UNIVERSITÉ DE LORRAINE

Institut des sciences du Digital Management & Cognition
COMPOSANTE DE L'UNIVERSITÉ DE LORRAINE

Data Mining for NLP

3- Unsupervised Approaches to Explore Data

These slides will be available on Arche

By Gaël Guibon, inspired by Benjamin Muller's course

Course Objective

Goal: Use Data Mining methods to improve NLP workflow

- Present methods to **explore textual data**
- Focus on **Machine Learning methods** to deal with NLP data
- Focus on **empirical approaches**

Course Logistics

- 3 sessions (3 lectures + 3 lab)
- 1h30 lectures
- 3 labs (google colab)

Material on Arche

Course Evaluation

- **Final Exam**
  - Mix between questions and code completion
  - 30% of the final grade Data Mining
  - Exam 29/01 5pm

Exploring Data: General Idea

1. **Represent the data**: one-hot, embeddings, features bags, etc.
2. **IF** there are some labels, use them to better understand the data in a **supervised learning** setting
    a. Feature importance score
    b. Leave One Out strategy
    c. Attention scores
3. Use **unsupervised learning** to better understand underlying distribution of the data
    a. PCA
    b. Simple unsupervised approaches
    c. Latent space (auto-encoders)
4. Mix both

How to represent input tokens?

Solution 1: 1-Hot Encoding

1. We associate each *token to a **1-hot vector of size D***

movie = [1, 0, ...,0, 0, 0]
hotel = [0, 1, ..., 0, 0, 0]
...
art = [0, 0,..., 0, 0, 1]

1. **Concatenate them** to get a unidimensional vector

1-Hot Encoding as inputs

$$dnn_\theta : \quad \{0,1\}^{|V|*K} \quad \rightarrow \quad [0,1]^V$$

$$x = ([x_1, .., x_K]) \mapsto \hat{p}$$

- ➔ First hidden layer is of size ***|V|*K***
- ➔ Taking as input **a sparse vector**

1-Hot Encoding as inputs

$$dnn_\theta : \quad \{0,1\}^{|V|*K} \rightarrow [0,1]^V$$
$$x = ([x_1, .., x_K]) \mapsto \hat{p}$$

First hidden layer:
assuming tanh as the activation function, dimension δ

$$h_1 = tanh(W.x) \text{ s.t. } W \in \mathbb{R}^{\delta\times(|V|*K))}$$

1-Hot Encoding as inputs

$$dnn_\theta : \quad \{0,1\}^{|V|*K} \rightarrow [0,1]^V$$
$$x = ([x_1, .., x_K]) \mapsto \hat{p}$$

Limits

➔ The representation of each token is fixed and a 1-hot vector

➔ In this approach, **we do not learn** a representation of **each input token**

How to represent input tokens?

Solution 2: **Integrate a Dense Embedding Layer**

How to represent input tokens?

Solution 2: **Integrate a Dense Embedding Layer**

We define a dense embedding layer $E \in \mathbb{R}^{\delta_e \times |V|}$.

This means that for each token $t \in V$ indexed by j in the vocabulary $V = \{t_1, .., t_{|V|}\}$) we have t_j *embedded* by the vector $E_{.j}$ (i.e. column of the matrix E indexed by j) of dimension δ_e (the dimension of the embedding vectors).

How to represent input tokens?

Solution 2: **Integrate a Dense Embedding Layer**

We define a dense embedding layer $E \in \mathbb{R}^{\delta_e \times |V|}$.

This means that for each token $t \in V$ indexed by j in the vocabulary $V = \{t_1, .., t_{|V|}\})$ we have t_j *embedded* by the vector $E_{.j}$ (i.e. column of the matrix E indexed by j) of dimension δ_e (the dimension of the embedding vectors).

