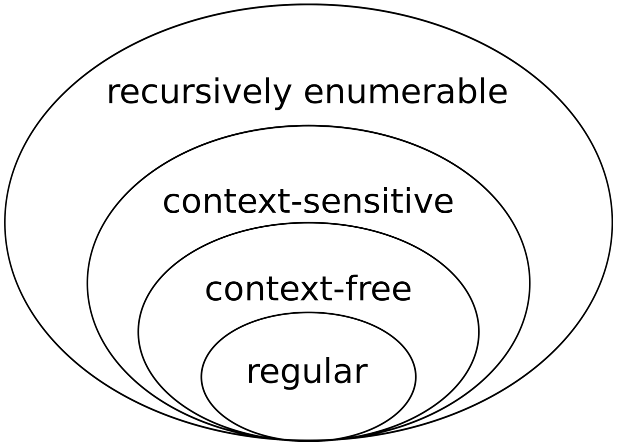
(Modern) Language Models

Fabrizio Silvestri

How to Model a Language?

- You may recall from early classes:
  - Chomsky' Hierarchy
  - Formally defined
  - In many cases automata can recognize a language
    - FSAs
    - Push-down Automata
    - Turing Machines
- In general NLP makes use of Statistical Language Models (LMs)
  - Better suited for natural language



$$P(w_1 w_2 \ldots w_T) = \prod_{i=1}^{n} P(w_1)P(w_2|w_1)P(w_i|w_1 w_2 \ldots w_{i-1})$$

Applications of LMs

- Information Retrieval
  - Each document is generated by a different LM
  - Queries are generated by a LM
  - Which of the documents' LMs are most similar to the query's LM?
- Speech Recognition
  - "I ate eight apples" more likely than "Aye eight ate apples"?
- Spell Correction
  - "Google" more likely than "Googel"?
- Natural Language Generation
  - What's the most likely continuation of this <X>?
    - Question, Sentence, Phrase, Paragraph, …
- Translation
  - ES → "El cafe negro me gusta mucho"
    - IT → "Il caffè nero mi piace molto"
    - EN → "I like black coffee very much" vs "The coffee black me pleases much"
- …

Formal LM Definition

- Goal:
  - Define a probability distribution over “tokens” to be able to measure likelihood of sequences s of tokens $p\left(s\right) = p(t_1, t_2, \ldots, t_n)$
- This way NLP tasks can be seen as computing a probability score for components of tasks
  - Information Retrieval
    - Probability of a document given a query: $p\left(d|q\right) = p\left(q|d\right)\frac{p(d)}{p(q)}$
  - Translation
    - Noisy channel: $p\left(d_{eng}|d_{ita}\right) \propto p\left(d_{eng}, d_{ita}\right) = p\left(d_{ita}|d_{eng}\right)p\left(d_{eng}\right)$

K-Gram Language Models

- Relative Frequency Estimate

$$p\left(\text{Computers are useless, they can only give you answers}\right) = \frac{\text{Count}\left(\text{Computers are useless, they can only give you answers}\right)}{\text{Count}\left(\text{all sentences ever spoken or written}\right))}$$

- Estimator is unbiased but, can we use it?
  - "*Count(all sentences ever spoken or written)*" is extremely large.
    - Even assuming $|V| = 10^5$ and max $|x| = 10$ we have that Count(...) is 10^{50}.
- Let's consider the probability of the sequence of tokens: $p\left(s\right) = p(t_1, t_2, \ldots, t_n)$
  - Which formally is: $p\left(t_1, \ldots, t_n\right) = p\left(t_1\right) p\left(t_2|t_1\right) p\left(t_3|t_2, t_1\right), \ldots, \left(t_n|t_1, t_2, \ldots, t_{n-1}\right)$
- We simplify by considering only subsequences of length *k*:

$$p(t_m|t_{m-1}, \ldots, t_1) \sim p(t_m| \underbrace{t_{m-1}, \ldots, t_{m-k+1}}_{\text{k-gram}})$$

SAPIENZA
UNIVERSITÀ DI ROMA

Berkeley Restaurant Project (BRP) sentences

- can you tell me about any good cantonese restaurants close by
- mid priced thai food is what i’m looking for
- tell me about chez panisse
- can you give me a listing of the kinds of food that are available
- i’m looking for a good place to eat breakfast
- when is caffe venezia open during the day

BRP - Raw Bigram Counts (out of 9,222 sentences)

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

BRP - Raw Bigram Probabilities

Unigram Counts

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

Bigram probabilities

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Bigram estimates of sentence probabilities

- P(<s> I want english food </s>) =
  - P(I | <s>) *
  - P(want | I) *
  - P(english | want) *
  - P(food | english) *
  - P(</s> | food) =
- 0.00031

P(lunch | want) = P(chinese | want) = 0

I *want lunch*

I *want chinese* food

K-Gram Language Models

```python
def main():
    nltk.download('reuters')
    nltk.download('punkt')
    text_sentences = reuters.sents()

    model = init_model()
    model = count_cooccurrences(model)
    model = generate_probabilities(model)

    sentence_to_be_continued = 'market analysis'

    current_sentence = sentence_to_be_continued
    n_continuations = 20
    for iteration in range(n_continuations):
        current_sentence = continue_sentence_greedy(model, current_sentence)
    print('Greedy:\t{}'.format(current_sentence))

    current_sentence = sentence_to_be_continued
    n_continuations = 20
    k = 5
    for iteration in range(n_continuations):
        current_sentence = continue_sentence_top_k(model, current_sentence, k)
    print('top-{}:\t{}'.format(k, current_sentence))
```

SAPIENZA
UNIVERSITÀ DI ROMA

K-Gram Language Models

```python
def count_cooccurrences(model):
    # Count frequency of co-occurrence
    for sentence in reuters.sents():
        for w1, w2, w3 in trigrams(sentence, pad_right=True, pad_left=True):
            model[(w1, w2)][w3] += 1

    return model


def generate_probabilities(model):
    # Let's transform the counts to probabilities
    for w1_w2 in model:
        total_count = float(sum(model[w1_w2].values()))
        for w3 in model[w1_w2]:
            model[w1_w2][w3] /= total_count

    return model
```

Greedy NL Generation

```python
def continue_sentence_greedy(model, prefix):
    # Use a greedy approach to generate sentences
    new_sentence = prefix
    last_bigram = prefix.lower().split()[-2:]
    last_bigram = (last_bigram[0], last_bigram[1])
    alternatives = dict(model[last_bigram])
    if len(alternatives) > 0:
        continuation = max(alternatives.items(), key=operator.itemgetter(1))[0]
        if continuation:
            new_sentence = ' '.join([new_sentence, continuation])

    return new_sentence
```

Top-K NL Generation

```python
def continue_sentence_top_k(model, prefix, k):
    # Use a top-k approach to generate sentences
    new_sentence = prefix
    last_bigram = prefix.lower().split()[-2:]
    last_bigram = (last_bigram[0], last_bigram[1])
    alternatives = dict(model[last_bigram])
    if len(alternatives) > 0:
        k = min(len(alternatives), k)
        continuations = nlargest(k, alternatives, key=alternatives.get)
        continuation = random.choice(continuations)
        if continuation:
            new_sentence = ' '.join([new_sentence, continuation])

    return new_sentence
```

Examples of Text Generated by N-Gram LMs

- ‘today they’
  - Greedy: **today they** found the United states since the beginning of the company ' s & lt ; BP
  - top-5: **today they** found the United states will make no contribution to this process .
- ‘the beginning’
  - Greedy: **the beginning** of the company ' s & lt ; BP
  - top-5: **the beginning** of a share for the current level , despite massive central bank , the company .
- ‘news articles’
  - Greedy: **news articles**
  - top-5: **news articles**
- ‘the base’
  - Greedy: **the base** rate cut , and the U
  - top-5: **the base** in data storage subsystems for the current level ' is rather confident currency stability will benefit all foreign meat processing

