Multi-tasking and character models for morphology prediction

Carlos Ramisch (carlos.ramisch@univ-amu.fr)

With the help of Benoit Favre & Alexis Nasr

Prédiction Structurée pour le Traitement Automatique des Langues
Master IAAA
Aix Marseille Université

Outline

Morphological analysis

Multi-tasking

Sub-lexical models

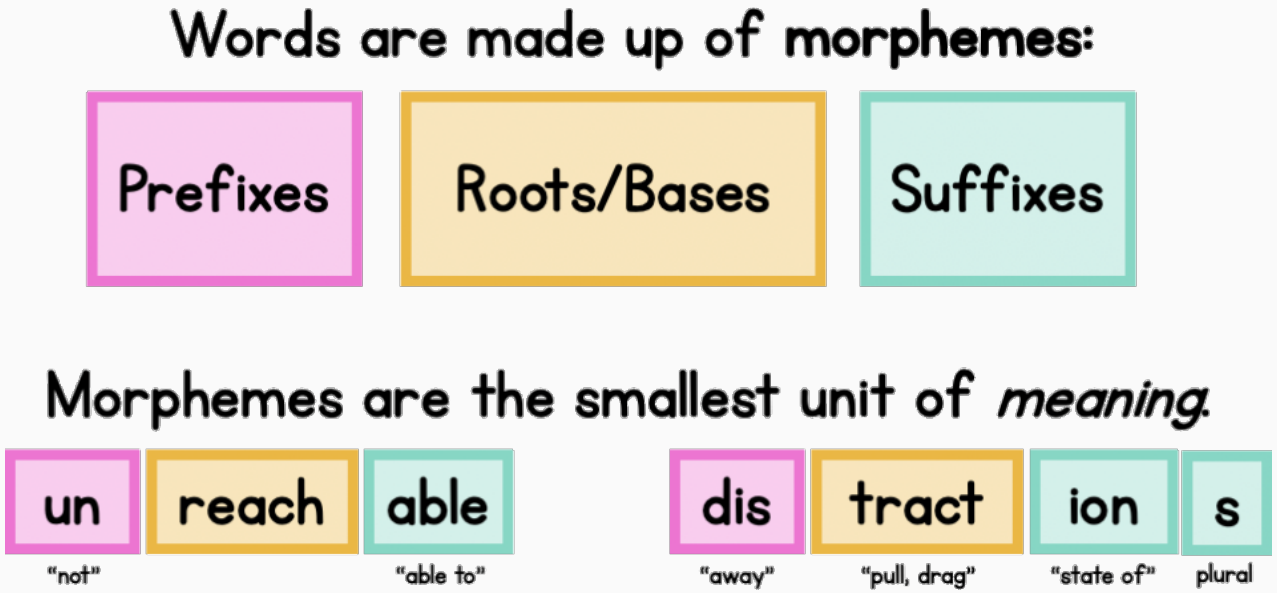
Convolution for text

What is morphology?

The study of words

- What is a word in a language?
- What are the different word classes?
- What is the internal structure of a word?
- Through which processes do we form new words?

Words and morphemes



- Free morphemes can appear as standalone words
  - → *reach*, *she*, *crocodile*, *the*...
- Bound morphemes only appear within other words
  - → *un-*, *dis-*, *-able*, *-s*, *rupt*...

Source: `https://sarahsnippets.com/understanding-morphology-part-1/`

Types of morphemes

- Root/base: core meaning of the word
  - → *read* in *unreadable*, *do* in *redoing*, . . .
- Affixes
  - Prefix: bound morpheme placed before the root
    - → re-, un-, mis-, over-. . .
  - Suffix: bound morpheme placed after the root
    - → -s, -ed, -ing, -est -ly, -ment. . .

Source: https://sarahsnippets.com/understanding-morphology-part-1/

Lemmas

- **Lemma**: Word seen as the canonical form of an (inflected) word
  - → *jouons* → *jouer*, *représentations* → *représentation*, *maison* → *maison*
  - → *elles* → *il*, *des* → *un*, *lucidité* → *lucidité*, *auxquelles* → *à* + *le* + *quel*
- Also defined as the least marked form of a word
- Correspond to the entries in a lexicon/dictionary
- Depend on the linguistic conventions of each language
  - → E.g. Arabic verbs: 3rd person singular masculin past

Root/base

Morpheme giving the core meaning of a word (may be a word or not)

→ *jouons* → *jou*, *représentations* → *présent*, *des* → *de*, *lucidité* → *lucid*

Word formation processes

- Inflection: resulting word has **same POS and lemma** as original
  - → *school* (NOUN) + *-s* = *schools* (NOUN)
  - → *présent-er* (VERB) + *-é* + *-es*= *présentées* (VERB)
- Derivation: resulting word has **new POS** and/or **new lemma**
  - → *settle* (VERB) + *-ment* + = *settlement* (NOUN)
  - → *re-* +*présent-er* (VERB) + *-atif* + *-e*= *représentative* (ADJ)
- Compounding: combine two or more roots
  - → *data* (NOUN) + *set* (NOUN) = *dataset* (NOUN)

Word formation processes

- Inflection: resulting word has **same POS and lemma** as original
  - → *school* (NOUN) + *-s* = *schools* (NOUN)
  - → *présent-er* (VERB) + *-é* + *-es*= *présentées* (VERB)
- Derivation: resulting word has **new POS** and/or **new lemma**
  - → *settle* (VERB) + *-ment* + = *settlement* (NOUN)
  - → *re-* +*présent-er* (VERB) + *-atif* + *-e*= *représentative* (ADJ)
- Compounding: combine two or more roots
  - → *data* (NOUN) + *set* (NOUN) = *dataset* (NOUN)

We will focus on inflection in this course.

Exercise: identify the morphemes

- Decompose these words into morphemes
- For each bound morpheme:
  - → What is its role in the word?
  - → Is it added by an inflectional or derivational process?

Word	Morphemes	Roles	Process
suppléantes			
déremboursement			
reparlerons			

Exercise: identify the morphemes

- Decompose these words into morphemes
- For each bound morpheme:
  - → What is its role in the word?
  - → Is it added by an inflectional or derivational process?

Word	Morphemes	Roles	Process
suppléantes	suppléant -e -s	féminin, pluriel	inflection
déremboursement	dé- re- bours -ment	non, retour, nom	dérivation
reparlerons	re- parl -er -ons	encore, futur, nous	dériv. + infl.

Note: the absence of some morphemes may also be relevant

→ *déremboursement* is a masculine singular noun

Morphological analysis

1. Identify words and morphemes
2. Assign them a meaning or function
   - → Bound morphemes modify roots in regular ways (e.g. *-s* = plural)
   - → Function can be described by morphological features

Morphological analysis

1. Identify words and morphemes
2. Assign them a meaning or function
   - $\rightarrow$ Bound morphemes modify roots in regular ways (e.g. *-s* = plural)
   - $\rightarrow$ Function can be described by morphological features

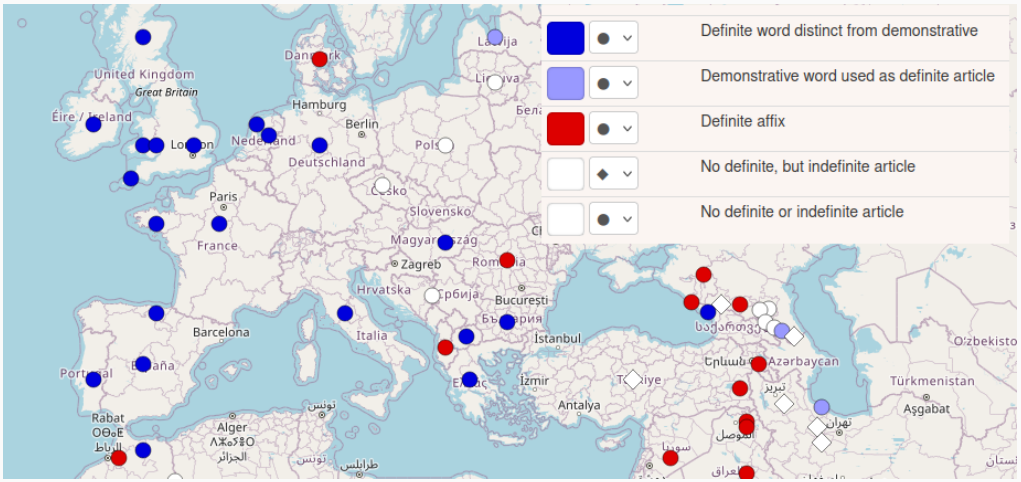
We will focus on predicting function (features) in this course

Morphological features

- Represent the roles of inflectional bound morphemes
- Relevant features depend on the language and POS
- For instance, for French:
  - → `NOUN`, `ADJ`: Number (`Sing`/`Plur`), Gender (`Masc`/`Fem`)
  - → `VERB`: Number (`Sing`/`Plur`), Person (`1`/`2`/`3`), Tense (`Past`/`Pres`/`Fut`)
- But for English:
  - → `NOUN`: Number (`Sing`/`Plur`)
  - → `ADJ`: Degree (`Pos`/`Cmp`/`Sup`)
  - → `VERB`: Number (`Sing`/`Plur`), Person (`1`/`2`/`3`), Tense (`Past`/`Pres`)

Linguistic typology

- Characterise and compare different languages of the world
- Languages represented as set of `key=value` pairs
  - → Idea: communication functions expressed using different strategies
  - → Example: definiteness in French (*un·e* vs. *le/la*) vs. Arabic (*al-* prefix)
- Includes (but not limited to) morphological features



Source: `https://wals.info/`

Morphologically rich languages

- Analytic: no or little inflection, few forms per lemma
  - → Chinese: yī tiān 'one day', sān tiān 'three day'
  - → Examples: English, Vietnamese, Thai, Yoruba
- Fusional: some inflection, some forms per lemma
  - → German nouns: `Sing/Plur`, `Masc/Fem/Neu`, `Acc/Dat/Nom/Gen`
  - → Examples: French, Polish, Arabic, Greek
- Agglutinative languages: many affixes, many forms per lemma
  - → Turkish: *mashin+ha+shun+ra* (car+s+their+at)
  - → Examples: Hungarian, Finnish, Japanese
- Polysynthetic languages: sentence-words, several roots per "word"
  - → Yupik: *tuntussuqatarniksaitengqiggtuq*
  - → Examples: Quechua, Guaraní, Inuktitut

Source: Wikipedia

Morphology-related NLP tasks

- Word segmentation ($\approx$ tokenisation)
- POS tagging
- Lemmatisation
- Morphological feature prediction

Goal: mitigate or neutralize variability (e.g. for information extraction)

Word segmentation is easy #not

Spanish contractions:

Vámonos	al	mar	.	Vamos	nos	a	el	mar	.
VERB?	X	NOUN	PUNCT	VERB	PRON	ADP	DET	NOUN	PUNCT

Chinese uses no space at all:

現在	我們	在	瓦倫西亞	。
Xiànzài	wǒmen	zài	Wǎlúnxīyǎ	.
Now	we	in	Valencia	.
ADV	PRON	ADP	PROPN	PUNCT

Vietnamese uses spaces within words:

Tất cả	đường	bêtông	nội đồng	là	thành quả	...
All	road	concrete	country	is	achievement	...
PRON	NOUN	NOUN	NOUN	AUX	NOUN	PUNCT

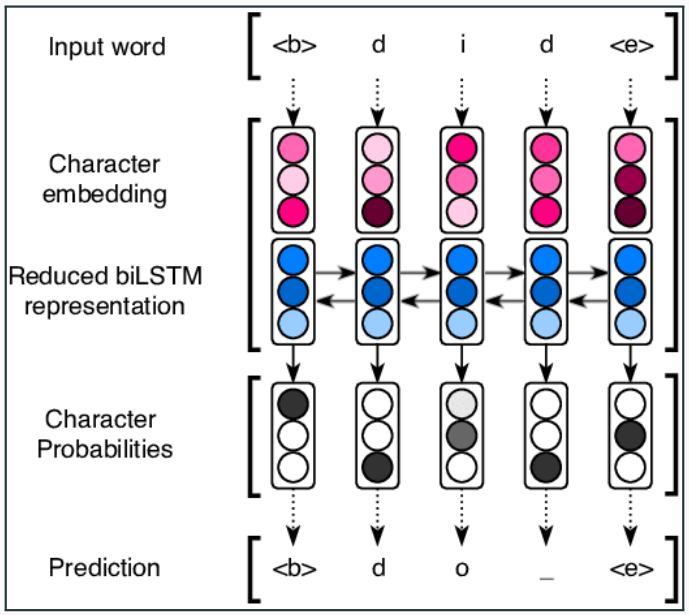
Source: Dan Zeman's Unidive webinar slides

POS tagging

- POS tags: high-level morpho-syntactic classes
- Historical task in NLP, single tag per word
- Moprhological features: fine-grained POS tags

Lemmatization

- Baseline: look up ⟨form,POS⟩ in dictionary
  - → Ambiguity: ⟨`suis`,`VERB`⟩ → *être* OR *suivre*
- Predict (suffix) transformation rules as tags
  - → *mangeons*: `ons -> r`, *parlons*: `ons -> er`
- Character-based sequence model (RNN or transformer)



Source: Adapted from Rybak & Wróblewska (2018)

Morphological features in UD

- Each word → set of Key=Value pairs (column 6: `FEATS`)
- Each key is unique in the set, and has a single value (in French)
- CoNLL-U: `Definite=Def|Number=Sing|PronType=Art`
  - → Equal sign = separetes key from value
  - → Both keys and values use `TitleCase`
  - → Pipe sign | separate Key=Value pairs
  - → If no morphological feature, use underscore _
- By convention, Key=Value pairs are ordered alphabetically
- In conllu library, python `dict` with Key=Value pairs

UD features: tagset

Languages uses subsets of keys and values

Lexical	Inflectional ("Nominal")	Inflectional ("Verbal")
PronType	Gender	VerbForm
NumType	Animacy	Mood
Poss	NounClass	Tense
Reflect	Number	Aspect
Foreign	Case	Voice
Abbr	Definite	Evident
Typo	Degree	Polarity
		Person
		Polite
		Clusivity

Source: Dan Zeman's Unidive webinar slides

POS vs. morphological features

Why can't we encode all morphological tags into the POS?

Example: `POS=DET_Def_Sing_Art`

Or consider POS as just another morphological feature?

Example: `FEATS="POS=DET|Definite=Def|Number=Sing|PronType=Art"`

POS vs. morphological features

Why can't we encode all morphological tags into the POS?
Or consider POS as just another morphological feature?

