Language Modeling

Pierre Colombo

pierre.colombo@centralesupelec.fr

MICS - CentraleSupelec

Introduction To (Deep) Natural Language Processing

Final Project

Lectures Outline

1. The Basics of Natural Language Processing

2. Representing Text with Vectors

3. Deep Learning Methods for NLP

4. **Language Modeling**

5. Sequence Labelling (Sequence Classification)

6. Sequence Generation Tasks

Outline

- **Causal Language Model with LSTM**

- **Causal Language Model with Transformers**

- **Evaluation**

Framework

Given $(t_1, .. t_D) \in V^D$, our goal is to estimate: $\quad P(t_{n+1} | t_1, .., t_n)$

We saw how to estimate that with n-gram models

To do better:
➔ Use a Deep-Learning Model

Why Deep Learning Models for LM?

Motivations

Theoretical Insights
- Deep Learning Models are **universal approximators**
- Recurrent Neural Network can in theory model infinite context

Practical Insights
- They can be trained on **very large amount of data**
- They can use **continuous representation** of input tokens capturing the *distributional hypothesis* efficiently

Framework

Given $(t_1, ..t_D) \in V^D$, **our goal is to estimate** :

$$p(t_{n+1}|t_1, .., t_n)$$

Framework

We want to find dnn_θ

$$dnn_\theta : \qquad V^D \qquad \rightarrow \quad [0,1]^V$$

$$(t_1, .., t_D) \mapsto \hat{p}$$

s.t. $\hat{p} = (p_i)_{i \in [|0, V-1|]}, \ \forall \ i \ \ p_i \in [0,1]$ and $\sum_i p_i = 1$

Design Questions

★ **What tokenization ?**

★ **What output activation function and loss?**

★ **What architecture?**

★ **How do you represent a token to feed the model?**

Design Questions

★ **What tokenization ?**

★ **What output activation function and loss?**

★ **What architecture?**

★ **How do you represent a token to feed the model?**

NB: Questions to ask for any NLP task approached with Deep Learning

Tokenization

- **Word-Level Tokenization: *e.g.* *"I, am, going"***
Pros: Easy to segment, Words are Linguistic Units
Cons: Out-of-Vocabulary (OOV) problem

Tokenization

- **Word-Level Tokenization: *e.g.* *"I, am, going"***

Pros: Easy to segment, Words are Linguistic Units

Cons: Out-of-Vocabulary (OOV) problem

- **Character-Level Tokenization: *e.g.* *"I, ,a, ,m, ,g,o,i,n,g"***

Pros: No OOV problem

Cons: Very long sequences

Tokenization

- **Word-Level Tokenization: *e.g.* *"I, am, going"***

Pros: Easy to segment, Words are Linguistic Units

Cons: Out-of-Vocabulary (OOV) problem

- **Character-Level Tokenization: *e.g.* *"I, ,a, ,m, ,g,o,i,n,g"***

Pros: No OOV problem

Cons: Very long sequence

- **SentencePiece Tokenization: *"_I, _am, _go, ing"***

Frequent "words" become tokens and infrequent ones are split into subwords

Tokenization

- **Word-Level Tokenization:** *e.g. "I, am, going"*
Pros: Easy to segment, Words are Linguistic Units
Cons: Out-of-Vocabulary (OOV) problem

- **Character-Level Tokenization:** *e.g. "I, ,a, ,m, ,g,o,i,n,g"*
Pros: No OOV problem
Cons: Very long sequence

- **SentencePiece Tokenization:** *"_I, _am, _go, ing"*
Frequent "words" are kept intact and infrequent ones are split into subwords
NB: SentencePiece is the most popular tokenization algorithm for language models

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

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

Output Activation & Loss

Softmax Function

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

Loss Function

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

NB: We will use them in all the tasks we will cover in this course

Architecture

- The Multi-Layer Perceptron

- Recurrent Neural Network: LSTM Model

- The Transformer

MLP for Language Modeling

Recall: The **MLP** works **on unidimensional data** (e.g. dimension d)

$$dnn_\theta : \qquad \mathbb{R}^d \qquad \rightarrow [0,1]^V$$

$$X \mapsto \hat{Y}$$

MLP for Language Modeling

Recall: The **MLP** works **on unidimensional data** (e.g. dimension d)

$$dnn_\theta : \qquad \mathbb{R}^d \qquad \to [0,1]^V$$

$$X \mapsto \hat{Y}$$

How can we model $(t_1, .. t_D) \in V^D$ with D **arbitrary high ?**

MLP for Language Modeling

Recall: The **MLP** works **on unidimensional data** (e.g. dimension d)

$$dnn_\theta : \quad \mathbb{R}^d \quad \rightarrow [0,1]^V$$
$$X \mapsto \hat{Y}$$

How can we model $(t_1, .. t_D) \in V^D$ with D **arbitrary high ?**

➔ Truncate input sequences: **Fixed-Window** **Language Modeling**

How to represent input tokens?

