Language modelling

Deep Natural Language Processing, 2022

Paweł Budzianowski

Discovering building blocks

1. Meaning
2. Word Vectors
3. New NLP task - language modelling
4. New family of ML models - recurrent neural networks

Language modelling

Language modelling is the task of predicting what word comes next.

Siedzimy właśnie w domu otwierając _________

Language modelling

Language modelling is the task of predicting what word comes next.

Siedzimy właśnie w domu otwierając _______

More formally: given a sequence of words $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$ compute the probability distribution of the next word:

$$P(\mathbf{x}_{t+1} | \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t)$$

where *x* can be any word in the vocabulary.

A system that does this a **language model.**

Language modelling

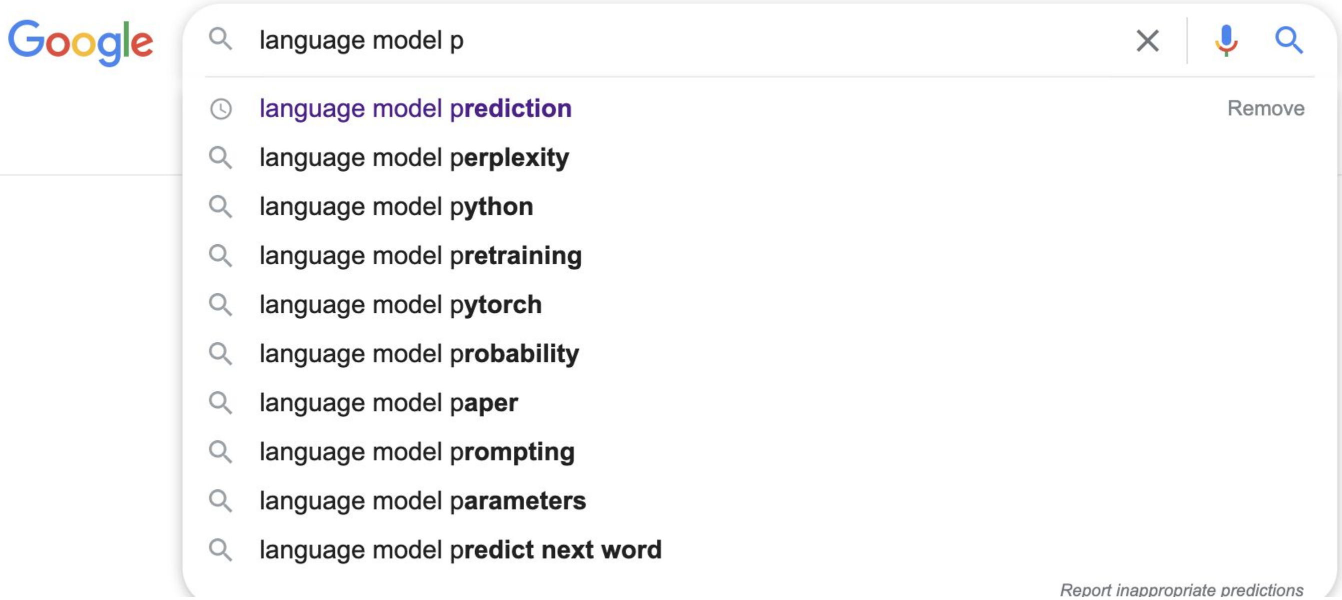
You can also think of a Language model as a system that assigns probability to a piece of text $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$.

For example if we have some text then the probability of this model is:

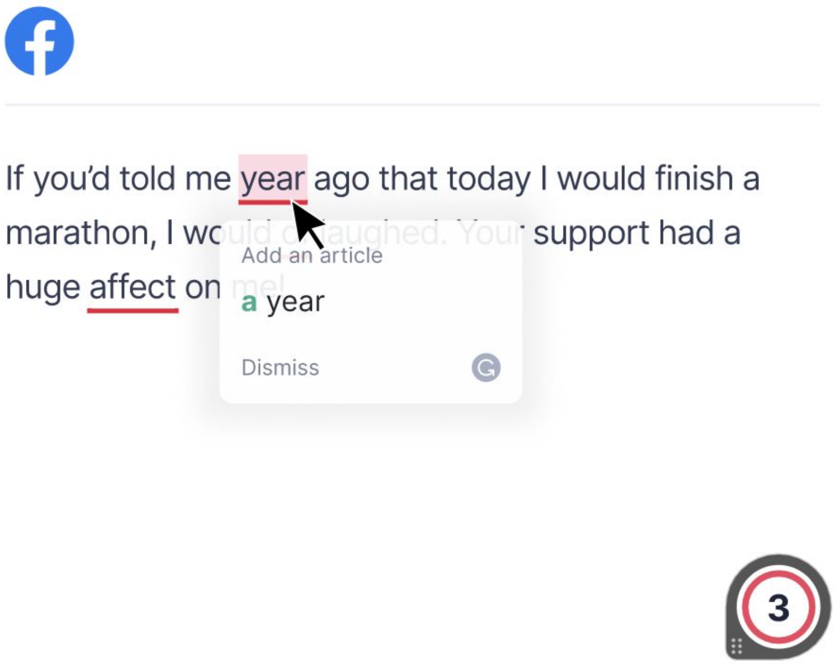
$$P(\mathbf{x}_{t+1}|\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t) = P(\mathbf{x}_1) \times P(\mathbf{x}_2|\mathbf{x}_1) \times \ldots \times P(\mathbf{x}_t|\mathbf{x}_1, \ldots, \mathbf{x}_{t-1})$$

$$= \prod_{t=1}^{T} P(\mathbf{x}_t|\mathbf{x}_1, \ldots, \mathbf{x}_{t-1})$$

LM are everywhere



LM are everywhere



N-gram Language Models

Question: How to learn a language model?

Answer: learn a *n*-gram language model

Definition: A ***n*-gram** is a chunk of *n* consecutive words:

unigrams: Siedzimy, właśnie, w, domu, otwierając

bigrams: Siedzimy właśnie, właśnie w, w domu, domu otwierając

trigrams: Siedzimy właśnie w, właśnie w domu, w domu otwierając

N-gram Language Models

Question: How to learn a language model?

Answer: learn a *n*-gram language model

Definition: A ***n*-gram** is a chunk of *n* consecutive words:

unigrams: Siedziemy, właśnie, w, domu, otwierając

bigrams: Siedziemy właśnie, właśnie w, w domu, domu otwierając

trigrams: Siedziemy właśnie w, właśnie w domu, w domu otwierając

Idea: Collect statistics about how frequent different *n*-grams are, and use these to predict next word.

N-gram language models

First we make a simplifying assumption: *x(t+1)* depends only on the preceding *n-1* words:

$$P(\mathbf{x}_{t+1}|\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t) = P(\mathbf{x}_{t+1}|\overbrace{\mathbf{x}_{t-n+2}, \ldots, \mathbf{x}_t}^{\text{(n-1 words)}})$$

$$= \frac{P(\mathbf{x}_{t-n+2}, \ldots, \mathbf{x}_t, \mathbf{x}_{t+1})}{P(\mathbf{x}_{t-n+2}, \ldots, \mathbf{x}_t)}$$

Question: How do we get these *n*-gram and *n*-gram probabilities?

N-gram language models

Question: How do we get these n-gram and n-gram probabilities?

