Regularization: if we have L2 regularization, then the loss will be:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) + \lambda \sum_{k} \theta_k^2$$

In the classic view, it is used for preventing overfitting but now, it is used for making the big models more generalized.

Dropout:

- How? For the train => For each data point each time, randomly set input to 0 with probability p which is “dropout ratio” (often p = 0.5 except p = 0.15 for input layer) via dropout mask
  For test => no other dropout and multiply all weights by 1-p
- Why? Prevent feature Co-adaptation and Good Regularization

Vectorization: use vectors and matrices rather than for loops as they are much faster and efficient.

Parameter initialization:

- Initialize hidden layer biases to 0 and output biases to optimal value if weights were 0 (like mean target or inverse sigmoid of mean target) and Initialize all other weights with Uniform(–r, r), with r chosen so numbers get neither too big or too small.
- Xavier initialization: has variance inversely proportional to fan-in (previous layer size) and fan-out (next layer size):

$$\mathrm{Var}(W_i) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

Optimizers:

- SGD, Adagrad (Simplest member of family, but tends to “stall early”), RMSprop, Adam, AdamW, NAdamW (Can be better with word vectors (W) and for speed (Nesterov acceleration))
- hand-tuning the learning rate (start it higher like 0.001 and halve it every k epochs)

Language Modeling:

- task of predicting what word comes next

$$P(\boldsymbol{x}^{(t+1)}|\ \boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)})$$

where $x^{(t+1)}$ can be any word in the vocabulary $V = \{\boldsymbol{w}_1, ..., \boldsymbol{w}_{|V|}\}$

- or a system that assigns a probability to a piece of text

$$P(\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(T)}) = P(\boldsymbol{x}^{(1)}) \times P(\boldsymbol{x}^{(2)}|\ \boldsymbol{x}^{(1)}) \times \cdots \times P(\boldsymbol{x}^{(T)}|\ \boldsymbol{x}^{(T-1)},\ldots,\boldsymbol{x}^{(1)})$$
$$= \prod_{t=1}^{T} P(\boldsymbol{x}^{(t)}|\ \boldsymbol{x}^{(t-1)},\ldots,\boldsymbol{x}^{(1)})$$

This is what our LM provides

- solution: Collect statistics about how frequent different n-grams are and use these to predict next word. => n-grams or RNNs
- evaluation metric: perplexity

$$\text{perplexity} = \prod_{t=1}^{T} \left(\frac{1}{P_{\text{LM}}(\boldsymbol{x}^{(t+1)}|\ \boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)})} \right)^{1/T}$$

Normalized by number of words

Inverse probability of corpus, according to Language Model

This is equal to the exponential of the cross-entropy loss $J(\theta)$:

$$= \prod_{t=1}^{T} \left(\frac{1}{\hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}}} \right)^{1/T} = \exp\left(\frac{1}{T} \sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}^{(t)}_{\boldsymbol{x}_{t+1}} \right) = \exp(J(\theta))$$

=> lower perplexity is better.

N-Grams: First we make a Markov assumption: x^(t-1) depends only on the preceding n-1 words and then count them in some large corpus of text

$$P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(1)}) = P(\boldsymbol{x}^{(t+1)}|\overbrace{\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)}}^{n\text{-1 words}})$$

$$\frac{\text{count}(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}{\text{count}(\boldsymbol{x}^{(t)},\ldots,\boldsymbol{x}^{(t-n+2)})}$$

- sparsity problem: Increasing n makes sparsity problems worse:
  1. count of soorat = 0 => smoothing
  2. count of makhraj = 0 => back-off (calculate the count without the first word)

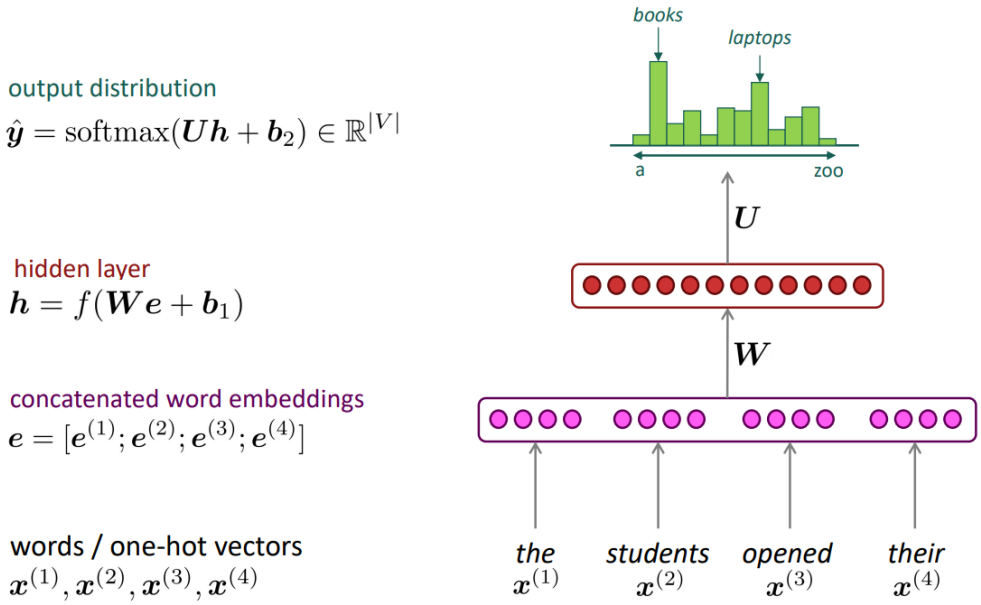
- storage problem: Need to store count for all n-grams you saw in the corpus.

A fixed-window neural Language Model: => solving the problem of sparsity and storage of the n-grams

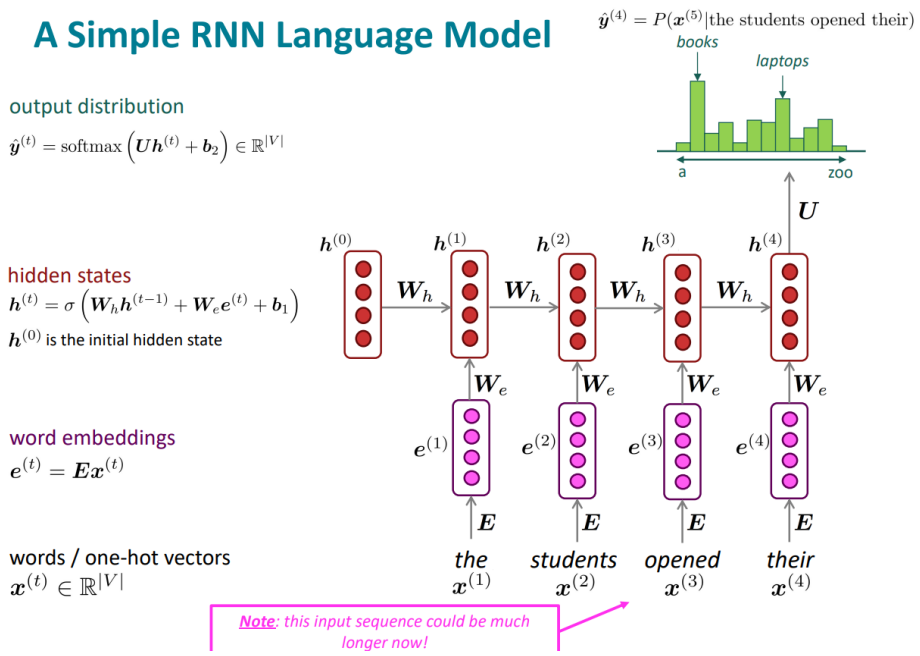
Problems:

1. Fixed window is too small
2. Enlarging window enlarges W
3. Window can never be large enough!
4. x(1) and x(2) are multiplied by completely different weights in W.
5. No symmetry in how the inputs are processed.

A fixed-window neural Language Model



RNN: => applying the same weight W and are not LM!



Advantages:

1. Can process any length input
2. Computation for step t can (in theory) use information from many steps back
3. Model size doesn't increase for longer input context
4. Same weights applied on every timestep, so there is symmetry in how inputs are processed.

Disadvantages:

1. Recurrent computation is slow
2. In practice, difficult to access information from many steps back
3. Vanishing gradient: in the chain rule, when these are small, the gradient signal gets smaller and smaller as it backpropagates further. So, model weights are updated only with respect to near effects, not long-term effects. (repeated multiplication by the same weight matrix) Solution: as in a simple rnn, the hidden state is constantly being rewritten, we need:
    1) LSTMs => an RNN with separate memory which is added to
    2) ResNet => add direct connection (known as skip connections or identity connections) to preserves information
    3) DenseNet => add dense connections and directly connect each layer to all future layers

4) HighwayNet => add highway connections and the identity
5) connection vs the transformation layer is controlled by a dynamic gate
6) Attention, residual connections => more direct and linear pass-through connections in model

4. Exploding gradient: If the gradient becomes too big, then the SGD update step becomes too big so we take too large a step and reach a weird and bad parameter configuration
   Solution: gradient clipping: if the norm of the gradient is greater than some threshold, scale it down before applying SGD update which will cause taking a step in the same direction, but a smaller step.
5. Linear interaction distance: Hard to learn long-distance dependencies (because gradient problems!) => O(sequence length) steps for distant word pairs to interact means is really large!
6. Lack of parallelizability: Forward and backward passes have O(sequence length) unparallelizable operations

Loss function:

Loss function on step *t* is cross-entropy between predicted probability distribution $\hat{\boldsymbol{y}}^{(t)}$, and the true next word $\boldsymbol{y}^{(t)}$ (one-hot for $\boldsymbol{x}^{(t+1)}$):

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{w \in V} \boldsymbol{y}_w^{(t)} \log \hat{\boldsymbol{y}}_w^{(t)} = -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

Average this to get overall loss for entire training set:

