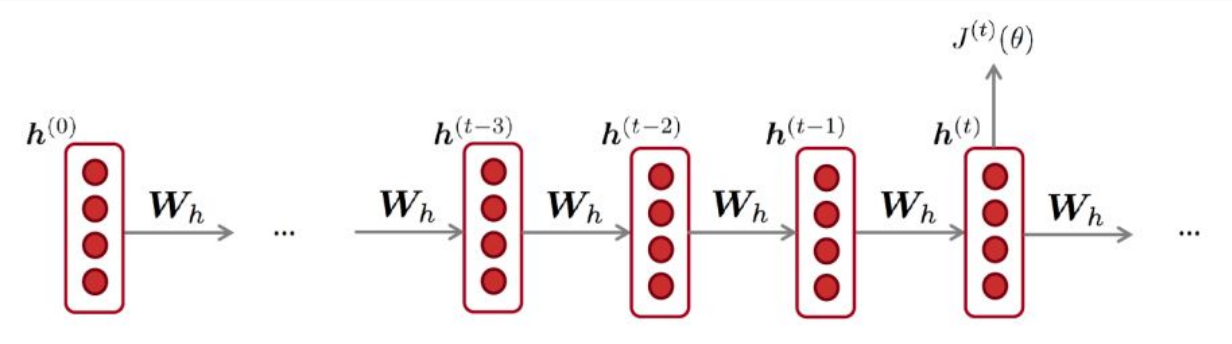


# Recursive NNs (RNNs)



# Recursive NNs (RNNs)

- **Minibatch learning:** Computing $\sum J^{(t)}(\theta)$ over all possible values of t (i.e. over all word positions in the corpus) in each iteration of Stochastic Gradient Descent (SGD) is too expensive. We rather computer $\sum J^{(t)}(\theta)$ only for a few random sentences in each iteration of SGD.
- **Backpropagation through time:** If we unfold the RNN along the previous time steps (t-1, t-2, t-3, … ), we can apply the chain rule to calculate the gradient of $\sum J^{(t)}(\theta)$ like in any other standard NN

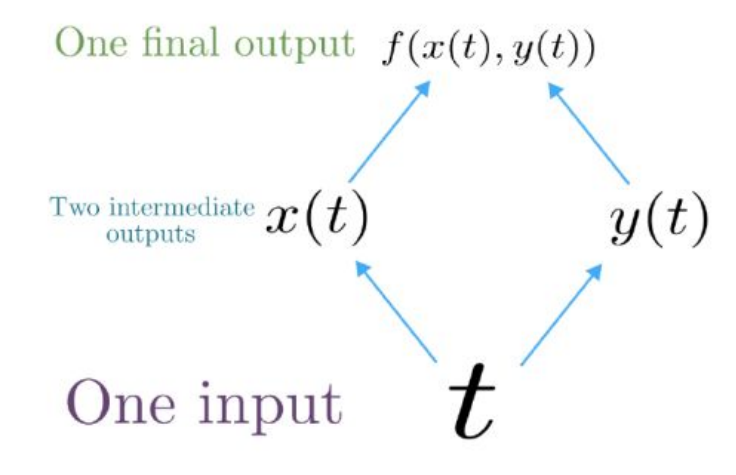



$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\bigg|_{(i)}$$

# Recursive NNs (RNNs)

- **The chain rule in RNNs:** Given a function f(x,y) that is a function of other two different functions of ‘t’ x(t) and y(t) ( making f(x,y) = f(x(t),y(t)) ), according to the chain rule the gradient of f is:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