Answer: By **counting** them in some large corpus of text.

$$\approx \frac{\text{count}(\mathbf{x}_{t-n+2}, \ldots, \mathbf{x}_t, \mathbf{x}_{t+1})}{\text{count}(\mathbf{x}_{t-n+2}, \ldots, \mathbf{x}_t)}$$

N-gram language models: example

Suppose we are learning a 4-gram language model.

Ostrożnie przesuwając odnóża wylazł z wykrotu, przepełzł przez zmurszały pień, trzema susami pokonał wiatrołom, jak duch przemknął przez ________

N-gram language models: example

Suppose we are learning a 4-gram language model.

Ostrożnie przesuwając odnóża wylazł z wykrotu, przepełzł przez zmurszały pień, trzema susami pokonał wiatrołom, jak duch przemknął przez _______

N-gram language models: example

Suppose we are learning a 4-gram language model.

Ostrożnie przesuwając odnóża wylazł z wykrotu, przepełzł przez zmurszały pień, trzema susami pokonał wiatrołom, jak duch przemknął przez ________

$$\approx \frac{\text{count}(\mathbf{x}_{t-n+2}, \ldots, \mathbf{x}_t, \mathbf{x}_{t+1})}{\text{count}(\mathbf{x}_{t-n+2}, \ldots, \mathbf{x}_t)}$$

Suppose that in the data we have observed:

duch przemknął przez - 1000 times

duch przemknął przez pole - 400 times

duch przemknął przez kościół - 200 times

Problems?

Problems?

1. Sparsity: (smoothing)
    - some context never appear in the data?
    - the probability will be zero - it's not impossible but might be wrong
    - lets add delta to every count to each probabilities - smoothing
    - but then you add all other n-grams
2. What happens if the denominator is zero? (back-off)
    - we have never seen any context
    - back of to condition on the last *n-1* words

Note: Increasing *n* makes sparsity problems worse. Typically we can't have *n* bigger than 5.

Storage problems with n-gram LM

We need to store count for all *n*-grams you saw in the corpus.

$$\approx \frac{\text{count}(\mathbf{x}_{t-n+2}, \ldots, \mathbf{x}_t, \mathbf{x}_{t+1})}{\text{count}(\mathbf{x}_{t-n+2}, \ldots, \mathbf{x}_t)}$$

Increasing *n* or increasing corpus increases model size.

N-gram LM in practice?

You can build a simple trigram LM over 1 million word corpus in a few seconds on your laptop.

Generating text with a *n*-gram LM

You can also use a LM to generate text.

Ostrożnie przesuwając odnóża wylazł z wykrotu, przepełzł przez zmurszały pień, trzema susami pokonał wiatrołom, jak duch przemknął przez ______

Generating text with a n-gram LM

You can also use a LM to generate text.

*Ostrożnie przesuwając odnóża wylazł z wykrotu, przepełzł przez zmurszały pień, trzema susami pokonał wiatrołom, jak duch przemknął przez **rozprysk** ________*

Generating text with a n-gram LM

You can also use a LM to generate text.

Ostrożnie przesuwając odnóża wylazł z wykrotu, przepełzł przez zmurszały pień, trzema susami pokonał wiatrołom, jak duch przemknął przez ***rozprysk ziemi*** ______

Sparsity problem - not much smoothness of probability distribution.

Generating text with a n-gram LM

You can also use a LM to generate text.

*Ostrożnie przesuwając odnóża wylazł z wykrotu, przepełzł przez zmurszały pień, trzema susami pokonał wiatrołom, jak duch przemknął przez **rozprysk ziemi ______***

Sparsity problem - not much smoothness of probability distribution.

But surprisingly grammatical!

But incoherent.

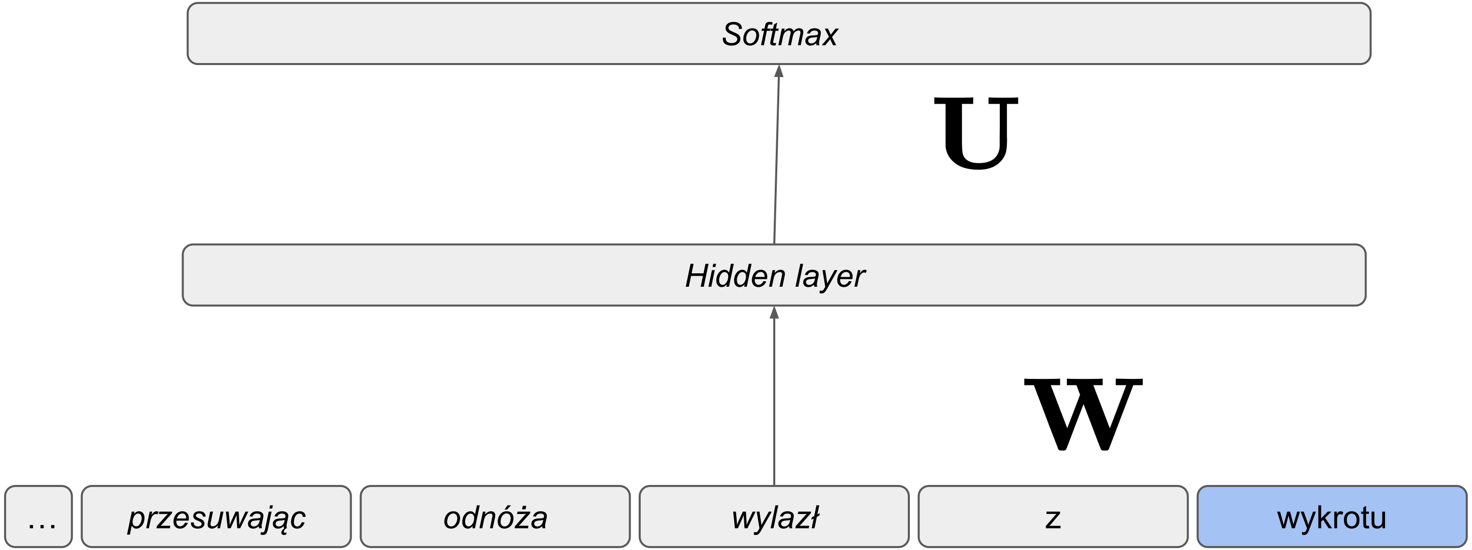
Let's go neural!