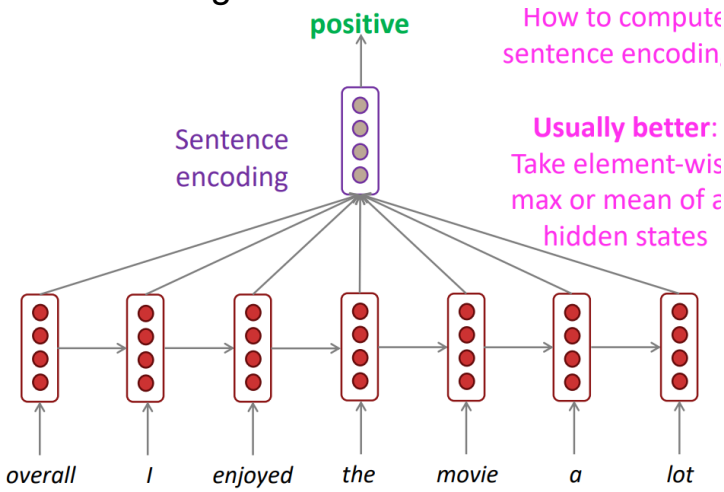
$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{\boldsymbol{y}}_{\boldsymbol{x}_{t+1}}^{(t)}$$

- Cross entropy:
  => Computing loss and gradients across entire corpus at once is too expensive (memory-wise)
- Stochastic Gradient Descent: consider x(1), x(2), …, x(T) as a sentence (or a document), compute loss for it, compute gradients and update weights. Repeat on a new batch of sentences

Backpropagation: $$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_{\boldsymbol{h}}} = \sum_{i=1}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_{\boldsymbol{h}}}\bigg|_{(i)}$$ => called backpropagation through time

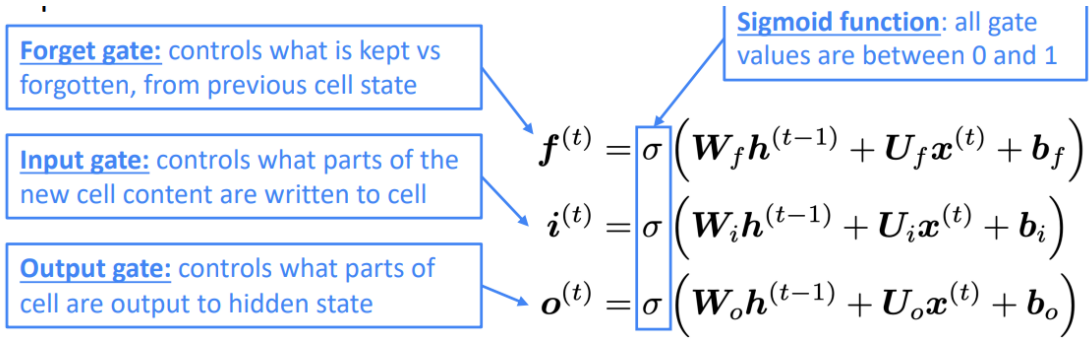
Generating text with an RNN: generate text by repeated sampling. Sampled output becomes next step's input.

RNN in SA: we can compute sentence encoding by using the final hidden state or taking element-wise max or mean of all hidden states

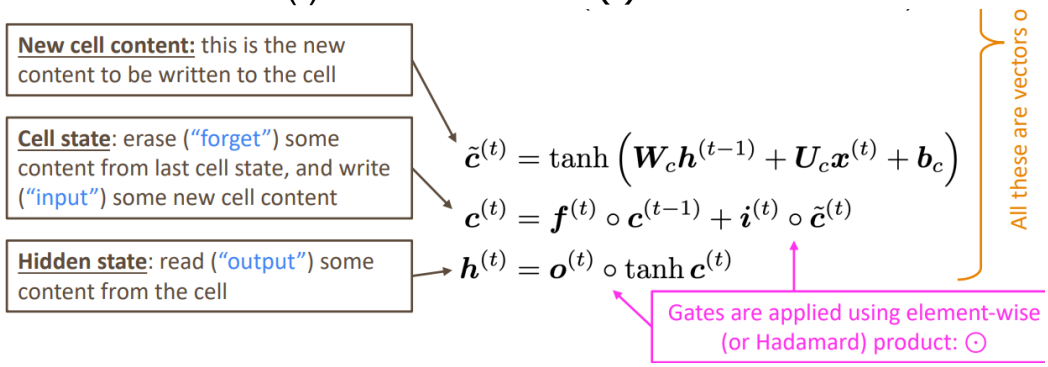


LSTMs:

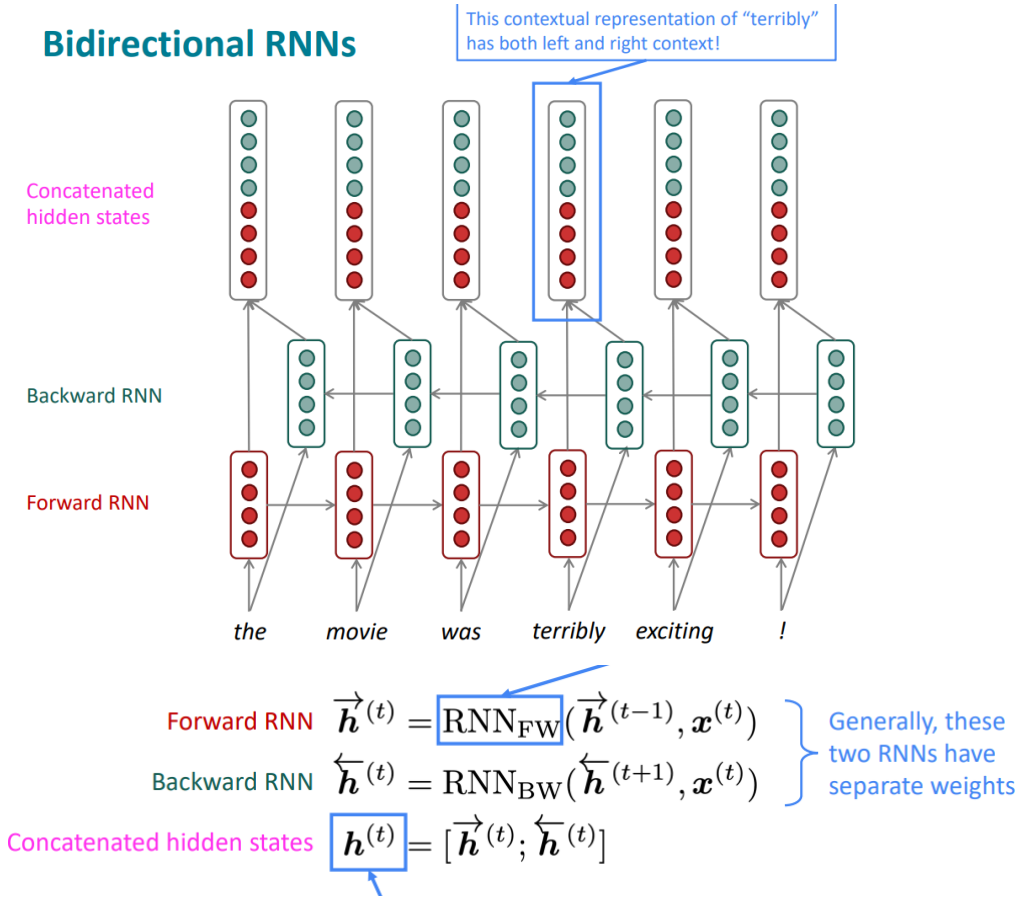
- On step t, there is a hidden state h(t) and a cell state c(t):
  - Both are vectors length n
  - The cell stores long-term information
  - The LSTM can read, erase, and write information from the cell
- The selection of which information is erased/written/read is controlled by three corresponding gates:
  - The gates are also vectors of length n
  - On each timestep, each element of the gates can be open (1), closed (0), or somewhere in-between
  - The gates are dynamic: their value is computed based on the current context
  -



- We have a sequence of inputs x(t) and we will compute a sequence of hidden states h(t) and cell states c(t):



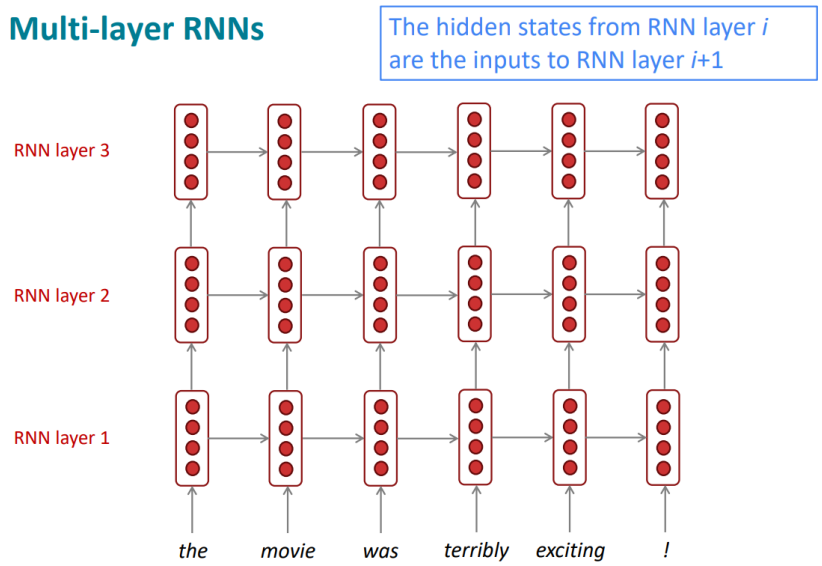
Bidirectional RNNs: in SA, we need the impression of the right context but in simple RNNs, contextual representations (a representation of a word in the context of this sentence.) only contain information about the left context. So we use bidirectional RNNs:



Bidirectional RNNs are use when you have access to the entire input

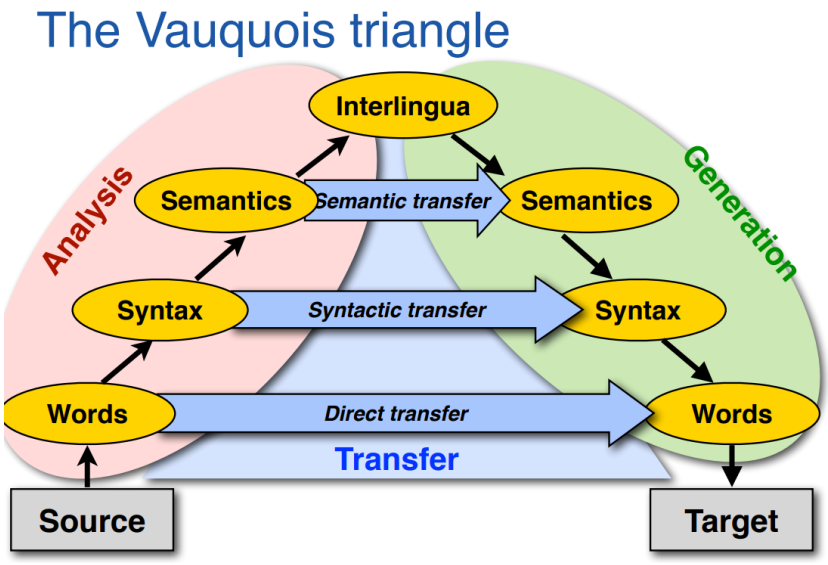
Sequence => They are not applicable to Language Modeling, because in LM you only have left context available.

