ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI FORLÌ

91258 / B0385
Natural Language Processing

Lesson 16. Recurrent Neural Networks

Alberto Barrón-Cedeño
a.barron@unibo.it

28/11/2024

Previously

- CNNs for text

Table of Contents

Chapter 8 of Lane et al. (2019)

Introduction

Introduction

CNNs

- Good for analysing *full* texts ($\sim$sentences)
- Words tending to appear close to each other are spotted and play a joint role

Introduction

CNNs

- Good for analysing *full* texts ($\sim$sentences)
- Words tending to appear close to each other are spotted and play a joint role
- Longer relationships —farther than $[3, 4]$ words are ignored

Introduction

CNNs

- Good for analysing *full* texts ($\sim$sentences)
- Words tending to appear close to each other are spotted and play a joint role
- Longer relationships —farther than $[3, 4]$ words are ignored

What is missing?

Introduction

CNNs

- Good for analysing *full* texts ($\sim$sentences)
- Words tending to appear close to each other are spotted and play a joint role
- Longer relationships —farther than $[3, 4]$ words are ignored

What is missing?

- Keeping track of what happened long ago
- Memory

Introduction

CNNs

- Good for analysing *full* texts (∼sentences)
- Words tending to appear close to each other are spotted and play a joint role
- Longer relationships —farther than $[3, 4]$ words are ignored

What is missing?

- Keeping track of what happened long ago
- Memory

- Language is not an image —no snapshots
- Language is a sequence; both text and speech

Keeping the past *in mind*

Remembering the Past

$$w_0\ w_1\ w_2\ w_3\ \dots\ w_{t-1}\ w_t\ w_{t+1}$$

(Lane et al., 2019, p. 250)

Remembering the Past

$$w_0\ w_1\ w_2\ w_3\ \ldots\ w_{t-1}\ w_t\ w_{t+1}$$

- To understand a text at time t, we need to consider what happened at time $t - k$

(Lane et al., 2019, p. 250)

Remembering the Past

$$w_0 \; w_1 \; w_2 \; w_3 \; \ldots \; w_{t-1} \; w_t \; w_{t+1}$$

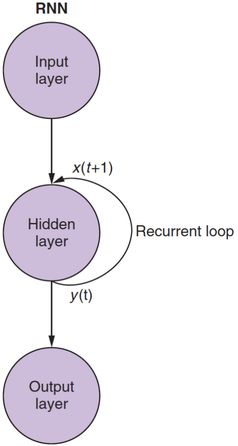
- To understand a text at time t, we need to consider what happened at time $t - k$
- Recurrent neural nets (RRN) come into play

(Lane et al., 2019, p. 250)

Remembering the Past

$$w_0\ w_1\ w_2\ w_3\ \ldots\ w_{t-1}\ w_t\ w_{t+1}$$

- To understand a text at time t, we need to consider what happened at time $t - k$
- Recurrent neural nets (RRN) come into play

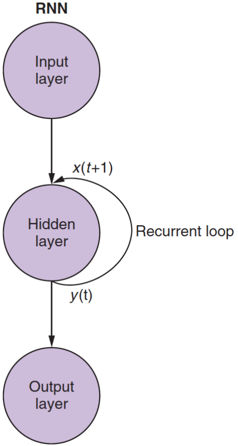


(Lane et al., 2019, p. 250)

Remembering the Past

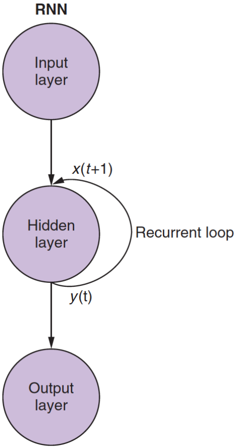
$$w_0\ w_1\ w_2\ w_3\ \dots\ w_{t-1}\ w_t\ w_{t+1}$$