Bigram estimates of sentence probabilities

- P(<s> I want english food </s>) =
  - P(I | <s>) *
  - P(want | I) *
  - P(english | want) *
  - P(food | english) *
  - P(</s> | food) =
- 0.00031

P(lunch | want) = P(chinese | want) = 0

I *want lunch*

I *want chinese* food

Smoothing

- What if p(w) = 0?
- **Smoothing** → add an imaginary "pseudo-count" to avoid situations where the probability of a token is zero
- Lidstone smoothing:

$$p_{\text{smooth}}\left(w_m | w_{m-1}\right) = \frac{\text{count}\left(w_{m-1}, w_m\right) + \alpha}{\sum_{w' \in \mathcal{V}} \text{count}\left(w_{m-1}, w'\right) + |\mathcal{V}|\alpha}$$

- Laplace's smoothing (*add one*) → $\alpha = 1$
- Jeffreys-Perks law → $\alpha = 0.5$
- Kneser-Ney smoothing → considers $p_{\text{continuations}}$

BRP - Laplace Smoothing

	i	want	to	eat	chinese	food	lunch	spend
i	6	828	1	10	1	1	1	3
want	3	1	609	2	7	7	6	2
to	3	1	5	687	3	1	7	212
eat	1	1	3	1	17	3	43	1
chinese	2	1	1	1	1	83	2	1
food	16	1	16	1	1	5	1	1
lunch	3	1	1	1	1	2	1	1
spend	2	1	2	1	1	1	1	1

BRP - Laplace Smoothed Bigram Probabilities

Unigram Counts

i	want	to	eat	chinese	food	lunch	spend
2534	928	2418	747	159	1094	342	279

Bigram probabilities

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.21	0.00025	0.0025	0.00025	0.00025	0.00025	0.00075
want	0.0022	0.00042	0.26	0.00084	0.0029	0.0029	0.0025	0.00084
to	0.00083	0.00026	0.0013	0.18	0.00078	0.00026	0.0018	0.055
eat	0.00046	0.00046	0.0014	0.00046	0.0078	0.0014	0.02	0.00046
chinese	0.0063	0.00062	0.00062	0.00062	0.00062	0.052	0.0012	0.00062
food	0.014	0.00039	0.0063	0.00039	0.00079	0.002	0.00039	0.00039
lunch	0.0059	0.00056	0.00056	0.00056	0.00056	0.0011	0.00056	0.00056
spend	0.0036	0.00058	0.0012	0.00058	0.00058	0.00058	0.00058	0.00058

Reconstituted Counts

$$c^*(w_{n-1}w_n) = \frac{[C(w_{n-1}w_n)+1] \times C(w_{n-1})}{C(w_{n-1})+V}$$

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.19

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.19

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Backoff and Interpolation

- Backoff
  - Use n-grams if you have enough evidence
  - Otherwise → Use (n-1)-grams if you have enough evidence of this kind
  - …
  - Finally → Use unigrams if you have enough evidence of this kind
- Katz Backoff
- Interpolation
  - Mix n-grams, (n-1)-grams, …, unigrams

Stupid Backoff

Interpolation works better

Linear Interpolation

- Simple. E.g., tri-gram
  - $P'(w_n|w_{n-1}, w_{n-2}) = \boldsymbol{\lambda}_1 P(w_n|w_{n-1}, w_{n-2}) + \boldsymbol{\lambda}_2 P(w_n|w_{n-1}) + \boldsymbol{\lambda}_3 P(w_n)$
    - $\boldsymbol{\lambda}_1 + \boldsymbol{\lambda}_2 + \boldsymbol{\lambda}_3 = 1$
- Lambdas conditional on context:
  - $P'(w_n|w_{n-1}, w_{n-2}) = \boldsymbol{\lambda}_1(w_{n-1}, w_{n-2})P(w_n|w_{n-1}, w_{n-2}) + \boldsymbol{\lambda}_2(w_{n-1}, w_{n-2})P(w_n|w_{n-1}) + \boldsymbol{\lambda}_3(w_{n-1}, w_{n-2})P(w_n)$

- To set $\boldsymbol{\lambda}$'s one can use a held-out dataset

Train	Held-Out	Test

OOV: Out Of Vocabulary

- If we know all the words in advanced
  - Vocabulary V is fixed
  - Closed vocabulary task
- Often we don’t know this
  - Out Of Vocabulary (OOV) words
  - Open vocabulary task
- Instead: create an unknown word token <UNK>
  - Training of <UNK> probabilities
  - Create a fixed vocabulary V
  - At text normalization phase, any training word not in V changed to <UNK>
  - Treat <UNK> like a normal word
  - At inference time use UNK probabilities for any word not in V

Language Models in IR: The Query Likelihood LM

- Given a query *q* the goal is to compute *P(d|q)* for each *d* in a collection
- Each document *d* is represented by a language model M_d
- Bayes rules is used to compute *P(d|q)* as *P(d)/P(q) * P(q|d)*
    - *P(d)* is a prior on document. We can consider it as uniform, i.e., constant
    - *P(q)* is the same for all the document
- We can conclude that *P(d|q) ~ P(q|d)*
    - Generative model for queries given a document

Estimating $P(q|M_d)$

- Given the LM for *d*, the formula then becomes $P(q|M_d)$ and we use multimodal unigram LM
  - $P(q|M_d) = K_q \prod_{t \epsilon V} P(t|M_d)$
  - K_q is a normalizing factor constant for each query *q*, and we can ignore it
- $P(t|M_d)$ can be estimated using MLE
  - $\prod_{t \epsilon V} P(t|M_d) = \prod_{t \epsilon q} P(t|M_d) = \prod_{t \epsilon q} tf(t,d)/len(d)$
- Smoothing can be applied to reduce the impact of 0 probabilities, or...
  - $P(t|d) = \boldsymbol{\lambda} P_{MLE}(t|M_d) + (1 - \boldsymbol{\lambda}) P_{MLE}(t|M_c)$;
    - M_c being the language model associated with the whole collection

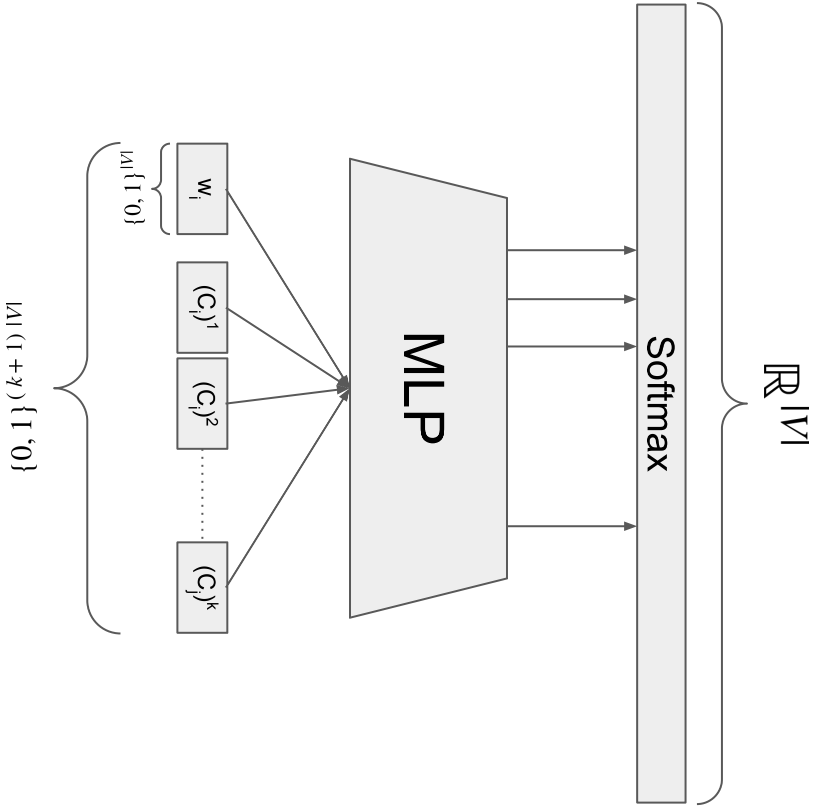
Is that it?