- The distinction is a bit arbitrary
  - → Penn Treebank has 2 noun POS: `NN` (singular) and `NNS` (plural)
  - → For verbs, it distinguishes 6 POS for English inflected forms
  - → The `PROPN` vs. `NOUN` distinction could be a morph. feature
- Cross-lingual compatibility
  - → Verbs and nouns exist across all languages, but inflection changes
  - → Proper and common nouns have different distributions (e.g. wrt `DET`)
  - → POS → lemma, morph. features → inflected forms

Exercise: morphological features in Sequoia

Write a python script using the `conllu` library to:

1. Count the number of different feature keys (column 6: `FEATS`)
2. List the possible values for each key
3. Count the average number of features per word
4. Count the number of featureless words
5. (Optional) Check some examples on Grew-match

   → `https://universal.grew.fr/?corpus=UD_French-Sequoia@2.14`

Exercise: morphological features in Sequoia

Write a python script using the `conllu` library to:

1. Count the number of different feature keys (column 6: `FEATS`)
   → Number of feature keys: 14
2. List the possible values for each key
   → Next slide
3. Count the average number of features per word
   → Average features per word: 1.63
4. Count the number of featureless words
   → Words with no feature: 24875/67955 (36.61%)
5. (Optional) Check some examples on Grew-match
   → `https://universal.grew.fr/?corpus=UD_French-Sequoia@2.14`

Morphological features in Sequoia: keys and values

```
Number of feature keys: 14
  Gender: ['Fem', 'Masc']
  Number: ['Sing', 'Plur']
  PronType: ['Dem', 'Art', 'Rel', 'Int', 'Prs']
  Person: ['1', '3', '2']
  Mood: ['Ind', 'Cnd', 'Sub', 'Imp']
  Tense: ['Pres', 'Imp', 'Past', 'Fut']
  VerbForm: ['Fin', 'Part', 'Inf']
  Definite: ['Def', 'Ind']
  NumType: ['Ord', 'Card']
  Voice: ['Pass']
  Poss: ['Yes']
  Polarity: ['Neg']
  Reflex: ['Yes']
  Foreign: ['Yes']
```

Morphological features in Sequoia per POS

	DET	NOUN	PRON	VERB	ADJ	PROPN	NUM	AUX	ADV	X
Gender	5656	13856	754	2130	2895	1382		10		
Number	10007	14004	1598	4056	4053	1521		1713		
PronType	9359		714						37	
Person			1761	2014				1703		
Mood				2020				1703		
Tense				4499				2013		
VerbForm				5640				2223		
Definite	8919									
NumType		117	55		204		1598			
Voice				735						
Poss	512									
Polarity									648	
Reflex			326							
Foreign										179

Exercise: morphological features in French

Tag the morphological features of the sentence below:

La	*gare*	*routière*	*attend*	*toujours*	*ses*	*illuminations*
DET	NOUN	ADJ	VERB	ADV	DET	NOUN

Exercise: morphological features in French

Tag the morphological features of the sentence below:

La	*gare*	*routière*	*attend*	*toujours*	*ses*	*illuminations*
DET	NOUN	ADJ	VERB	ADV	DET	NOUN
Definite=Def Gender=Fem Number=Sing PronType=Art	Gender=Fem Number=Sing	Gender=Fem Number=Sing	Mood=Ind Number=Sing Person=3 Tense=Pres VerbForm=Fin	_	Number=Plur Poss=Yes	Gender=Fem Number=Plur

Evaluating morphological analysis

- Accuracy for the whole "supertag"
  - → Strict: single error makes whole word false
- Accuracy for each feature key
  - → Some keys are rare ⟹ class imbalance!
- Precision, recall and F-score of each key
  - → Precision: proportion of correct among predicted
  - → Recall: proportion of correct among gold
  - → Micro- and macro-average
  - → Ignore frequent (easy) featureless words

Exercise: morphological analysis evaluation

Calculate:

1. Overall accuracy for concatenation of both tags
2. Precision and recall per key
3. Micro-and macro-averaged precision and recall

	La DET	*gare* NOUN	*routière* ADJ	*attend* VERB	*toujours* ADV	*ses* DET	*illuminations* NOUN
Gold	Number=Sing Gender=Fem	Number=Sing Gender=Fem	Number=Sing Gender=Fem	Number=Sing	_	Number=Plur	Number=Plur Gender=Fem
Pred	Number=Sing	Number=Sing Gender=Masc	Number=Sing Gender=Fem	Number=Sing	Number=Plur Gender=Fem	Number=Plur Gender=Masc	Gender=Fem

Exercise: morphological analysis evaluation

Calculate:

1. Overall accuracy for concatenation of both tags
2. Precision and recall per key
3. Micro-and macro-averaged precision and recall

	La DET	*gare* NOUN	*routière* ADJ	*attend* VERB	*toujours* ADV	*ses* DET	*illuminations* NOUN
Gold	Number=Sing Gender=Fem	Number=Sing Gender=Fem	Number=Sing Gender=Fem	Number=Sing	_	Number=Plur	Number=Plur Gender=Fem
Pred	Number=Sing	Number=Sing Gender=Masc	Number=Sing Gender=Fem	Number=Sing	Number=Plur Gender=Fem	Number=Plur Gender=Masc	Gender=Fem

	✓	Pred	Gold	Precision	Recall
Both	2	7		Accuracy=2 / 7 ≈ 28.6%	
Gender	2	5	4	2/5≈40%	2/4≈50%
Number	5	6	6	5/6≈83.3%	5/6≈83.3%
Micro-avg	7	11	10	7/11≈63.6%	7/10≈70%
Macro-avg				$\frac{2/5+5/6}{2}$ ≈61.7%	$\frac{2/4+5/6}{2}$ ≈66.7%

Evaluation script

```
./accuracy.py --tagcolumn feats --pred <test-pred>.conllu
              --gold <test>.conllu --train <train>.conllu
```

```
Accuracy on all feats: 87.22 ( 8371/ 9598)
Accuracy on OOV feats: 57.02 (  520/  912)

Precision, recall, and F-score for feats:
Definite   : P= 98.23 ( 1222/ 1244) / R= 98.47 ( 1222/ 1241) / F= 98.35
Foreign    : P=  0.00 (    0/   33) / R=  0.00 (    0/    0) / F=  0.00
Gender     : P= 92.69 ( 3464/ 3737) / R= 91.81 ( 3464/ 3773) / F= 92.25
Mood       : P= 79.09 (  416/  526) / R= 92.04 (  416/  452) / F= 85.07
NumType    : P= 79.70 (  212/  266) / R= 94.64 (  212/  224) / F= 86.53
Number     : P= 95.81 ( 4991/ 5209) / R= 96.43 ( 4991/ 5176) / F= 96.12
Person     : P= 83.16 (  657/  790) / R= 97.91 (  657/  671) / F= 89.94
Polarity   : P= 95.45 (   84/   88) / R= 95.45 (   84/   88) / F= 95.45
Poss       : P= 44.19 (   38/   86) / R=100.00 (   38/   38) / F= 61.29
PronType   : P= 94.84 ( 1342/ 1415) / R= 97.81 ( 1342/ 1372) / F= 96.30
Reflex     : P= 86.27 (   44/   51) / R=100.00 (   44/   44) / F= 92.63
Tense      : P= 76.35 (  681/  892) / R= 91.04 (  681/  748) / F= 83.05
VerbForm   : P= 81.99 (  892/ 1088) / R= 97.81 (  892/  912) / F= 89.20
Voice      : P= 35.11 (   33/   94) / R= 82.50 (   33/   40) / F= 49.25

micro-avg  : P= 90.70 (14076/15519) / R= 95.24 (14076/14779) / F= 92.92
macro-avg  : P= 74.49               / R= 88.28               / F= 80.80
```

Outline

Morphological analysis

Multi-tasking

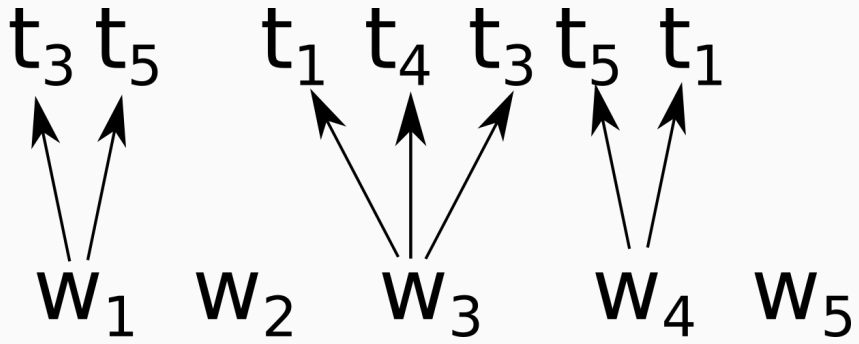
Sub-lexical models

Convolution for text

Multi-tag sequence tagging

Task definition

- **Input** (x): a sequence of n inputs (words) w_1 to w_n
- **Output** (y): zero, one, or several category tags t_i per word w_i



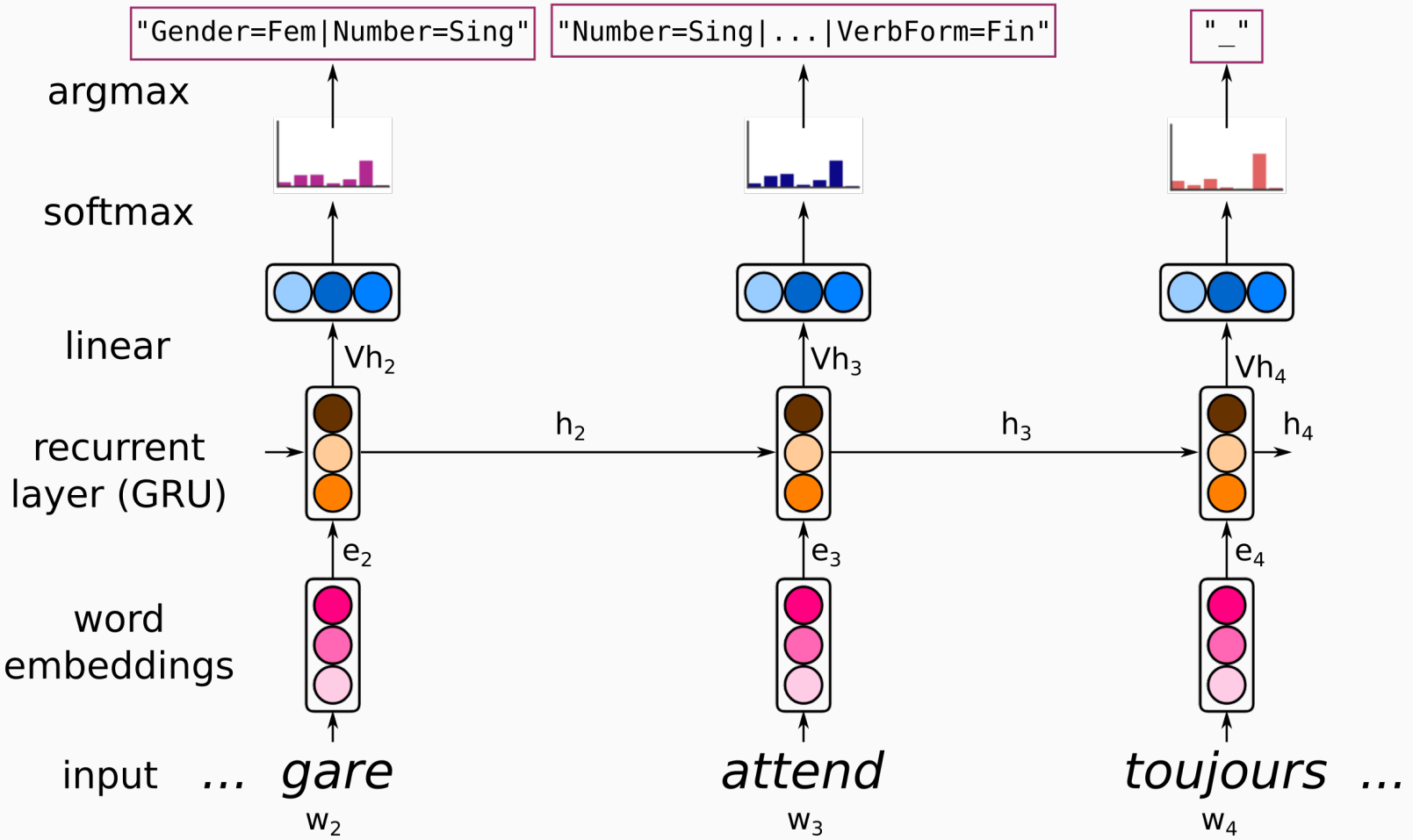
Multi-tag sequence tagger

1. Single supertag tagging
2. Separate independent taggers
3. Joint model with multi-task learning

Supertag tagging

- Idea: concatenate features as a sigle morphological "supertag"
  - → No internal structure, tag as string `"Key1=Value1|Key2=Value2..."`
- Careful! order tags deterministically
  - → We do not want `Gender=Fem|Number=Plur` ≠ `Number=Plur|Gender=Fem`
- Adapt existing sequence tagging model, e.g. RNN POS tagger (TP1)

Supertag tagging with RNN



Supertag tagging: pros and cons

Pros

- Simple model, easy to adapt from POS tagger
- Effective for morphology-poor languages with enough data

Cons

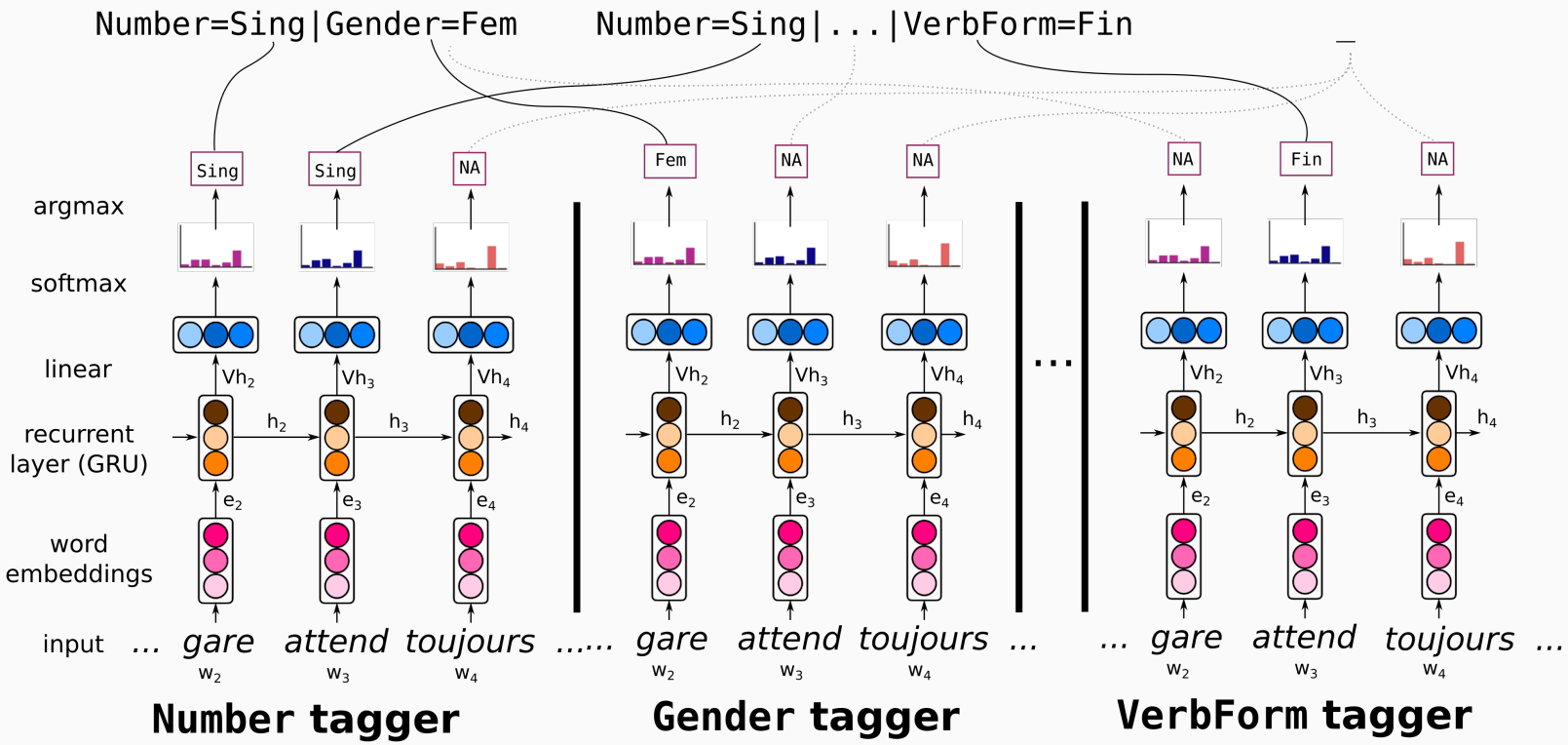
- Many different tags (e.g. 125 in Sequoia)

  $\rightarrow$ More categories $\Longrightarrow$ harder classification task
- Orthogonal outputs, ignores tag structure

Separate models

- Train N separate sequence taggers
  - → One for each of the N feature keys
  - → Example: Tagger 1 (Gender) predicts `Fem` or `Masc`
- The "Not Applicable" trick: special `NA` tag
  - → If a key is not present for word w_i
  - → Example: the `Gender` of French verbs is `NA`
- Inference: give input sentence N to each tagger, obtain N sequences
  - → Concatenate output from classifiers word-wise
- If all N classifiers predict `NA`, tag is underscore "_"

Separate RNN taggers



Separate models: pros and cons

Pros

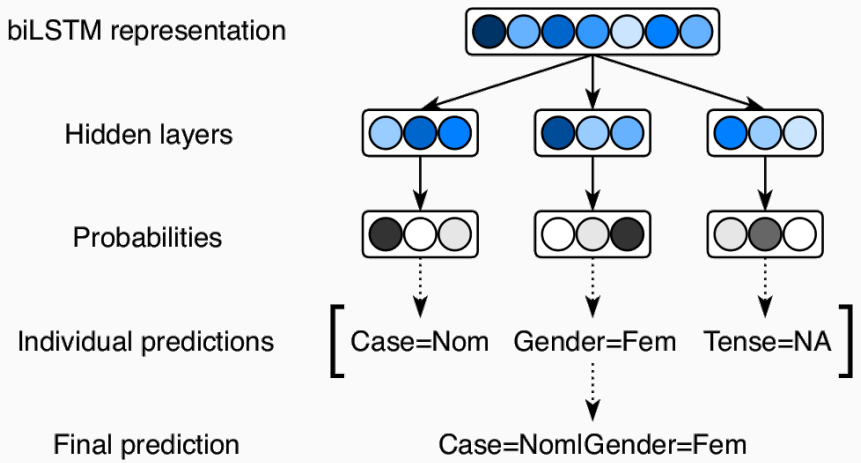
- Individual taggers are simple, easy to adapt from POS tagger
- Each tagger is specialised, must chose among few labels
  - → At most 6 (`PronType` 5 + `NA`) in Sequoia

Cons

- Training and storing N classifiers is expensive
  - → Multiply parameters by N (=14 in Sequoia)
- Overall system is quite complex and slow
  - → Final labels depend on 14 independent predictions
- No sharing among related tasks (e.g. `Tense` depends on `Mood`)