- To understand a text at time t, we need to consider what happened at time $t-k$
- Recurrent neural nets (RRN) come into play
- RNNs combine what happened before with what is happening now



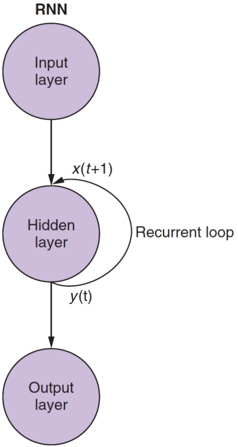
(Lane et al., 2019, p. 250)

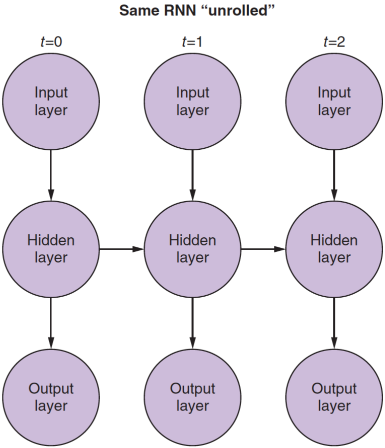
Full feed-forward networks that consider their own output



(Lane et al., 2019, p. 252)

Full feed-forward networks that consider their own output

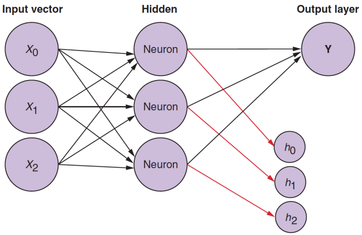




(all three columns are the same)

(Lane et al., 2019, p. 252)

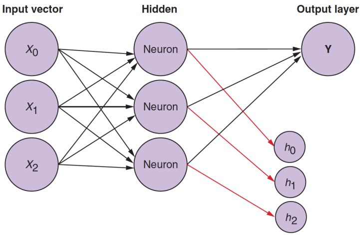
Zooming into the unrolled RNN: t and $t+1$



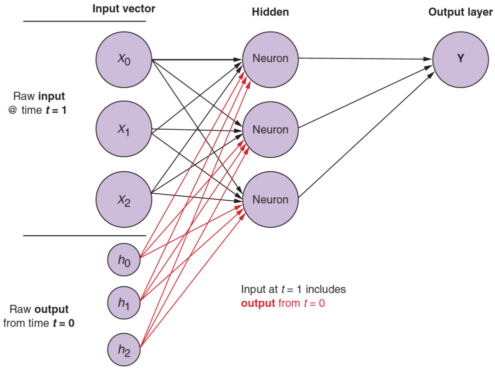
$$t = 0$$

(Lane et al., 2019, p. 252–253)

Zooming into the unrolled RNN: t and $t + 1$



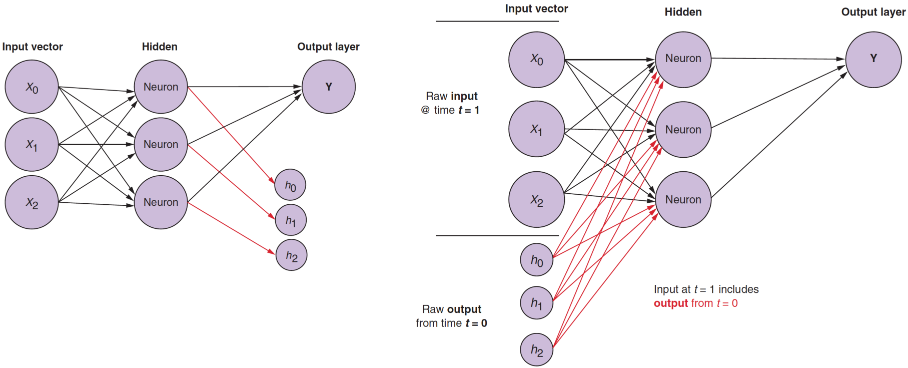
$t = 0$



$t = 1$

(Lane et al., 2019, p. 252–253)