Solution 1

How to represent input tokens?

Solution 1
1. **1-Hot Encoding**

How to represent input tokens?

Solution 1: 1-Hot Encoding

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

$$\text{movie} = [1, 0, \ldots, 0, 0, 0]$$
$$\text{hotel} = [0, 1, \ldots, 0, 0, 0]$$
$$\ldots$$
$$\text{art} = [0, 0, \ldots, 0, 0, 1]$$

2. **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

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

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

First hidden layer:

assuming tanh as the activation function, dimension $\quad \delta$

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

1-Hot Encoding as inputs

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

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

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 an Dense Embedding Layer

How to represent input tokens?

Solution 2: Integrate an 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 an 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 $E_{\cdot j}$ (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**

How to represent input tokens?

Solution 2: **Integrate an 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**
See how to define it in torch

Dense Embedding Layer

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

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

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

Dense Embedding Layer

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

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

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

Dense Embedding Layer

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

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

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

➔ 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 $\cos(x, x') \sim 1$)

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 $\cos(x,x') \sim 1$)
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**

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 between 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 between random**

Can we do better?

➔ In lecture 2 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

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)

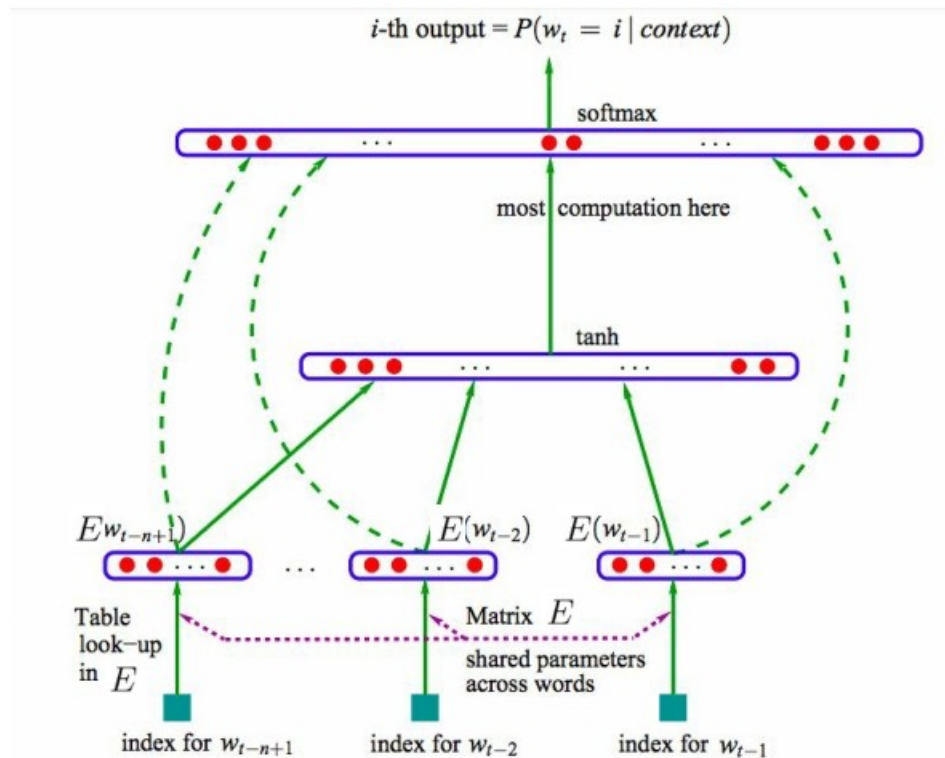- They can be used with all Deep Learning Architectures

- **For all NLP tasks**

MLP for Fixed-Window Language Modeling

$$\hat{t}_{n+1} = argmax_{t \in V} \, p(t|t_1, .., t_n)$$

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

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

MLP for Fixed-Window Language Modeling

Limits of MLP for language modeling

- *Windows* is Fixed

➔ Use Recurrent Neural Network (e.g. LSTM)

Recurrent Neural Network for LM

Recall:

$$h_{i+1,t+1} = \varphi_i(W_i h_{i,t} + U_i h_{i+1,t} + +b_i), \forall\, i \in [|1, L-1|]$$

$$\text{with } h_{1,t} = X_t \text{ and } \hat{Y}_t = dnn(X_t) = h_{L,t} \,\forall\, t \in [|1, T-1|]$$