Multi-layer RNNs or stacked RNNs: allows the network to compute more complex representations. The lower RNNs should compute lower-level features and the higher RNNs should compute higher-level features.



Machine Translation:

- Statistical Machine Translation:



  -

- Suppose we're translating French → English.
- We want to find best English sentence *y*, given French sentence *x*

$$\text{argmax}_y P(y|x)$$

- Use Bayes Rule to break this down into two components to be learned separately:

$$= \text{argmax}_y \underbrace{P(x|y)}_{\text{Translation Model}} \underbrace{P(y)}_{\text{Language Model}}$$

Translation Model

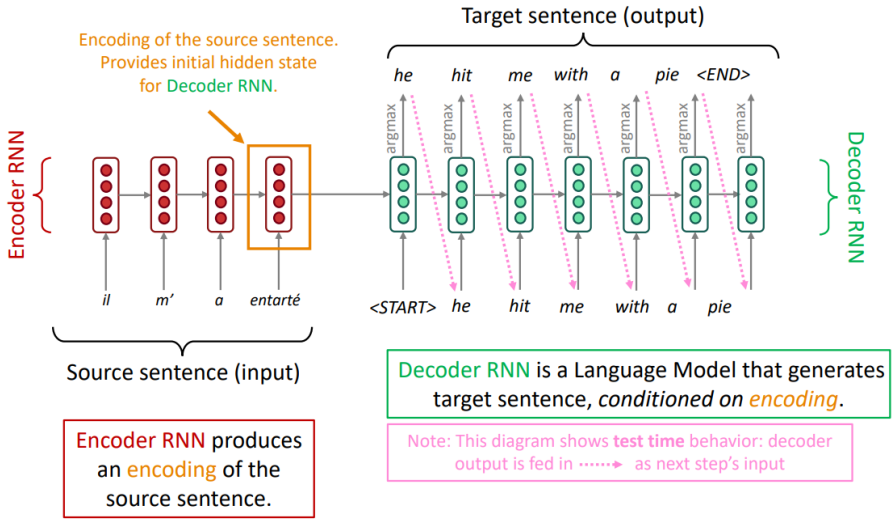
Models how words and phrases should be translated (*fidelity*). Learned from parallel data.

Language Model

Models how to write good English (*fluency*). Learned from monolingual data.

- Properties:
    1. extremely complex
    2. systems had many separately-designed subcomponents
    3. Lots of feature engineering
    4. Required compiling and maintaining extra resources
    5. Lots of human effort to maintain

- Neural Machine Translation (Seq2Seq model): a way to do Machine Translation with a single end-to-end neural network by using two RNNs:
    - Encoder-decoder model => One neural network takes input and produces a neural representation and another network produces output based on that neural representation
    - Consisting of an encoder (encoding of the input) and a decoder (LM)

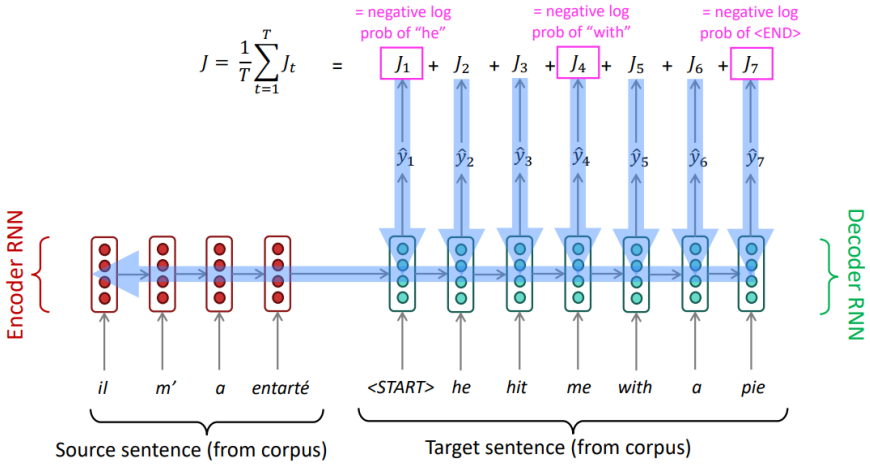


-
  - NMT directly calculates $P(y|x)$:

$$P(y|x) = P(y_1|x)\,P(y_2|y_1,x)\,P(y_3|y_1,y_2,x)\ldots P(y_T|y_1,\ldots,y_{T-1},x)$$

Probability of next target word, given target words so far and source sentence *x*

-
- Optimization: Seq2seq is optimized as a single system. Backpropagation operates "end-to-end":

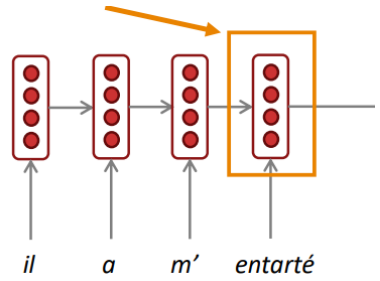


- Seq2Seq is used also in:
  1) Summarization (long text → short text)
  2) Dialogue (previous utterances → next utterance)
  3) Parsing (input text → output parse as sequence)
  4) Code generation (natural language → Python code)
- Seq2Seq is a conditional LM: Language Model because the decoder is predicting the next word of the target sentence y.

Conditional because its predictions are also conditioned on the source sentence x.

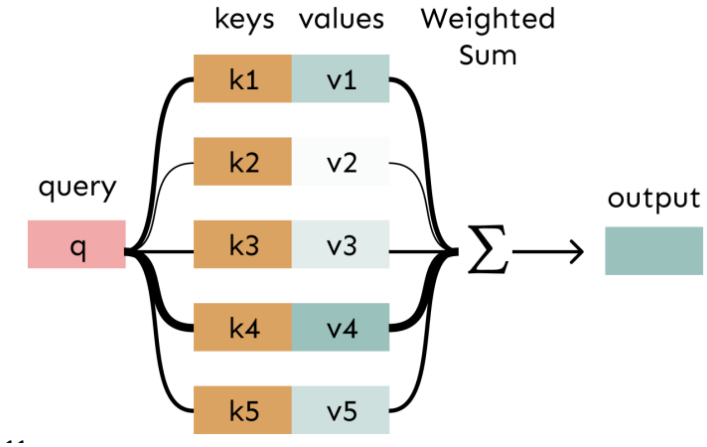
Problems of Seq2Seq models:

the bottleneck problem: there is an information bottleneck where all information about the source sentence is there. => Solution: attention



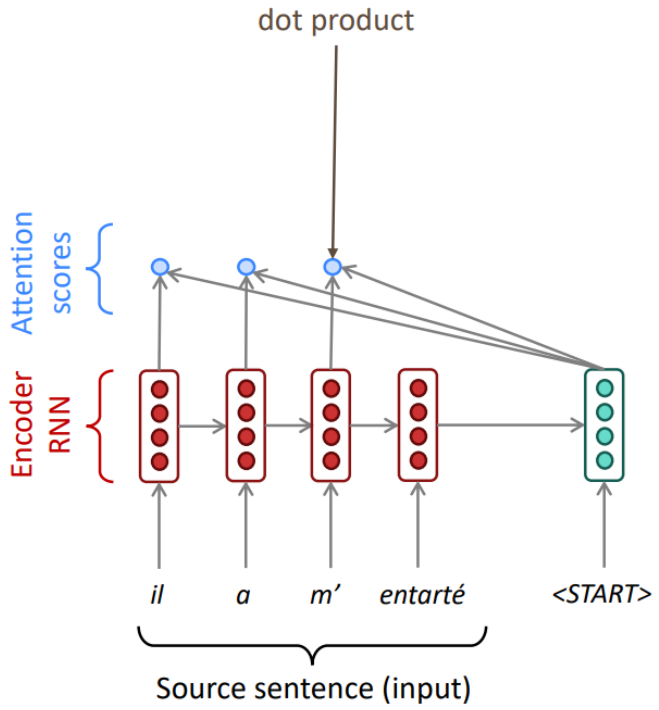
Attention: => on each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence.

Attention is just a weighted average: In attention, the query matches all keys softly, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed. => treats each word's representation as a query to access and incorporate information from a set of values.