Neural Language Models

The Key Idea

- Model the problem as a discriminative learning task
  - P(word | context; θ)
- Predict the occurrence of a word w_i given a set of words C_j, as
  - $$P\left(w_i|C_j\right) = \frac{e^{\vartheta_{w_i} \cdot \vartheta_{C_j}}}{\sum_{w' \in V} e^{\vartheta_{w'} \cdot \vartheta_{C_j}}}$$
  - $$\text{softmax}(\vec{z})_i = \frac{e^{z_i}}{\sum_{1 \le j \le K} e^{z_j}}$$
  - $$P\left(-|C_j\right) = \text{softmax}\left(\vartheta_{w_1} \cdot \vartheta_{C_j}, \vartheta_{w_2} \cdot \vartheta_{C_j}, \ldots, \vartheta_{w_{|V|}} \cdot \vartheta_{C_j}\right)$$

Simple Idea: Use an MLP



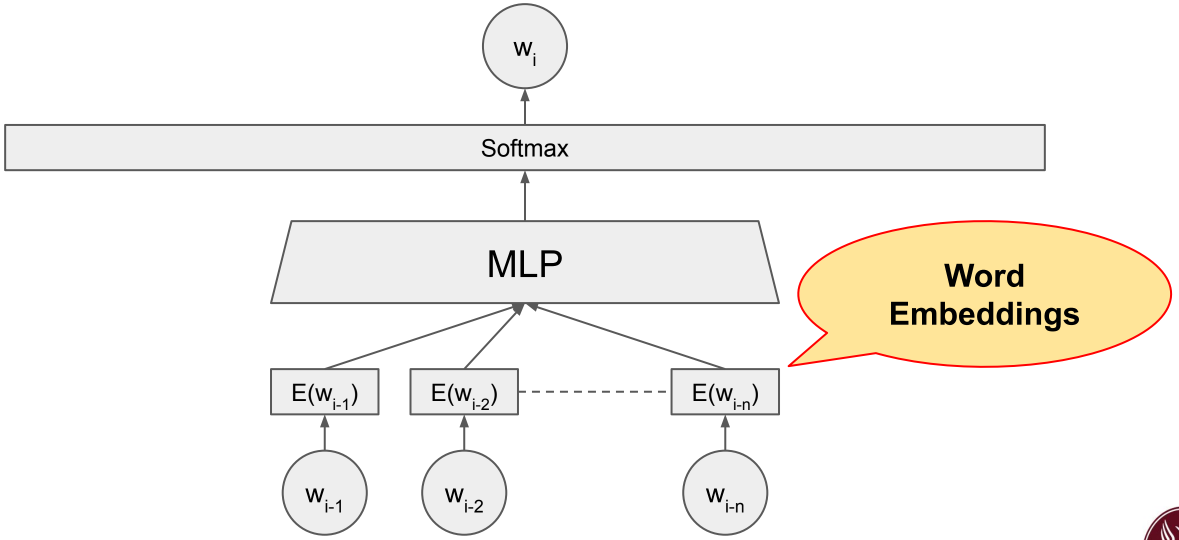
- Input is a concatenation of *w* with each context word $(C_i)^j$
    - 1-hot encoding: overall $(k+1)|V|$ (bits)
- Output is $|V|$ real numbers
- The MLP, then, uses a matrix of $(k+1)|V|^2 + |V|$ parameters.
    - E.g., for a (very small) vocabulary of 10^3 words, and a context of 10 words we will need $11*10^6+10^3 \sim 10^7$ parameters for a single layer MLP.
        - If $|V| = 10^5$ (more realistic), number of parameters is $\sim 10^{11}$

SAPIENZA UNIVERSITÀ DI ROMA

Word Embeddings (a quick intro)

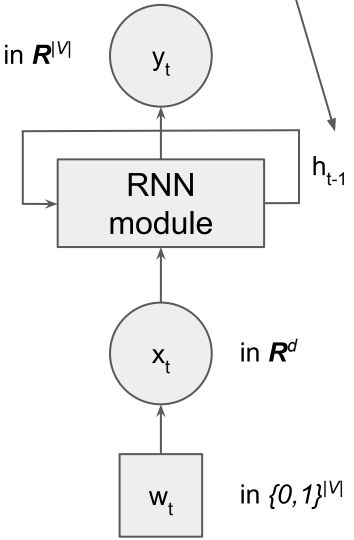
Let C, the context, be the “sequence” of n preceding tokens to a word w_i
We want to compute $P(w_i \mid w_{i-1}, w_{i-2}, \dots, w_{i-n})$

```python
    def forward(self, input):
        emb = self.embedding(input)
        x = emb.reshape(emb.size()[0], self.embedding_dim * self.context_size)
        x = self.mlp(x)

        return x
```



Recurrent Neural Network LMs

What is an RNN?



h_t is in $\boldsymbol{R}^k$, k being the hidden size

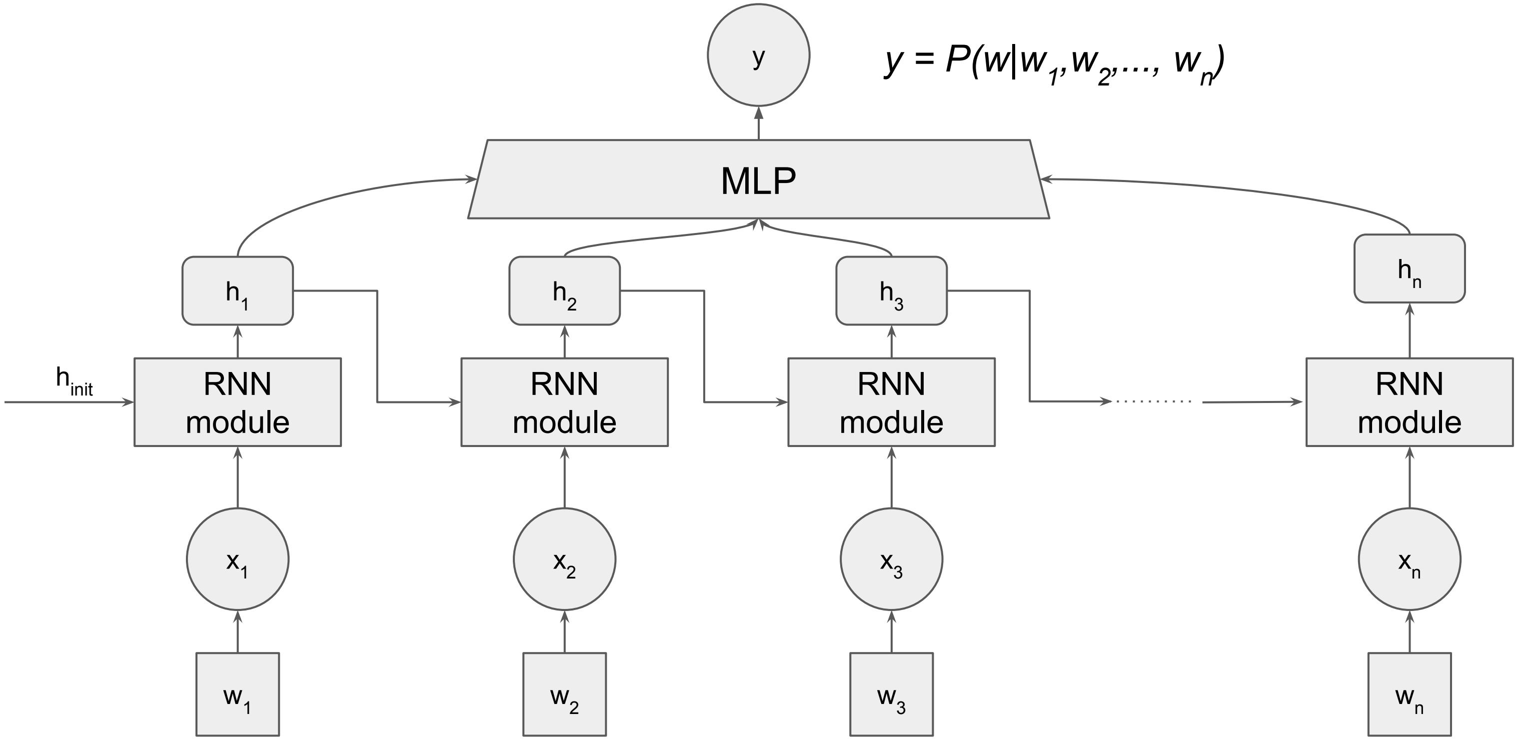
ϕ in $\boldsymbol{R}^{d \times |V|}$ is a matrix containing a word embedding