Recurrent Neural Network for LM

Recall:

$$h_{i+1,t+1} = \varphi_i(W_i h_{i,t} + U_i h_{i+1,t} + + b_i), \forall\, i \in [|1, L-1|]$$

$$\text{with } h_{1,t} = X_t \text{ and } \hat{Y}_t = dnn(X_t) = h_{L,t} \,\forall\, t \in [|1, T-1|]$$

For Language Modeling , like we did for the MLP
- **We use an Embedding layer**
- **We use the softmax layer as output**

Recurrent Neural Network for LM

For an sequence of token

$$h_{i+1,t+1} = \varphi_i(W_i h_{i,t} + U_i h_{i+1,t} + b_i), \forall\, i \in [|1, L|]\, \forall\, t \in [|1, T|]$$

$$\text{with } h_{1,t} = Emb(x_t) \text{ and } \hat{p_{t+1}} = h_{L+1,t+1}$$

$$\text{with } \varphi_L = softmax$$

We estimate $\hat{p_{t+1}} = p(x_{t+1}|x_1, .. x_t)$ **directly with the RNN**

Recurrent Neural Network for LM

Written in a more synthetic way

$$h_{i+1,t+1} = RNN_i(h_{i,t}, h_{i+1,t}), \forall i \in [|1, L|] \, \forall t \in [|1, T|]$$

$$\text{with } h_{1,t} = Emb(x_t) \text{ and } \hat{p_{t+1}} = h_{L+1,t+1}$$

$$\text{with } \varphi_L = softmax$$

We estimate $\quad \hat{p_{t+1}} = p(x_{t+1}|x_1, ..x_t) \quad$ directly with the RNN

Recurrent Neural Network for LM
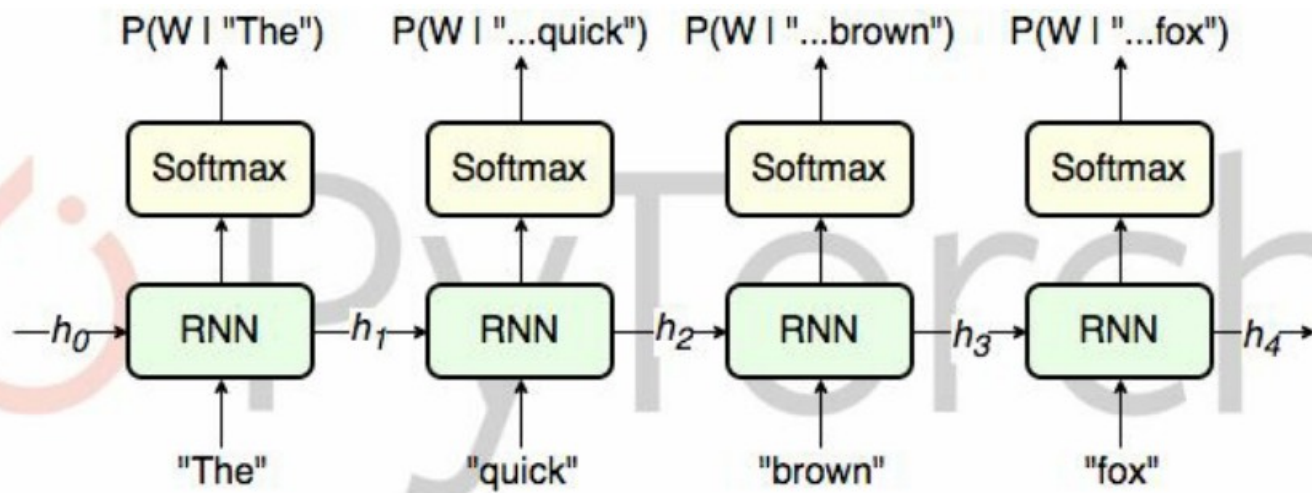
With a LSTM, we have a dependency on the *Cell* Vector:

$$h_{i+1,t+1}, C_{i+1,t+1} = LSTM_i(h_{i,t}, h_{i+1,t}, C_{i+1,t}), \forall\, i \in [|1, L|]\, \forall\, t \in [|1, T|]$$

$$\text{with } h_{1,t} = Emb(x_t) \text{ and } \hat{p_{t+1}} = h_{L+1,t+1}$$

$$\text{with } \varphi_L = softmax$$

We estimate $\hat{p_{t+1}} = p(x_{t+1}|x_1, ..x_t)$ **directly with the LSTM**

Recurrent Neural Network for LM

Transformer for Causal Language Modeling

Inputs: Transformers requires a fixed sequence at input (we note it $\mathcal{T}$)