Steps:

1. For each hidden state of the encoder, we compute the attention score with dot product of the output of the hidden state and the decoder's that step's hidden state



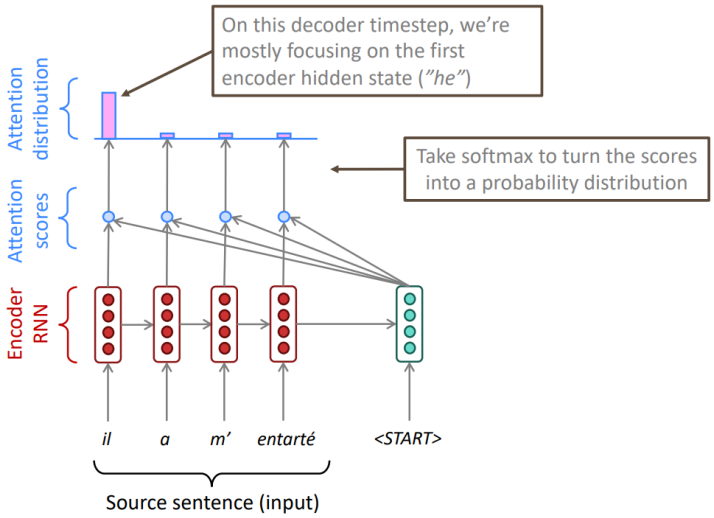
We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$

On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$

We get the attention scores $\boldsymbol{e}^t$ for this step:

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

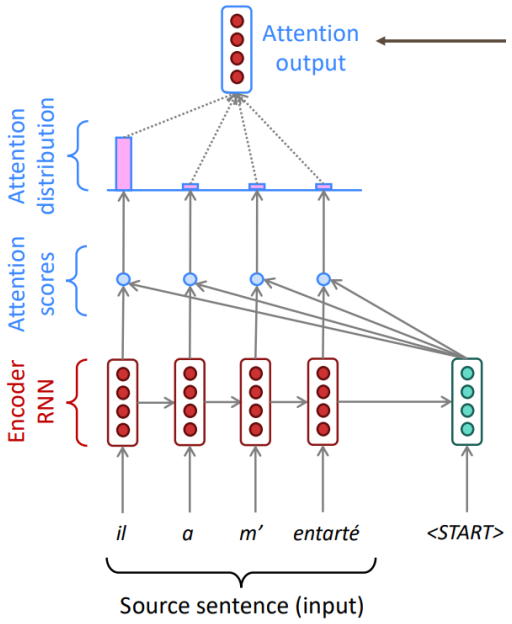
2. Then, we compute attention distribution using softmax:



We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \text{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

3. Then, use the attention distribution to take a weighted sum of the encoder hidden states (called context vector) => the attention output mostly contains information from the hidden states that received high attention. => The weighted sum is a selective summary of the information contained in the values, where the query determines which values to focus on



We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

4. Concatenate attention output with decoder hidden state, then use to compute yhat(1) as before

   Finally we concatenate the attention output $\boldsymbol{a}_t$ with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

5. For the following steps, sometime we take the attention output from the previous step, and also feed it into the decoder (along with the usual decoder input)

Advantages:

- parallelizable => number of unparallelizable operations does not increase with sequence length
- solves bottleneck issues (bypass bottleneck) => maximum interaction distance: O(1), since all words interact at every layer
- improves NMT performance
- a more “human-like” model of the MT process
- helps with the vanishing gradient problem => Provides shortcut to faraway states
- provides some interpretability => the network just learned alignment by itself
- flexible
- general way pointer and memory manipulation

For step 1 (computing e from h1, h2, …, hN, s) where $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N \in \mathbb{R}^{d_1}$ and $\boldsymbol{s} \in \mathbb{R}^{d_2}$ there are different methods:

1. Basic dot-product attention: => if d1=d2:

$$\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{h}_i \in \mathbb{R}$$

2. Multiplicative attention (bilinear attention): use W which $\boldsymbol{W} \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix.

$$\boldsymbol{e}_i = \boldsymbol{s}^T \boldsymbol{W} \boldsymbol{h}_i \in \mathbb{R}$$

3. Reduced-rank multiplicative attention:
For low rank matrices $\boldsymbol{U} \in \mathbb{R}^{k \times d_2}$, $\boldsymbol{V} \in \mathbb{R}^{k \times d_1}$, $k \ll d_1, d_2$

$$e_i = s^T (\boldsymbol{U}^T \boldsymbol{V}) h_i = (\boldsymbol{U}s)^T (\boldsymbol{V}h_i)$$

4. Additive attention:
Where $\boldsymbol{W}_1 \in \mathbb{R}^{d_3 \times d_1}$, $\boldsymbol{W}_2 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $\boldsymbol{v} \in \mathbb{R}^{d_3}$ is a weight vector. and d3 (the attention dimensionality) is a hyperparameter:

$$\boldsymbol{e}_i = \boldsymbol{v}^T \tanh(\boldsymbol{W}_1 \boldsymbol{h}_i + \boldsymbol{W}_2 \boldsymbol{s}) \in \mathbb{R}$$

Attention in general: Given a set of vector values, and a vector query, attention is a technique to compute a weighted sum of the values, dependent on the query. => query attends to the values

Cross attention: paying attention to the input x to generate y(t)

Self-attention: to generate y(t), we need to pay attention to y(<t) => key, query and values are from the same sequence:

Let $\boldsymbol{w}_{1:n}$ be a sequence of words in vocabulary V, like *Zuko made his uncle tea*.

For each $\boldsymbol{w}_i$, let $\boldsymbol{x}_i = E\boldsymbol{w}_i$, where $E \in \mathbb{R}^{d \times |V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices Q, K, V, each in $\mathbb{R}^{d \times d}$

$\boldsymbol{q}_i = Q\boldsymbol{x}_i$ (queries) $\quad \boldsymbol{k}_i = K\boldsymbol{x}_i$ (keys) $\quad \boldsymbol{v}_i = V\boldsymbol{x}_i$ (values)

2. Compute pairwise similarities between keys and queries; normalize with softmax

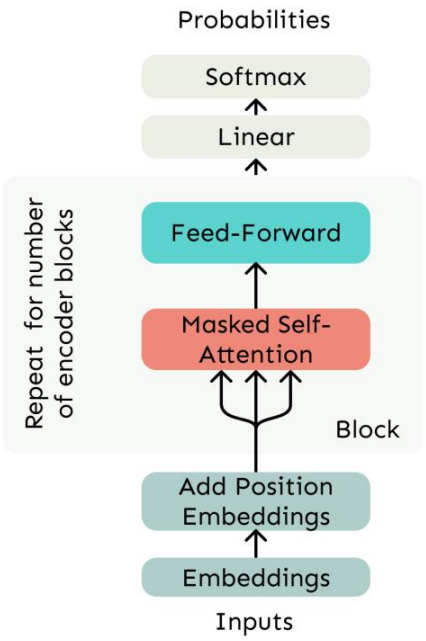
$$\boldsymbol{e}_{ij} = \boldsymbol{q}_i^\top \boldsymbol{k}_j \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'} \exp(\boldsymbol{e}_{ij'})}$$

3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \boldsymbol{v}_i$$

$$\boldsymbol{o}_i = \sum_j \boldsymbol{\alpha}_{ij} \boldsymbol{v}_i$$

34



Problems of self-attention and their solution:

1. Doesn't have an inherent notion of order: self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
   Solution: add position representations to the inputs:

Consider representing each **sequence index** as a **vector**

$$\boldsymbol{p}_i \in \mathbb{R}^d, \text{ for } i \in \{1,2,\dots,n\} \text{ are position vectors}$$

Then $\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$ where xi is the embedding of the word at index i and x~i is the positioned embedding.
Pi:

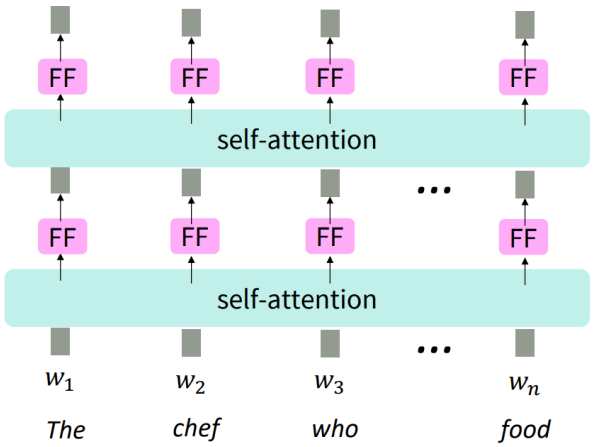
- Sinusoidal position representations:

$$\boldsymbol{p}_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$

  which because of its periodicity, “absolute position” isn’t as important and can extrapolate to longer sequences as periods restart but is not learnable!
- Learned absolute position representations: Let all pi be learnable parameters! Learn a matrix p ∈ ℝd×n, and let each pi be a column of that matrix.
  Which is flexible but can’t extrapolate to indices outside 1, … , n
- Relative linear position attention
- Dependency syntax-based position

2. No nonlinearities for deep learning! It’s all just weighted averages => there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors:
   Solution: apply the same feedforward network to each self-attention output.

$$\begin{aligned} m_i &= MLP(\text{output}_i) \\ &= W_2 * \text{ReLU}(W_1\, \text{output}_i + b_1) + b_2 \end{aligned}$$



3. Need to ensure we don't "look at the future" when predicting a sequence
   Solution:
   1) At every timestep, we could change the set of keys and queries to include only past words => inefficient!
   2) mask out the future by artificially setting attention weights to 0 (which will parallelize operations):

$$e_{ij} = \begin{cases} q_i^\top k_j, j \leq i \\ -\infty, j > i \end{cases}$$

Multi-headed attention: In self-attention, we look where (Q.xi)^T * K.xj is high. But sometimes we want to focus on different j => we will define multiple Q,K,V matrices:

Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and ℓ ranges from 1 to h.

Each attention head performs attention independently:

- $\text{output}_\ell = \text{softmax}\left(XQ_\ell K_\ell^\top X^\top\right) * XV_\ell$, where $\text{output}_\ell \in \mathbb{R}^{d/h}$

Then the outputs of all the heads are combined!

- $\text{output} = [\text{output}_1; \ldots; \text{output}_h]Y$, where $Y \in \mathbb{R}^{d \times d}$

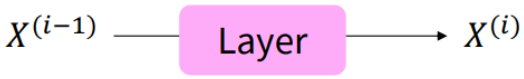
⇨ it's not really more costly.
⇨ when d becomes large, the dot product will be large and inputs to the softmax function can be large, making the gradients small. So we use scaled dot product:

We divide the attention scores by $\sqrt{d/h}$, to stop the scores from becoming large just as a function of d/h (The dimensionality divided by the number of heads.)

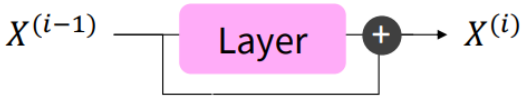
$$\text{output}_\ell = \text{softmax}\left(\frac{XQ_\ell K_\ell^\top X^\top}{\sqrt{d/h}}\right) * XV_\ell$$

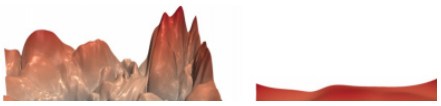
Residual connections: it helps models train better:

Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where i represents the layer)



We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$ (so we only have to learn "the residual" from the previous layer)





⇨ gradient is great through the residual connection is 1.
⇨ bias towards the identity function.

Layer normalization: it helps models train faster: => cutting down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation within each layer:

Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.