$$x_t = \Phi w_t$$
$$h_t = \tanh\left(W_{hh} h_{t-1} + W_{xh} x_t\right)$$
$$y_t = W_{hy} h_t$$

W_{hh} is in $\boldsymbol{R}^{kxk}$
W_{xh} is in $\boldsymbol{R}^{dxk}$
W_{hy} is in $\boldsymbol{R}^{kx|V|}$

- Total number of parameters:
  - $d|V|+dk+k^2+k|V| = (d+k)|V| + (d+k)k = (d+k)(|V|+k)$
- E.g., $|V| = 10^5$, $d=10^2$, $k=10^2$
  - Total parameters = $10^7 \ll 10^{11}$

“Unrolling” an RNN: Training an LM



Backpropagation Through Time
(BTT)

PyTorch (Lightning) Model

```python
class RNNModel(pl.LightningModule):
  def __init__(self, num_tokens, config_params):
    super(RNNModel, self).__init__()
    self.config_params = config_params

    self.embedding = nn.Embedding(
        num_embeddings=num_tokens,
        embedding_dim=self.config_params.embedding_dim,
        _weight=self.init_embedding(
            num_tokens,
            self.config_params.embedding_dim
        )
    )

    self.rnn = nn.RNN(
        self.config_params.embedding_dim,
        self.config_params.rnn_hidden_dim,
        self.config_params.num_layers,
        batch_first=True
    )
    self.mlp = nn.Sequential(
        nn.Linear(
            self.config_params.rnn_hidden_dim * self.config_params.num_layers,
            self.config_params.hidden1_dim
        ),
        nn.ReLU(),
        nn.Dropout(self.config_params.dropout_probability),
        nn.Linear(self.config_params.hidden1_dim, num_tokens)
    )

  def forward(self, input):
    h = self.init_hidden(input.size(0))

    x = self.embedding(input)
    out, h = self.rnn(x, h)
    x = h.transpose(0, 1)
    x = x.reshape(
        x.size()[0],
        self.config_params.rnn_hidden_dim * self.config_params.num_layers
    )
    x = self.mlp(x)

    return x

```

SAPIENZA
UNIVERSITÀ DI ROMA

Examples

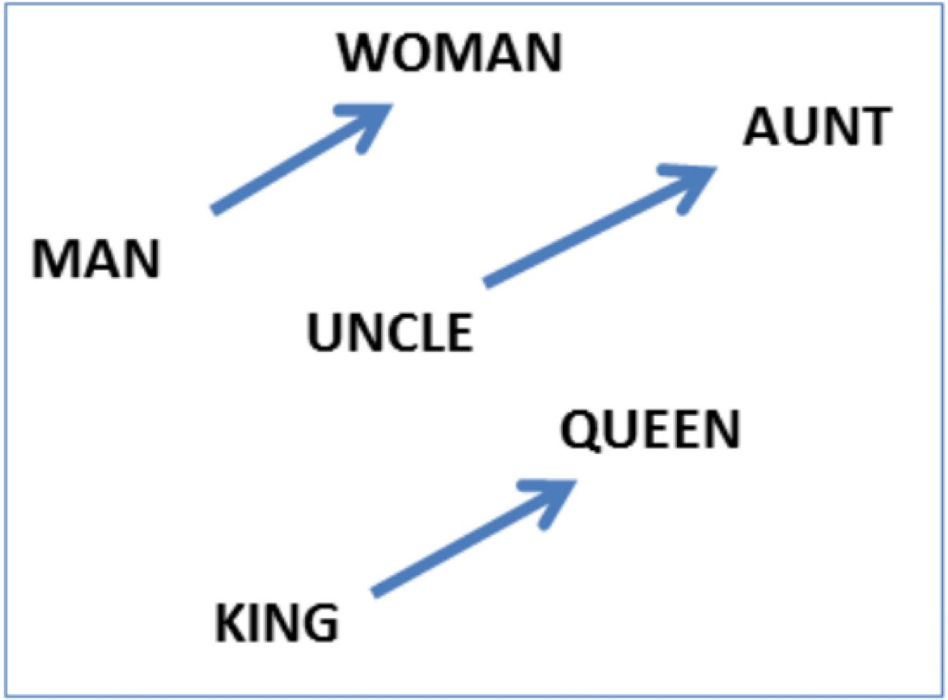
- *MLP Language Model*
- *RNN Language Model*