Joint multi-task model

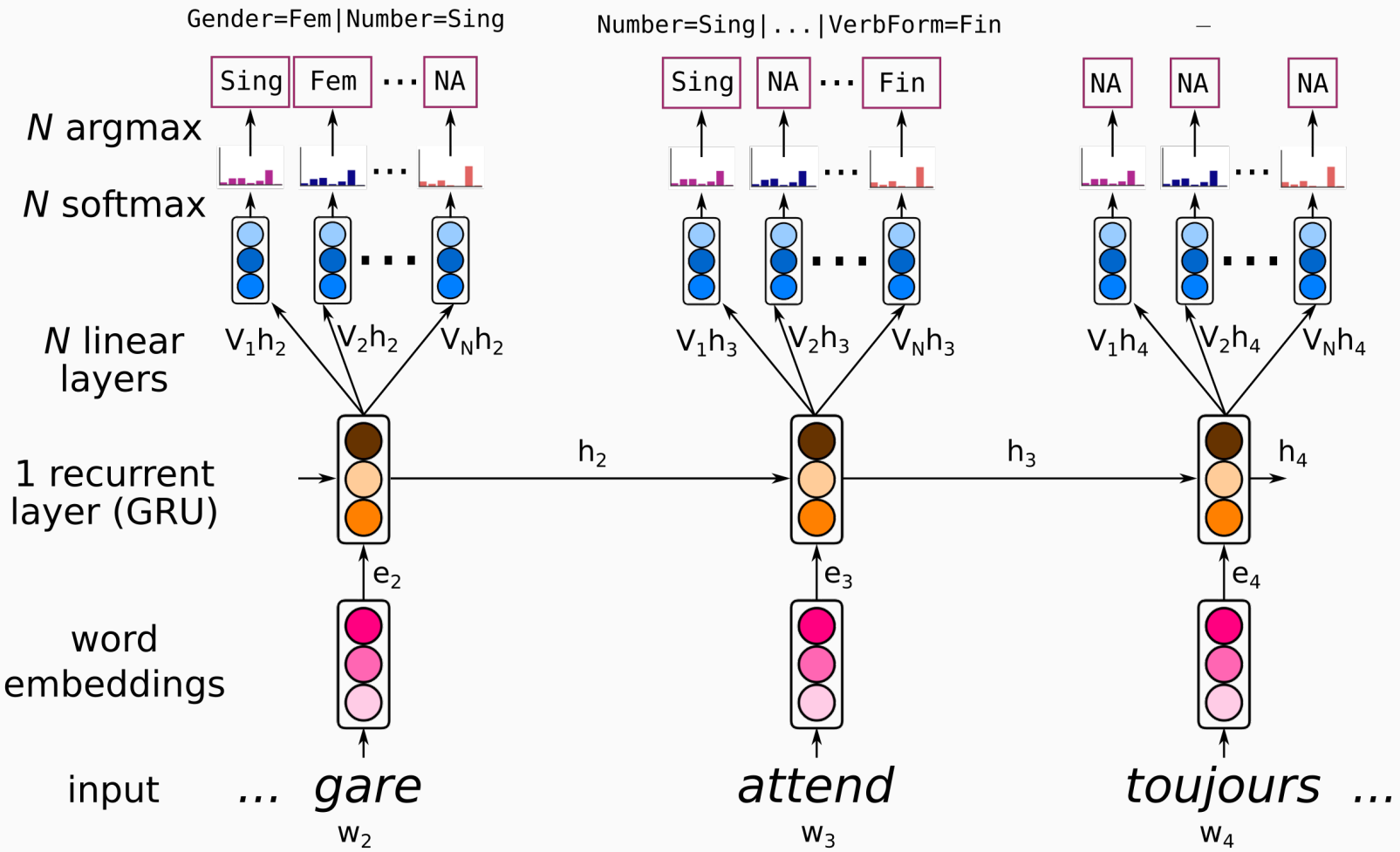
- Single model encodes (embeddings) and contextualises (RNN)
- Separate prediction heads (linear) specialise for each key
- Inspiration: Rybak & Wróblewska – `https://aclanthology.org/K18-2004/`



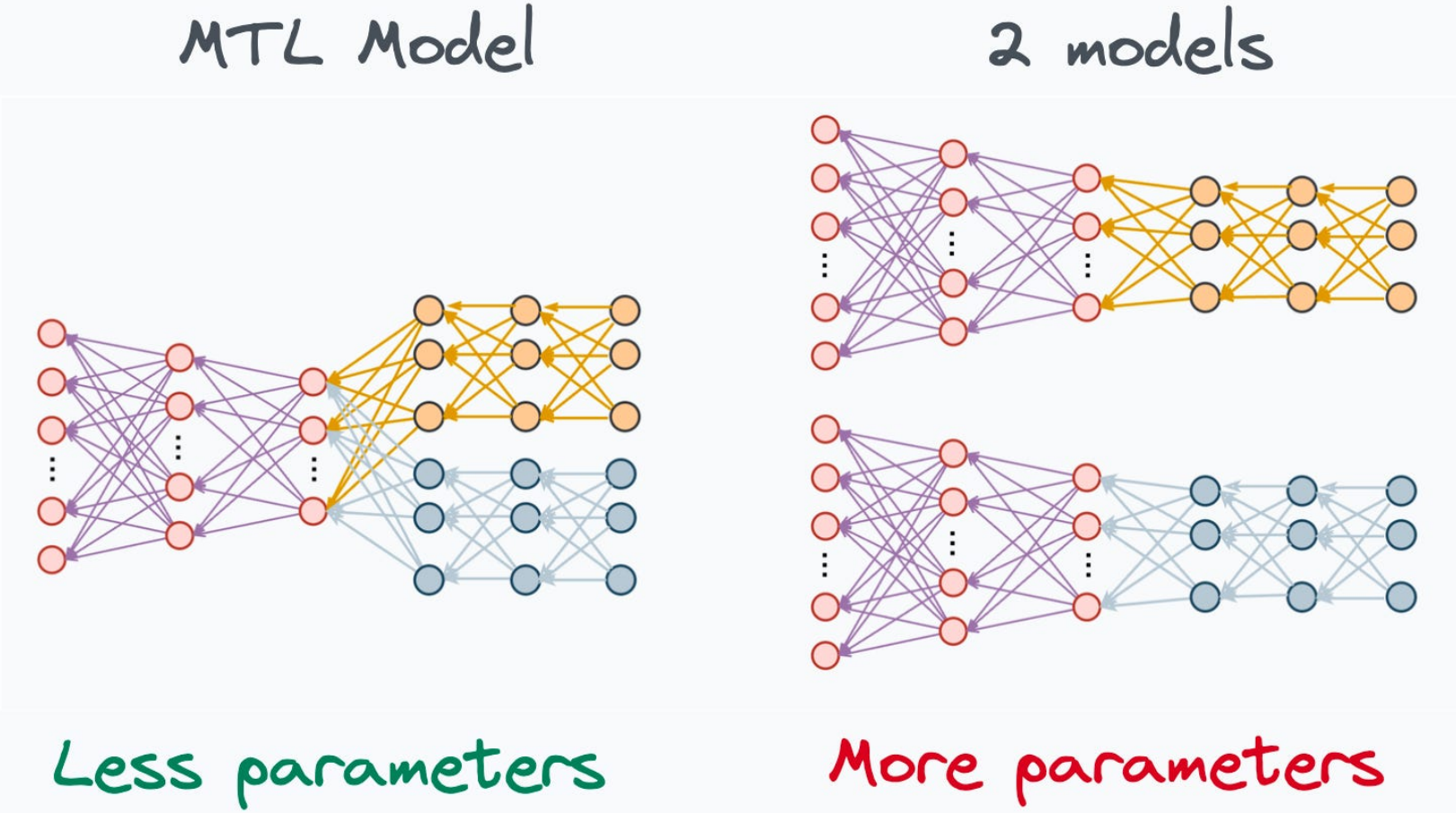
Multi-task learning (MTL)

- Each head deals with a task, so the model is multi-task
- Training: One softmax per task (feature key), N separate losses
  $\rightarrow$ combine (sum) losses from different heads
- Inference: concatenate predictions from each head
  $\rightarrow$ Also use `NA` tag, like in separate models

Multi-task morphology tagger



Multi-task learning (MTL) vs. separate models



Source: https://blog.dailydoseofds.com/p/transfer-learning-vs-fine-tuning

MTL model: pros and cons

Pros

- Parameter sharing, more compact than N separate models
- Training and inference faster than N separate models
- Sentence encoding shared among related tasks (e.g. `Tense` & `Mood`)
- Each head is specialised, must chose among few labels

  $\rightarrow$ At most 6 (`PronType` 5 + `NA`) in Sequoia

Cons

- We must generate several outputs per word: how?
- We must combine gradients from several tasks: how?

MTL model: pros and cons

Pros

- Parameter sharing, more compact than N separate models
- Training and inference faster than N separate models
- Sentence encoding shared among related tasks (e.g. `Tense` & `Mood`)
- Each head is specialised, must chose among few labels

  $\rightarrow$ At most 6 (`PronType` 5 + `NA`) in Sequoia

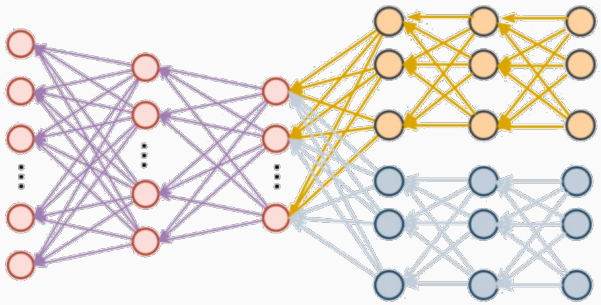
Cons

- We must generate several outputs per word: how?
- We must combine gradients from several tasks: how?

Good news: with `torch`, this is quite easy to implement!

Multiple output heads

- Module's `forward` returns a tuple of tensors
- Each output is generated by a specialised linear layer
- Same input (e.g. RNN states) can be reused several times



Source: `https://blog.dailydoseofds.com/p/a-practical-and-intuitive-guide-to`

Minimal MTL example

```
from torch import tensor, nn

class DummyMTL (nn.Module):
  def __init__(self):
    super().__init__()
    self.a = nn.Parameter(tensor([2.],requires_grad=True))
    self.b = nn.Parameter(tensor([3.],requires_grad=True))

  def forward(self):
    c = self.a + self.b # output of task 1 (sum)
    d = self.a * self.b # output of task 2 (product)
    return (c, d)       # Two "task" outputs

  def print_gradients(self):
    print([f"{n}.grad={p.grad and p.grad.item()}" \
           for (n, p) in self.named_parameters()])
```

Minimal MTL example: gradients

```
model = DummyMTL()
(c,d) = model() # calls forward
model.print_gradients()
```

Minimal MTL example: gradients

```
model = DummyMTL()
(c,d) = model() # calls forward
model.print_gradients()
# ['a.grad=None', 'b.grad=None']
c.backward()
model.print_gradients()
```

Minimal MTL example: gradients

```
model = DummyMTL()
(c,d) = model() # calls forward
model.print_gradients()
# ['a.grad=None', 'b.grad=None']
c.backward()
model.print_gradients()
# ['a.grad=1.0', 'b.grad=1.0']
d.backward()
model.print_gradients()
```

Minimal MTL example: gradients

```
model = DummyMTL()
(c,d) = model() # calls forward
model.print_gradients()
# ['a.grad=None', 'b.grad=None']
c.backward()
model.print_gradients()
# ['a.grad=1.0', 'b.grad=1.0']
d.backward()
model.print_gradients()
# ['a.grad=4.0', 'b.grad=3.0']
```

In `torch`, gradients are cumulative!

Minimal MTL example: gradients

Now imagine c and d are 2 loss functions we combine

```
 model = DummyMTL()
(c,d) = model() # calls forward
model.print_gradients()
```

Minimal MTL example: gradients

Now imagine c and d are 2 loss functions we combine

```
 model = DummyMTL()
(c,d) = model() # calls forward
model.print_gradients()
# ['a.grad=None', 'b.grad=None']
loss = c + d # combined during training
loss.backward()
model.print_gradients()
```

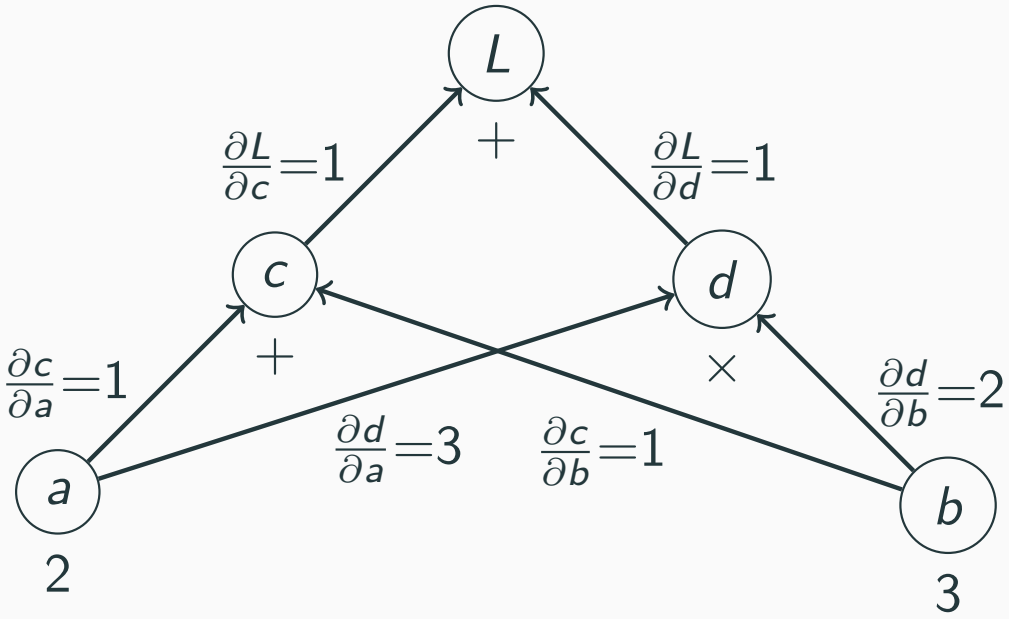
Minimal MTL example: gradients

Now imagine c and d are 2 loss functions we combine

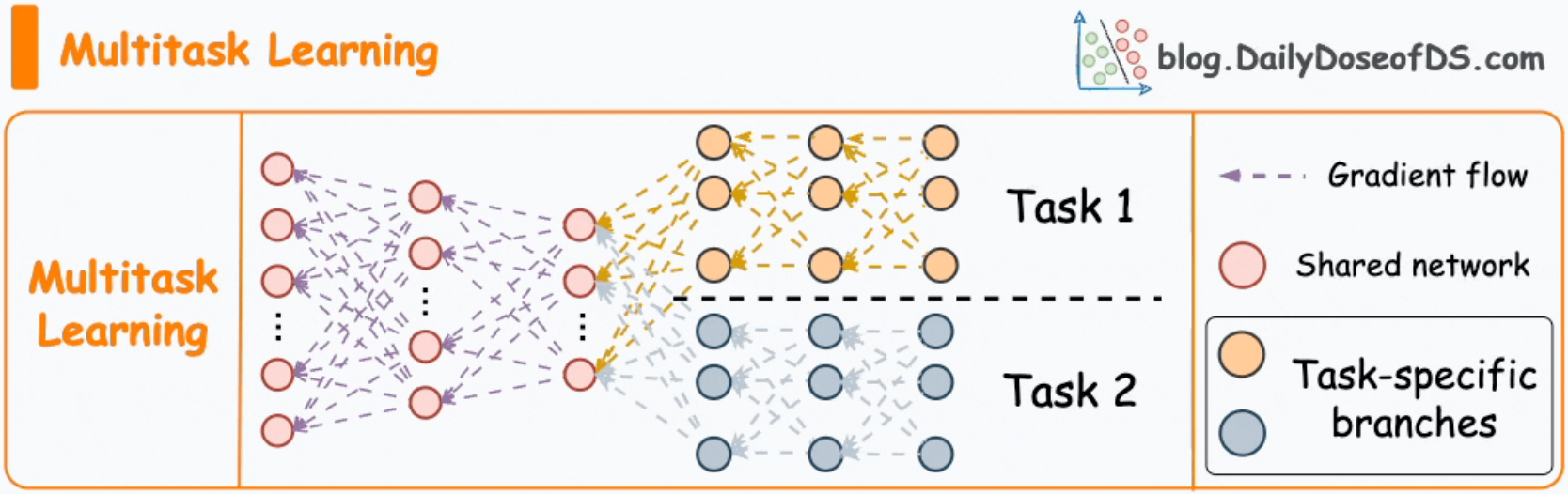
```
 model = DummyMTL()
(c,d) = model() # calls forward
model.print_gradients()
# ['a.grad=None', 'b.grad=None']
loss = c + d # combined during training
loss.backward()
model.print_gradients()
# ['a.grad=4.0', 'b.grad=3.0']
```

By summing the losses, gradients are updated wrt. both tasks!