Let's assume we have a sequence $(x_1, ... x_T)$

We simply append it with a ***PADDING*** token

We append $(x_{T+1}, .., x_{\mathcal{T}})$ with $\quad x_t = [PAD] \, \forall \, t \geq T + 1$

We get a sequence of length $\mathcal{T} : (x_1, ... x_{\mathcal{T}})$

We make the model ignore those tokens by setting the softmax scores to 0 in the self-attention

Transformer for Causal Language Modeling

Input Embeddings:

$$(x_1, \ldots x_{\mathcal{T}})$$

Embedding:

$$(Emb(x_1), \ldots Emb(x_{\mathcal{T}}))$$

such that $Emb(x_i) = PositionEmb(x_i) + TokenEmb(x_i)$

Transformer for Causal Language Modeling

Given a sequence of tokens: $(x_1, .., x_T)$

$$H_{i+1} = FeedForward(A_{i+1}) \text{ and } A_{i+1} = SelfAttention(H_i) \quad \forall i \in [|1, L|]$$

$$\text{with} \quad SelfAttention(H_i) = softmax(\frac{Q\,K^T}{\sqrt{\delta_K}})V$$

$$H_0 = (Emb(x_1), ... Emb(x_{\mathcal{T}}))$$

Transformer for Causal Language Modeling

Given a sequence of tokens: $(x_1, .., x_T)$

$$\mathbf{H}_{i+1} = FeedForward(A_{i+1}) \text{ and } A_{i+1} = SelfAttention(H_i) \quad \forall\, i \in [|1, L|]$$

$$\text{with} \quad SelfAttention(\mathbf{H}_i) = softmax\left(\frac{Q\,K^T}{\sqrt{\delta_K}}\right)V$$

$$\mathbf{H}_0 = (Emb(x_1), ... Emb(x_\mathcal{T}))$$

- **Residual Connection** and **Layer Norm** are not included in those equations

- **FeedForward is position-wise** two layer MLP (i.e. applied independently from the position of each hidden vector)

- Self-Attention is actually a **Multi-Head Self-Attention**

The Transformer Architecture

The Transformer Architecture is

- Stack of [Self-Attention + FF Layer]

- With Skip-Layer and Normalization Layers in between

- Encoding the position with positional vector
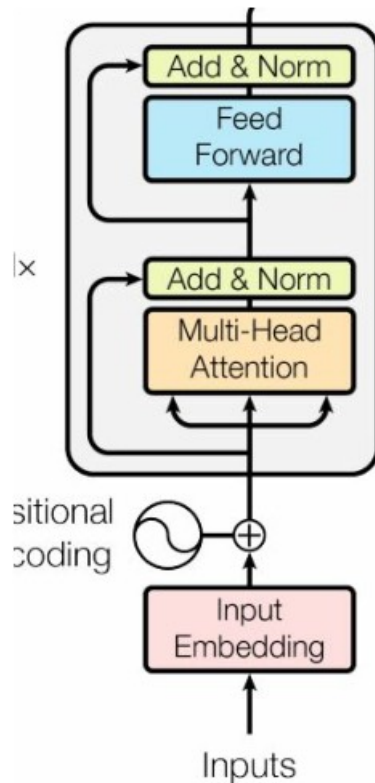
Transformer for Causal Language Modeling

Given a sequence of tokens: $(x_1, .., x_T)$

$\Rightarrow$ **Last element of the sequence of the hidden states** of the last layer fed to a softmax

$$p_{\hat{x_{T+1}}} = softmax(h_T) \quad \forall t \leq T$$

Training

- We train on large corpus of text (+1G of text)

- We train them with backpropagation

- We usually do " teacher-forcing ", for each step, we use the "gold" sequence" and not the predicted one.

- For Transformers, we train on sequences as long as possible (~1000 tokens)

Evaluation

$$perplexity(\hat{x}, x) = 2^{-\sum_i x_i log(\hat{x}_i)}$$

The lower the perplexity the better the language model

Empirical Performance

Language Model Performance Comparison

➔ **Transformer Models outperform LSTM-based models**

Lecture Summary

- **Causal Language Modeling Framework**

- **Representing input tokens for language modeling**

- **Recurrent Neural Network for Language Modeling**

- **Transformer for Language Modeling**

Bibliography and Acknowledgment

All these class have been taken from https://nlp-ensae.github.io/materials/ and is taken from Benjamin Muller