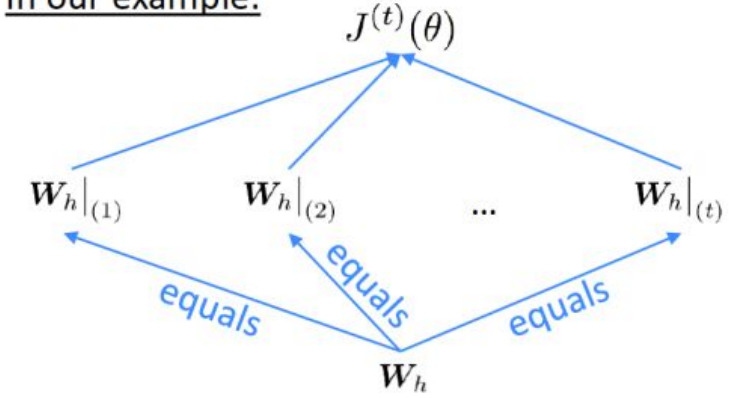


# Recursive NNs (RNNs)

➢ **The chain rule in RNNs:** Given a function f(x,y) that is a function of other two different functions of 't' x(t) and y(t) ( making f(x,y) = f(x(t),y(t)) ), according to the chain rule the gradient of f is:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

In our example:




Apply the multivariable chain rule:

$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\Big|_{(i)} \frac{\partial W_h|_{(i)}}{\partial W_h}$$