Input: $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$

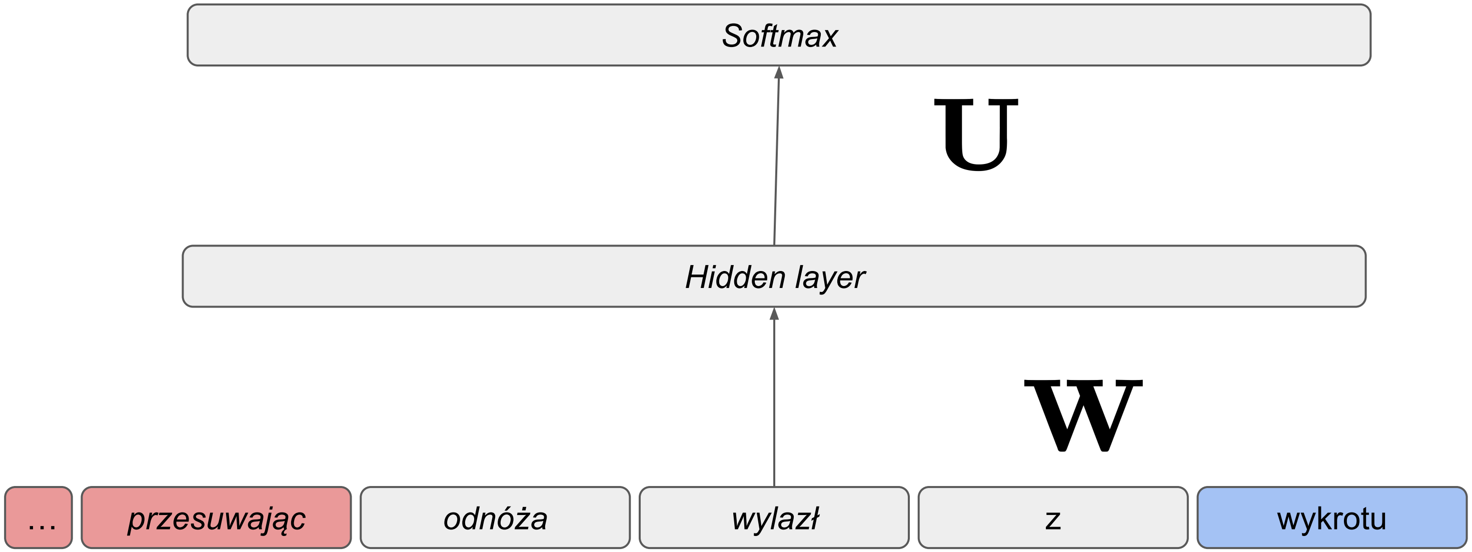
Output: $P(\mathbf{x}_{t+1} | \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t)$

How about a window-based neural model?

A fixed-window Neural LM



A fixed-window Neural LM



A fixed-window Neural LM

Improvements over *n*-gram LM

- no sparsity problem (no problems of zeros)
- don’t need to store all observed *n*-grams - just store word vectors

Remaining problems:

- fixed window is too **small**
- enlarging window enlarges ***W***
- window can never be large enough
- we are not learning things efficiently cause ***W*** learns transformation for specific word place

A fixed-window Neural LM

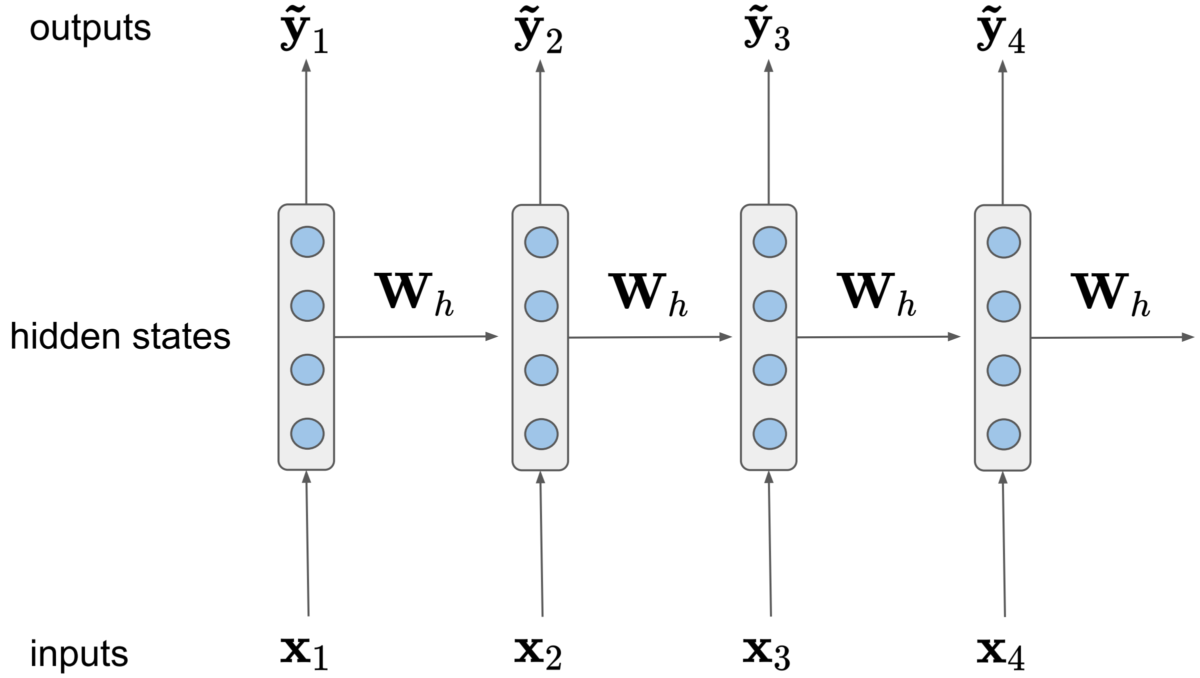
Improvements over *n*-gram LM

- no sparsity problem (no problems of zeros)
- don’t need to store all observed *n*-grams - just store word vectors

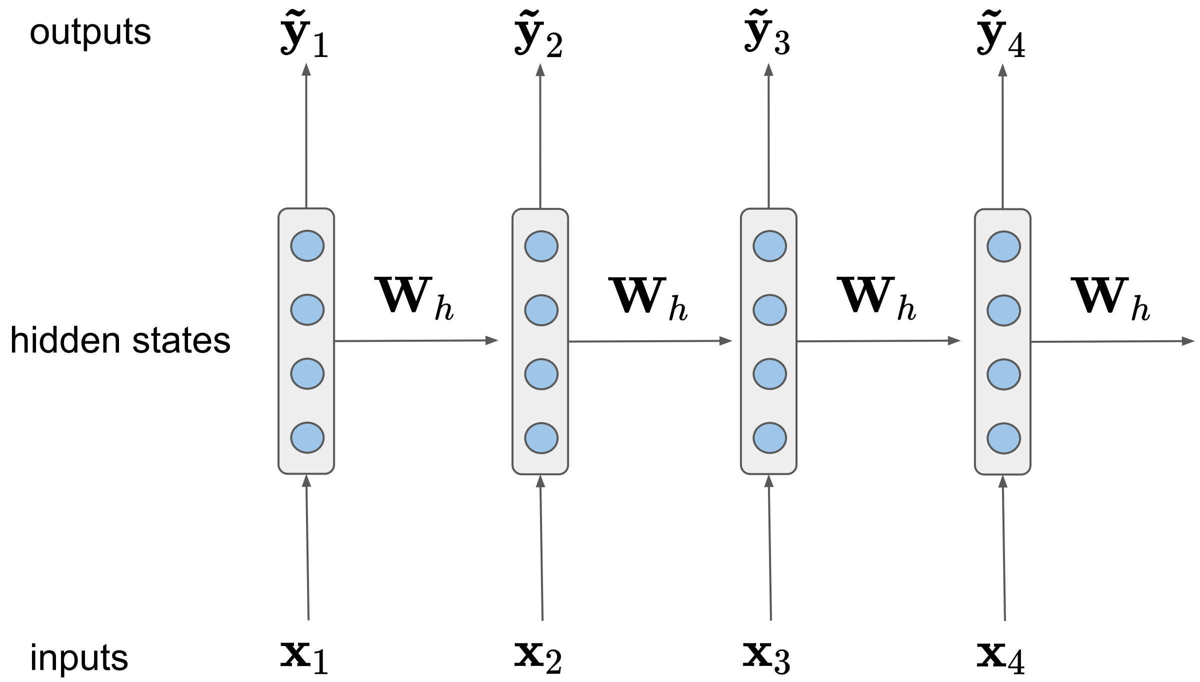
Remaining problems:

- fixed window is too **small**
- enlarging window enlarges ***W***
- window can never be large enough
- we are not learning things efficiently cause ***W*** learns transformation for specific word place - one section of **W** is in charge of specific word in the context. It’s inefficient.