➔ ***E* is part of the parametrization** of the model like any other layers

➔ We can train it during **backprop end-to-end**

Dense Embedding Layer

$$dnn_\theta : \quad \boxed{\mathbb{R}^{|K|*\delta_e}} \rightarrow [0,1]^V$$

$$x = ([x_1, .., x_K]) \mapsto \hat{p}$$

s.t. $x_i = E_{.j} \in \mathbb{R}^{\delta_e}$ with token t_i indexed by j in V

Dense Embedding Layer

$$dnn_\theta : \quad \mathbb{R}^{|K| * \delta_e} \rightarrow [0,1]^V$$

$$x = ([x_1, .., x_K]) \mapsto \hat{p}$$

$$\boxed{\text{s.t. } x_i = E_{.j} \in \mathbb{R}^{\delta_e}} \text{with token } t_i \text{ indexed by } j \ in \ V$$

Dense Embedding Layer

$$dnn_\theta : \quad \mathbb{R}^{|K| * \delta_e} \quad \rightarrow \quad [0,1]^V$$

$$x = ([x_1, .., x_K]) \mapsto \hat{p}$$

$$\boxed{\text{s.t. } x_i = E_{.j} \in \mathbb{R}^{\delta_e}} \text{with token } t_i \text{ indexed by } j \ in \ V$$

- ➔ *E* is a dense embedding matrix
- ➔ We can **learn a representation vector** for **each token in the vocabulary**

Why is an Embedding Layer much better?

Trainable Dense Embedding layers are a **"game changer"** for Deep Learning Models in NLP i.e. Generalization is much better compared to 1-hot encoding

Why? **(intuition)**

t and t' that have the embedding vectors (in E) x and x'. e.g. t = "dog" and t' = "cat"

Why is an Embedding Layer much better?

Trainable Dense Embedding layers are a **"game changer"** for Deep Learning Models in NLP i.e. Generalization is much better compared to 1-hot encoding

Why? (intuition)

t and t' that have the embedding vectors (in E) x and x'. e.g. t = "dog" and t' = "cat"

1. Let's assume that during training token the model has seen much less frequently ***cat*** than ***dog***

Why is an Embedding Layer much better?

Trainable Dense Embedding layers are a **"game changer"** for Deep Learning Models in NLP i.e. Generalization is much better compared to 1-hot encoding

Why? **(intuition)**

t and t' that have the embedding vectors (in E) x and x'. e.g. t = "dog" and t' = "cat"

1. Let's assume that during training token the model has seen much less frequently ***cat*** than ***dog***
2. But "luckily" ***x* and *x'* have similar embedding** vectors (i.e cos(x,x') ~ 1)

Why is an Embedding Layer much better?

Trainable Dense Embedding layers are a **“game changer“** for Deep Learning Models in NLP i.e. Generalization is much better compared to 1-hot encoding

Why? (intuition)

t and t' that have the embedding vectors (in E) x and x'. e.g. t = "dog" and t' = "cat"

1. Let’s assume that during training token the model has seen much less frequently ***cat*** than ***dog***
2. But “luckily” ***x* and *x’* have similar embedding** vectors (i.e cos(*x*,*x’*) ~ 1)
3. **When the model *dnn* sees**, at test time, ***cat*** it will be likely to model ***dog*** much better than in a 1-hot modeling case **by using this similarity**

How to initialize an Embedding Layer?

Similarly to all other parameters in a deep learning model

- Before starting training: **we can simply** **initialize the embedding matrix randomly**
- Before training, the similarity between **embedding word vectors is random**

How to initialize an Embedding Layer?

Similarly to all other parameters in a deep learning model

- Before starting training: **we can simply initialize the embedding matrix randomly**
- Before training, the similarity between **embedding word vectors is random**

Can we do better?

How to initialize an Embedding Layer?

Similarly to all other parameters in a deep learning model

- Before starting training: **we can simply** **initialize the embedding matrix randomly**
- Before training, the similarity between **embedding word vectors is random**

Can we do better?

➔ In previous lecture we have seen how to represent good dense embedding vector with skip-gram word2vec model

➔ **We can simply initialize our word embedding matrix with word2vec vectors**

How to initialize an Embedding Layer?