Zooming into the unrolled RNN: t and $t+1$

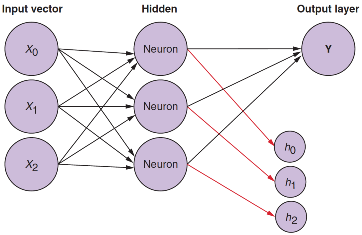


$t = 0$ $\qquad$ $t = 1$

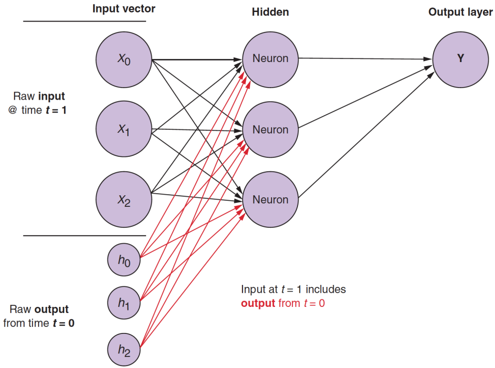
- The red arrows are just *standard* connections, with weights

(Lane et al., 2019, p. 252–253)

Zooming into the unrolled RNN: t and $t+1$



$t = 0$

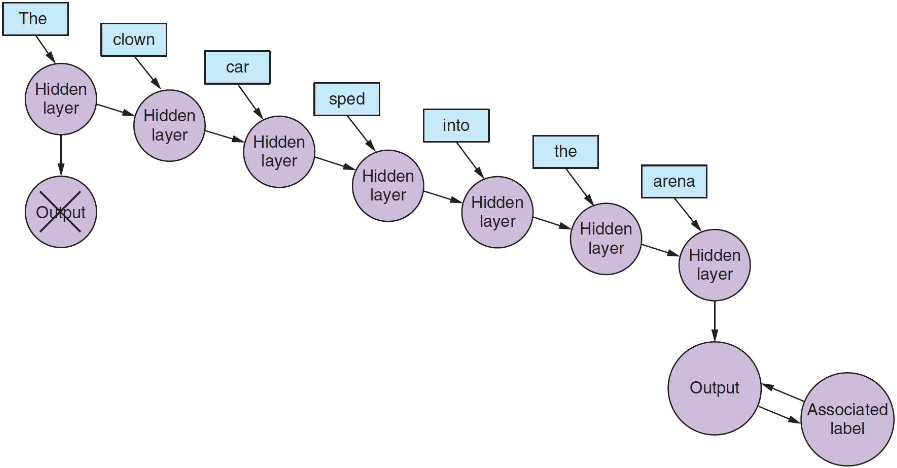


$t = 1$

- The red arrows are just *standard* connections, with weights
- Now we can feed the text, one word at a time

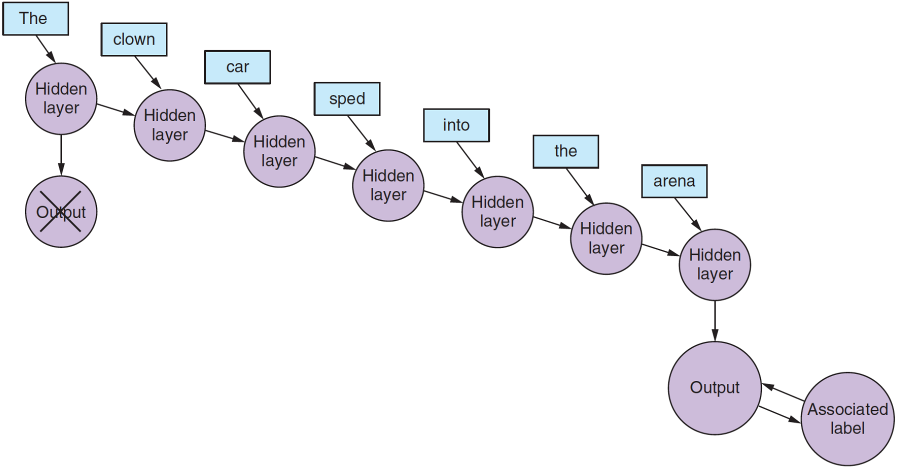
(Lane et al., 2019, p. 252–253)