Combined loss: calculation graph



MTL gradient flow



Source: `https://blog.dailydoseofds.com/p/transfer-learning-vs-fine-tuning`

MTL for morphological features

- Complex task structure: incremental development
  - → Start with predicting 1 feature (e.g. Number)
- Data preparation: several output vocabularies V_t
  - → $V_{\texttt{Number}}$, $V_{\texttt{Gender}}$, ..., $V_{\texttt{VerbForm}}$
  - → All V_i contain special tokens `<PAD>` and `NA`
- Model: multiple heads (linear layers) and outputs
  - → Store prediction heads in `ParameterDict`
- Learning curve: accuracy per task (misleading)
- Inference: combine outputs from each head
  - → Predict `w['feats']=None` if `NA` for all tasks

Training loop for MTL

Example:

```
for (x, y) in train_loader:     # dict y['task']=tensor
  optimizer.zero_grad()
  y_hat_tuple = model(x)        # 1 logit vector per task
  loss = torch.tensor([0.])     # initialise cumulative loss
  for (task, y_hat) in enumerate(y_hat_tuple) :
    loss += criterion(y_hat.flatten(0, 1), y[task].flatten())
  loss.backward()               # combined loss for all tasks
  optimizer.step()
```

Training loop for MTL

Example:

```
for (x, y) in train_loader:     # dict y['task']=tensor
  optimizer.zero_grad()
  y_hat_tuple = model(x)        # 1 logit vector per task
  loss = torch.tensor([0.])     # initialise cumulative loss
  for (task, y_hat) in enumerate(y_hat_tuple) :
    loss += criterion(y_hat.flatten(0, 1), y[task].flatten())
  loss.backward()               # combined loss for all tasks
  optimizer.step()
```

Why not calling `loss.backward()` for each task?

Training loop for MTL

Example:

```
for (x, y) in train_loader:     # dict y['task']=tensor
  optimizer.zero_grad()
  y_hat_tuple = model(x)        # 1 logit vector per task
  loss = torch.tensor([0.])     # initialise cumulative loss
  for (task, y_hat) in enumerate(y_hat_tuple) :
    loss += criterion(y_hat.flatten(0, 1), y[task].flatten())
  loss.backward()               # combined loss for all tasks
  optimizer.step()
```

Why not calling `loss.backward()` for each task?

→ Dynamic calculation graph is freed after `backward()`

→ Call `loss.backward(retain_graph=True)` possible, but inefficient

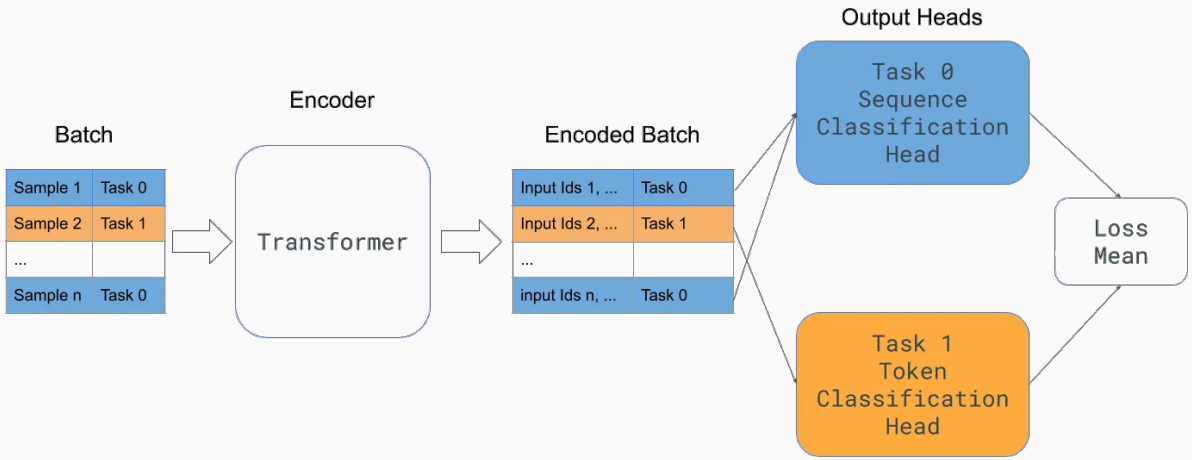
Why does MTL work?

- Single-task models can be enriched with auxiliary tasks
  - $\rightarrow$ Usually improves performance of main task
- Implicit data augmentation
- Less overfitting, implicit regularisation
- Soft introduction of inductive biases
- Focus on relevant features, better representations

Source: `https://www.ruder.io/multi-task/`

MTL with different datasets

- For morphological analysis, tasks are aligned on same data
- Can we learn from multiple supervision from different datasets?
- **Yes!** By alternating batches from both tasks!



Source: Image by Amine Elhattami

Further reading for MTL

- Rich Caruana's PhD thesis
  → `http://reports-archive.adm.cs.cmu.edu/anon/1997/CMU-CS-97-203.pdf`
  → `https://www.cs.cornell.edu/~caruana/mlj97.pdf` (MLJ paper)
- Avi Chawla's blog (overall view)
  → `https://blog.dailydoseofds.com/p/transfer-learning-vs-fine-tuning`
- Adam K's blog – (text classification tutorial in `torch`)
  → `https://medium.com/gumgum-tech/an-easy-recipe-for-multi-task-learning-in-pytorch-that-you-can-do-at-home-1e529a8df`
- Sebastian Ruder's blog
  → `https://www.ruder.io/multi-task/` (MTL principles for NLP)
  → `https://www.ruder.io/multi-task-learning-nlp/` (MTL examples in NLP)
- Amine Elhattami's blog (MTL with transformers)
  → https://towardsdatascience.com/how-to-create-and-train-a-multi-task-transformer-model-18c54a146240

Seq2seq generation

- Predict Key=Value pairs with specialised LM
- Generate sequence in auto-regressive fashion
- Special token to mark the end of generation
- No hard constraint on well-formed of output
  - $\rightarrow$ Number of Key=Value pairs, redundancy

Outline

Morphological analysis

Multi-tasking

Sub-lexical models

Convolution for text

Generalisation for morphology

- Word-based representations (embeddings) are learned on the fly
  - $\rightarrow$ Morphologically similar words $\Longrightarrow$ close embeddings
- Embedding of OOV words with similar suffixes: `<UNK>`
  - $\rightarrow$ Examples: *enregistrera*, *amélioreront*, *modernisation*, *isolée* (Sequoia)
  - $\rightarrow$ Mixes rare words from different POS/paradigms
  - $\rightarrow$ Concerns $912/9598 = 9.50\%$ of words in Sequoia
- We need representations based on units smaller than words!

Sub-lexical units in NLP

1. Character n-gram embeddings
2. Sub-lexical tokenisation (BPE, Wordpiece...)
3. Character RNNs
4. Character CNNs

Sub-lexical units in NLP

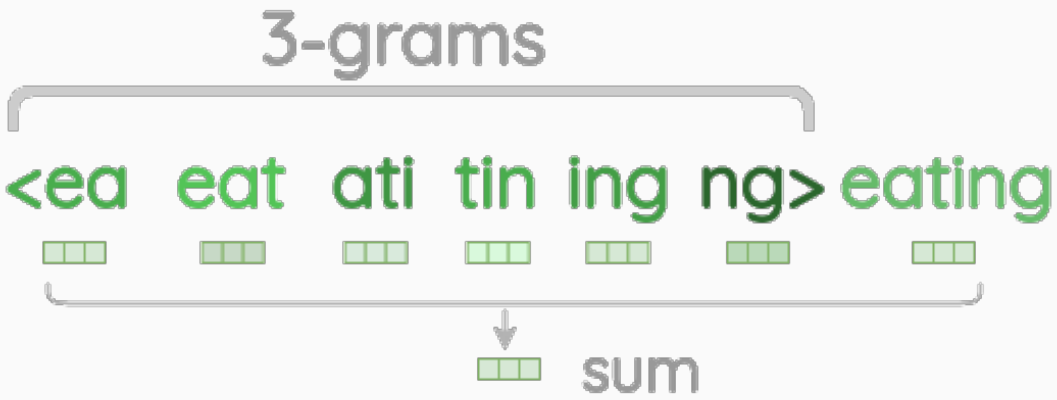
1. Character n-gram embeddings
2. Sub-lexical tokenisation (BPE, Wordpiece...)
3. Character RNNs ← our solution
4. Character CNNs

Affix embeddings

- Pre-processing: decompose words into sub-lexical units
  - → Handcrafted morpheme dictionaries
  - → Automatic morphological analysers
  - → Proxy: character n-gram suffixes/prefixes
- Embed sub-lexical units as we did for words
- Combine (sum, concatenate) sub-word embeddings

Character n-grams: FastText

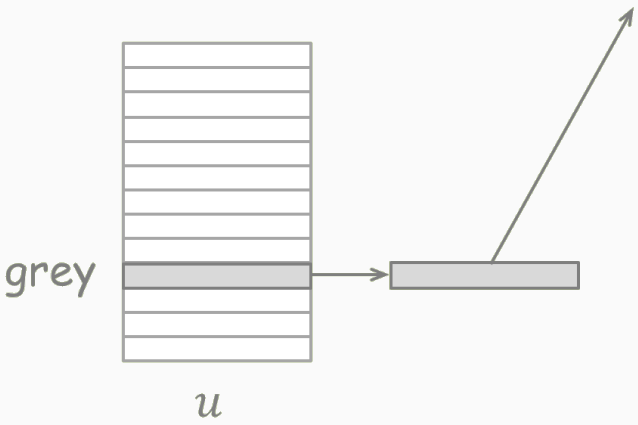
- Sum word and character n-gram embeddings
- Can create embeddings for OOV on the fly
  - → Average the character n-gram embeddings
- Pre-trained embeddings available: `https://fasttext.cc/`
  - → Can be reimplemented in specific `torch` model

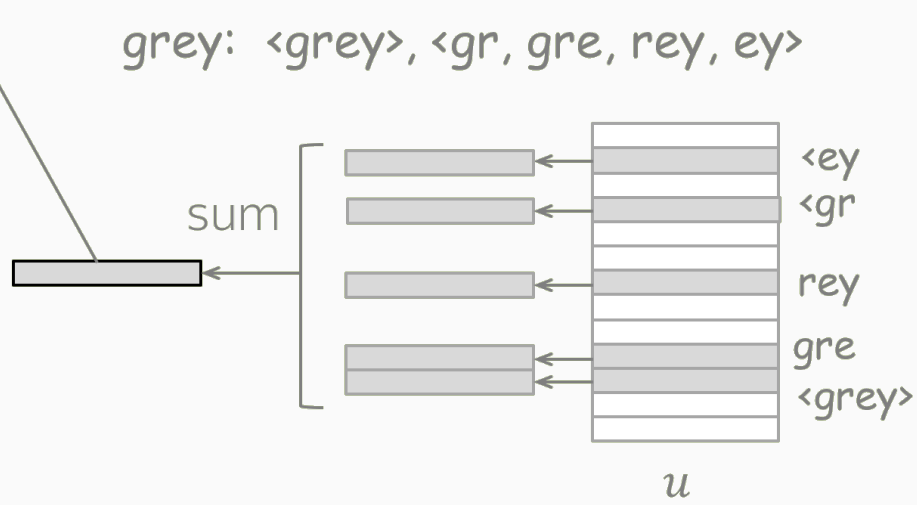


Source: `https://amitness.com/2020/06/fasttext-embeddings/`

Word2Vec vs. FastText

… I saw a cute grey cat playing in the garden …





Word2Vec

Vocabulary consists of:
- words

Word vector is:
- one vector from the look-up table

FastText

Vocabulary consists of:
- words and character n-grams

Word vector is:
- sum of word vector and vectors for its n-grams

Source: `https://lena-voita.github.io/nlp_course/word_embeddings.html`

BPE tokenisation

- Solution adopted in most (all) modern LLMs
- Split text into frequency-based sub-words, e.g. BPE:
  - → Start with the set of c characters in the corpus
  - → Merge most frequent pair AB into a new symbol AB
  - → Stop after k merges → vocabulary of size $|V| = c + k$

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

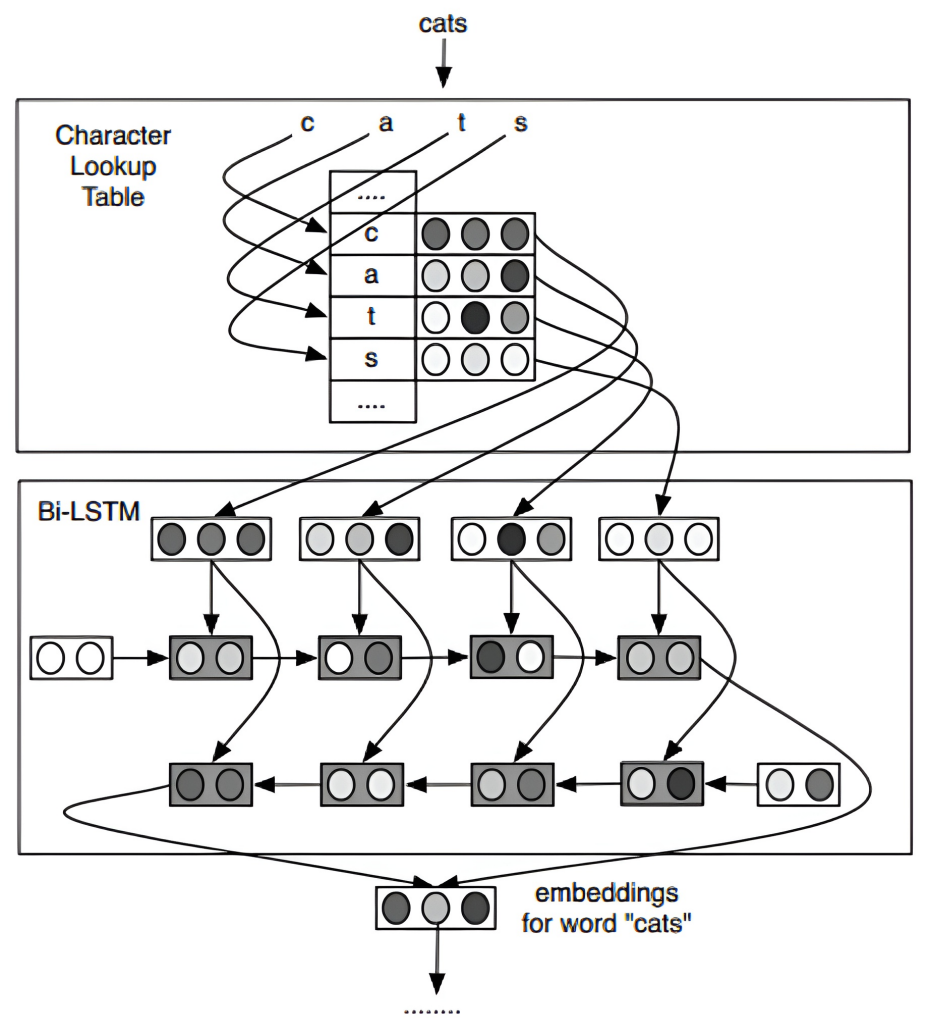
$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens til k times
  $t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
  $t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
  $V \leftarrow V + t_{NEW}$ # update the vocabulary
  Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

Source: Jurafsky and Martin

Character-level RNN

- Idea: split text into characters instead of words
  - → Word embeddings → character embeddings
- RNN's context can "encode" character sequences
  - → Limited *n*-gram → unlimited character sequences (in theory)
- OOVs are represented by their characters
  - → Good news: no need for OOV word embedings!
- OOV characters may still occur, but very rarely
  - → Example: zero OOV characters in Sequoia

Character-level RNN: example



Source: https://aclanthology.org/Y18-1061

Character RNN for text classification

- Goal: adapt word-based RNN text classifier to deal with characters

Data preparation:

```
if in_type == "char":
  fields = " ".join(fields[1:])
else :
  fields = fields[1:]
```

- Everything else kept identical!