Recurrent Neural Networks (RNN)

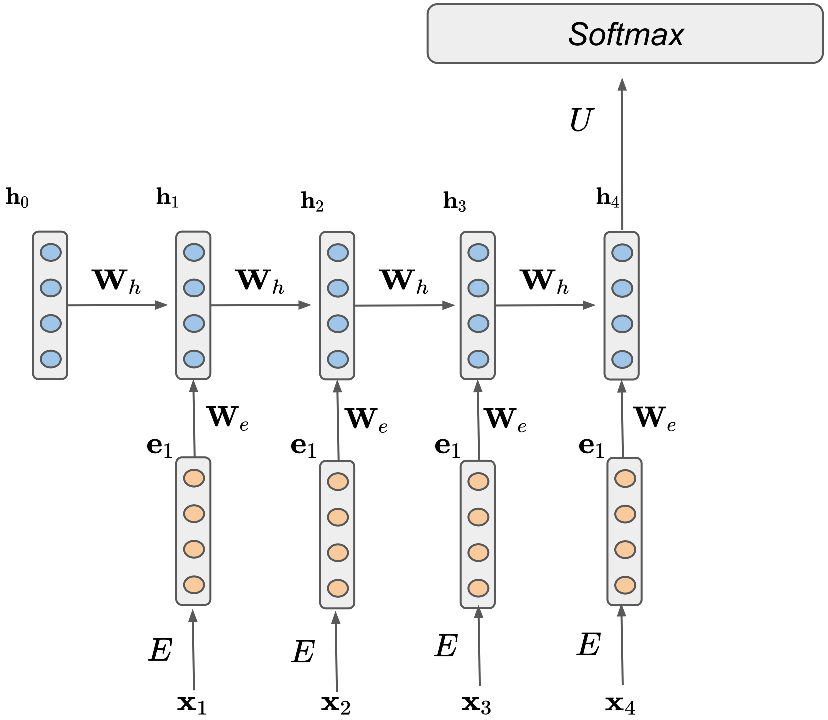


Recurrent Neural Networks (RNN)



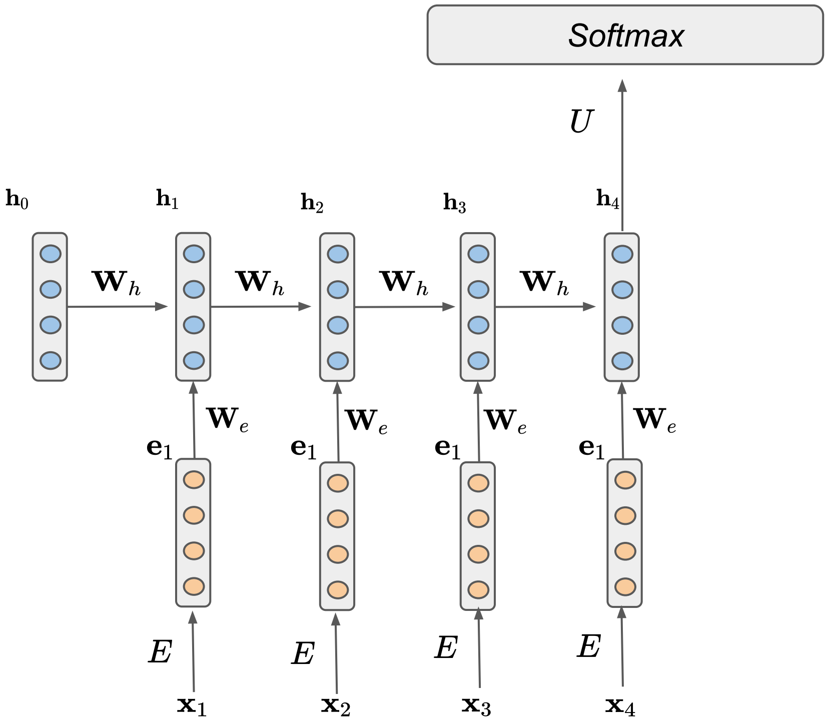
$$\mathbf{W}_h$$

RNN for language modelling



odnóża wylazł z wykrotu

RNN for language modelling



odnóża wylazł z wykrotu

$$\mathbf{y}_t = \mathrm{softmax}(\mathbf{U}\mathbf{h}_t + b_2)$$

$$\mathbf{h}_t = \sigma(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_e \mathbf{e}_t + b_1)$$

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t$$

RNN for LM

RNN advantages:

- can process **any length** input
- computation for step *t* can use information from **many steps back**
- model size **doesn’t increase** for longer input
- same weights applied on every timestep so there is symmetry in how inputs are processed

RNN disadvantages:

- recurrent computation is **slow**
- in practice, **difficult to access** information from many steps back

Training a RNN LM

- Get a big **corpus** of text which is a sequence of words $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t$
- Feed into RNN-LM, compute output distribution *y_t* **for every step *t***
  - i.e. predict probability distribution of every word, given words so far

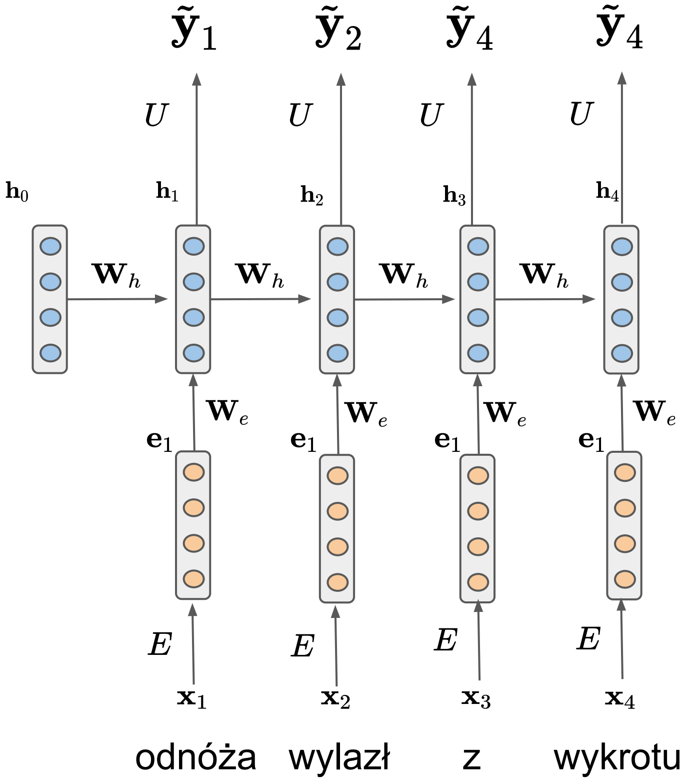
- **Loss function** on step *t* is cross-entropy between predicted probability and true next word

$$\mathbf{J}_t(\theta) = CE(y_t, \tilde{y}_t) = -\sum_{w \in V} y_t^w \log \tilde{y}_t^w = -\log \tilde{y}_t^w$$

- Average this to get **overall loss** for entire training set:

$$\mathbf{J}(\theta) = \frac{1}{T} \sum_{t=1}^{T} \mathbf{J}_t(\theta) = \frac{1}{T} \sum_{t=1}^{T} -\log \tilde{y}_t^w$$