Initializing with a **pretrained embedding** layer was also a ***gamechanger*** for many NLP tasks and many Deep Learning architecture

Conditions to use a pretrained embedding layer:

- ➔ The token in our vocabulary must be in the training of the word2vec model
- ➔ For the one that were not seen, we can simply initialize them randomly
  - For LLMs it is even better to average some parts:
    - https://nlp.stanford.edu/~johnhew/vocab-expansion.html

Transfer Learning in NLP

Initializing with a **pretrained embedding** layer is also a ***game changer*** for many NLP tasks and many Deep Learning architecture

It is called Transfer Learning

Embedding Layer Summary

- Trainable Dense Embedding Layer are a *game changer* for Deep Learning Models
- Even more when we can use a pretrained embedding layers (e.g. with word2vec)
- They can be used with all Deep Learning Architectures
- **For all NLP tasks**

General DL Framework

Framework

We want to model $(X_1, .., X_T)$ i.e. find the correct label Y

$$dnn_\theta : \quad \mathbb{R}^{d,T} \quad \rightarrow \mathbb{R}^p \; or \; [|0, K|]^p$$
$$(X_1, .., X_T) \mapsto \hat{Y}$$

- Output space is $\mathbb{R}^p$ for **Regression** tasks
- Output space is $[|0, K|]^p$ for **Classification** tasks

Framework

We want to model $(X_1, .., X_T)$ i.e. find the correct label Y

$$dnn_\theta : \quad \mathbb{R}^{d,T} \rightarrow \mathbb{R}^p \; or \; [|0, K|]^p$$
$$(X_1, .., X_T) \mapsto \hat{Y}$$

Questions: when we do Deep Learning…

- **How do we define** dnn_θ **?**
- **How do we train** dnn_θ **with data ?**

Framework

Given a sequence of vectors $(X_1, .., X_T)$ we want to predict *Y*

$$dnn_\theta : \quad \mathbb{R}^{d,T} \rightarrow \mathbb{R}^p \; or \; [|0, K|]^p$$
$$(X_1, .., X_T) \mapsto \hat{Y}$$

Most Deep Learning Models:

- are parametric (i.e. $\theta \in \mathbb{R}^D$)
- defined as a composition of “simple” functions (linear & non-linear)
- are trained in an end-to-end fashion with backpropagation

NB: In Deep Learning, **the parametrization of *dnn*** is called the Architecture

Different Types of Architecture

How can we define our predictive function dnn_θ ?

➔ Multi-Layer Perceptron

➔ Recurrent Layers

➔ Attention Layers

➔ Self-Attention Layers (in a Transformer Architecture)

Different Types of Architecture

How can we define our predictive function dnn_θ ?

➔ Multi-Layer Perceptron

➔ Recurrent Layers

➔ Attention Layers

➔ Self-Attention Layers (in a Transformer Architecture)

How do we **train our model**? (i.e. estimate the parameters of the model)

➔ **Stochastic Gradient Descent** also called **backpropagation** in this context

Output Activation & Loss

Softmax Function

$$softmax(s) = \left(\frac{e^{s_i}}{\sum_k e^{s_k}}\right)_{i \in [|1, V|]}, \text{for } s \in \mathbb{R}^V$$

Loss Function

$$l(p, \hat{p}) = CE(p, \hat{p}) = \sum_{i \in [|0, V-1|]} p_i \, log(\hat{p_i})$$

Loss Functions

Based on the task we aim at modeling, we can use:

For Regression: Mean-Square Error

$$l(y, \hat{y}) = \|y - \hat{y}\|_2^2 = \sum_i (y_i - \hat{y}_i)^2 \text{ assuming } y_i,\ \hat{y}_i \in \mathbb{R}$$

For Classification: Cross-Entropy Loss

$$l(y, \hat{y}) = CE(y, \hat{y}) = \sum_i y_i \, log(\hat{y}_i) \text{ assuming } y_i,\ \hat{y}_i \in \ [0, 1]$$

Most NLP tasks will be based on the **Cross-Entropy loss**

Encoder - Decoder

Framework

We assume an input sequence of tokens $(x_1, .., x_T) \in V^T$
a target sequence of tokens $(y_1, .., y_{T'}) \in V'^{T'}$.

Framework

We assume an input sequence of tokens $(x_1, .., x_T) \in V^T$
a target sequence of tokens $(y_1, .., y_{T'}) \in {V'}^{T'}$.

Framework

We assume an input sequence of tokens $(x_1, .., x_T) \in V^T$
a target sequence of tokens $(y_1, .., y_{T'}) \in V'^{T'}$.