Let $\mu = \sum_{j=1}^{d} x_j$; this is the mean; $\mu \in \mathbb{R}$.

Let $\sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2}$; this is the standard deviation; $\sigma \in \mathbb{R}$.

Let $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ be learned "gain" and "bias" parameters. (Can omit!)

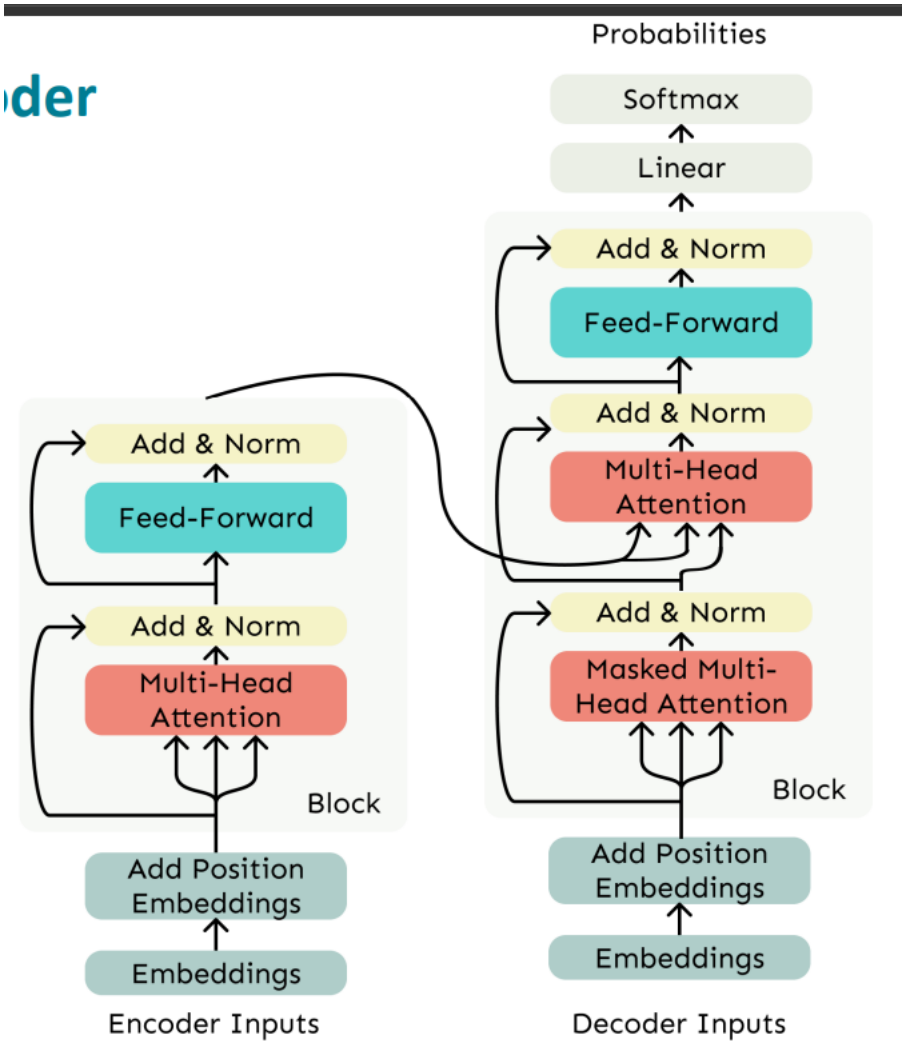
Then layer normalization computes:

$$\text{output} = \frac{x - \mu}{\sqrt{\sigma + \epsilon}} * \gamma + \beta$$

Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias

Transformers:



- its decoder block consists of:
    1) Masked Multi-Head Attention
    2) Add & Norm
    3) Feed-Forward
    4) Add & Norm
- its decoder constrains to unidirectional context, as for language models.
- its encoder has bidirectional context and it does not have any masking in Multi-Head Attention part.

- Cross-attention in transformer:

  In the decoder, we have attention that looks more like what we saw last week.

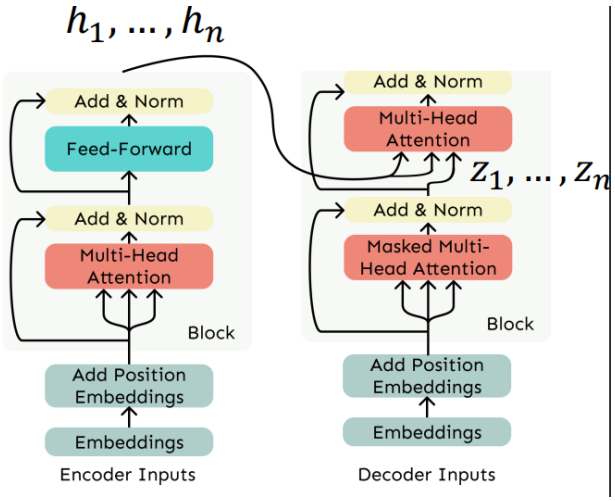
  Let $h_1, \dots, h_n$ be **output** vectors **from** the Transformer **encoder**; $x_i \in \mathbb{R}^d$

  Let $z_1, \dots, z_n$ be input vectors from the Transformer **decoder**, $z_i \in \mathbb{R}^d$

  Then keys and values are drawn from the **encoder** (like a memory):

  - $k_i = Kh_i, v_i = Vh_i$.

  And the queries are drawn from the **decoder**, $q_i = Qz_i$.



Problems with transformers:

- Quadratic compute in self-attention => our computation grows quadratically with the sequence length which was linearly for RNNs:
  However, its total number of operations grows as $O(n^2 d)$, where n is the sequence length, and d is the dimensionality.

  Solution: Linformer => map the sequence length dimension to a lower-dimensional space for values, keys
- Position representations => use Relative linear position attention or Dependency syntax-based position

---

Pretraining:

- Make sure your model can process large-scale, diverse datasets
- Don't use labeled data (otherwise you can't scale!)
- Compute-aware scaling

- Why it helps?

  Consider, provides parameters $\hat{\theta}$ by approximating $\min_{\theta} \mathcal{L}_{\text{pretrain}}(\theta)$.

  - (The pretraining loss.)

  Then, finetuning approximates $\min_{\theta} \mathcal{L}_{\text{finetune}}(\theta)$, starting at $\hat{\theta}$.

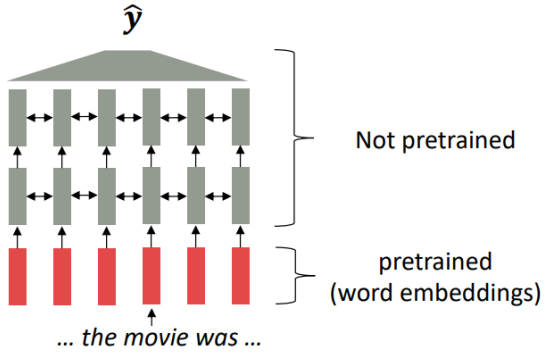
  - (The finetuning loss)

  The pretraining may matter because stochastic gradient descent sticks (relatively) close to $\hat{\theta}$ during finetuning.

  - So, maybe the finetuning local minima near $\hat{\theta}$ tend to generalize well!
  - And/or, maybe the gradients of finetuning loss near $\hat{\theta}$ propagate nicely!

⇨ pretrained word embeddings:

- Start with pretrained word embeddings
- The training data we have for our downstream task (like question answering) must be sufficient to teach all contextual aspects of language



⇨ pretraining whole models:

- effective at representations of language, parameter initializations, Probability distributions

⇨ Pretraining through language modeling:

1. train a neural network to perform language modeling on a large amount of text (learn general things). And then, save the network parameters
2. finetune on the specific task

- loss function: cross-entropy => we want the model to assign a high probability to true word w.
-

A language's vocabulary: a fixed vocab of tens of thousands of words, built from the training set.

All novel words seen at test time are mapped to a single UNK.

Subword modeling: include a wide range of methods for reasoning about structure below the word level. => at training and testing time, each word is split into a sequence of known subwords.

Byte-pair encoding: a strategy for defining a subword vocabulary. Steps:

1. Start with a vocabulary containing only characters and an “end-of-word” symbol.
2. Using a corpus of text, find the most common adjacent characters a,b”; add “ab” as a subword.
3. Replace instances of the character pair with the new subword; repeat until desired vocab size

⇨ Common words end up being a part of the subword vocabulary, while rarer words are split into components.

⇨ In the worst case, words are split into as many subwords as they have characters.

Pretraining for three types of LLMs’ architectures:

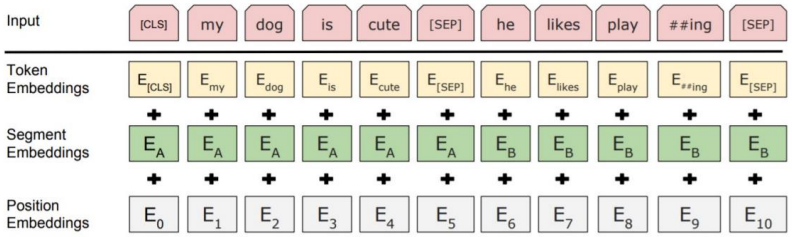
1. Encoders:
    - gets bidirectional context
    - can condition on future
    - are usually fine-tuned (trained on supervised data)
    - used for classification tasks
    - replace some fraction of words in the input with a special [MASK] token and predict these words

$$h_1, \ldots, h_T = \text{Encoder}(w_1, \ldots, w_T)$$
$$y_i \sim Aw_i + b$$

Only add loss terms from words that are “masked out.” If $\tilde{x}$ is the masked version of x, we’re learning $p_\theta(x|\tilde{x})$. Called **Masked LM**.