"Multiple inputs, one output"



(Lane et al., 2019, p. 254)

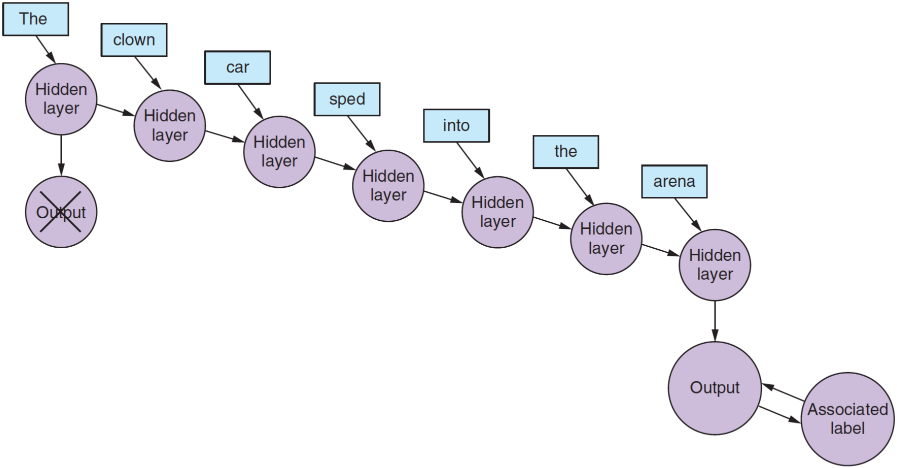
“Multiple inputs, one output”



- No more length constraints (although we have to be reasonable)

(Lane et al., 2019, p. 254)

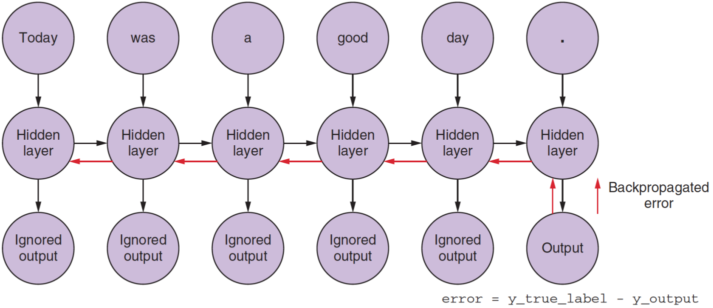
“Multiple inputs, one output”



- No more length constraints (although we have to be reasonable)
- No more a bunch of snapshots; there is a sense of time

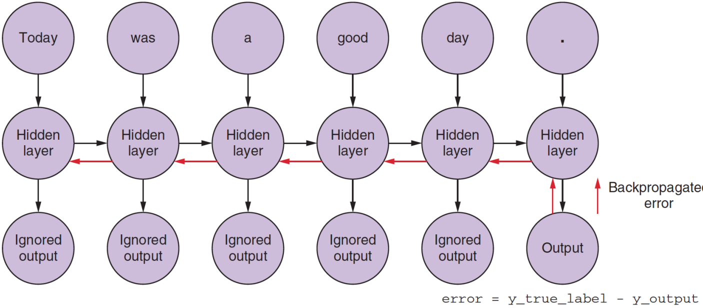
(Lane et al., 2019, p. 254)

Backpropagation through Time: the "Vanilla" Way



(Lane et al., 2019, p. 256)