Our goal is to estimate:

$$\boxed{p_\theta(y_1, .., y_{T'} | x_1, .., x_T)}$$

Framework

We assume an input sequence of tokens $(x_1, .., x_T) \in V^T$
a target sequence of tokens $(y_1, .., y_{T'}) \in V'^{T'}$.

Our goal is to estimate:

$$p_\theta(y_1, .., y_{T'} | x_1, .., x_T)$$

We frame it as a classification task:

$$\boxed{\hat{y}_t = argmax_{y \in V'} \, p_\theta(y | (x_1, .., x_T), (y_t, .., y_{t-1})) \;\; \forall \, t \in [|1, T'|]}$$

What Architecture?

$$p_\theta(y_t|(x_1,..,x_T),(y_t,..,y_{t-1}))$$

We want to model an output sequence conditioned on an input *sequence*

With deep learning, we do that with: a encoder-decoder model

NB: also called “sequence to sequence” or “*seq2seq*”

The Encoder-Decoder Paradigm

$$p_\theta(y_t | (x_1, .., x_T), (y_t, .., y_{t-1}))$$

Intuition:

- **We know how to model a single sequence at a time with a DL model**
  E.g. with a LSTM or a Transformer
- **Here we want to model two sequences together**
  One conditioned on the other

The Encoder-Decoder Paradigm

$$\boxed{p_\theta(y_t|(x_1, .., x_T), (y_t, .., y_{t-1}))}$$

Intuition:

- **We know how to model a single sequence at a time with a DL model**
  E.g. with a LSTM or a Transformer
- **Here we want to model two sequences together**
  One conditioned on the other

➔ **Combine two Deep Learning Architectures together**

The Encoder-Decoder Paradigm

$$p_\theta(y_t | (x_1, .., x_T), (y_t, .., y_{t-1}))$$

Encode **input sequence**

$$\boxed{\begin{array}{rcl} enc_{\theta_e} : \quad V^T & \rightarrow & \mathbb{R}^T \\ (x_1, .., x_T) & \mapsto & (h_1, ..., h_T) \end{array}}$$

The Encoder-Decoder Paradigm

$$p_\theta(y_t | (x_1, .., x_T), (y_t, .., y_{t-1}))$$

***Encode* input sequence**

$$enc_{\theta_e} : \quad V^T \rightarrow \mathbb{R}^T$$
$$(x_1, .., x_T) \mapsto (h_1, ..., h_T)$$

***Decode* target sequence** given hidden states of the encoder

$$\boxed{\begin{aligned} dec_{\theta_d} : \quad \mathbb{R}^T \times V'^t &\rightarrow [0,1]^V \\ ((h_1, .., h_T), (y_1, .., y_t)) &\mapsto \hat{p} \end{aligned}}$$

The Encoder-Decoder Paradigm

How to integrate (h1,..hT) in the *decoder* ?

➔ **It depends what architecture is chosen for the *encoder* and the *decoder***

The Encoder-Decoder Paradigm

How to integrate (h1,..hT) in the *decoder* ?

➔ **It depends what architecture is chosen for the *encoder* and the *decoder***

- **RNN encoder-decoder** (possibly with an Attention Mechanism)
- **Transformer Model**

The Encoder-Decoder with RNNs

Recall: Vanilla RNN with *L'* and time step *sequence of length T'*

$$h_{i+1,t+1} = \varphi_i(W_i h_{i,t} + U_i h_{i+1,t} + b_i), \forall\, i \in [|1, L' - 1|]$$

$$\text{with } \; h_{1,t} = Emb(y_t) \; \text{ and } \; \hat{p_{t+1}} = h_{L',t+1} \quad \forall\, t \in [|1, T' - 1|]$$

$$\text{with } \; \varphi_{L'} = softmax$$

The Encoder-Decoder with RNNs

Given $(x_1, .., x_T)^T$ and $(y_1, .., y_t) \in V'^{T'}$, we predict $\hat{p}_{t+1}$, distribution over V' with:

Decoder

$$h_{dec,i+1,t+1} = \varphi_i(W_i' h_{dec,i,t} + U_i' h_{dec,i+1,t} + b_i' + \boxed{V_i h_{enc,L+1,T+1}}), \forall\, i \in [|1, L'|]$$

$$\text{with } \; h_{dec,1,t} = Emb(y_t) \; \text{ and } \; \hat{p_{t+1}} = h_{dec,L'+1,t+1} \quad \forall\, t \in [|1, T'|]$$

$$\text{with } \; \varphi_{L'} = softmax$$

The Encoder-Decoder with RNNs

Given $(x_1, .., x_T)^T$ and $(y_1, .., y_t) \in V'^{T'}$, we predict $\hat{p}_{t+1}$, distribution over V' with:

Decoder

$$h_{dec,i+1,t+1} = \varphi_i(W_i' h_{dec,i,t} + U_i' h_{dec,i+1,t} + b_i' + V_i h_{enc,L+1,T+1}), \forall\, i \in [|1, L'|]$$

$$\text{with } h_{dec,1,t} = Emb(y_t) \text{ and } \hat{p_{t+1}} = h_{dec,L'+1,t+1} \quad \forall\, t \in [|1, T'|]$$

$$\text{with } \varphi_{L'} = softmax$$

- Decoder of *with **L' layer**, with parameters **W'i, U'i, b', Vi for all i***
- It decodes sequentially the target sequence

The Encoder-Decoder with RNNs

Given $(x_1, .., x_T)^T$ and $(y_1, .., y_t) \in V'^{T'}$, we predict $\hat{p}_{t+1}$, distribution over V' with:

Decoder

$$h_{dec,i+1,t+1} = \varphi_i(W'_i h_{dec,i,t} + U'_i h_{dec,i+1,t} + b'_i + \boxed{V_i h_{enc,L+1,T+1})}, \forall\, i \in [|1, L'|]$$

$$\text{with } h_{dec,1,t} = Emb(y_t) \text{ and } \hat{p_{t+1}} = h_{dec,L'+1,t+1} \quad \forall\, t \in [|1, T'|]$$

$$\text{with } \varphi_{L'} = softmax$$

- Decoder of *with **L' layer**, with parameters **W'i, U'i, b', Vi for all i***
- It decodes sequentially the target sequence
- It is conditioned on **the input sequence through the encoding output**

The Encoder-Decoder with RNNs

Given $(x_1, .., x_T)^T$ and $(y_1, .., y_t) \in V'^{T'}$, we predict $\hat{p}_{t+1}$, distribution over V' with:

Decoder

$$h_{dec,i+1,t+1} = \varphi_i(W_i' h_{dec,i,t} + U_i' h_{dec,i+1,t} + b_i' + \boxed{V_i h_{enc,L+1,T+1}}), \forall\, i \in [|1, L'|]$$

$$\text{with } h_{dec,1,t} = Emb(y_t) \text{ and } \hat{p_{t+1}} = h_{dec,L'+1,t+1} \quad \forall\, t \in [|1, T'|]$$