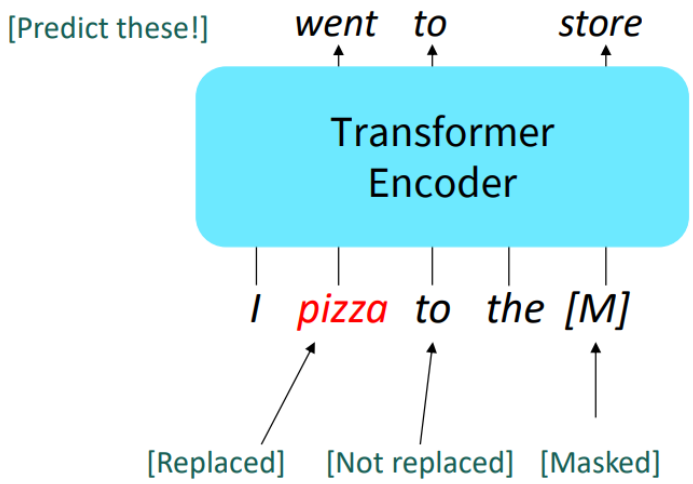
    - a sample of masked LLM is BERT. Properties of BERT:
        - Pretraining is expensive and impractical on a single GPU

- Finetuning is practical and common on a single GPU
- The pretraining input to BERT was two separate contiguous chunks of text. BERT was trained to predict whether one chunk follows the other or is randomly sampled



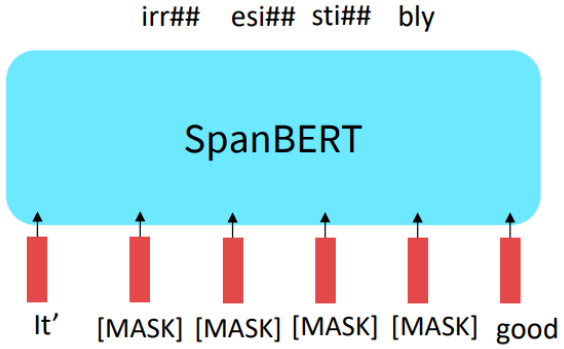
- Is pr-trained for two reasons:
    1) Masked Language Model (MLM)
    2) Next Sentence Prediction (NSP)

- BERT steps:
    1) Predict a random 15% of (sub)word tokens.
    2) Replace input word with [MASK] 80% of the time
    3) Replace input word with a random token 10% of the time
    4) Leave input word unchanged 10% of the time (but still predict it!)



- Extensions of BERT:
    1) RoBERTa: mainly just train BERT for longer and remove next sentence prediction.

2) SpanBERT: masking contiguous spans of words makes a harder, more useful pretraining task.

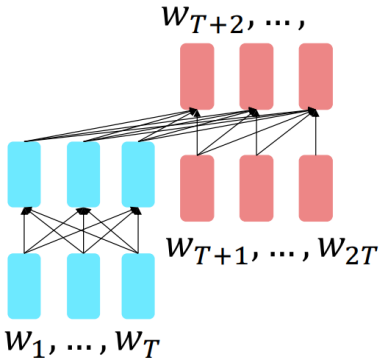


- Disadvantages of encoders:
  - If your task involves generating sequences, BERT and other pretrained encoders don't naturally lead to nice autoregressive (1-word-at-a-time) generation methods.

2. Encoder-Decoders:
   - a prefix of every input is provided to the encoder and is not predicted.

$$h_1, \dots, h_T = \text{Encoder}(w_1, \dots, w_T)$$
$$h_{T+1}, \dots, h_2 = Decoder(w_1, \dots, w_T, h_1, \dots, h_T)$$
$$y_i \sim Ah_i + b, i > T$$

The **encoder** portion benefits from bidirectional context; the **decoder** portion is used to train the whole model through language modeling.



   -
   - The encoder portion benefits from bidirectional context.
   - The decoder portion is used to train the whole model through language modeling.
   - Used in text preprocessing
   - Trained to map from one sequence to another => like in MT or SR
   - a sample of encoder-decoder model is T5 => using span corruption and it can be finetuned to answer a wide range of questions, retrieving knowledge from its parameters
   - T5's steps:

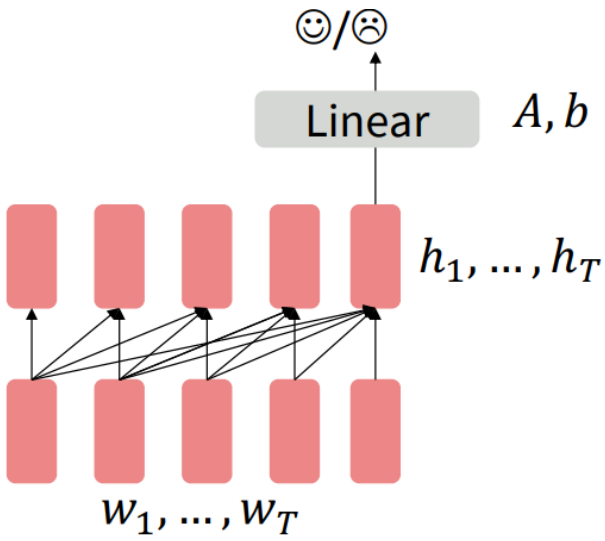
1) Replace different-length spans from the input with unique placeholders
2) Decode out the spans that were removed!

3. Decoders:
   - can't condition on future words
   - All the biggest pretrained models are Decoders
   - Are called Left-to-right LLMs or Autoregressive LLMs or Causal LLMs
   - can be finetune them by training a classifier on the last word's hidden state.

$$h_1, \dots, h_T = \text{Decoder}(w_1, \dots, w_T)$$
$$y \sim Ah_T + b$$

Where A and b are randomly initialized and specified by the downstream task.

Gradients backpropagate through the whole network.

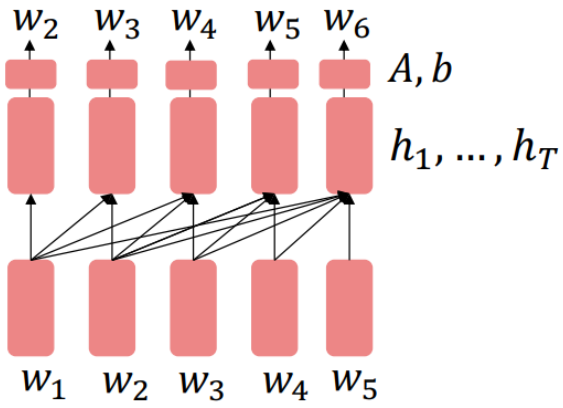
   - 

   - pretrain decoders as language models and then use them as generators

- helpful in tasks where the output is a sequence with a vocabulary like that at pretraining time
  - Dialogue (context=dialogue history)
  - Summarization (context=document)

$$h_1, \ldots, h_T = \text{Decoder}(w_1, \ldots, w_T)$$
$$w_t \sim Ah_{t-1} + b$$

Where A, b were pretrained in the language model!

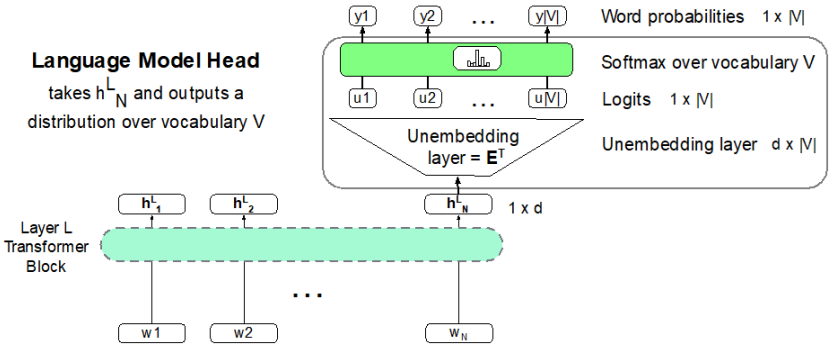


- samples are GPT, GPT-2, GPT-3, Llama 2
- GPT-3 perform some kind of learning without gradient steps simply from examples you provide within their contexts => In-Context learning which seems to specify the task to be performed, and the conditional distribution mocks performing the task to a certain extent.

LLMs:

- Difference of LLMs and N-Grams: n-grams are trained on counts computed from lots of text but llms are trained by learning to guess the next word.
- Both the LLMs and N-Grams assign probabilities to sequences of words and generate text by sampling possible next words.
- Properties of LLMs:
  1. Self-supervised learners

2. There is the need of a lot of text and computation
3. are built out of transformers => the input of transformers is word embedding + position embedding!
4. First, we have transformers and then its output is sent into the LMs' Head:



In LM Head we have these parts: Unembedding layer, Logits, Softmax over vocabulary, word probabilities (generating word)

Types of word embedding:

1. static word embedding: The embedding for a word doesn't reflect how its meaning changes in context
2. contextual word embedding: a representation of meaning of a word should be different in different contexts.
    - Each word has a different vector that expresses different meanings depending on the surrounding words
    - how to compute? By attention!
      Attention: A mechanism for helping compute the embedding for a token by selectively attending to and integrating information from surrounding tokens (at the previous layer). => a method for doing a weighted sum of vectors.

Harms of LLMs:

1. Hallucination => to address: using RAGs
2. Copyright
3. Privacy
4. Toxicity and Abuse
5. Misinformation

Conversational agents:

- Conversational agents (dialogue systems, dialogue agents, chatbots) are systems designed to interact with humans via conversations, either in text or speech
- Two kinds:
    1. Chatbots:
        - can be extended conversations with the goal of mimicking the unstructured conversations or ‘chats’ characteristic of informal human-human interaction
        - used for fun
        - Good for narrow, scriptable applications
        - They don't really understand
        - Giving the appearance of understanding may be problematic
        - Three types:
            1) Rule-based: Pattern-action rules or A mental model. => expensive and brittle
            2) Corpus-based: large datasets of human-human conversations => using information retrieval to copy a human response from a previous conversation, or using an encoder-decoder system to generate a response from a user utterance. => mirror training data
            3) Hybrid Architectures: using a series of different generators
    2. Goal-based (frame-based) dialogue agents: solve some tasks => interfaces to personal assistants like booking, robots, …
        - Done by building around a knowledge structure representing user intentions called the frame (having One or more frames)
        - Frames (or domain ontology) are a set of slots, to be filled with information of a given type. Each frame associated

with a question to the user.

Slot	Type	Question
ORIGIN	city	"What city are you leaving from?
DEST	city	"Where are you going?
DEP DATE	date	"What day would you like to leave?
DEP TIME	time	"What time would you like to leave?
AIRLINE	line	"What is your preferred airline?