Character RNN for (word) sequence tagging

- Text classification: tag each sentence
  - $\rightarrow$ Always take RNN state at last position
- Morphological feature prediction: tag each word
  - $\rightarrow$ Retrieve one RNN state per word
  - $\rightarrow$ How to align character-based RNN states with words?
- Idea: word representation = last character's RNN state
  - $\rightarrow$ We need to keep a **mapping** between words & characters

Character-word alignment in RNN

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
chars=	PAD	L	a	␣	g	a	r	e	␣	r	o	u	t	i	è	r	e	␣	a	t	t	e	n	d
in_enc=	0	3	4	2	5	4	6	7	2	6	8	9	10	11	12	6	7	2	4	10	10	7	13	14
ends=	2	7	16	23	32	36	50																	

- `in_enc`: encoded input: each character gets unique identifier
  - → Space identifier ≠ padding identifier
- `ends`: positions of last characters of each word
- Both `in_enc` and `ends` are inputs
  - → Part of `DataLoader`, given to `forward`

Tagger's `forward` function

- Embed and contextualise `in_enc` using a `GRU`
- Obtain GRU states corresponding to `ends` positions
  - → Use a `for` loop within `forward`
  - → OR use `gather` function (GPU-friendly)

Examples of gather

```
a = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
mask = torch.tensor([[1], [1]])
print(a.gather(dim=1, index=mask))
```

Examples of gather

```
a = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
mask = torch.tensor([[1], [1]])
print(a.gather(dim=1, index=mask))
# tensor([[2.], [5.]])
print(a.gather(dim=0, index=mask))
```

Examples of gather

```
a = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
mask = torch.tensor([[1], [1]])
print(a.gather(dim=1, index=mask))
# tensor([[2.], [5.]])
print(a.gather(dim=0, index=mask))
# tensor([[4.], [4.]])
print(a.gather(0, torch.tensor([[1, 0, 1],
                                [0, 1, 0]])))
```

Examples of gather

```
a = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
mask = torch.tensor([[1], [1]])
print(a.gather(dim=1, index=mask))
# tensor([[2.], [5.]])
print(a.gather(dim=0, index=mask))
# tensor([[4.], [4.]])
print(a.gather(0, torch.tensor([[1, 0, 1],
                                [0, 1, 0]])))
# tensor([[4., 2., 6.],
#         [1., 5., 3.]])
print(a.gather(1, torch.tensor([[2, 1],
                                [0, 2]])))
```

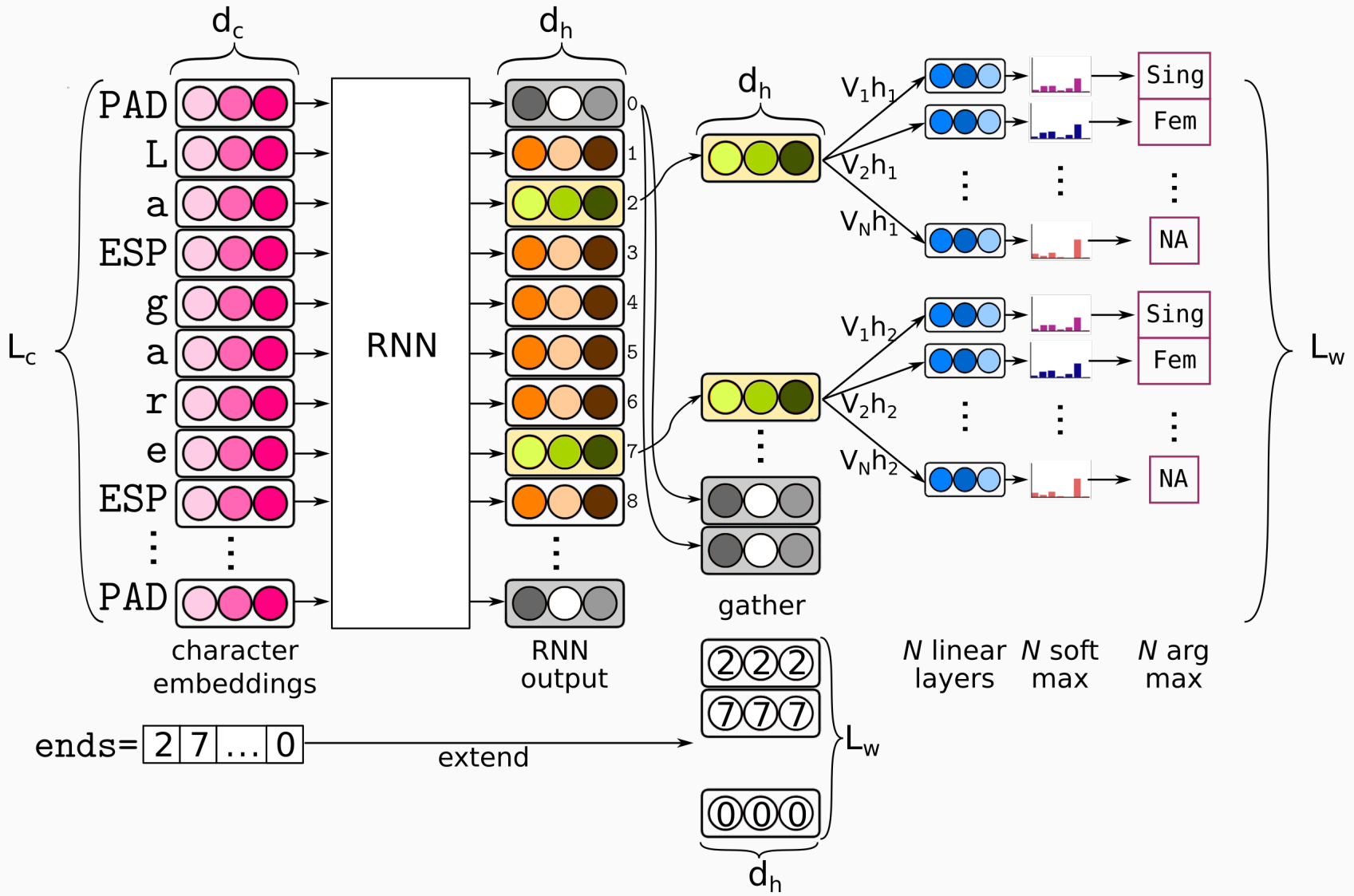
Examples of gather

```
a = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
mask = torch.tensor([[1], [1]])
print(a.gather(dim=1, index=mask))
# tensor([[2.], [5.]])
print(a.gather(dim=0, index=mask))
# tensor([[4.], [4.]])
print(a.gather(0, torch.tensor([[1, 0, 1],
                                [0, 1, 0]])))
# tensor([[4., 2., 6.],
#         [1., 5., 3.]])
print(a.gather(1, torch.tensor([[2, 1],
                                [0, 2]])))
#tensor([[3., 2.],
#        [4., 6.]])
```

`gather` trick for padding

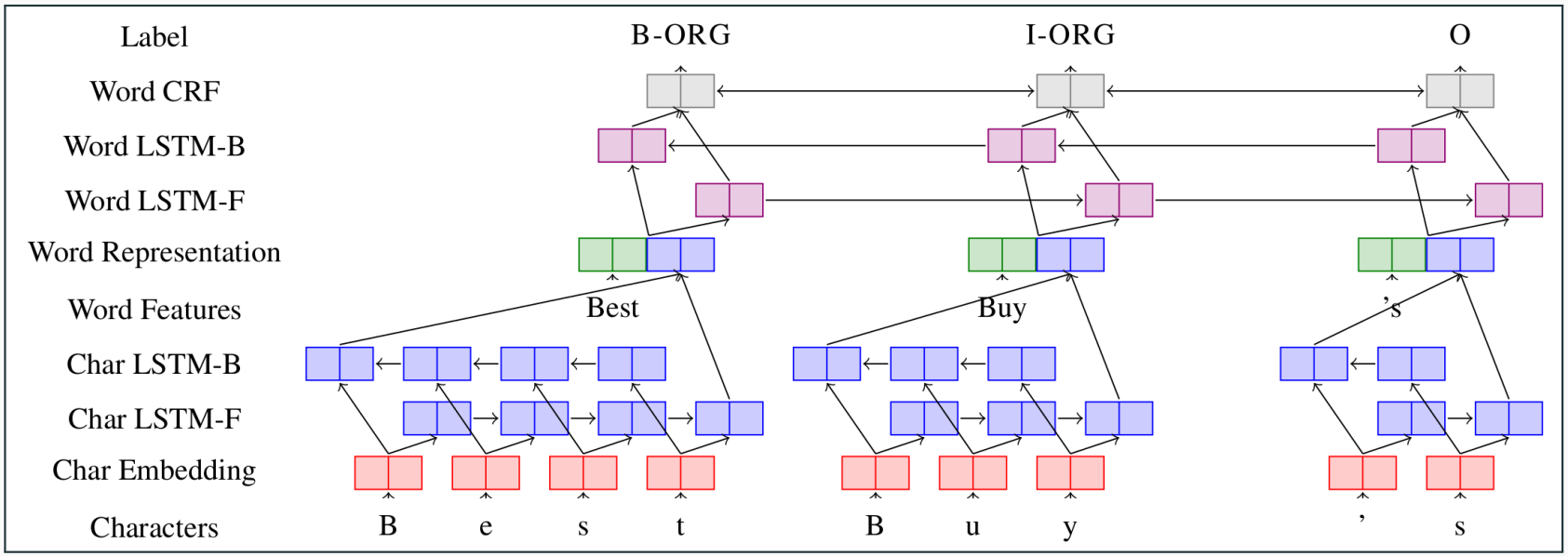
- Do we need to start the sentence with `<PAD>`?
  - → No, but there is a trick!
- Since `ends` is an input, it is padded with zeroes
- `gather` gets RNN-state at position zero
  - → Corresponding to padding vector (should be zero)
- Everything will be fine ☺

Everything everywhere all at once



Word + character RNN

- Use char embeddings ~~instead of~~ combined with word embeddings
  - → Usually works best
- Bidirectional RNN: concatenate first and last character



Source: https://www.aclweb.org/anthology/C18-1182

Character RNNs: pros and cons

Pros

- Able to represent OOV words via their characters
- No morpheme decomposition: implicitly extract relevant affixes

Cons

- RNNs are hard to parallelise (other than batch)
- RNN character models tend to be very slow

Outline

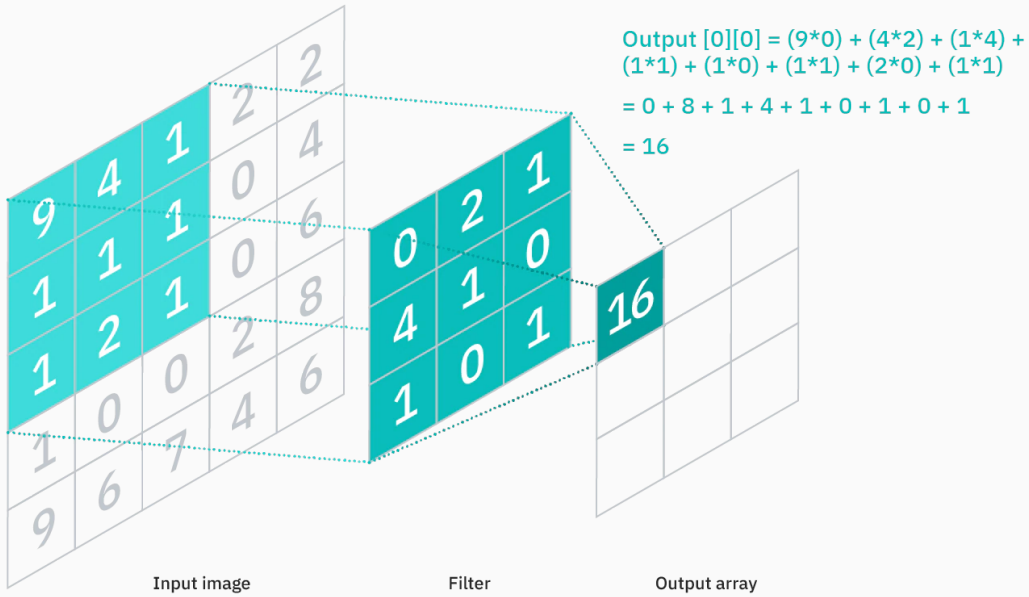
Morphological analysis

Multi-tasking

Sub-lexical models

Convolution for text

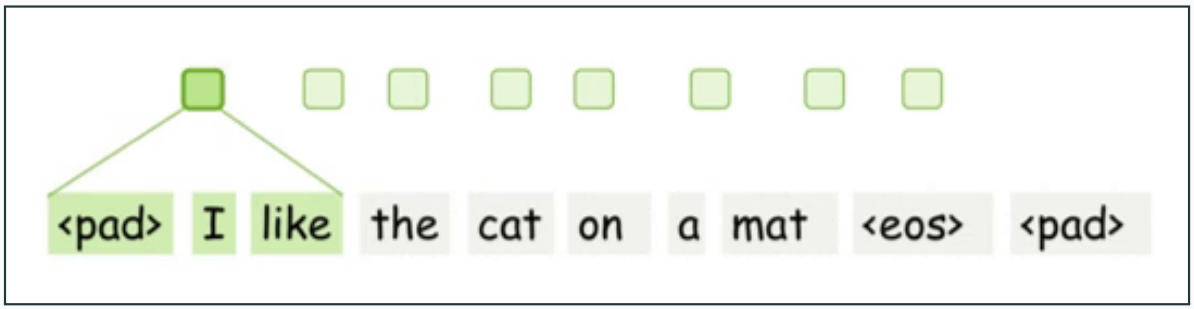
Convolution for images



- Dot product of small kernel matrix (filter) with input
- Move filter across image with a given step (stride)
- Obtain a "summary" of the image with the filter
- Local pixel patterns, robust to translation
- Multiple filters extract different characteristics

1D convolution for words

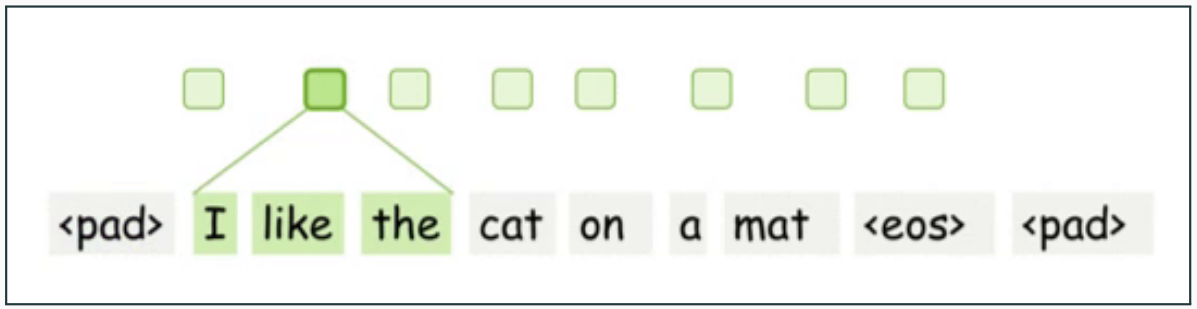
- Apply filter (or kernel) over k adjacent words
- Sliding window over L words of the sentence
- Generate $L - k + 1$ scalar values for the filter



Source: https://lena-voita.github.io/nlp_course/models/convolutional.html

1D convolution for words

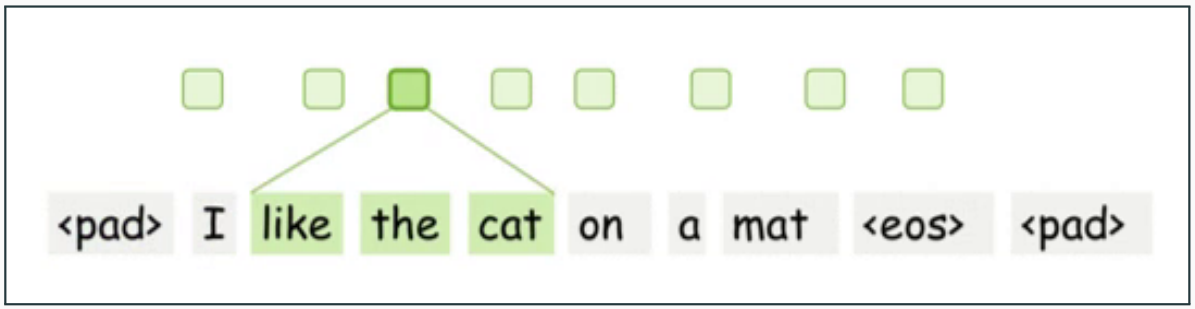
- Apply filter (or kernel) over k adjacent words
- Sliding window over L words of the sentence
- Generate $L - k + 1$ scalar values for the filter



Source: https://lena-voita.github.io/nlp_course/models/convolutional.html

1D convolution for words

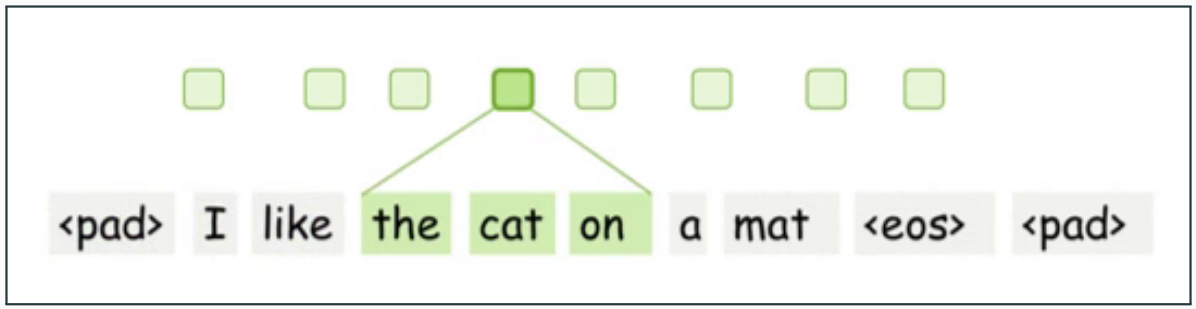
- Apply filter (or kernel) over k adjacent words
- Sliding window over L words of the sentence
- Generate $L - k + 1$ scalar values for the filter