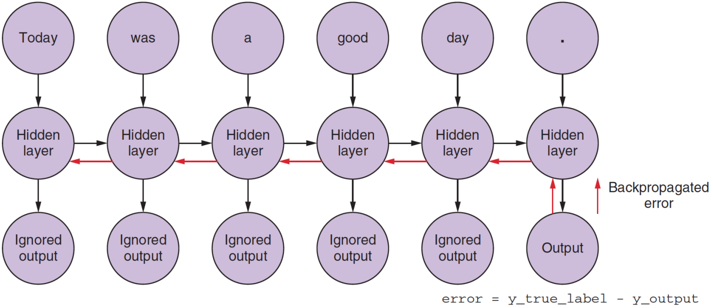
Backpropagation through Time: the "Vanilla" Way



- All intermediate outputs are ignored; the loss is computed at the end

(Lane et al., 2019, p. 256)

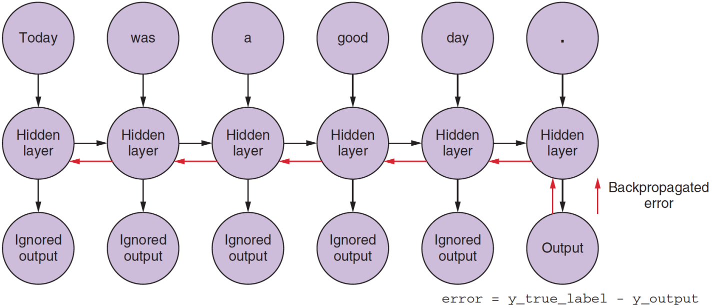
Backpropagation through Time: the “Vanilla” Way



- All intermediate outputs are ignored; the loss is computed at the end
- The same chain rule is applied to do backpropagation; but this time it heads to “the past”

(Lane et al., 2019, p. 256)

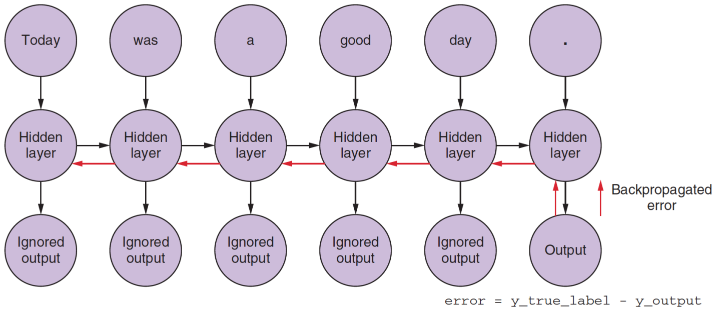
Backpropagation through Time: the "Vanilla" Way



- All intermediate outputs are ignored; the loss is computed at the end
- The same chain rule is applied to do backpropagation; but this time it heads to "the past"
- The weight corrections are calculated for each t

(Lane et al., 2019, p. 256)

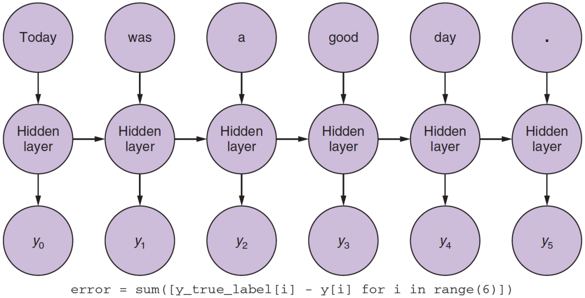
Backpropagation through Time: the "Vanilla" Way



- All intermediate outputs are ignored; the loss is computed at the end
- The same chain rule is applied to do backpropagation; but this time it heads to "the past"
- The weight corrections are calculated for each t
- The combined updates are applied only until reaching $t = 0$