Example - RNN for language modelling



$$\mathbf{y}_t = \mathrm{softmax}(\mathbf{U}\mathbf{h}_t + b_2)$$

$$\mathbf{h}_t = \sigma(\mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{W}_e\mathbf{e}_t + b_1)$$

$$\mathbf{e}_t = \mathbf{E}\mathbf{x}_t$$

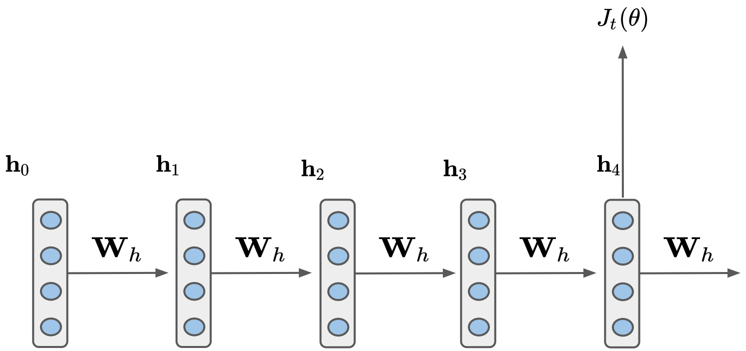
Training a RNN LM

- However, computing loss and gradients across **entire corpus** is **too expensive**:

$$\mathbf{J}(\theta) = \frac{1}{T}\sum_{t=1}^{T}\mathbf{J}_t(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\log \tilde{y}_t^{w}$$

- In practice consider as a **sentence** or a **document**
- Recall: SGD allows us to compute loss and gradients for small chunk of data and update
- Compute loss for a sentence, compute gradients and update weights and repeat

Backpropagation for RNNs



Question: what's the derivative of $J_t(\theta)$ w.r.t the repeated weight matrix $\mathbf{W}_h$?

Backpropagation Basics

Backpropagation Basics

- Chain rule - if y = f(u) and u = g(x), i.e. y = f(g(x)), then:

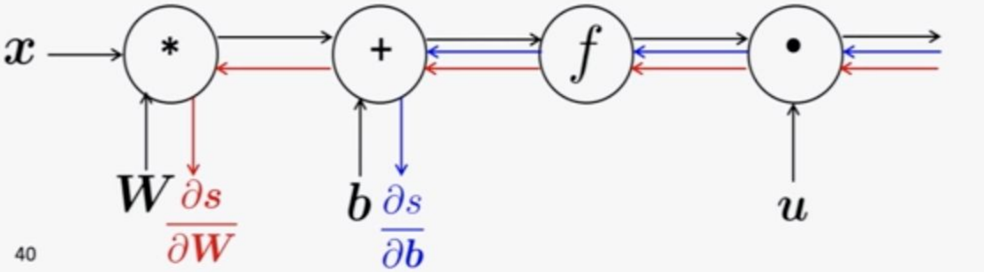
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

Backpropagation Basics [Manning, 2019]

Efficiency: compute all gradients at once

- Incorrect way of doing backprop:
  - First compute $\frac{\partial s}{\partial \boldsymbol{b}}$
  - Then independently compute $\frac{\partial s}{\partial \boldsymbol{W}}$
  - Duplicated computation!

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$

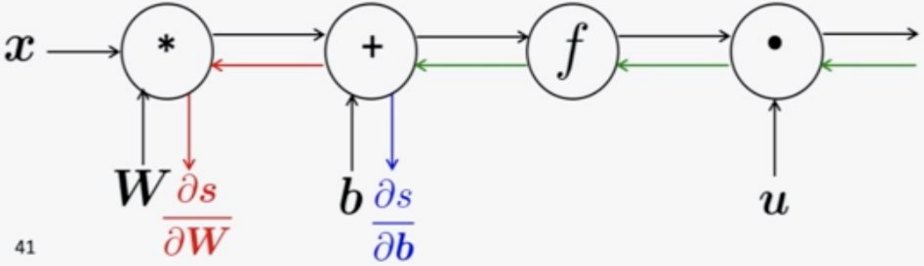


Backpropagation Basics [Manning, 2019]

Efficiency: compute all gradients at once

- Correct way:
  - Compute all the gradients at once
  - Analogous to using $\boldsymbol{\delta}$ when we computed gradients by hand

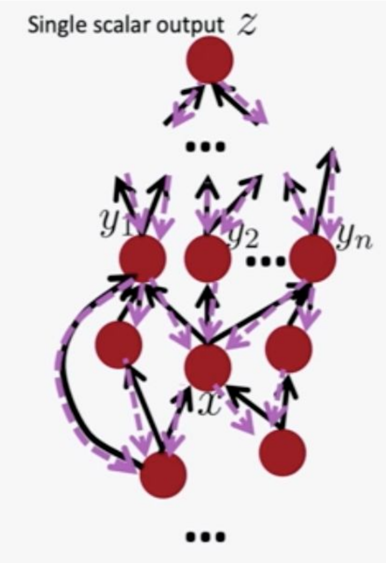
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



General Backprop

Forward propagation:

- visit nodes in topological sort order
- compute value node given predecessors



Backward propagation:

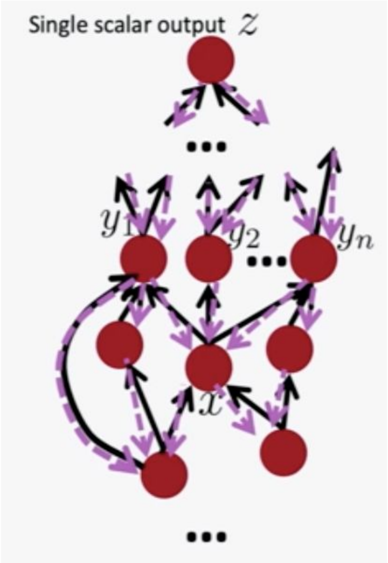
- initialize output gradient 1
- visit nodes in reversed order
- compute gradient wrt each node using gradient wrt successors

General Backprop

- Chain rule:

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

- Computationally forward pass equals backward pass!



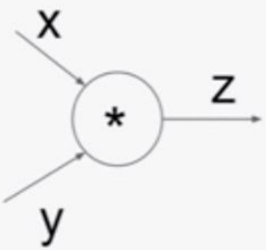
Automatic differentiation

- The gradient computation can be automatically inferred from the symbolic expression of the forward propagation
- Each node type needs to know how to compute its output and how to compute the gradient w.r.t. its inputs given the gradient wrt its output
- Modern deep learning frameworks can do backpropagation for you!