Source: https://lena-voita.github.io/nlp_course/models/convolutional.html

1D convolution for words

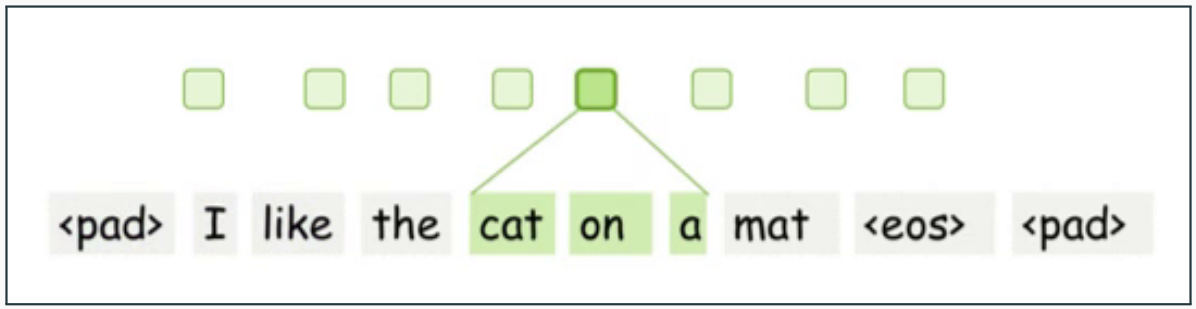
- Apply filter (or kernel) over k adjacent words
- Sliding window over L words of the sentence
- Generate $L - k + 1$ scalar values for the filter



Source: https://lena-voita.github.io/nlp_course/models/convolutional.html

1D convolution for words

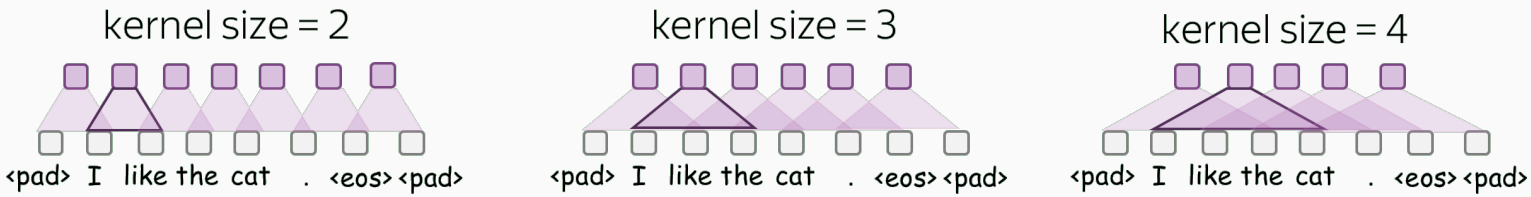
- Apply filter (or kernel) over k adjacent words
- Sliding window over L words of the sentence
- Generate $L - k + 1$ scalar values for the filter



Source: `https://lena-voita.github.io/nlp_course/models/convolutional.html`

1D convolution: kernel size

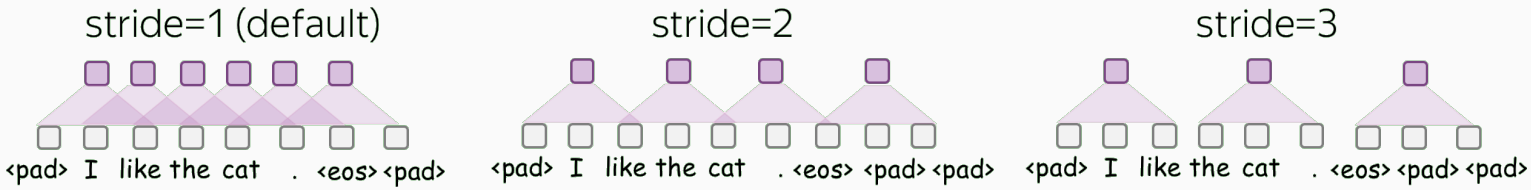
- Kernel size k is the number of adjacent words covered
  - $\rightarrow$ $k = 2$: dot product of filter with 2 adjacent words
  - $\rightarrow$ $k = 3$: dot product of filter with 3 adjacent words
  - $\rightarrow$ ...
- Equivalent to extracting relevant k-grams from sentence



Source: https://lena-voita.github.io/nlp_course/models/convolutional.html

1D convolution: stride

- As for images, the stride is the window's increment step
  - → Stride=1: apply convolution, then move 1 step right
  - → Stride=2: apply convolution, then move 2 steps right
  - → ...

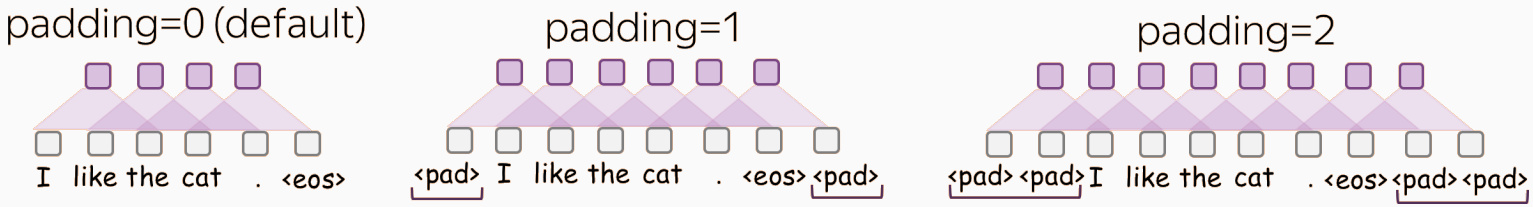


Source: `https://lena-voita.github.io/nlp_course/models/convolutional.html`

1D convolution: padding

- As for images, we can pad the text on the left/right
- Narrow convolution: result of size $L - k + 1$ (stride=1)
- Wide convolution: result of size L (stride=1)

  $\rightarrow$ Add $k - 1$ padding vectors at beginning/end of sentence

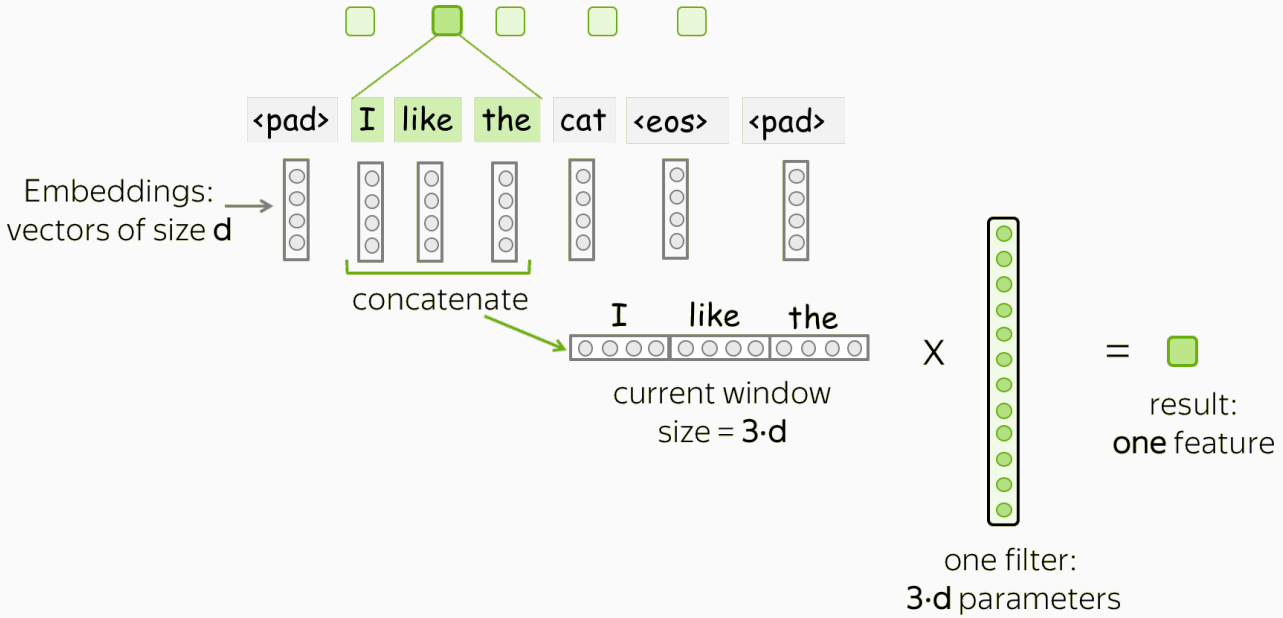


Source: `https://lena-voita.github.io/nlp_course/models/convolutional.html`

1D convolution and word embeddings

- Apply filter over k adjacent ~~words~~ word embeddings
- Sliding window over L word embeddings of size d

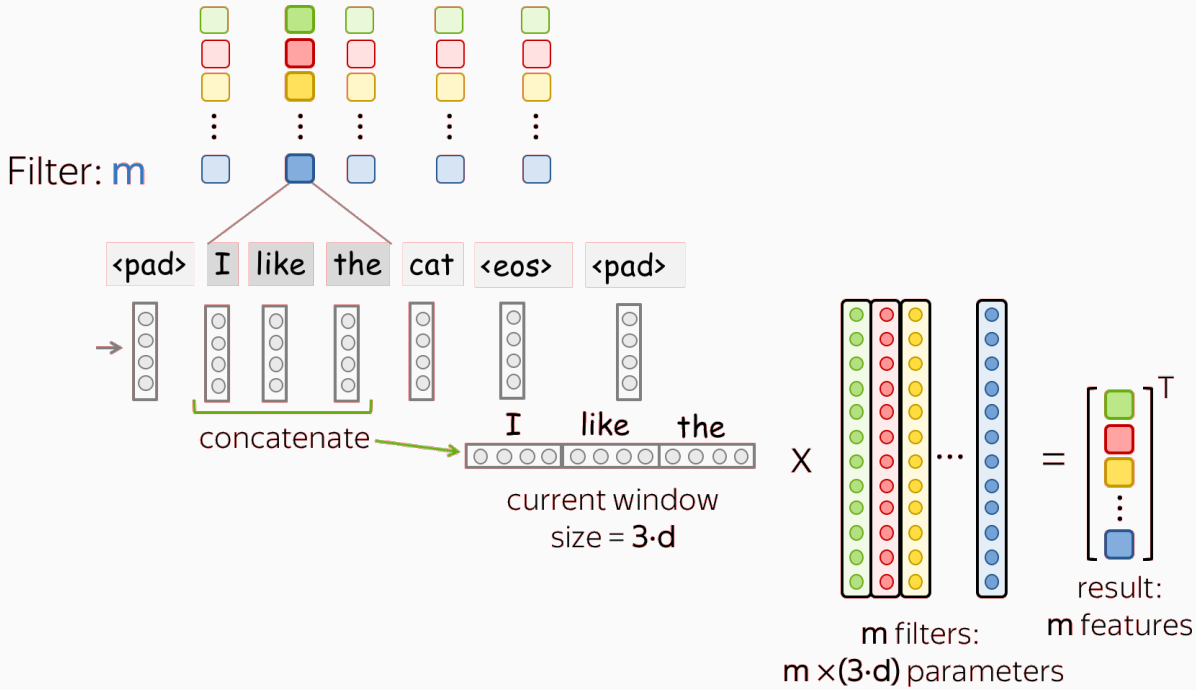
  $\rightarrow$ Each filter is of size $k \times d$



Source: `https://lena-voita.github.io/nlp_course/models/convolutional.html`

Multiple filters

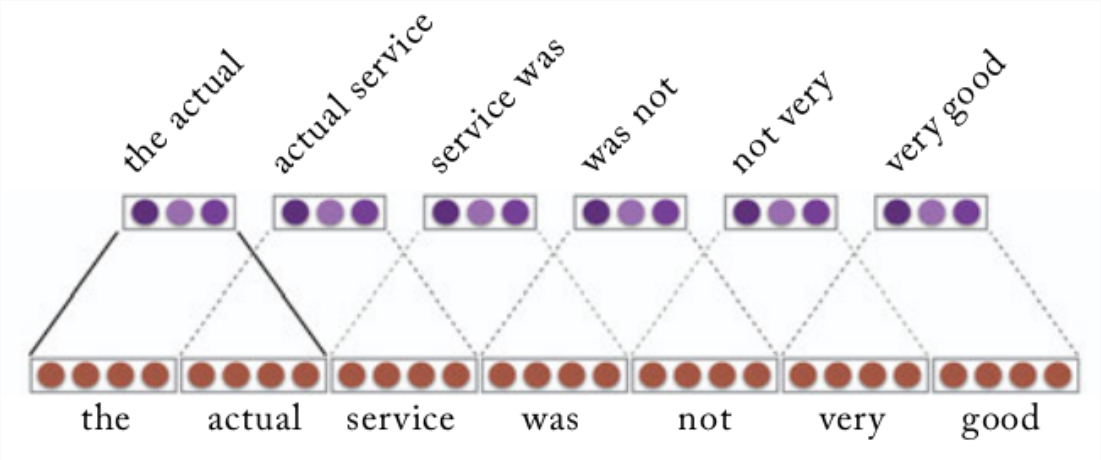
- Generate $L - k + 1$ scalar values for one filter
- Use m filters to extract different characteristics
- Result: m-dimensional vector for each word



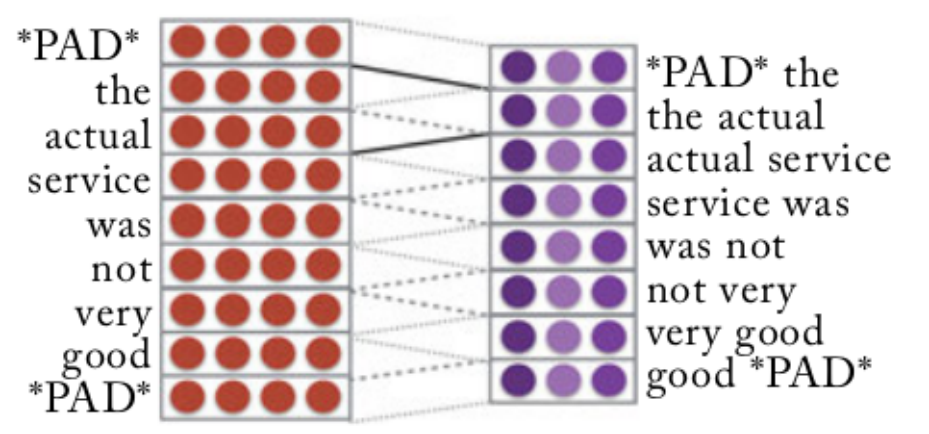
Source: `https://lena-voita.github.io/nlp_course/models/convolutional.html`

Concatenation vs. stacked notation

Concatenation



Stacked



Source: Goldberg, 2017

1D convolution in `torch`

```
conv = nn.Conv1d(in_channels, out_channels, kernel_size)
result = conv(embedded.transpose(2,1))
```

- `in_channels`: Dimension d of input word embeddings
- `out_channels`: Number of filters m to apply in parallel
- `kernel_size`: Number of adjacent words covered by filters

- Input: dimension $d \times L$ (transposition needed!)
  $\rightarrow$ Whereas `nn.Embedding` gives $L \times d$ matrix
- Output: dimension $m \times (L - k + 1)$

Conv1d examples in torch

```
cnn = nn.Conv1d(in_channels=3, out_channels=1, kernel_size=2)
print(cnn(mat).shape)
```

Conv1d examples in torch

```
cnn = nn.Conv1d(in_channels=3, out_channels=1, kernel_size=2)
print(cnn(mat).shape)
# torch.Size([1, 7])
cnn = nn.Conv1d(in_channels=3, out_channels=5, kernel_size=3)
print(cnn(mat).shape)
```

Conv1d examples in torch

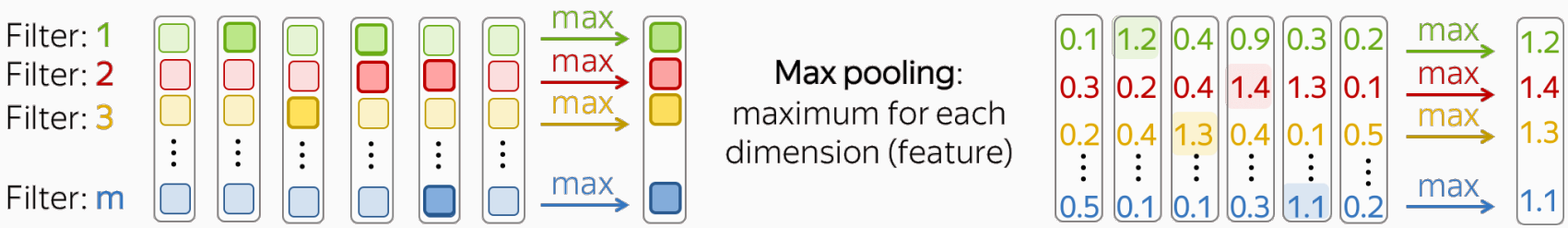
```
cnn = nn.Conv1d(in_channels=3, out_channels=1, kernel_size=2)
print(cnn(mat).shape)
# torch.Size([1, 7])
cnn = nn.Conv1d(in_channels=3, out_channels=5, kernel_size=3)
print(cnn(mat).shape)
# torch.Size([5, 6])
cnn = nn.Conv1d(in_channels=2, out_channels=5, kernel_size=3)
print(cnn(mat).shape)
```