Sapienza Università di Roma

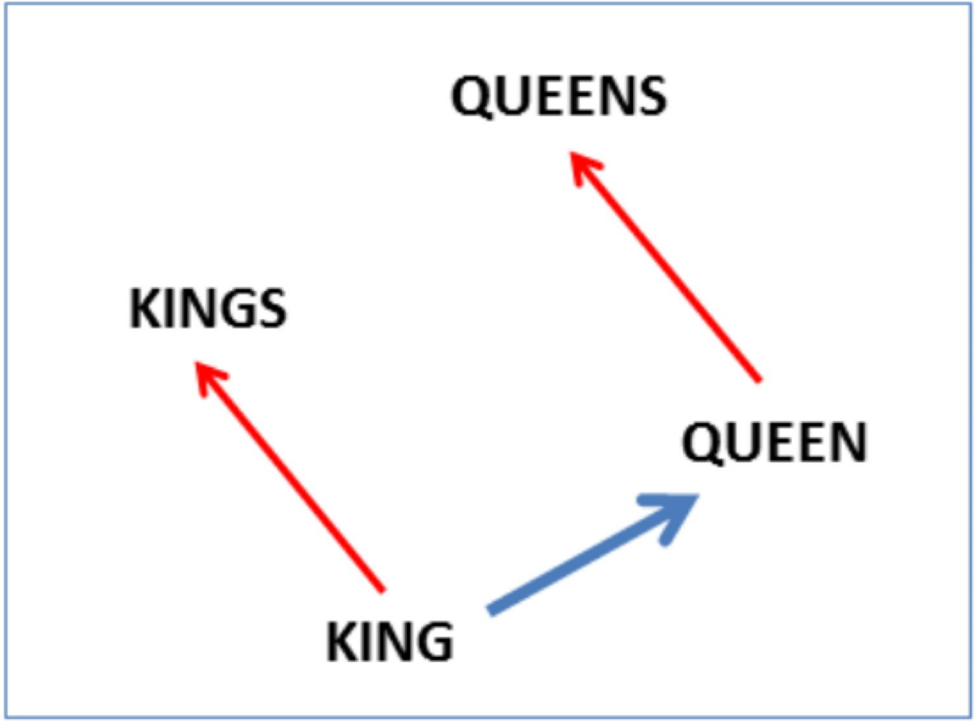
Word2Vec

- Computing softmax for large vocabularies is expensive (typically $O(|V|^2)$)
- Use Noise Contrastive Estimation (or Negative Sampling) to “cast” the problem into a classification problem $P(w_i| \text{context}_i)$ into $P(D=1 \mid w_i, \text{context}_i)$
- How to pick D=0 cases?
  - Randomly sample w_j, context_j pairs according to a given distribution
  - For each positive, sample 5~10 negatives
  - Based on: Gutmann and Hyvärinen. “Noise-contrastive estimation: A new estimation principle for unnormalized statistical models“. JMLR, 13:307-361, 2012.
    - Dyer’s notes on differences between NCE and NS.

Word2Vec: Some Nice Properties



Word2Vec: Some Nice Properties



Word2Vec: Some Nice Properties

Table 8: *Examples of the word pair relationships, using the best word vectors from Table 4 (Skip-gram model trained on 783M words with 300 dimensionality).*

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Word2Vec: Some Nice Properties

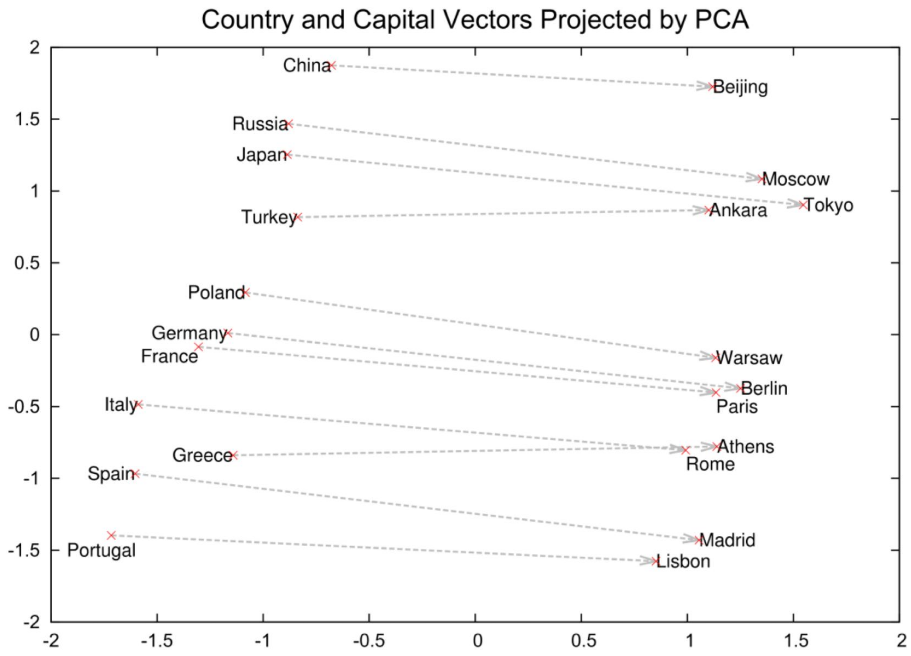
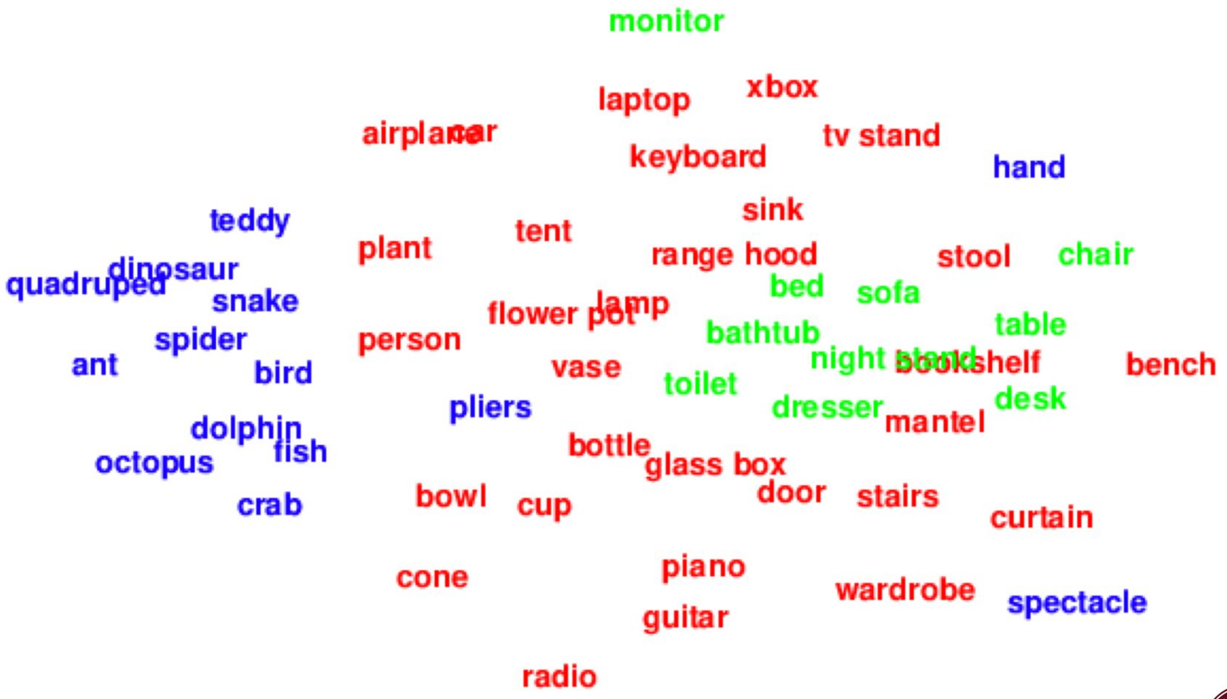


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.

SAPIENZA
UNIVERSITÀ DI ROMA

Word2Vec: Some Nice Properties



After Word2Vec

- GloVe
- FastText
- Misspelling Oblivious Embeddings
- ...

Evaluating Language Models

- *Intrinsic Evaluation*
  - Metrics to evaluate LMs in isolation, not taking into account the specific tasks it’s going to be used for.
- *Extrinsic Evaluation*
  - Employing LMs in actual tasks (such as machine translation) and looking at their final loss/accuracy.

Intrinsic Metric: Perplexity

- It evaluates the normalised inverse probability of the test set

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1, w_2, \ldots, w_N)}}$$

Test Set

"Yesterday I went to the cinema"

"Hello, how are you?"