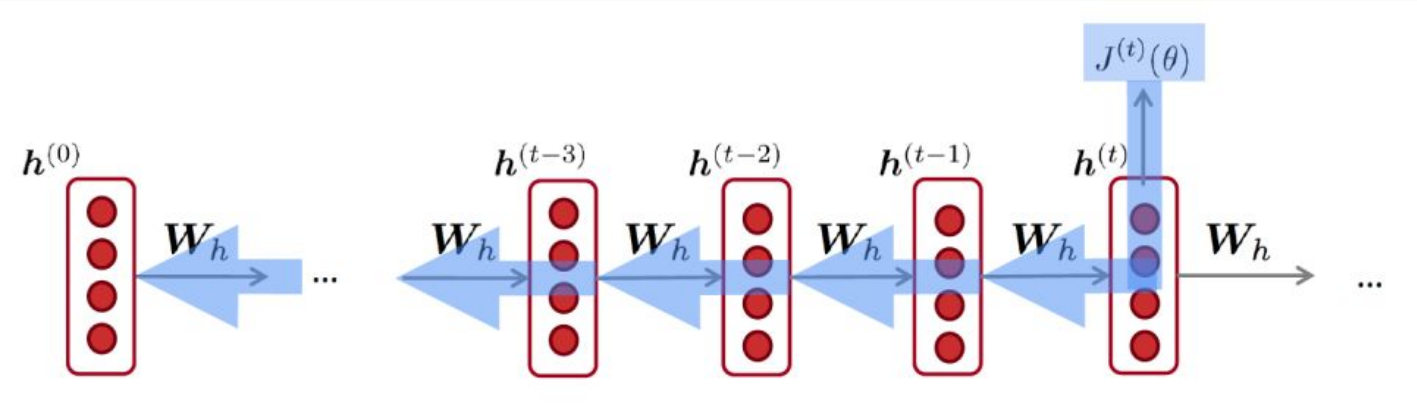
= 1

$$= \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial W_h}\Big|_{(i)}$$

# Recursive NNs (RNNs)

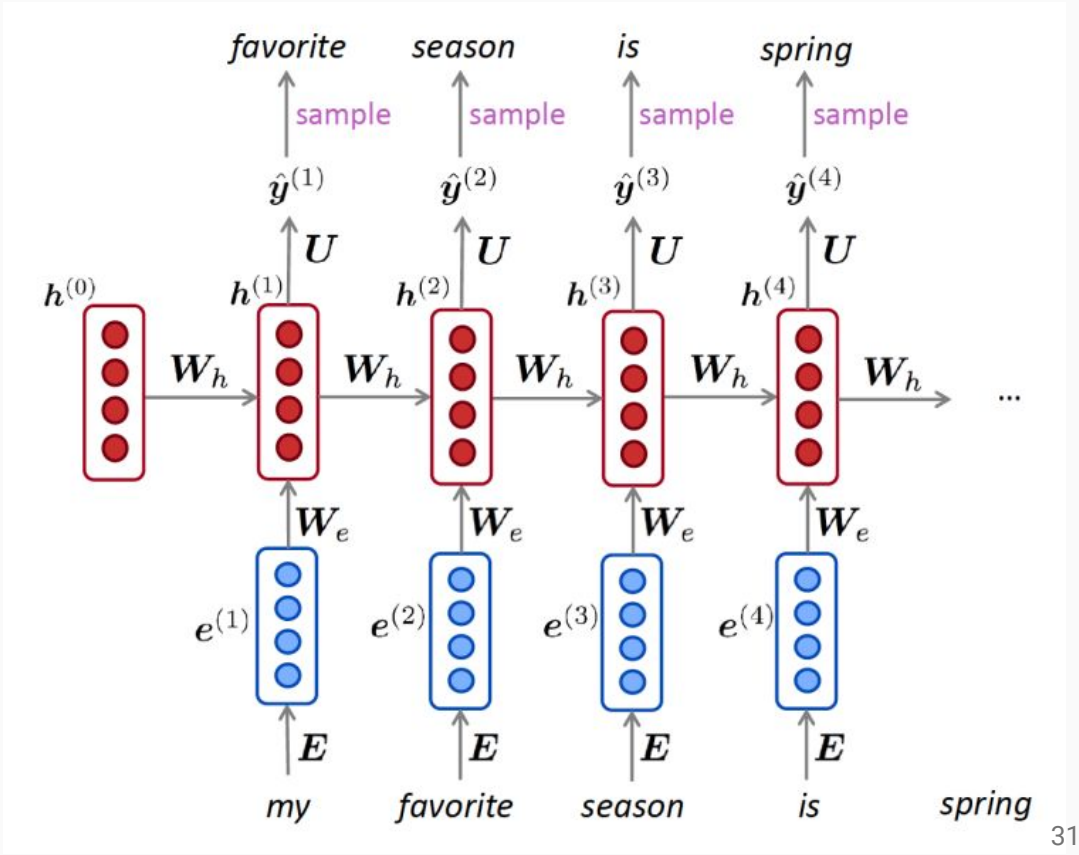
- **The chain rule in RNNs:** Given a function f(x,y) that is a function of other two different functions of 't' x(t) and y(t) ( making f(x,y) = f(x(t),y(t)) ), according to the chain rule the gradient of f is:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$



# Recursive NNs (RNNs)

- Once trained on a large corpus, you can generate text with a RNN language model by sampling the output distribution $y^{(t)}$ in each time step t, and using that word for time step t+1



# Recursive NNs (RNNs)

- RNN trained on obama speeches[Samin @ Medium]:

*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 trained on Harry Potter[Max Deutsch @ Medium]:

"Sorry," Harry shouted, panicking—"I'll leave those brooms in London, are they?"

"No idea," said Nearly Headless Nick, casting low close by Cedric, carrying the last bit of treacle Charms, from Harry's shoulder, and to answer him the common room perched upon it, four arms held a shining knob from when the spider hadn't felt it seemed. He reached the teams too.

# Recursive NNs (RNNs)

- RNN trained on paint color names[Janelle Shane]:


- This RNNs runs at the character level, not word level

# Evaluating quality of language models

- **Perplexity:** This is the standard evaluation metric for language models. It measures how 'surprised' the NN is of finding that the real word at step t+1 is $x^{(t+1)}$ after reading as input all previous words $x^{(t)}$, $x^{(t-1)}$, $x^{(t-2)}$, $x^{(t-3)}$, …

$$perplexity(t) = \prod_{t=1}^{T} \left( \frac{1}{P_{RNN}\left( x^{(t+1)} \mid x^{t}, x^{t-1}, x^{t-2}, \dots \right)} \right)^{1/T} = \prod_{t=1}^{T} \left( \frac{1}{\underset{V}{[x_{t+1}]} \cdot \underset{V}{[y_t]}} \right)^{1/T}$$

- Surprisingly enough, if we apply the exponential rules $e^{ab}=e^{a}+e^{b}$ and $e^{log(a)}=a$ , it turns out that perplexity is equal to the exponential of the total loss J:

$$\prod_{t=1}^{T} \left( \frac{1}{\underset{V}{[x_{t+1}]} \cdot \underset{V}{[y_t]}} \right)^{1/T} = exp\left( \frac{1}{T} \sum_{t=1}^{T} -log\left( \underset{V}{[x_{t+1}]} \cdot \underset{V}{[y_t]} \right) \right) = exp(J(\theta))$$