Backprop Implementations

```python
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Implementation: forward API



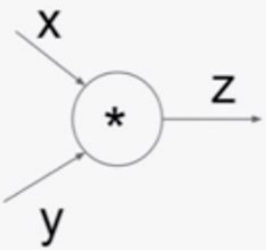
(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$$\boxed{\frac{\partial L}{\partial z}}$$

$$\boxed{\frac{\partial L}{\partial x}}$$

Implementation: backward API



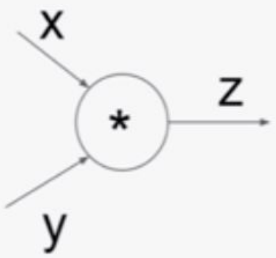
(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$\boxed{\frac{\partial L}{\partial z}}$

$\boxed{\frac{\partial L}{\partial x}}$

Implementation: backward API

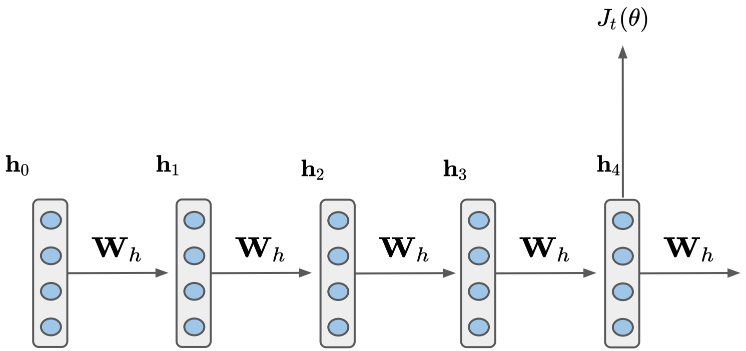


(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

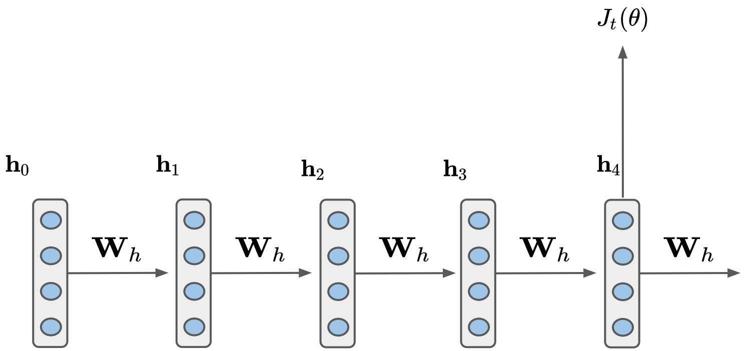
Back to RNN

Backpropagation for RNNs



Question: what's the derivative of $J_t(\theta)$ w.r.t the repeated weight matrix $\mathbf{W}_h$?

Backpropagation for RNNs



Question: what's the derivative of $J_t(\theta)$ w.r.t the repeated weight matrix $\mathbf{W}_h$?

Answer: the gradient w.r.t. a repeated weight is the sum of the gradient w.r.t. each time it appears:

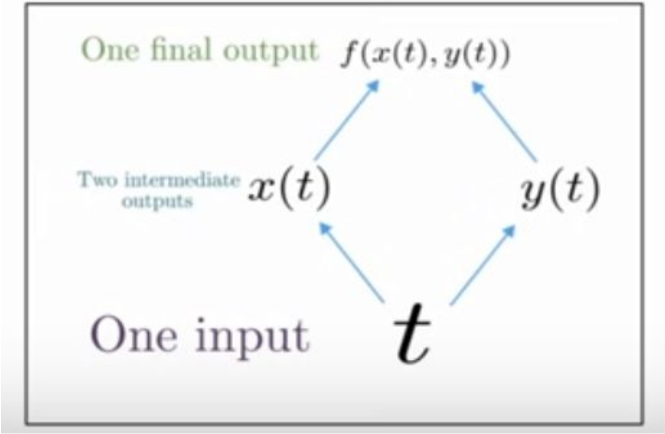
$$\frac{\partial J_t}{\partial \mathbf{W}_h} = \sum_{i=1}^{t} \frac{\partial J_t}{\partial \mathbf{W}_h}\bigg|_{i}$$

Multivariable Chain Rule

- Given a multivariable function f(x,y) and two single variable functions x and y, the chain rule is:

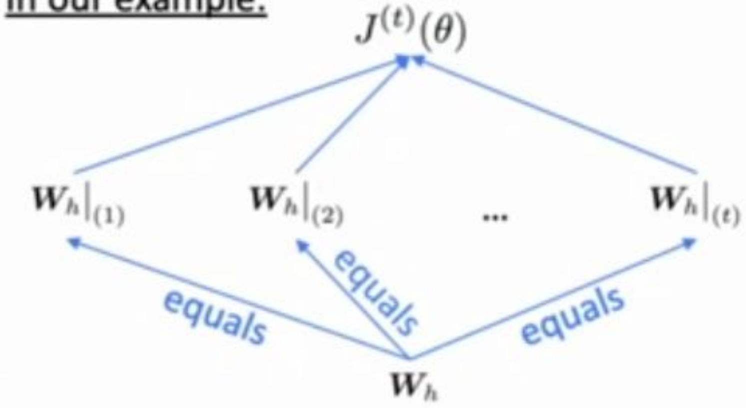
$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}$$

- In our example we have:



Backpropagation for RNNs: sketch [Manning, 2019]

In our example:



Apply the multivariable chain rule:

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\bigg|_{(i)} \overbrace{\boxed{\frac{\partial \boldsymbol{W}_h\big|_{(i)}}{\partial \boldsymbol{W}_h}}}^{=1}$$

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

Backpropagation for RNNs

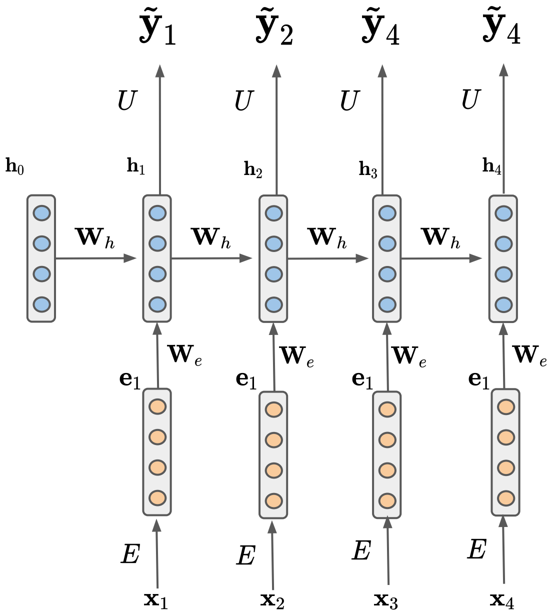
How do we calculate this in practice?

Answer: Backpropagation over timesteps $i=t,...,0$ summing gradients as you go.

This algorithm is called **backpropagation through time**.

Generating text with a RNN LM

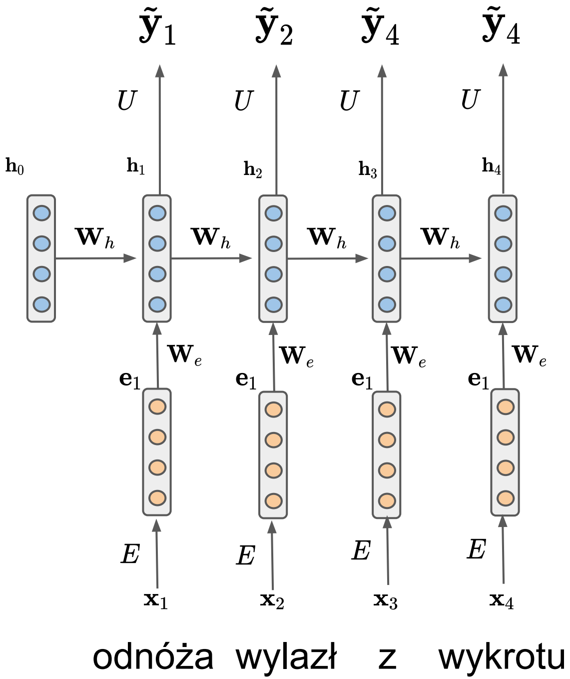
Just like a *n*-gram LM, you can use a RNN LM to **generate text** by repeated sampling. Samples output is next step's input.



odnóża

Generating text with a RNN LM

Just like a *n*-gram LM, you can use a RNN LM to **generate text** by repeated sampling. Samples output is next step’s input.



Generating text with a RNN LM

- You can train a RNN-LM on any kind of text, then generate text in that style

Najsmaczniejszy polski owoc to właśnie serek homogeniczny – i jak to się mawia, to mi się nigdy więcej nie spodobało…. A na sam koniec, to, co mi się podobało, to, że po prostu smakuje…

A

Generating text with a RNN LM

Bardzo lubię chodzić na wykłady ze sztucznej inteligencji ale nie z taką sobie miałem przyjemność, wolałem zajęcia z nią i po prostu się bałem na nią na co dzień – to że nie nauczyłem się tego nie jest niczym dziwnym, no ale trzeba mieć dystans

Codex [OpenAI, 2022]