- High precision
- Can provide coverage if the domain is narrow
- Can be expensive and slow to create rules
- Can suffer from recall problems

Properties of Human Conversation:

Turn: each contribution is a turn:

- a turn can be a sentence or a word or multiple sentences.
- Issue: turn taking and interruptions
- Barge-in: understanding when the user is talking which will allow the user to interrupt
- Latching: speakers start their turns almost immediately after the other speaker finishes, without a long pause
- End-pointing: The task for a speech system of deciding whether the user has stopped talking => hard because of noise or the people's pause in the middle of their turns

Speech Acts:

1. Constatives: متعهد ساختن گوینده به چیزی (پاسخ دادن، ادعا، تأیید، انکار، مخالفت، بیان)
2. Directives: تلاش های گوینده برای وادار کردن مخاطب به انجام کاری (توصیه، درخواست، منع، دعوت، سفارش، درخواست)
3. Commissives: متعهد کردن گوینده به برخی از اقدامات آینده (وعده، برنامه ریزی، نذر، شرط بندی، مخالفت)
4. Acknowledgments: بیان نگرش گوینده در مورد شنونده با توجه به برخی از اقدامات اجتماعی (عذرخواهی، سلام، تشکر، پذیرش یک تشکر)

Grounding: means acknowledging that the hearer has understood the speaker

- common ground: the participants to establish what they both agree on => by saying OK, repeating or asking new related question
- Principle of closure: Agents performing an action require evidence that they have succeeded in performing it.

Structures of conversation:

1. Local structure between adjacency pairs:
   - Question… Answer
   - Proposal… Acceptance/Rejection
   - Compliments ("Nice jacket!") … Downplayer ("Oh, this old thing?")
2. Subdialogue: dialogue acts aren't always followed immediately by their second pair part. The two parts can be separated by a side sequence.
3. Presequences: a user starts with a question about the system's capabilities before making a request.
4. Inference: The speaker seems to expect the hearer to draw certain inferences => not directly answer the question

Conversational Initiative: Some conversations are controlled by one person => Most human conversations have mixed initiative (I lead, then you lead, then I lead) which is hard for NLP systems!

- User initiative: user asks or commands, system responds
- System initiative: system asks user questions to fill out a form, user can't change the direction

Rule-based chatbots:

ELIZA:

- was designed to simulate a Rogerian psychologist, based on a branch of clinical psychology whose methods involve drawing the patient out by reflecting patient's statements back at them => knowing almost nothing of the real world but ask you to gain knowledge without showing that they do not know!
- Pattern rule: Rules are organized by **keywords** and each keyword has a pattern and a list of possible transform

- Keywords are associated with a rank, with specific words being more highly ranked, and more general words ranking lower

```
function ELIZA GENERATOR(user sentence) returns response

Find the word w in sentence that has the highest keyword rank
  if w exists
      Choose the highest ranked rule r for w that matches sentence
      response ← Apply the transform in r to sentence
      if w = 'my'
          future ← Apply a transformation from the 'memory' rule list to sentence
          Push future onto memory stack
   else (no keyword applies)
     either
         response ← Apply the transform for the NONE keyword to sentence
     or
         response ← Pop the top response from the memory stack
   return(response)
```

-
- If no keyword matches, ELIZA chooses a non-committal response like “PLEASE GO ON”, “THAT’S VERY INTERESTING”, or “I SEE”.
- Whenever “MY” is highest keyword, randomly select a transform on the MEMORY list, apply to sentence, then store on a (first-in-first-out) queue. Later, if no keyword matches a sentence, return the top of the MEMORY queue instead => memory list:

```
(MEMORY MY
 (0 MY 0 = LETS DISCUSS FURTHER WHY YOUR 3)
 (0 MY 0 = EARLIER YOU SAID YOUR 3)
 (0 MY 0 = DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT
YOUR 3))
```

- Ethical implications: deeply emotionally involved, Privacy
- Value-sensitive design: consider during the design process the benefits, harms and possible stake-holders of the resulting system

PARRY:

- Another chatbot with a clinical psychology focus
- The first system to pass a version of the Turing test
- Used to study schizophrenia
- Much richer than Eliza in:
    - control structure
    - language understanding capabilities
    - model of mental state

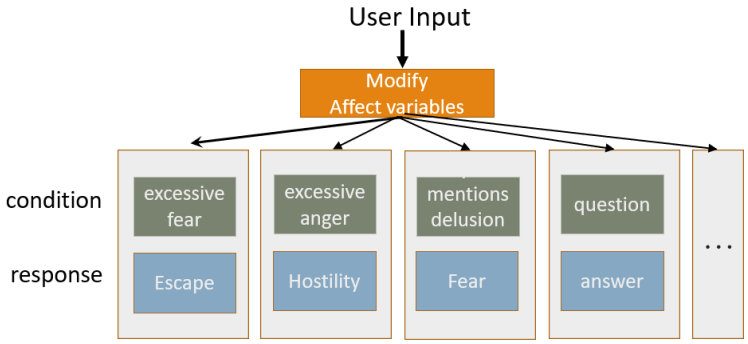
- variables modeling levels of Anger, Fear, Mistrust

- variables: Fear (0-20), Anger (0-20), Mistrust (0-15)

  Start with all variables low

  After each user turn
  - Each user statement can change Fear and Anger
    - E.g., Insults increases Anger, Flattery decreases Anger
    - Mentions of his delusions increase Fear
  - Else if nothing malevolent in input
    - Anger, Fear, Mistrust all drop

- Parry's responses depend on mental state



Corpus-based chatbots: (response generation systems)

- Two kinds:
  1. Response by retrieval: Use information retrieval to grab a response (that is appropriate to the context) from some corpus:
     - Classic IR method:

       Response by retrieval: classic IR method

       1. Given a user turn *q*, and a training corpus *C* of conversation
       2. Find in C the turn *r* that is most similar (tf-idf cosine) to *q*
       3. Say r

       $$\text{response}(q,C) = \underset{r \in C}{\text{argmax}} \frac{q \cdot r}{|q||r|}$$

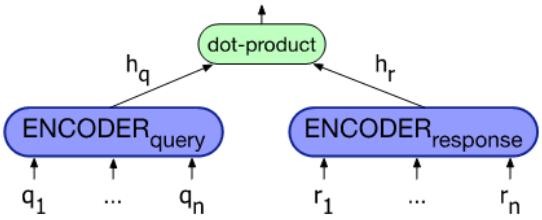
     - neural IR method: (bi-encoder model) => we train two separate encoders, one to encode the user query and one to encode the candidate response, and use the dot product between these two vectors as the score.

Response by retrieval: neural IR method

1. Given a user turn *q*, and a training corpus *C* of conversation
2. Find in C the turn *r* that is most similar (BERT dot product) to *q*
3. Say r

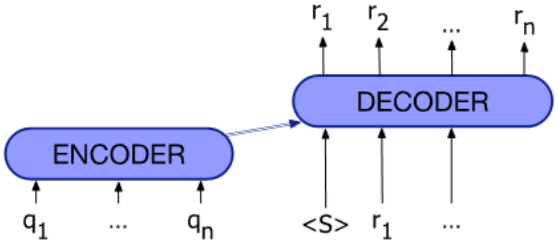
$$h_q = \text{BERT}_Q(\text{q})[\text{CLS}]$$
$$h_r = \text{BERT}_R(\text{r})[\text{CLS}]$$
$$\text{response}(q,C) = \underset{r \in C}{\text{argmax}}\, h_q \cdot h_r$$

represent the query and candidate response as the [CLS] token of the respective encoders, then we choose whichever turn in our corpus has the highest dot product with the query.



(a) Response by Retrieval

2. Response by generation: Use a language model or encoder-decoder to generate the response given the dialogue context:
   - fine-tune a large language model on conversational data and use it as a response generator method => like gpt-2
   - Generate each token rt of the response by conditioning on the encoding of the entire query q and the response so far r1...rt−1 method



(b) Response by Generation

    - we normally include a longer context, forming the query not just from the user's turn but from the entire conversation
    - basic encoder-decoder have a tendency to produce predictable but repetitive and therefore dull responses => we can use beam search or diversity-focused training objectives rather than choosing the most likely response or add minimum length constraints
    - combination of those two methods: use IR to retrieve passages from Wikipedia (retrieving and refining knowledge), concatenate each Wikipedia sentence to the dialogue context with a separator token, Give as encoder context to the encoder-decoder model, which learns to incorporate text into its response

- Modern ones are very data-intensive
- They commonly require hundreds of millions or billions of words
- Datasets:
    - Transcripts of telephone conversations between volunteers
    - Movie dialogue
    - Hire human crowdworkers to have conversations
    - Pseudo-conversations from public posts on social media

Frame-based dialogue agents:

1. GUS (frame-based) architecture:
    - used in industrial task-based dialogue agents
    - Control structure: System asks questions of user (using pre-specified question templates associated with each slot of each frame), filling any slots that user specifies, when frame is filled, do database query. => a set of hand-build production rules for filling frames and taking actions
    - System must detect which slot of which frame user is filling and switch dialogue control to that frame.
    - In GUS, there are three main steps:
        1) Domain classification

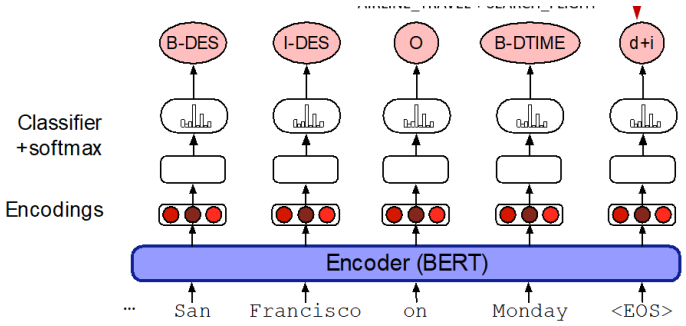
2) Intent Determination: what general task or goal is the user trying to accomplish
3) Slot Filling: Extract the actual slots and fillers

- Rule-based Slot-filling: using handwritten rules, often as part of the condition action rules like writing regular expressions or grammar rules
- Generating responses: template-based generation => sentences created by these templates are often called prompts.