# Evaluating quality of language models

➢ Language models have greatly improved perplexity during the last years[Grave & Joulin]

*n*-gram model →

Increasingly complex RNNs ↓

| Model | Perplexity |
|---|---|
| Interpolated Kneser-Ney 5-gram (Chelba et al., 2013) | 67.6 |
| RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013) | 51.3 |
| RNN-2048 + BlackOut sampling (Ji et al., 2015) | 68.3 |
| Sparse Non-negative Matrix factorization (Shazeer et al., 2015) | 52.9 |
| LSTM-2048 (Jozefowicz et al., 2016) | 43.7 |
| 2-layer LSTM-8192 (Jozefowicz et al., 2016) | 30 |
| **Ours small** (LSTM-2048) | 43.9 |
| **Ours large** (2-layer LSTM-2048) | 39.8 |

Perplexity improves (lower is better) ↓

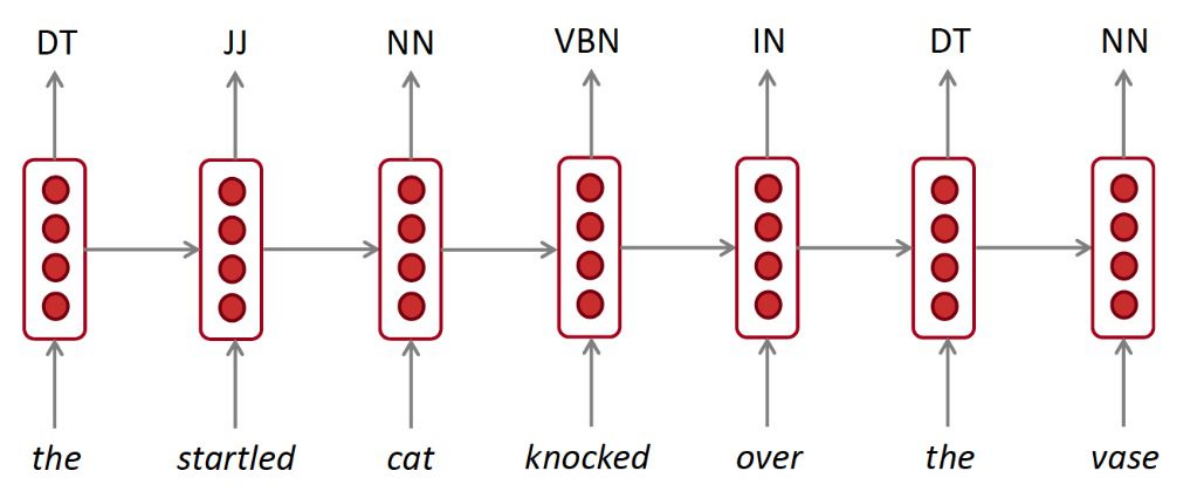
[Gave & Joulin] = https://research.fb.com/building-an-efficient-neural-language-model-over-a-billion-words/

# Applications of language models

- Language models become a subcomponent task of many other more complex tasks:
  - Predictive typing
  - Speech recognition
  - Handwriting recognition
  - Spelling/grammar correction
  - Text classification
  - Machine translation
  - Summarization
  - Dialogue
  - etc...