Conv1d examples in torch

```
cnn = nn.Conv1d(in_channels=3, out_channels=1, kernel_size=2)
print(cnn(mat).shape)
# torch.Size([1, 7])
cnn = nn.Conv1d(in_channels=3, out_channels=5, kernel_size=3)
print(cnn(mat).shape)
# torch.Size([5, 6])
cnn = nn.Conv1d(in_channels=2, out_channels=5, kernel_size=3)
print(cnn(mat).shape)
# Error: expected input to have 2 channels,
# but got 3 channels instead
```

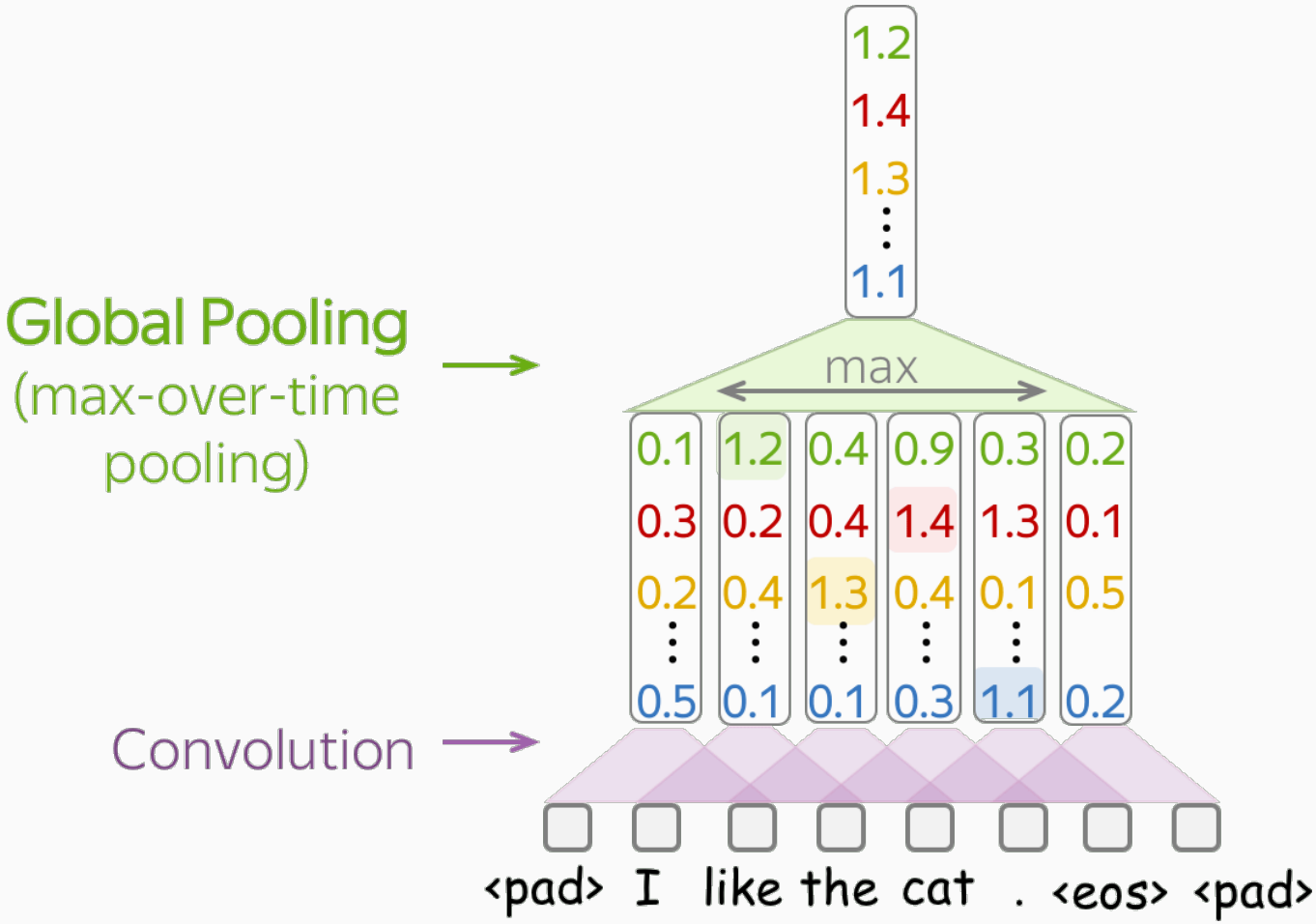
(Max-)pooling over sentence

- Output of dimension $m \times (L - k + 1)$
  - $\rightarrow$ Where L depends on sentence length
- Variable-length issue for text classification (again!)
- Idea: apply max-pooling over $L - k + 1$ results
  - $\rightarrow$ Encode whole sentence as fixed m-dimensional vector



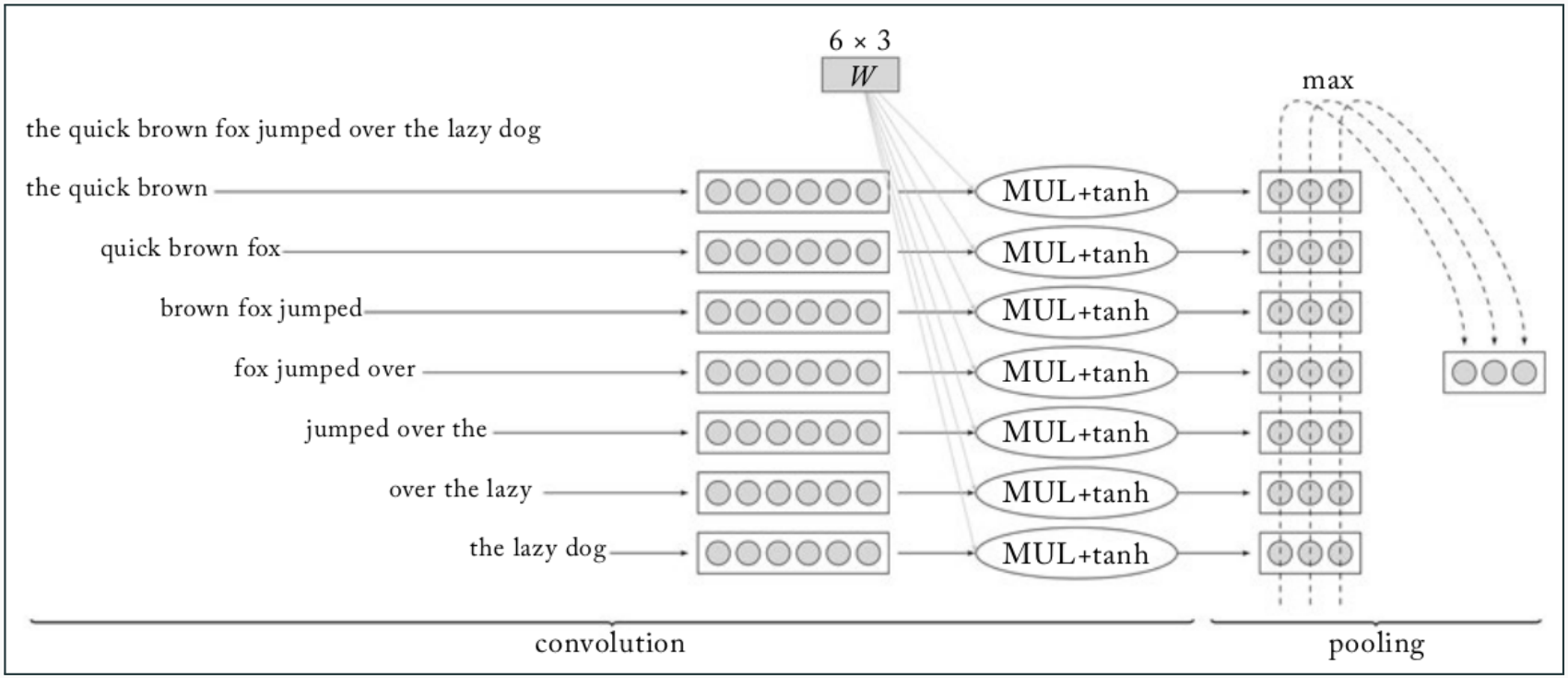
Source: `https://lena-voita.github.io/nlp_course/models/convolutional.html`

Max-pooling for text classification



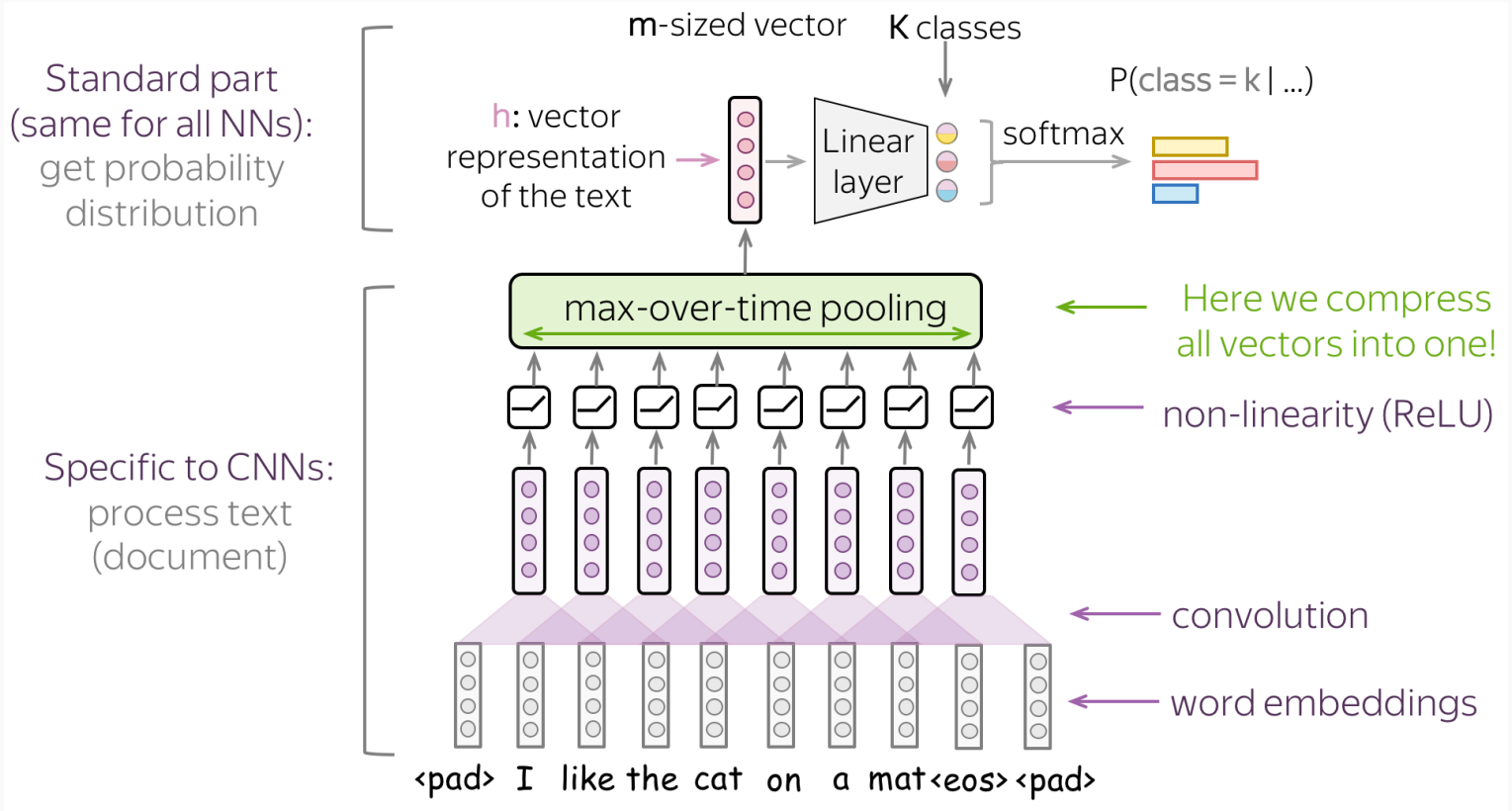
Source: `https://lena-voita.github.io/nlp_course/models/convolutional.html`

Word convolution + max-pooling



Source: Goldberg, 2017

Full CNN-based text classifier



Source: https://lena-voita.github.io/nlp_course/models/convolutional.html

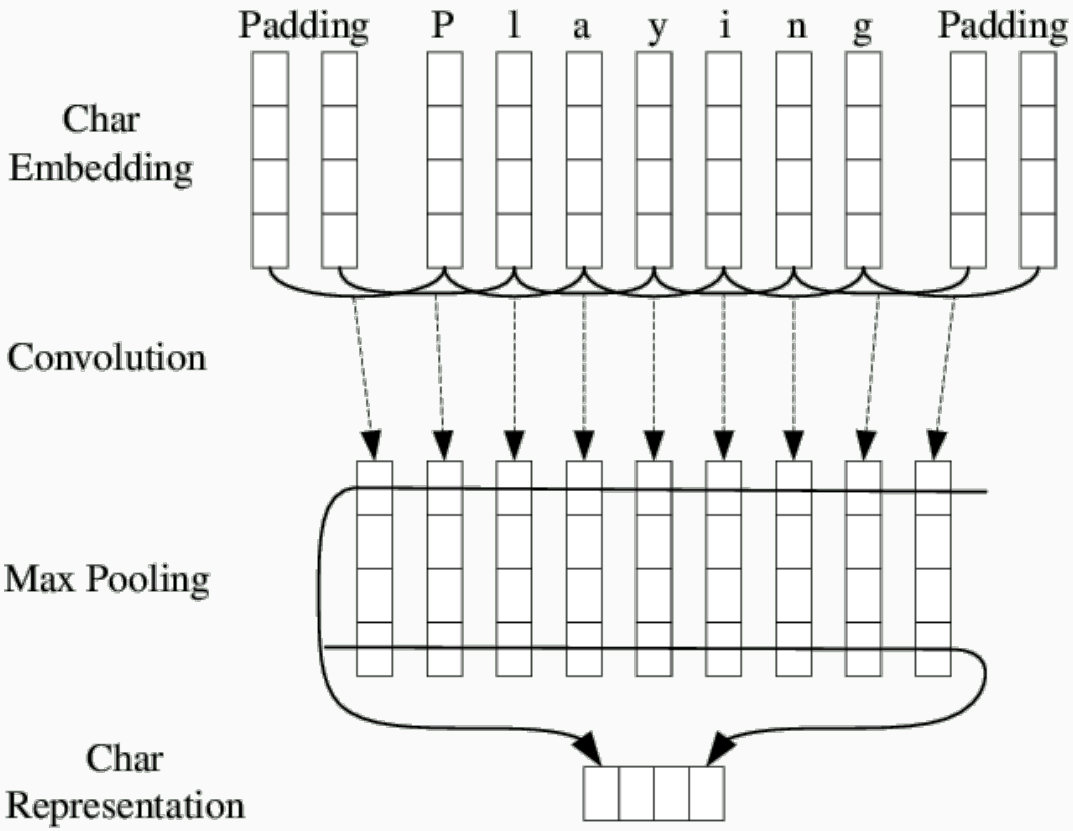
CNN text classifier

```
class CNNClassifier(nn.Module):

  def __init__(self, d_embed, d_hidden, d_in, d_out):
    super().__init__()
    self.embed = nn.Embedding(d_in, d_embed, padding_idx=0)
    self.cnn = nn.Conv1d(d_embed, d_hidden, kernel_size=5)
    self.dropout = nn.Dropout(0.1)
    self.nonlin = torch.nn.ReLU()
    self.decision = nn.Linear(d_hidden, d_out)

  def forward(self, idx_words):
    emb = self.embed(idx_words)
    conv = self.nonlin(self.cnn(emb.transpose(2,1)))
    hidden = nn.functional.max_pool1d(conv, conv.shape[-1])
    return self.decision(self.dropout(hidden.squeeze()))
```