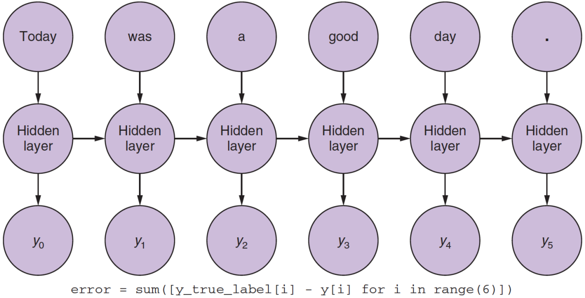
(Lane et al., 2019, p. 256)

Backpropagation through Time: the Better Way



(Lane et al., 2019, p. 258)

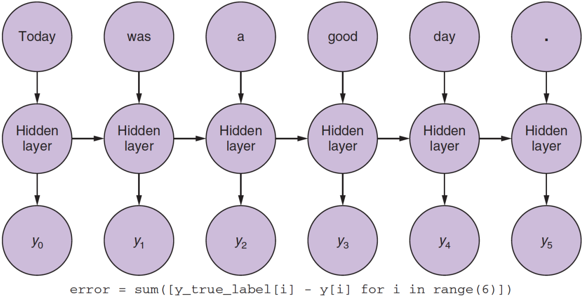
Backpropagation through Time: the Better Way



- We compute the loss combining all intermediate outputs

(Lane et al., 2019, p. 258)

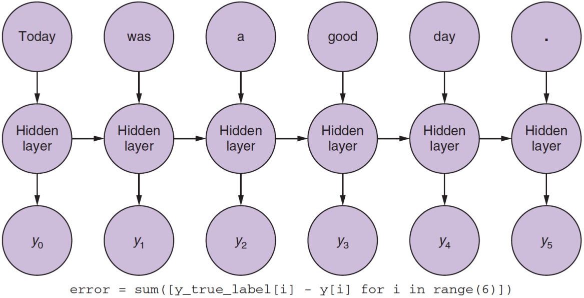
Backpropagation through Time: the Better Way



- We compute the loss combining all intermediate outputs
- The weight corrections are still additive: the update is applied until
    1. computing all errors and

(Lane et al., 2019, p. 258)

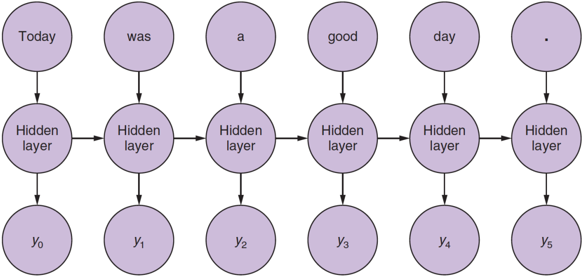
Backpropagation through Time: the Better Way



- We compute the loss combining all intermediate outputs
- The weight corrections are still additive: the update is applied until
    1. computing all errors and
    2. reaching back to the weight adjustments in $t = 0$

(Lane et al., 2019, p. 258)

Backpropagation through Time: the Better Way




```
error = sum([y_true_label[i] - y[i] for i in range(6)])
```

- We compute the loss combining all intermediate outputs
- The weight corrections are still additive: the update is applied until
  1. computing all errors and
  2. reaching back to the weight adjustments in $t = 0$

Let us see

(Lane et al., 2019, p. 258)

RNNs in Keras

RNN in Keras: what we have so far

We have setup a simple recurrent neural network

RNN in Keras: what we have so far

We have setup a simple recurrent neural network

- The input sequences have fixed length: 400 tokens (each 300D)

RNN in Keras: what we have so far

We have setup a simple recurrent neural network

- The input sequences have fixed length: 400 tokens (each 300D)
- Our recurrent layer contains 50 units

RNN in Keras: what we have so far

We have setup a simple recurrent neural network

- The input sequences have fixed length: 400 tokens (each 300D)
- Our recurrent layer contains 50 units
- The output will be 400×50:

RNN in Keras: what we have so far

We have setup a simple recurrent neural network

- The input sequences have fixed length: 400 tokens (each 300D)
- Our recurrent layer contains 50 units
- The output will be 400×50:
  - 400 elements
  - one 50D vector each