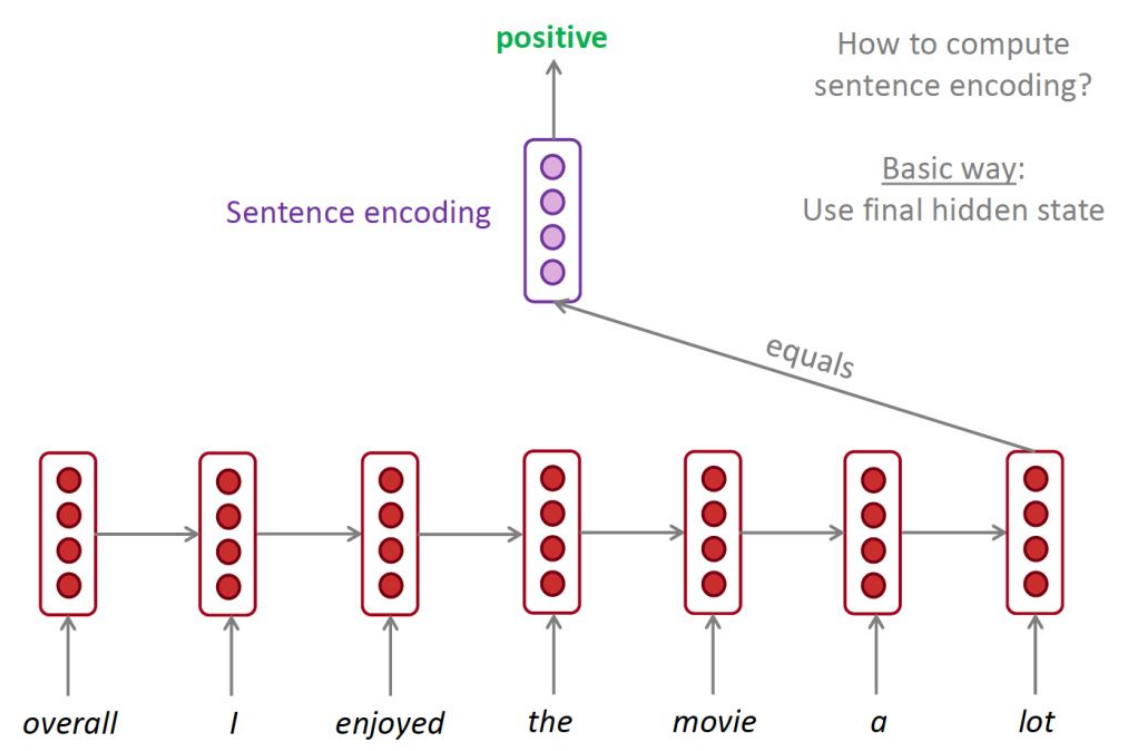
# Applications of language models

- Language models can be used for Part Of Speech (POS) or Named Entity Recognition (NER):