Character-based CNN

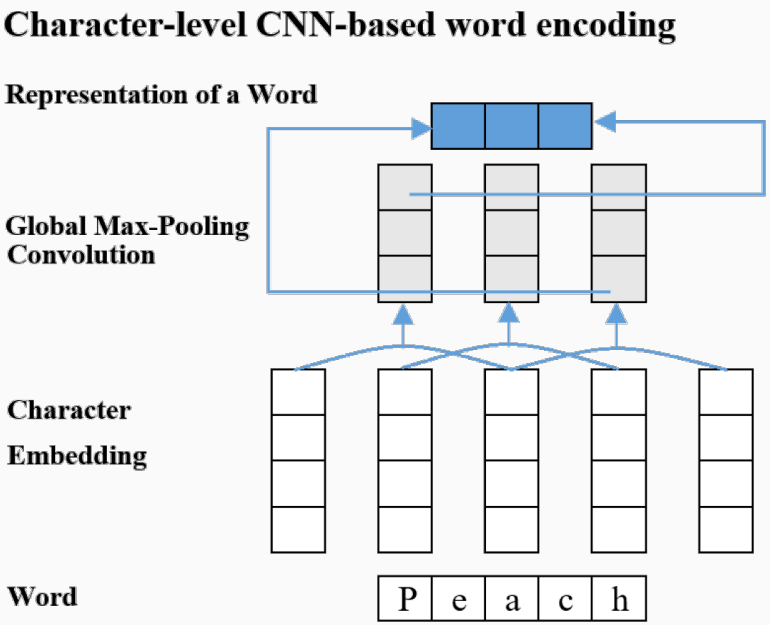
- Max-pooling over word length

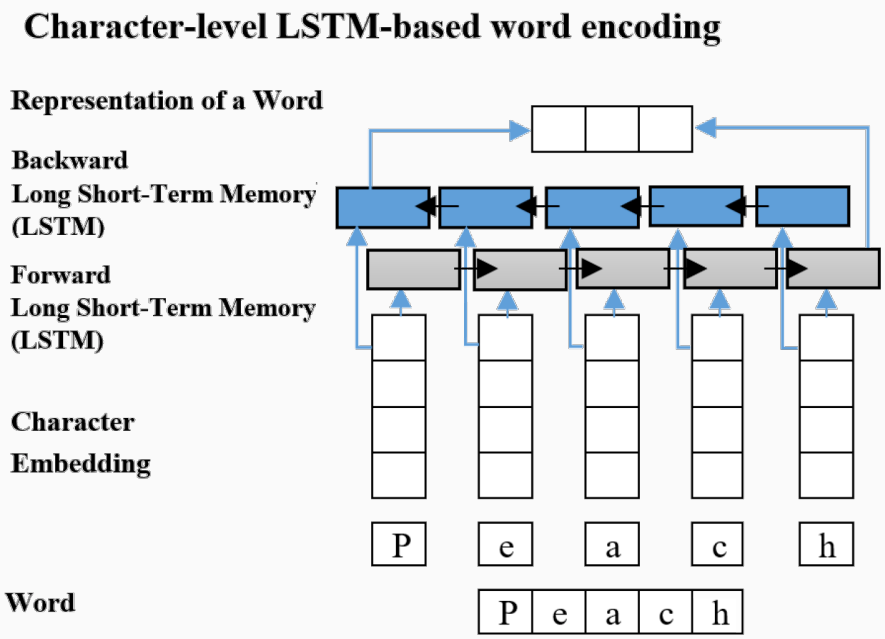
  → Result: 1 m-dimensional vector per word



Source: Ma & Hovy, 2016

Character CNN vs. character RNN





Source: https://www.mdpi.com/2076-3417/10/21/7557

Character CNN: pros and cons

Pros

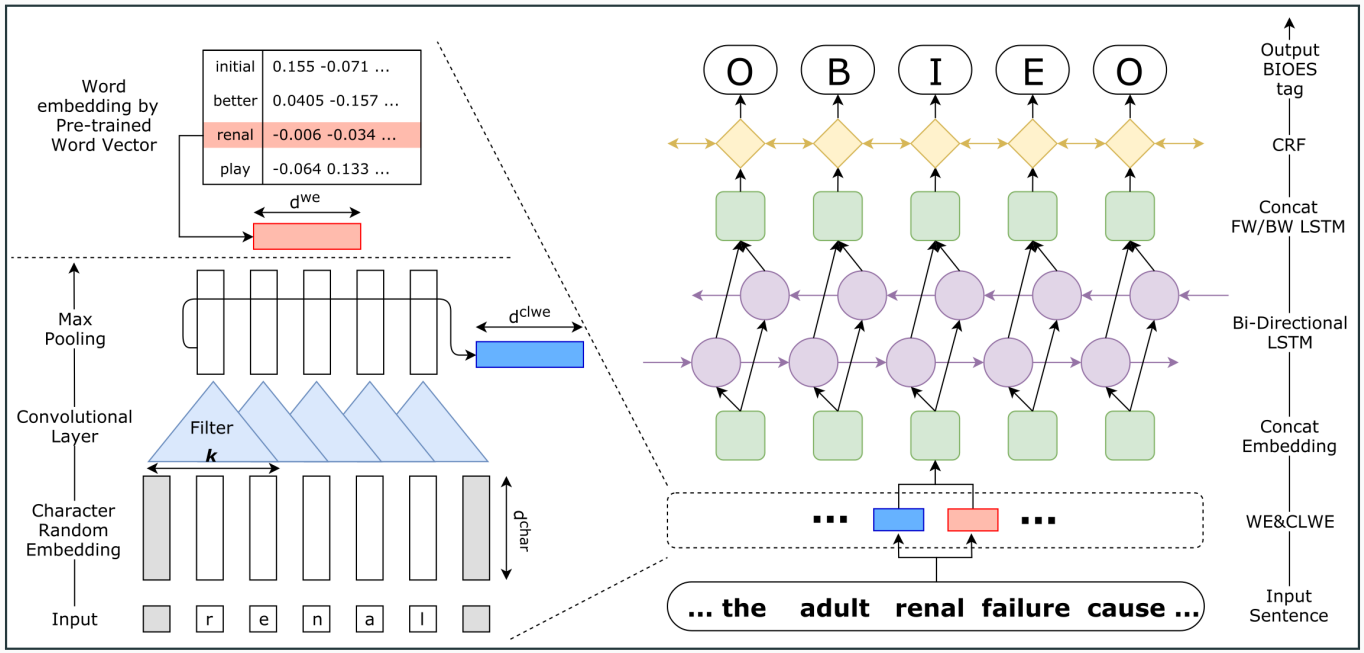
- Much faster than character RNN
- Generalises well for typos, morphology, OOVs

Cons

- Limited coverage: fixed-length character *n*-grams
  - $\rightarrow$ Works less well for OOV URLs, numbers, long names (use placeholders)
- Does not guarantee the absence of OOV characters (<UNK>)

Character + word embeddings

- Using only characters is usually too strict
  - → Although some models work (e.g. Flair)
- We usually combine character-based CNN with word embeddings



Source: https://doi.org/10.1186/s12859-019-2813-6

Thanks!

That's all for today

Carlos Ramisch (carlos.ramisch@univ-amu.fr)

With the help of Benoit Favre & Alexis Nasr

Prédiction Structurée pour le Traitement Automatique des Langues
Master IAAA
Aix Marseille Université

Sources i

- Dan Zeman's Unidive webinar slides – https://github.com/UniDive/2023-unidive-webinar/
- Dan Zeman's course – https://ufal.mff.cuni.cz/courses/npfl094
- Tamanna's blog post – https://medium.com/@tam.tamanna18/power-of-character-level-rnns-and-embeddings-in-natural-language-processing-b84321d1
- Avi Chawla's blog post – https://blog.dailydoseofds.com/p/transfer-learning-vs-fine-tuning and https://blog.dailydoseofds.com/p/a-practical-and-intuitive-guide-to
- Adam K's blog posts – https://medium.com/gumgum-tech/an-easy-recipe-for-multi-task-learning-in-pytorch-that-you-can-do-at-home-1e529a8df
- Sebastian Ruder's blog posts – https://www.ruder.io/multi-task/ and https://www.ruder.io/multi-task-learning-nlp/

Sources ii

- Vincent Dumoulin, Francesco Visin – https://github.com/vdumoulin/conv_arithmetic
- Sarah Paul's blog – https://sarahsnippets.com/understanding-morphology-part-1/
- Rybak & Wróblewska (2018) – https://aclanthology.org/K18-2004/
- Yadav & Bethard (2018) – https://aclanthology.org/C18-1182/
- Benoit Favre's PSTALN website – https://pageperso.lis-lab.fr/benoit.favre/pstaln/
- Alexis Nasr's slides from TLNL and ML course – master 2 IAAA
- Dan Jurafsky & James H. Martin, Speech and Language Processing 3 (Online edition Aug 2024) – https://web.stanford.edu/~jurafsky/slp3/
- Yoav Goldberg, Neural network methods for NLP – https://www.morganclaypool.com/doi/abs/10.2200/S00762ED1V01Y201703HLT037

Sources iii

- Lena Voita's "NLP course For You" – `https://lena-voita.github.io/nlp_course.html`
- Universal Dependencies – `https://universaldependencies.org/`
- Discussions with Bruno Guillaume, Marie Candito, Benoit Favre, Alexis Nasr, Frédéric Béchet
- Feedback from participants of previous course editions
- Slides illustrated with the help of: Google images, `imgupscaler.com`, Canva
- Slides written with the help of: ChatGPT, Google Bard, DeepL, Linguee, Overleaf

Backup slides

1D convolution for text

- Document D contains L words $w_1 \dots w_L$
- $\mathbf{E}_{[w_i]} = \mathbf{w_i} \rightarrow d$-dimensional embeddings of w_i
- Filter $\mathbf{u} \in \mathbb{R}^{k \times d}$ covers k words
  - Summarises an n-gram of length k
- Operator $\bigoplus(\mathbf{w_{i:i+k-1}})$ concatenates embeddings $\mathbf{w_{i:i+k-1}}$
- Window $i \rightarrow \mathbf{x_i} = \bigoplus(\mathbf{w_{i:i+k-1}}) = [\mathbf{w_i}; \mathbf{w_{i+1}}; \dots; \mathbf{w_{i-k+1}}]$
  - $\mathbf{x_i} \in \mathbb{R}^{k \times d}$
- Apply filter to each window i:

$$p_i = g(\mathbf{x_i} \cdot \mathbf{u}) + b$$

- With $b, p_i \in \mathbb{R}$, g non-linear activation

Multiple filters

- Use l different filters $\mathbf{u_1}, \ldots, \mathbf{u_l}$ arranged in matrix $\mathbf{U}$

$$\mathbf{p_i} = g(\mathbf{x_i} \cdot \mathbf{U}) + \mathbf{b}$$

- With $\mathbf{b}, \mathbf{p_i} \in \mathbb{R}^l$ and $\mathbf{U} \in \mathbb{R}^{k \times d \times l}$
- Slide window over all i positions, obtain $L - k + 1$ vectors $\mathbf{p_i}$

Pooling

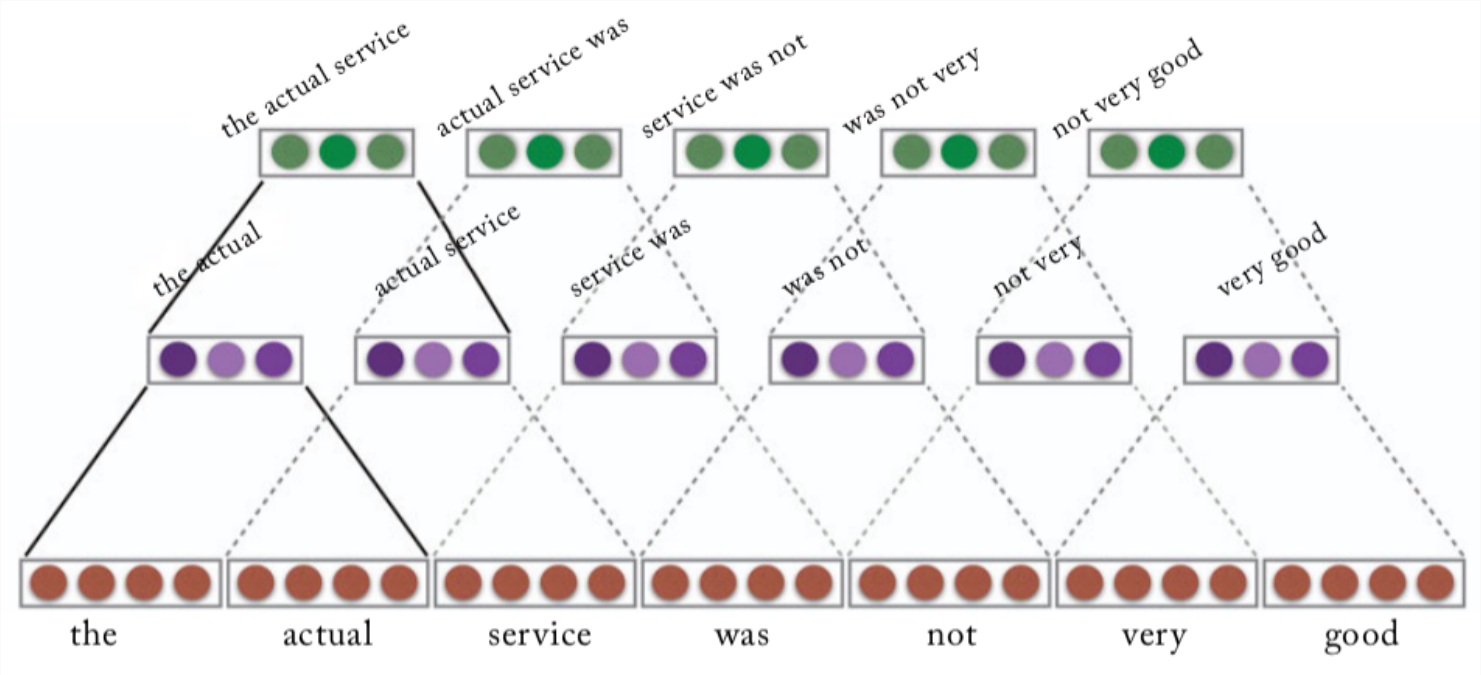
- Goal
  - Reduce input dimension
  - Keep most informative features
- For images:
  - Cover a $m \times p$ region of the feature map
- For text:
  - Cover the entire sentence or document of length n

$$\mathbf{c}_{[j]} = \max_{1<i\leq m} \mathbf{p}_{\mathbf{i}[j]} \qquad \forall j \in [1, l]$$

- $\mathbf{c} \in \mathbb{R}^l$ summarises D with fixed length l
- Average pooling: average of n-grams

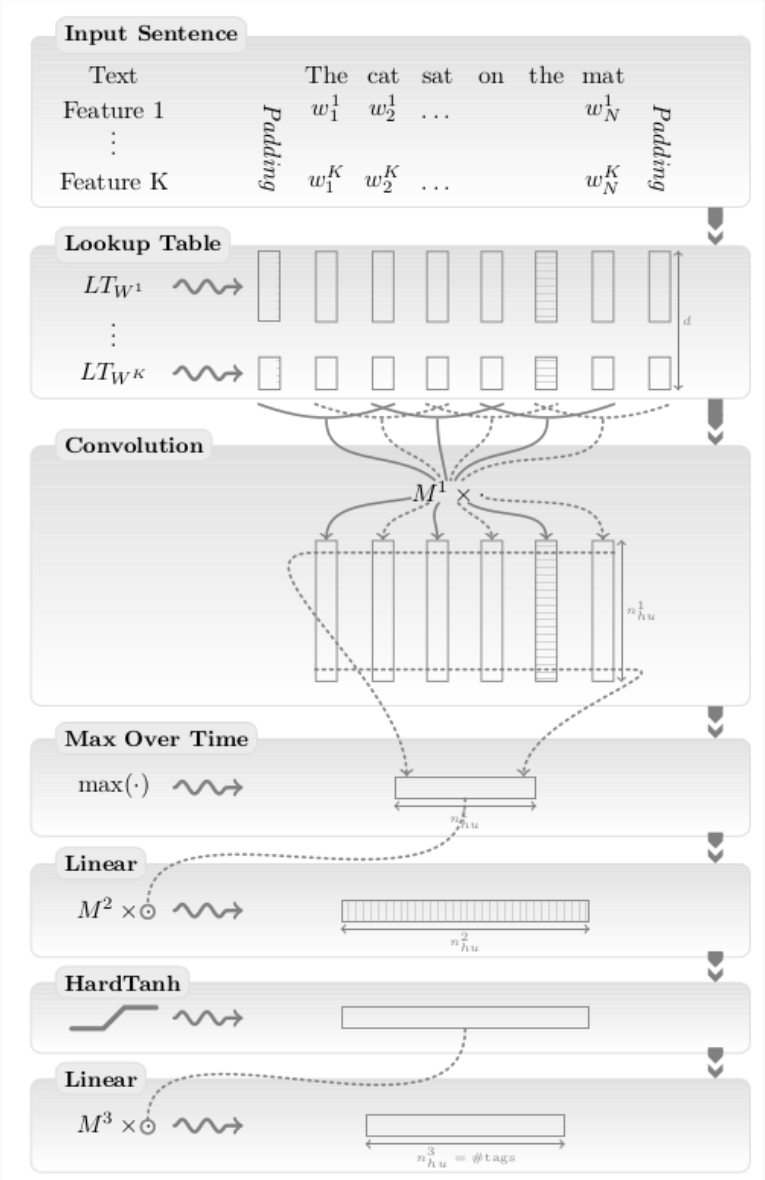
Hierarchical convolution

- Larger n-grams, potentially discontinuous



Source: Goldberg, 2017

Historical note



Dumas text classification results

Model	Input	Acc (%)
BOW	word	**52.93**
BOW	char	39.87
GRU	word	45.67
GRU	char	42.47
CNN	word	47.47
CNN	char	46.33