RNN in Keras: what we have so far

We have setup a simple recurrent neural network

- The input sequences have fixed length: 400 tokens (each 300D)
- Our recurrent layer contains 50 units
- The output will be 400×50:
  - 400 elements
  - one 50D vector each

```
return_sequences=True
```

RNN in Keras: what we have so far

We have setup a simple recurrent neural network

- The input sequences have fixed length: 400 tokens (each 300D)
- Our recurrent layer contains 50 units
- The output will be 400×50:
  - 400 elements
  - one 50D vector each

`return_sequences=True`

True return the network value at each t: 400 50D vectors

RNN in Keras: what we have so far

We have setup a simple recurrent neural network

- The input sequences have fixed length: 400 tokens (each 300D)
- Our recurrent layer contains 50 units
- The output will be 400×50:
  - 400 elements
  - one 50D vector each

`return_sequences=True`

True return the network value at each t: 400 50D vectors

False return a single 50D vector (default)

RNN in Keras: what we have so far

We have setup a simple recurrent neural network

- The input sequences have fixed length: 400 tokens (each 300D)
- Our recurrent layer contains 50 units
- The output will be 400×50:
  - 400 elements
  - one 50D vector each

`return_sequences=True`

True return the network value at each t: 400 50D vectors

False return a single 50D vector (default)

True → this is why we are padding

RNN in Keras: what we have so far

We have setup a simple recurrent neural network

- The input sequences have fixed length: 400 tokens (each 300D)
- Our recurrent layer contains 50 units
- The output will be 400×50:
  - 400 elements
  - one 50D vector each

`return_sequences=True`

True return the network value at each t: 400 50D vectors

False return a single 50D vector (default)

True → this is why we are padding

Let us see

RNN in Keras: further details

- A `Dense` layer expects a *flat* vector

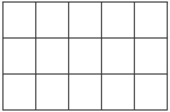
Example derived from https://stackoverflow.com/questions/43237124/role-of-flatten-in-keras

RNN in Keras: further details

- A `Dense` layer expects a *flat* vector

```
model.add(Flatten())
```

5×3



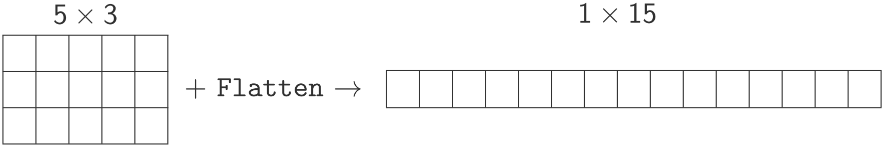
$+$ `Flatten` $\rightarrow$

Example derived from
https://stackoverflow.com/questions/43237124/role-of-flatten-in-keras

RNN in Keras: further details

- A `Dense` layer expects a *flat* vector

```
model.add(Flatten())
```

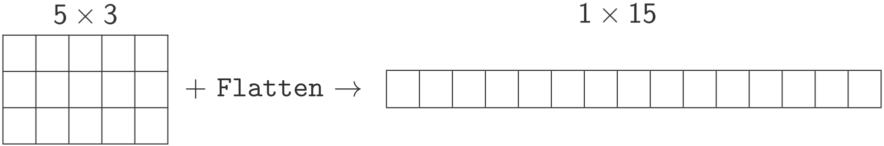


Example derived from `https://stackoverflow.com/questions/43237124/role-of-flatten-in-keras`

RNN in Keras: further details

- A `Dense` layer expects a *flat* vector

```
model.add(Flatten())
```



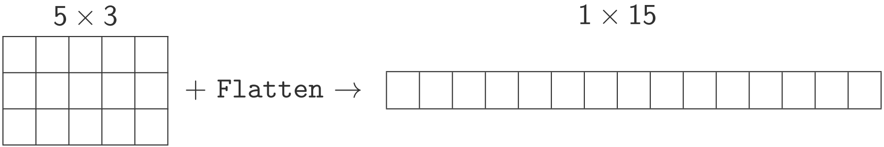
- In our case: $400 \times 50 \rightarrow 1 \times 20,000$