"The dog was wagging its tail"

High probability
Low perplexity

Fake/incorrect sentences

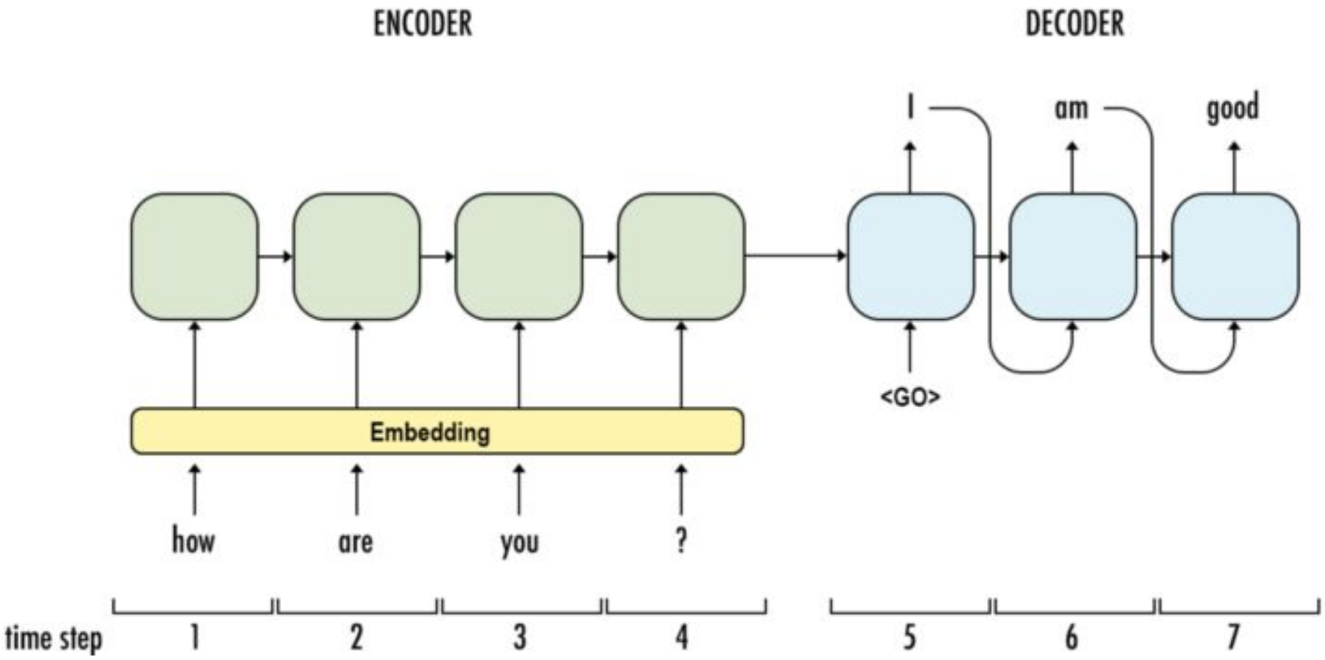
"Can you does it?"

"For wall a driving"

"She said me this"

Low probability
High perplexity

Encoder Decoder Architectures



Modern Language Models: Transfomers

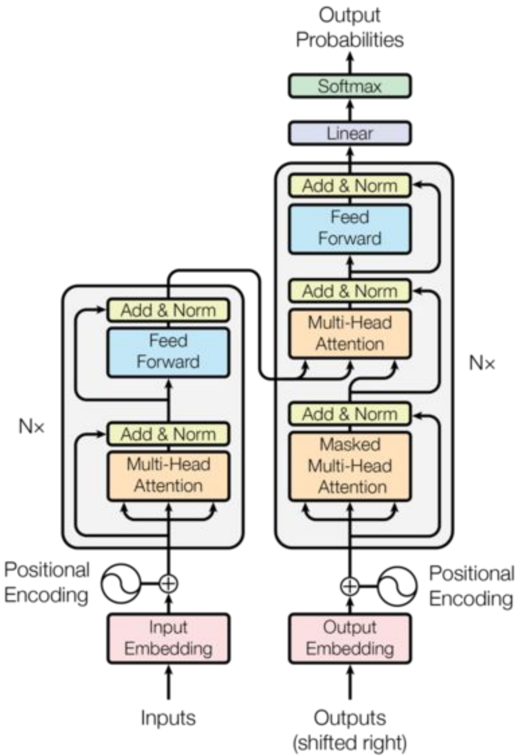
- Vaswani, Ashish, et al. "Attention is All you Need." NIPS. 2017.

```python
class EncoderDecoder(nn.Module):
    """
    A standard Encoder-Decoder architecture. Base for this and many
    other models.
    """
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(EncoderDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def forward(self, src, tgt, src_mask, tgt_mask):
        "Take in and process masked src and target sequences."
        return self.decode(self.encode(src, src_mask), src_mask,
                            tgt, tgt_mask)

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```

```python
class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        return F.log_softmax(self.proj(x), dim=-1)
```

Modern Language Models: Transfomers

- Vaswani, Ashish, et al. "Attention is All you Need." NIPS. 2017.



Transfomers: Encoder

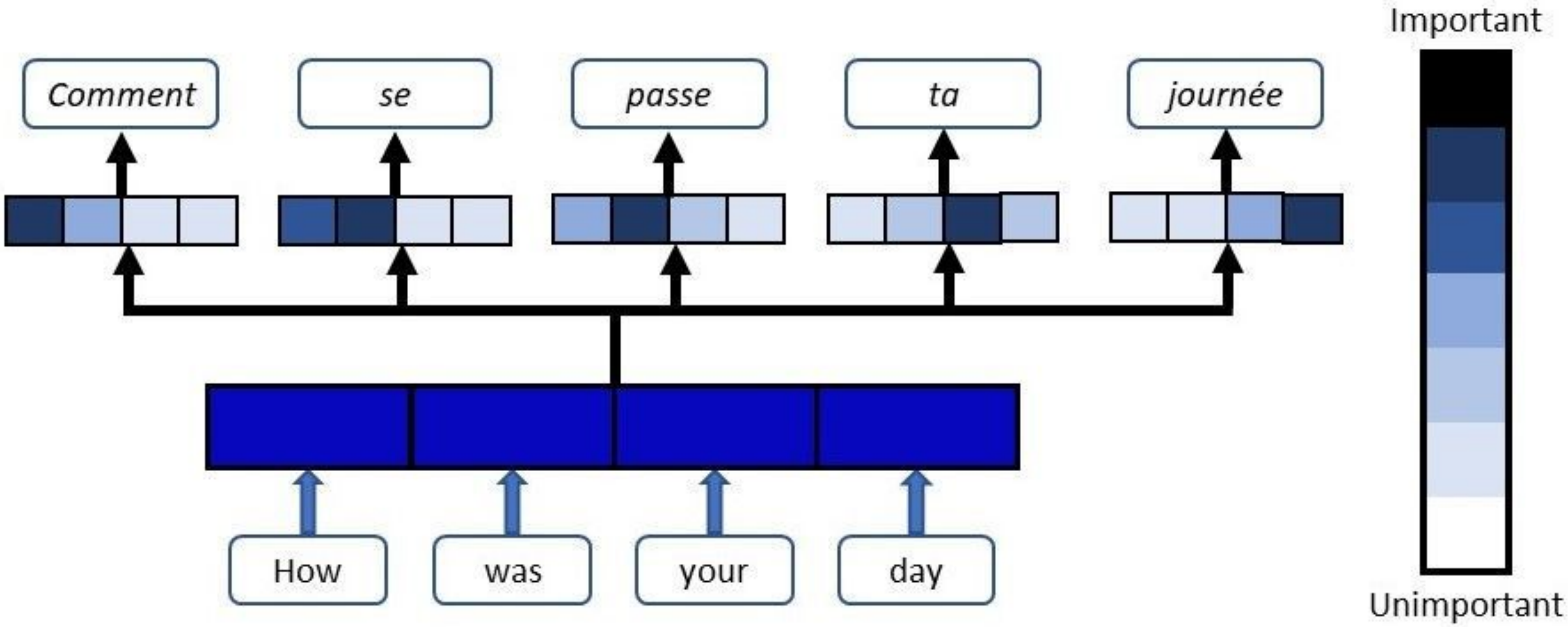
```python
def clones(module, N):
    "Produce N identical layers."
    return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
```