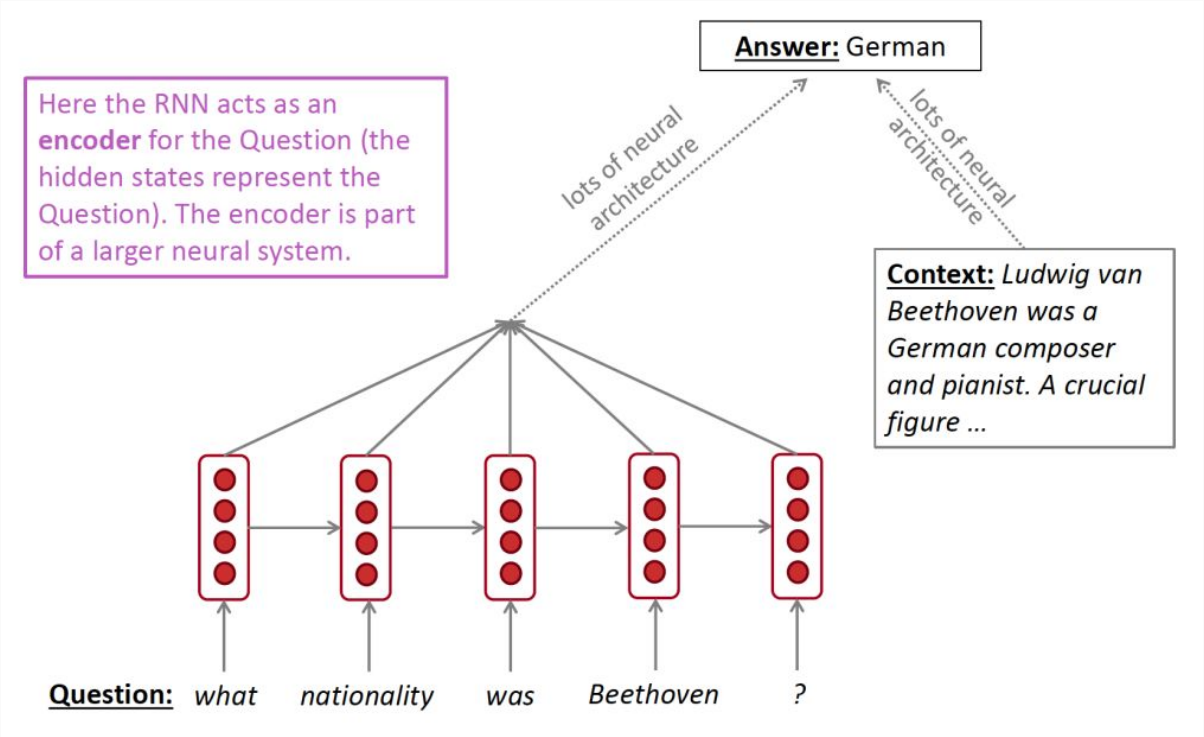
# Applications of language models

➢ Language models can be used for text classification (e.g. sentiment analysis):



# Applications of language models

- Language models can be used for question answering



# Next lecture!

- In this lecture: We have described the simplest type of RNN → "vanilla RNN"
- Next lecture: We will learn more complex and powerful ones

→ GRU  → LSTM  → Multiplayer RNNs 

- By end of course: We will learn much more advanced ones

bidirectional
attention
self-attention
residual networks
transformers
reformers...

# Course structure

- **Introduction:** What is NLP. Why it is hard. Why NNs work well ← Lecture 9 (NLP 1)
- **Word representation:** How to represent the meaning of individual words
  - Old technology: One-hot representations, synsets ← Lecture 9 (NLP 1)
  - Embeddings: First trick that boosted the performance of NNs in NLP ← Lecture 9 (NLP 1)
    - Word2vec: Single layer NN. CBOW and skip-gram ← Lecture 10 (NLP 2)
    - Co-occurrence matrices: Basic counts and SVD improvement ← Lecture 10 (NLP 2)
    - Glove: Combining word2vec and co-occurrence matrices idea ← Lecture 10 (NLP 2)
    - Evaluating performance of embeddings ← Lecture 10 (NLP 2)
- **Named Entity Recognition (NER):** How to find words of specific meaning within text
  - Multilayer NNs: Margin loss. Forward- and back-propagation ← Lecture 11 (NLP 3)
  - Better loss functions: margin loss, regularisation ← Lecture 11 (NLP 3)
  - Better initializations: uniform, xavier ← Lecture 11 (NLP 3)
  - Better optimizers: Adagrad, RMSprop, Adam... ← Lecture 11 (NLP 3)

# Course structure

- **Language modelling:** How to represent the meaning of full pieces of text
  - Old technology: N-grams ← Lecture 12 (NLP 4)
  - Recursive NNs language models (RNNs) ← Lecture 12 (NLP 4)
  - Evaluating performance of language models ← Lecture 12 (NLP 4)
  - Vanishing gradients: Problem. Gradient clipping ← Lecture 13 (NLP 5)
  - Improved RNNs: LSTM, GRU ← Lecture 13 (NLP 5)
- **Machine translation:** How to translate text
  - Old technology: Georgetown–IBM experiment and ALPAC report ← Lecture 16 (NLP 6)
  - Seq2seq: Greedy decoding, encoder-decoder, beam search ← Lecture 16 (NLP 6)
  - Attention: Simple attention, transformers, reformers ← Lecture 16 (NLP 6)
  - Evaluating performance: BLEU ← Lecture 16 (NLP 6)

# Literature

➢ Papers =

- “N-gram language models”, Jurafsky et al., 2018. https://web.stanford.edu/ jurafsky/slp3/3.pdf
- “The unreasonable effectiveness of recurrent neural networks”, Karpathy, 2015. http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- Sections 10.1 and 10.2 of “Deep Learning”, Goodfellow et al., 2016. http://www.deeplearningbook.org/contents/rnn.html