Example derived from
https://stackoverflow.com/questions/43237124/role-of-flatten-in-keras

RNN in Keras: further details

- A `Dense` layer expects a *flat* vector

`model.add(Flatten())`



- In our case: $400 \times 50 \rightarrow 1 \times 20,000$

Let us see

Example derived from https://stackoverflow.com/questions/43237124/role-of-flatten-in-keras

Some parameters are "free"

embedding_dims comes from the embedding space; hard to change, but possible: other embeddings, 1-hot

[1] See, for instance, Fernicola et al. (2020).

Some parameters are "free"

embedding_dims comes from the embedding space; hard to change, but possible: other embeddings, 1-hot

num_neurons kind of arbitrary; can be changed

[1] See, for instance, Fernicola et al. (2020).

Some parameters are "free"

embedding_dims comes from the embedding space; hard to change, but possible: other embeddings, 1-hot

num_neurons kind of arbitrary; can be changed

maxlen kind of arbitrary; can be changed (or neglected)

[1] See, for instance, Fernicola et al. (2020).

Some parameters are "free"

embedding_dims comes from the embedding space; hard to change, but possible: other embeddings, 1-hot

num_neurons kind of arbitrary; can be changed

maxlen kind of arbitrary; can be changed (or neglected)

batch_size bigger→faster (higher local minimum risk)

[1] See, for instance, Fernicola et al. (2020).

Some parameters are "free"

embedding_dims comes from the embedding space; hard to change, but possible: other embeddings, 1-hot

num_neurons kind of arbitrary; can be changed

maxlen kind of arbitrary; can be changed (or neglected)

batch_size bigger→faster (higher local minimum risk)

epochs trivial to increase (don't start from scratch each time)

[1] See, for instance, Fernicola et al. (2020).

Some parameters are "free"

embedding_dims comes from the embedding space; hard to change, but possible: other embeddings, 1-hot

num_neurons kind of arbitrary; can be changed

maxlen kind of arbitrary; can be changed (or neglected)

batch_size bigger→faster (higher local minimum risk)

epochs trivial to increase (don't start from scratch each time)

Let us see

[1] See, for instance, Fernicola et al. (2020).

Some parameters are "free"

embedding_dims comes from the embedding space; hard to change, but possible: other embeddings, 1-hot

num_neurons kind of arbitrary; can be changed

maxlen kind of arbitrary; can be changed (or neglected)

batch_size bigger→faster (higher local minimum risk)

epochs trivial to increase (don't start from scratch each time)

Let us see

Important: unless you have access to HPC, don't go *bananas* when exploring parameters (and perhaps even in that case)

[1] See, for instance, Fernicola et al. (2020).

Some parameters are "free"

embedding_dims comes from the embedding space; hard to change, but possible: other embeddings, 1-hot

num_neurons kind of arbitrary; can be changed

maxlen kind of arbitrary; can be changed (or neglected)

batch_size bigger→faster (higher local minimum risk)

epochs trivial to increase (don't start from scratch each time)

Let us see

Important: unless you have access to HPC, don't go *bananas* when exploring parameters (and perhaps even in that case)

Try some sensitive configurations and keep track of all the settings and outputs[1]

[1] See, for instance, Fernicola et al. (2020).

References

Fernicola, F., S. Zhang, F. Garcea, P. Bonora, and A. Barrón-Cedeño
2020. Ariemozione: Identifying emotions in opera verses. In *Italian Conference on Computational Linguistics*.

Lane, H., C. Howard, and H. Hapkem
2019. *Natural Language Processing in Action*. Shelter Island, NY: Manning Publication Co.