```python
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

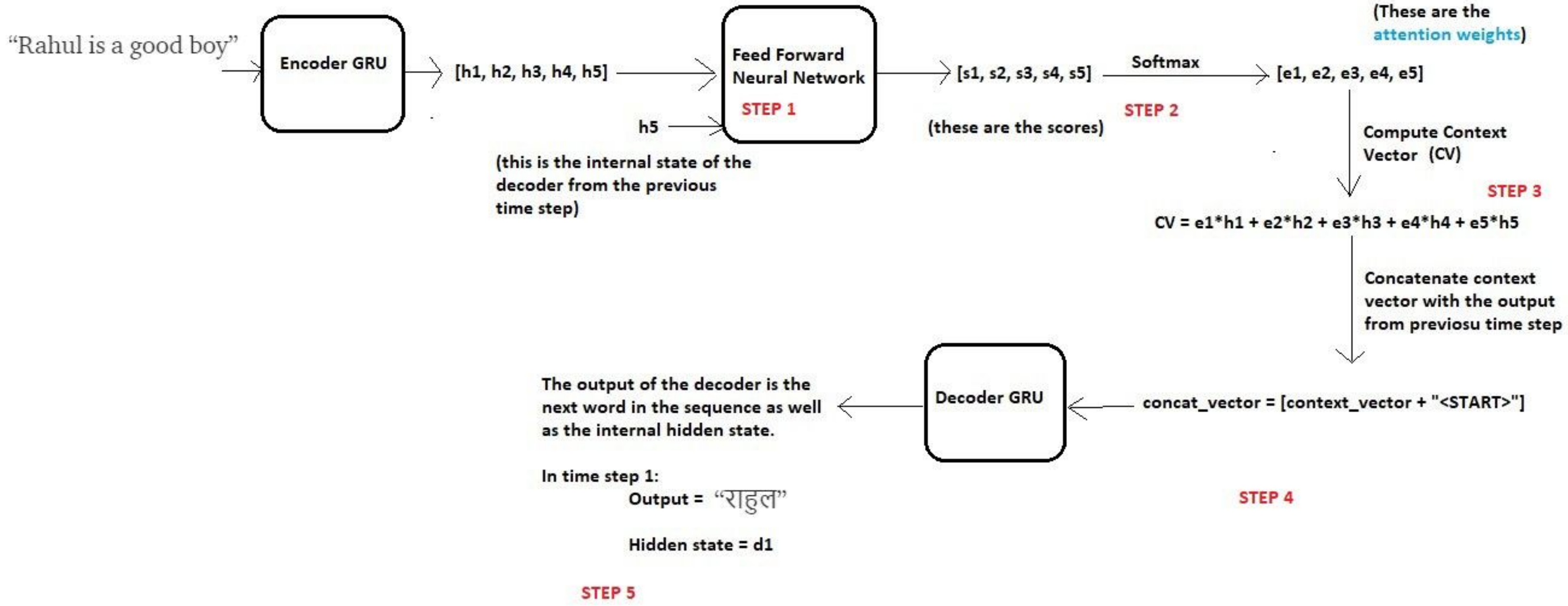
We employ a residual connection (cite) around each of the two sub-layers, followed by layer normalization (cite).

```python
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."
    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

Attention Mechanism



Attention Mechanism



Transfomers: Encoder

- That is, the output of each sub-layer is LayerNorm(x + SubLayer(x))
    - SubLayer(x) implements the sub-layer that we are about to describe

```python
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """
    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

Transfomers: Encoder

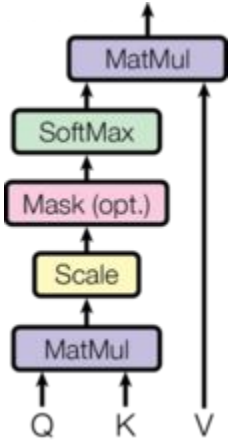
- Each layer has two sub-layers:
  - multi-head self-attention mechanism
  - position-wise fully connected feed-forward network

```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

Transfomers: Attention

- It maps a query and a set of key-value pairs to an output
  - where the query, keys, values, and output are all vectors.
- The output is computed as a weighted sum of the values
  - the weight assigned to each value is computed by a **compatibility** function of the query with the corresponding key.

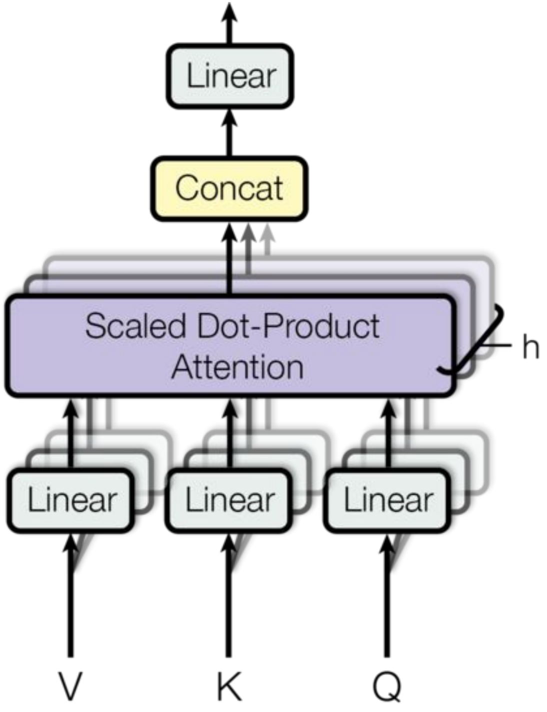
$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$




```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) \
             / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

Transfomers: Multi-Head Attention

- Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.
- With a single attention head, averaging inhibits this.



$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head_1}, \ldots, \mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

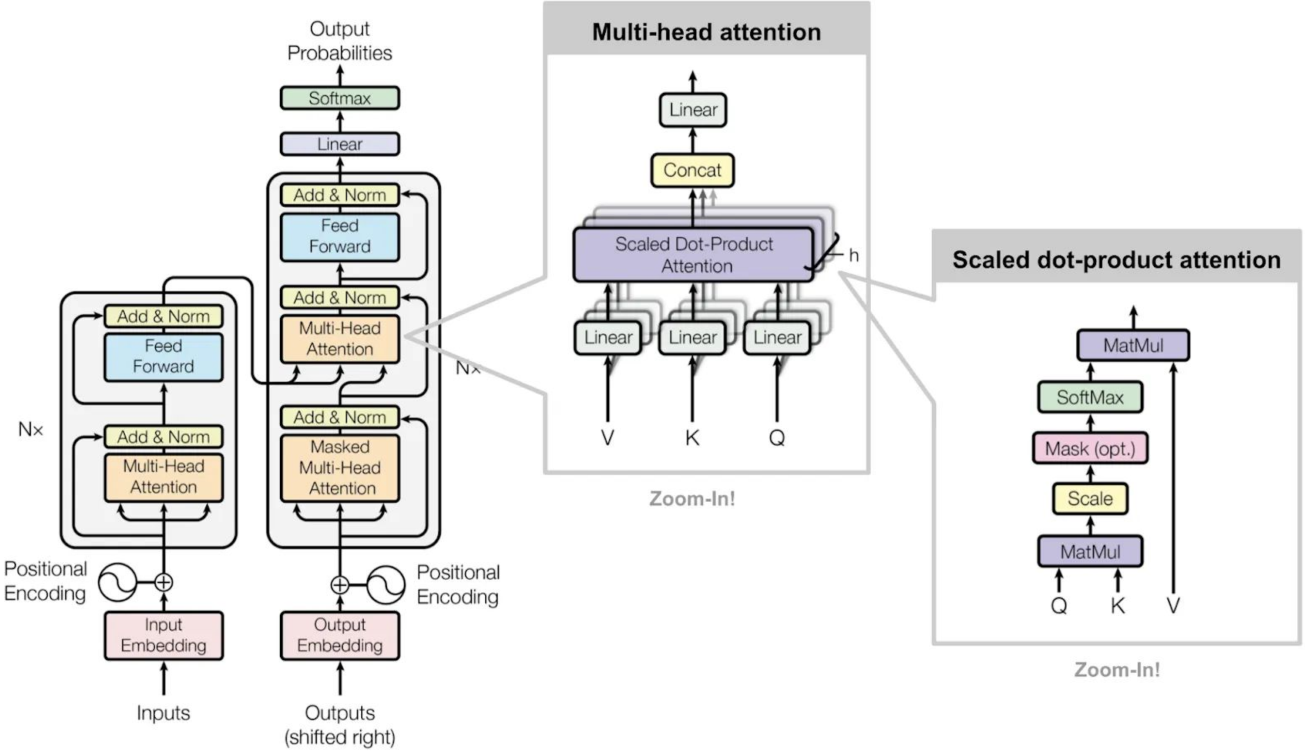
$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$