$$\text{with } \varphi_{L'} = softmax$$

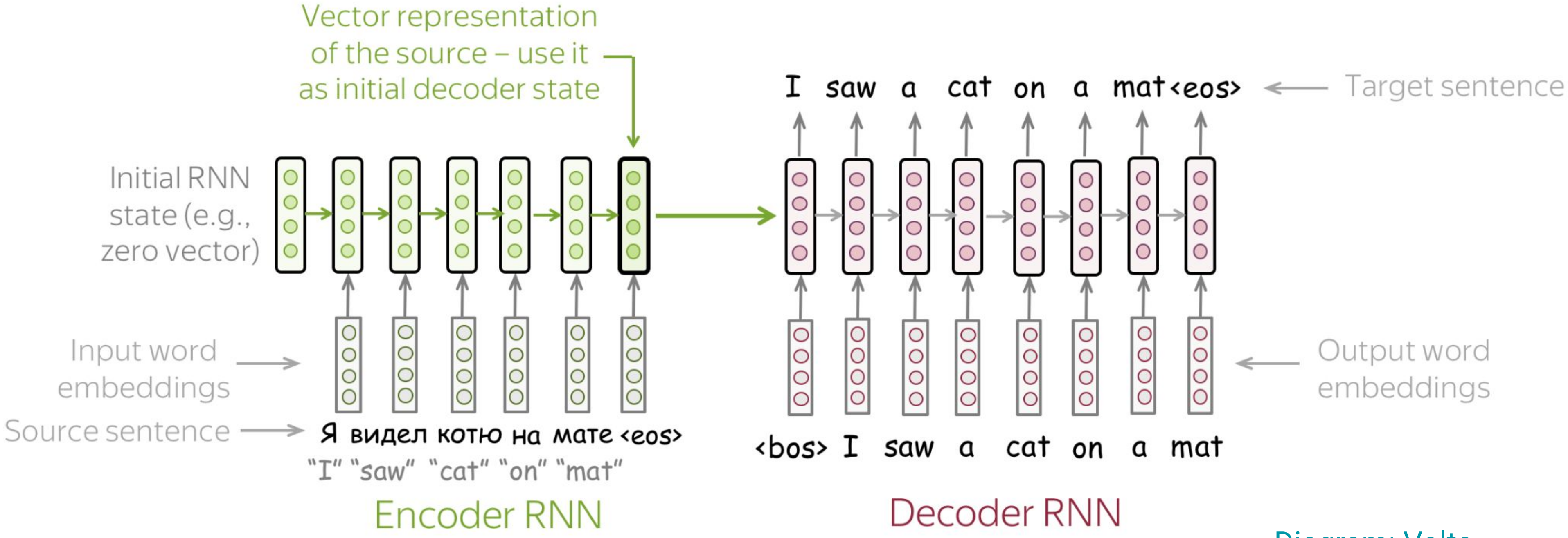
Encoder: also a RNN that encodes the input sequence in a fixed vector

$$\boxed{h_{enc,i+1,t+1}} = \varphi_i(W_i h_{enc,i,t} + U_{enc,i} h_{i+1,t} + b_{enc,i})$$

$$\forall\, i \in [|1, L|]\ \forall\, t \in [|1, T|] \quad \text{with } h_{enc,1,t} = Emb(x_t)$$

The Encoder-Decoder with RNNs

Simple RNN-based Encoder-Decoder Model:



[Diagram: Volta]()

The Encoder-Decoder Training

With Backpropagation

1. **We feed the model with both the input and output sequence**
2. **We compute the loss based on the “gold“ output sequence**
3. **We update all the parameters of the model with backpropagation**

Diagram: Volta

The Encoder-Decoder with RNNs: Limits

Limits: ***At test time in the encoder-decoder that we introduced***
The input sequence has a fixed representation ($h_{enc,L+1,T+1}$ **is fixed)**

The Encoder-Decoder with RNNs: Limits

Limits: ***At test time in the encoder-decoder that we introduced***
The input sequence has a fixed representation ($h_{enc,L+1,T+1}$ is fixed)

Example:

Je vois un chat sur un matelas ⇒ *I see a cat on a mat*

The Encoder-Decoder with RNNs: Limits

Limits: *At test time in the encoder-decoder that we introduced*
The input sequence has a fixed representation ($h_{enc,L+1,T+1}$ **is fixed)**

Example:

Je vois un chat sur un matelas ⇒ *I see a cat on a mat*

Step 4:
Given: *Je vois un chat sur un matelas* ⇒ *cat*
Step 7:
Given: *Je vois un chat sur un matelas* ⇒ *mat*

The Encoder-Decoder with RNNs: Limits

Limits: ***At test time in the encoder-decoder that we introduced***
The input sequence has a fixed representation ($h_{enc,L+1,T+1}$ is fixed)

Example:

Je vois un chat sur un matelas ⇒ *I see a cat on a mat*

Step 4:
Given: *Je vois un chat sur un matelas* ⇒ *cat*
Step 7:
Given: *Je vois un chat sur un matelas* ⇒ *mat*

➔ **We need "decoding-dependent" representation of the input sequence**

How to build more flexible encoder-decoder?

Solution 1:

- **Integrate an Attention Mechanism in a RNN-based encoder-decoder**

Solution 2:

- **Use an Encoder-Decoder Transformer**

In lab you will only make a classic seq2seq!

To Be Seen in Lab

- Classic Unsupervised learning to explore data (K-Means)
- PyTorch tutorial
- Unsupervised learning with PyTorch (Autoencoder)