```python
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    Examples
    solution([5, 8, 7, 1]) ==>12
    solution([3, 3, 3, 3, 3]) ==>9
    solution([30, 13, 24, 321]) ==>0
    """
    return sum(lst[i] for i in range(0,len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

Character based RNN LM

```
Recipe via Meal-Master (tm) v8.05
     Title: FLUMIL WITH PEANUT CARROT - PATTY PEANUT BUTTER STEW
Categories: Poultry, German, Casseroles
     Yield: 8 Servings
     1 pk Guennisin
     1 c  Mayonnaise
     1 lb Lean bag in microwave
     1    Onion; chopped fine
 Saute the peas in the refrigerator at least 8 hours.

 Prepare pastry and expanpie with a layer of the squares, on the salads and lightly, and set
 over a slotted serving plate to cook.

 In a large saucepan, combine the vegetables and blend well.

 From:                                                  Fine, Help-jellini
 by Market Alaskarel Cookbook_ by Inrow
```

Handwriting generation [Graves, 2013]

recurrent neural network handwriting generation demo

Type a message into the text box, and the network will try to write it out longhand (this paper explains how it works, source code is available here). Be patient, it can take a while!

Text --- up to 100 characters, lower case letters work best

Style --- either let the network choose a writing style at random or prime it with a real sequence to make it mimic that writer's style.

- take the breath away when they are
- He dismissed the idea
- prison welfare Officer complement
- She looked closely as she
- at Hunterscombe is being adapted for

The Unreasonable Effectiveness of Recurrent Neural Networks [Karpathy, 2014]