Multi-Head Attention in PyTorch

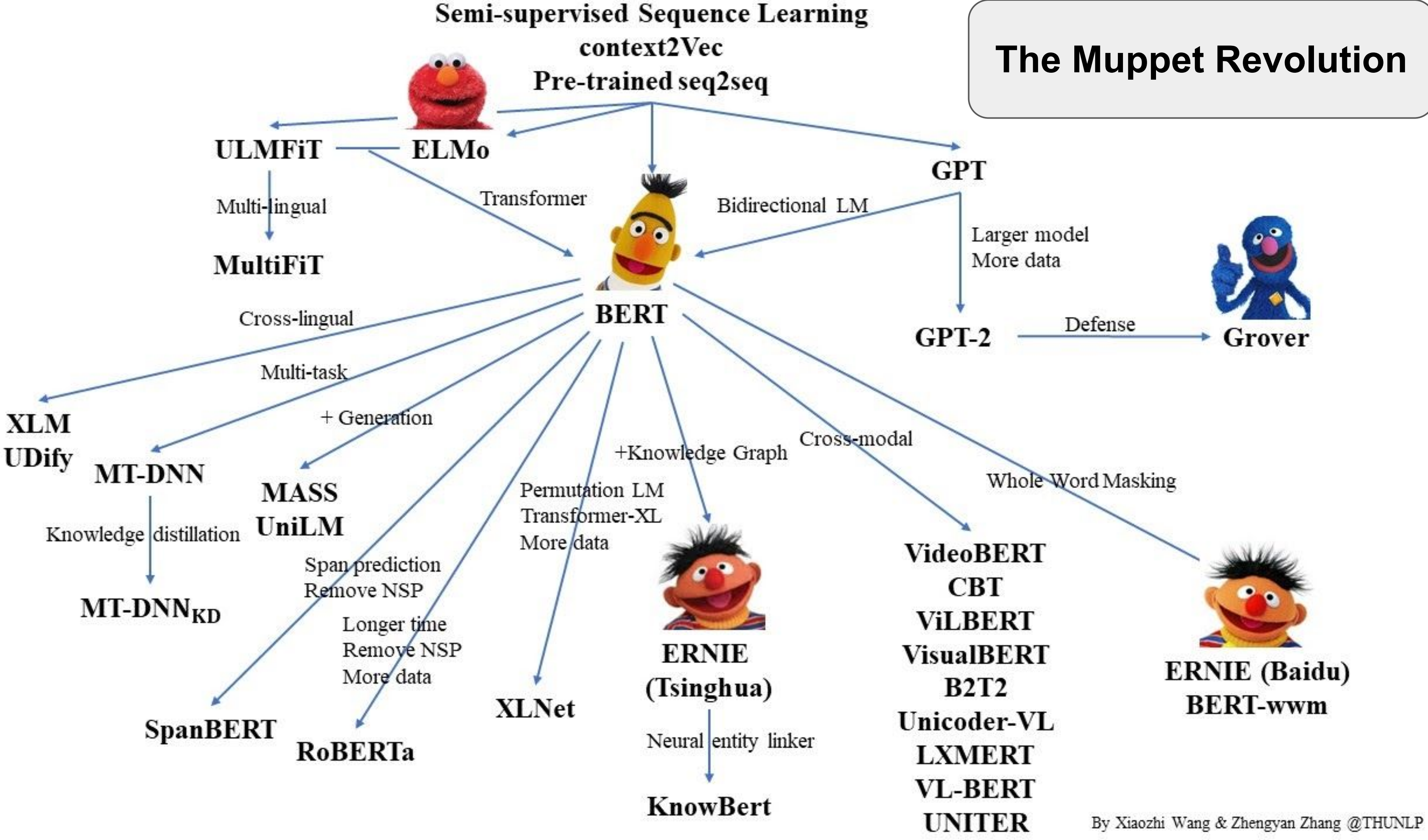
```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        "Implements Figure 2"
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = \
            [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (query, key, value))]

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous() \
             .view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

SAPIENZA
UNIVERSITÀ DI ROMA

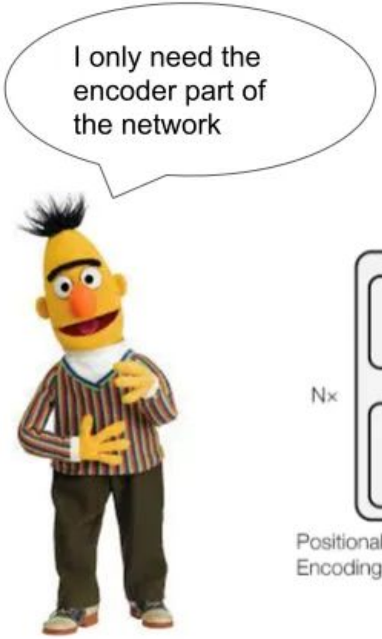
The Full Picture

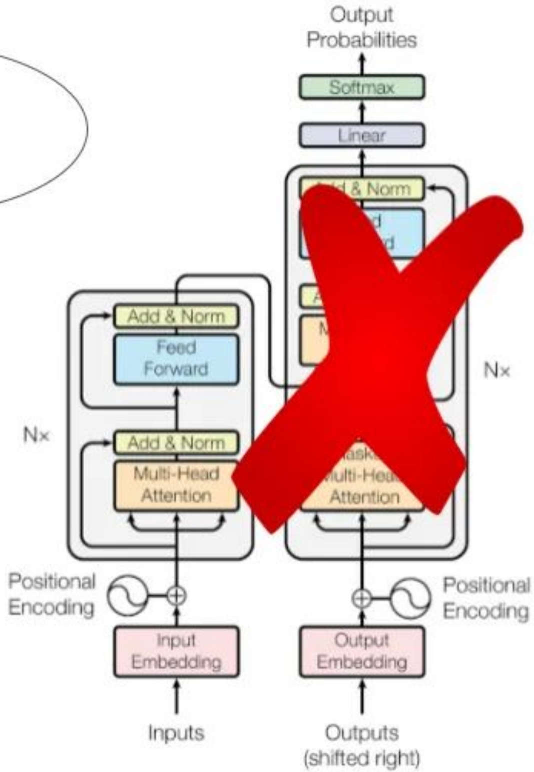


The Muppet Revolution



BERT: Bidirectional Encoder Representations from Transformers





A Colab Example

1. https://colab.research.google.com/drive/1XqA6oMGaJvmHdC4Mlyz1gUs10cUY-w-f
2. https://colab.research.google.com/drive/12NWHoUjmZjVtxYD4RAipU7b5Qx8yHaHp