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.
```

Evaluating Language Models

Evaluating Language Models

The standard evaluation metric for Language Models is perplexity:

$$\text{perplexity} = \prod_{i=1}^{T} \left(\frac{1}{P_{LM}(x_{t+1} | x_1, \ldots, x_t)} \right)^{1/T}$$

Inverse probability of corpus according to Language Model normalized by number of words.

Evaluating Language Models

The standard evaluation metric for Language Models is perplexity:

$$\text{perplexity} = \prod_{i=1}^{T} \left(\frac{1}{P_{LM}(x_{t+1} | x_1, \ldots, x_t)} \right)^{1/T}$$

Inverse probability of corpus according to Language Model normalized by number of words.

This is equal to the exponential of the cross-entropy loss *J*:

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

Evaluating Language Models

The standard evaluation metric for Language Models is perplexity:

$$\text{perplexity} = \prod_{i=1}^{T} \left(\frac{1}{P_{LM}(x_{t+1} | x_1, \ldots, x_t)} \right)^{1/T}$$

Inverse probability of corpus according to Language Model normalized by number of words.

This is equal to the exponential of the cross-entropy loss *J*:

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

LOWER PERPLEXITY IS BETTER

Computing perplexity

Let’s analyze 2 language models.

100 words in the test set.

$$P(\text{Who are you}) = P(\text{Who | <s>})P(\text{are | <s> Who})P(\text{you | <s> Who are })$$

Computing perplexity

Let’s analyze 2 language models.

100 words in the test set.

$$P(\text{Who are you}) = P(\text{Who} \mid \text{<s>})P(\text{are} \mid \text{<s> Who})P(\text{you} \mid \text{<s> Who are })$$

$$P(W) = 0.9 => PP(W) = 0.9^{-1/100} = 1.00105416$$

Computing perplexity

Let’s analyze 2 language models.

100 words in the test set.

$$P(\text{Who are you}) = P(\text{Who} \mid \text{<s>})P(\text{are} \mid \text{<s> Who})P(\text{you} \mid \text{<s> Who are })$$

$$P(W) = 0.9 => PP(W) = 0.9^{-1/100} = 1.00105416$$

$$P(W) = 10^{-250} => PP(W) = (10^{-250})^{-1/100} \approx 316$$

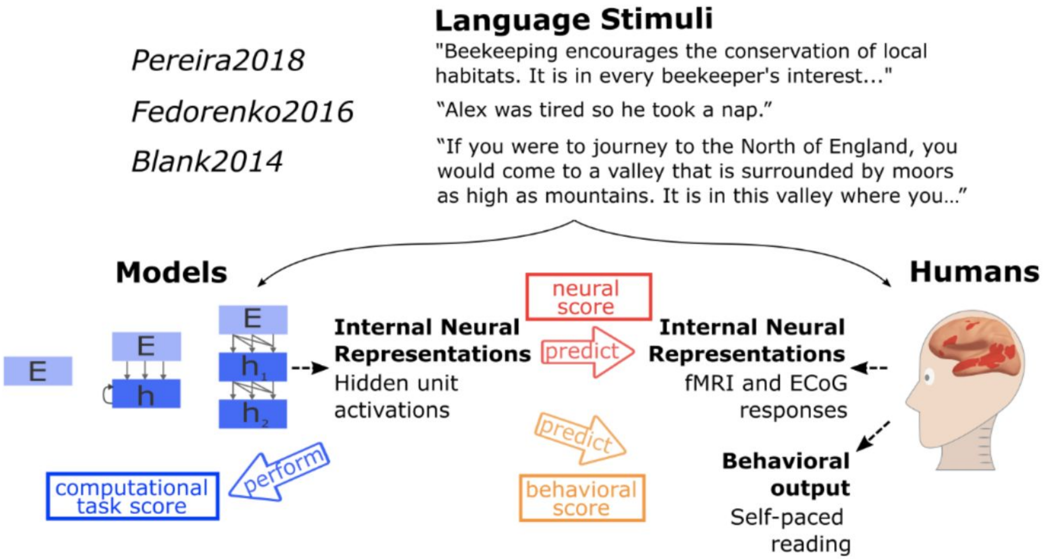
RNNs have greatly improved perplexity [Grave et al. 2016]

Model	Test perplexity
Interpolated Kneser-Ney 5-gram (Chelba et al., 2013)	67.6
Feedforward NN + D-Softmax (Chen et al., 2015)	91.2
4-layer IRNN-512 (Le et al., 2015)	69.4
RNN-2048 + BlackOut sampling (Ji et al., 2015)	68.3
Sparse Non-negative Matrix Language Model (Shazeer et al., 2015)	52.9
RNN-1024 + MaxEnt 9-gram (Chelba et al., 2013)	51.3
LSTM-2048-512 (Jozefowicz et al., 2016)	43.7
2-layer LSTM-8192-1024 + CNN inputs (Jozefowicz et al., 2016)	30.0
Ours (LSTM-2048)	43.9
Ours (2-layer LSTM-2048)	39.8

Table 2. One Billion Word benchmark. Perplexity on the test set for single models. Our result is obtained after 5 epochs.

Why should we care about Language Modelling?

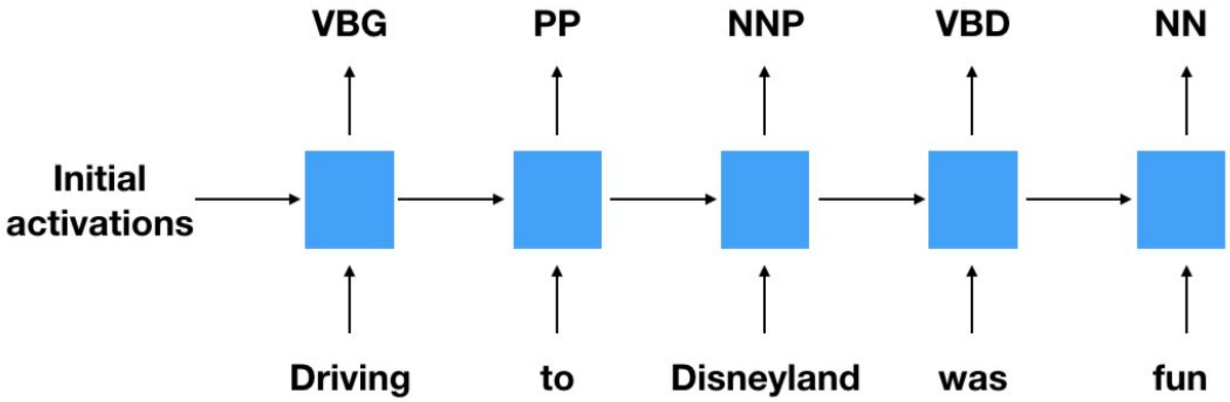
Language processing is fundamentally predictive



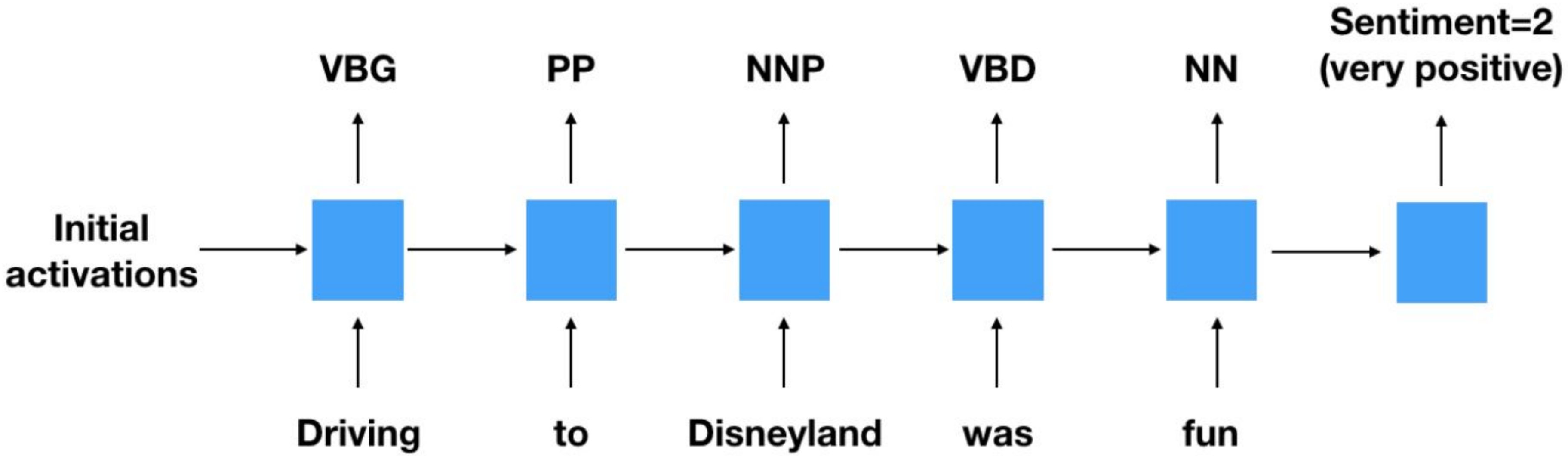
Why should we care about Language Modelling?

- Language Modelling is a benchmark task that helps us measure our progress on understanding language.
- Language modelling is a subcomponent of many NLP tasks, especially those involving generating text or estimating the probability of the text
    - predictive typing
    - speech recognition
    - handwriting recognition
    - summarization
    - dialogue
    - spelling correction

LM can be used for tagging



RNNs can be used for sentence classification



RNNs can be used for sentence classification

- How to compute sentence encoding?
- Use the final hidden state

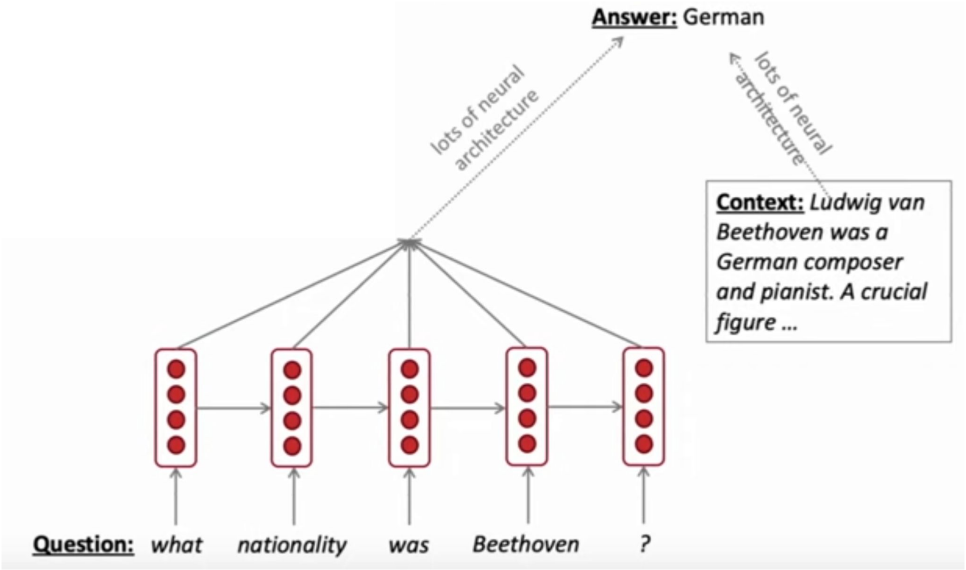
- But the whole pooling works better - taking an average of all vectors or max.

RNNs can be used as an encoder module

- Question-answering, machine translation and many other tasks!

RNNs can be used as an encoder module [Manning, 2019]

- Question-answering, machine translation and many other tasks!
- RNN acts as an encoder for the question
- The encoder is part of a larger neural system



RNN-LMs can be used to generate test

- Speech recognition
- Machine translation
- Summarization

- These are example of **conditional generation**

Literature

1. Grave et al., 2016 - https://arxiv.org/pdf/1609.04309.pdf
2. Graves, 2013 - https://www.cs.toronto.edu/~graves/handwriting.cgi
3. Karpathy, 2013 - http://karpathy.github.io/2015/05/21/rnn-effectiveness/