2. Dialogue-state architecture (Belief-State Architecture):
   - Extension of GUS
   - Has dialogue acts, more ML, better generation
   - More common in research systems
   - Has 6 components:
     1) Automatic Speech Recognition (ASR)
     2) Spoken Language Understanding (SLU): extracts slot fillers from the user's utterance using machine learning => Build a classifier to map from one to the other which requirements lots of labeled data.
        We create a B and I tag for each slot-type and convert the training data to this format

```
O O    O  O   O  B-DES I-DES     O  B-DEPTIME I-DEPTIME  O
I want to fly to San   Francisco on Monday    afternoon  please
```

        How to create tags: using contextual embeddings: The input is a series of words w1...wn, which is passed through a contextual embedding model to get contextual word representations. This is followed by a feedforward layer and a softmax at each token position over possible BIO tags, with the output a series of BIO tags s1...sn.



After having BIO tags, extract the filler string for each slot, then normalize it to the correct form in the ontology

3) Dialog State Tracker (DST): maintains the current state of the dialogue (user's most recent dialogue act) plus the entire set of slot-filler constraints from user

   Dialogue act interpretation algorithm: done by supervised classification trained on hand-labeled dialog acts, predicting the dialogue act tag based on embeddings representing the current input sentence and the prior dialogue acts.

   Sometimes correction is needed => use will do by repeating, rephrasing or saying "no" to a confirmation question

features	examples
lexical	words like "no", "correction", "I don't", swear words, utterance length
semantic	similarity (word overlap or embedding dot product) between the candidate correction act and the user's prior utterance
phonetic	phonetic overlap between the candidate correction act and the user's prior utterance (i.e. "WhatsApp" may be incorrectly recognized as "What's up")
prosodic	hyperarticulation, increases in F0 range, pause duration, and word duration, generally normalized by the values for previous sentences
ASR	ASR confidence, language model probability

4) Dialog Policy: decides what the system should do or say next (when to answer questions, when to ask a clarification question, …)

At turn *i* predict action A_i to take, given entire history:

$$\hat{A}_i = \underset{A_i \in A}{\operatorname{argmax}} P(A_i | (A_1, U_1, ..., A_{i-1}, U_{i-1})$$

Simplify by just conditioning on the current dialogue state (filled frame slots) and the last turn and turn by system and user:

$$\hat{A}_i = \underset{A_i \in A}{\operatorname{argmax}} P(A_i | \text{Frame}_{i-1}, A_{i-1}, U_{i-1})$$

To make sure that we understood user, there are two mechanisms:

- Confirming: understandings with the user => Explicit: asks the user a direct question to confirm the system's understanding.
  Implicit: demonstrate its understanding as a grounding strategy, for example repeating back the system's understanding as part of asking the next question => increase the length of conversation but it is easier for user to answer!
- Rejecting: utterances that the system is likely to have misunderstood. (give the user a prompt like *I'm sorry, I didn't understand that*.) => using progressive prompting or escalating detail (ask in detail about what is misunderstood!)

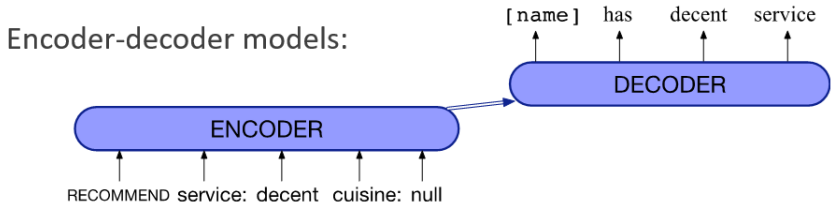
Systems could use set confidence thresholds:

$< \alpha$	low confidence	reject
$\geq \alpha$	above the threshold	confirm explicitly
$\geq \beta$	high confidence	confirm implictly
$\geq \gamma$	very high confidence	don't confirm at all

5) Natural Language Generation (NLG): produce more natural, less templated utterances
   There are two stages:

- content planning (what to say): assume this is done by dialog policy (Chosen the dialogue act to generate and chosen some attributes (slots and values) that the planner wants to say to the user)
- sentence realization (how to say it):

Delexicalization: replacing words in the training set that represent slot values with a generic placeholder token => done by encoder decoder models trained on large hand-labeled corpora of task-oriented dialogue



6) Text to Speech (TTS)

- Dialog acts: Combine the ideas of speech acts and grounding into a single representation

Tag	Sys	User	Description
HELLO($a=x, b=y, \ldots$)	✓	✓	Open a dialogue and give info $a=x, b=y, \ldots$
INFORM($a=x, b=y, \ldots$)	✓	✓	Give info $a=x, b=y, \ldots$
REQUEST($a, b=x, \ldots$)	✓	✓	Request value for a given $b=x, \ldots$
REQALTS($a=x, \ldots$)	χ	✓	Request alternative with $a=x, \ldots$
CONFIRM($a=x, b=y, \ldots$)	✓	✓	Explicitly confirm $a=x, b=y, \ldots$
CONFREQ($a=x, \ldots, d$)	✓	χ	Implicitly confirm $a=x, \ldots$ and request value of d
SELECT($a=x, a=y$)	✓	χ	Implicitly confirm $a=x, \ldots$ and request value of a
AFFIRM($a=x, b=y, \ldots$)	✓	✓	Affirm and give further info $a=x, b=y, \ldots$
NEGATE($a=x$)	χ	✓	Negate and give corrected value $a=x$
DENY($a=x$)	χ	✓	Deny that $a=x$
BYE()	✓	✓	Close a dialogue

Evaluating chatbots and task-based dialogue:

- Task-based dialogue: mainly by measuring task performance: two measures:
  1. End-to-end evaluation (Task Success)
  2. Slot Error Rate for a Sentence:

$$\frac{\text{\# of inserted/deleted/subsituted slots}}{\text{\# of total reference slots for sentence}}$$

  3. User Satisfaction Survey: users interact with a dialogue system to perform a task and complete a questionnaire. Here are some sample multiple-choice questions responses are mapped into the range of 1 to 5, and then averaged over all questions to get a total user satisfaction rating.
  4. Efficiency cost:
     - total elapsed time for the dialogue in seconds
     - the number of total turns or of system turns

        - total number of queries
        - “turn correction ratio”: % of turns that were used to correct errors
    5. Quality cost:
        - number of ASR rejection prompts
        - number of times the user had to barge in
- Chatbots: mainly by human evaluation:
    1. Participant evaluation: human rate 8 dimensions of quality: avoiding repetition, interestingness, making sense, fluency, listening, inquisitiveness, humanness, engagingness
    2. observer (third-party) evaluation: look two conversations and compare their Engagingness, Interestingness, Humanness, Knowledgeable
    3. Automatic evaluation: They correlate poorly with human judgements so are not been used! => Adversarial Evaluation is looking to which is train a ``Turing-like'' classifier to distinguish between human responses and machine responses.The more successful a dialogue system is at fooling the evaluator, the better the system.

Ethical design:

1. Safety: Systems abusing users, distracting drivers, or giving bad medical advice
2. Representational harm: Systems demeaning particular social groups
3. Privacy: Information Leakage => Accidental information leakage or Intentional information leakage

---

Application 1: (MT)

- Statistical MT:

- Suppose we're translating French → English.
- We want to find best English sentence *y*, given French sentence *x*

$$\text{argmax}_y P(y|x)$$

- Use Bayes Rule to break this down into two components to be learned separately:

$$= \text{argmax}_y P(x|y)P(y)$$

Translation Model

Models how words and phrases should be translated (*fidelity*). Learned from parallel data.

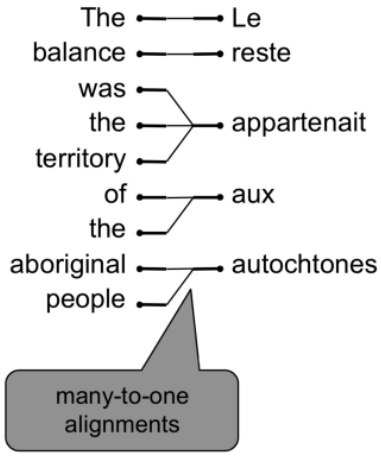
Language Model

Models how to write good English (*fluency*). Learned from monolingual data.

For computing p(x|y), we should learn alignment and then compute p(x,a|y) where a is the alignment. => the model concluded from Bayes Rule is called "noisy-channel model".

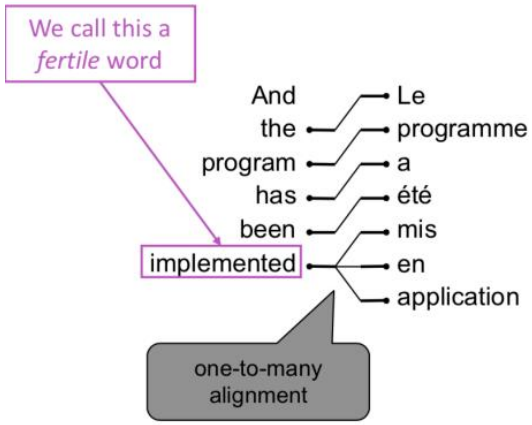
Alignment is the correspondence between particular words in the translated sentence pair:

- Typological differences between languages lead to complicated alignments
- Are latent variables: They aren't explicitly specified in the data! Require the use of special learning algorithms (like Expectation-Maximization) for learning them.
- Some words have no counterpart
- Different types of alignment:

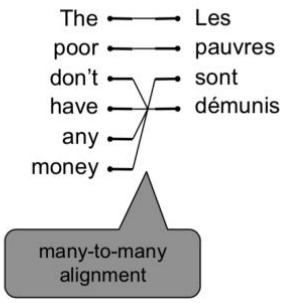


1. many-to-one

2. one-to-many



3. many-to-many (phrase-level)



- problems in finding alignment:

1. Lexical divergences:
    - The different senses of homonymous (همنام) words generally have different translations
    - The different senses of polysemous (چند معنایی) words may also have different translations
    - Lexical specificity
    - Morphological divergences (تغییر کلمات با در نظر گرفتن جنسیت)
2. Syntactic divergences:
    - Word order => [SVO (Sbj-Verb-Obj), SOV, VSO,…] or what?
    - Head-marking vs. dependent-marking

      Dependent-marking (English) *the man's house*

      Head-marking (Hungarian) *the man house-his*
    - Pro-drop languages => omit pronouns
3. Semantic differences:

- progressive aspect
  - English has a **progressive aspect**:
    'Peter swims' vs. 'Peter is swimming'
  - German can only express this with **an adverb**:
    'Peter schwimmt' vs. 'Peter schwimmt gerade' ('swims currently')
- Motion events

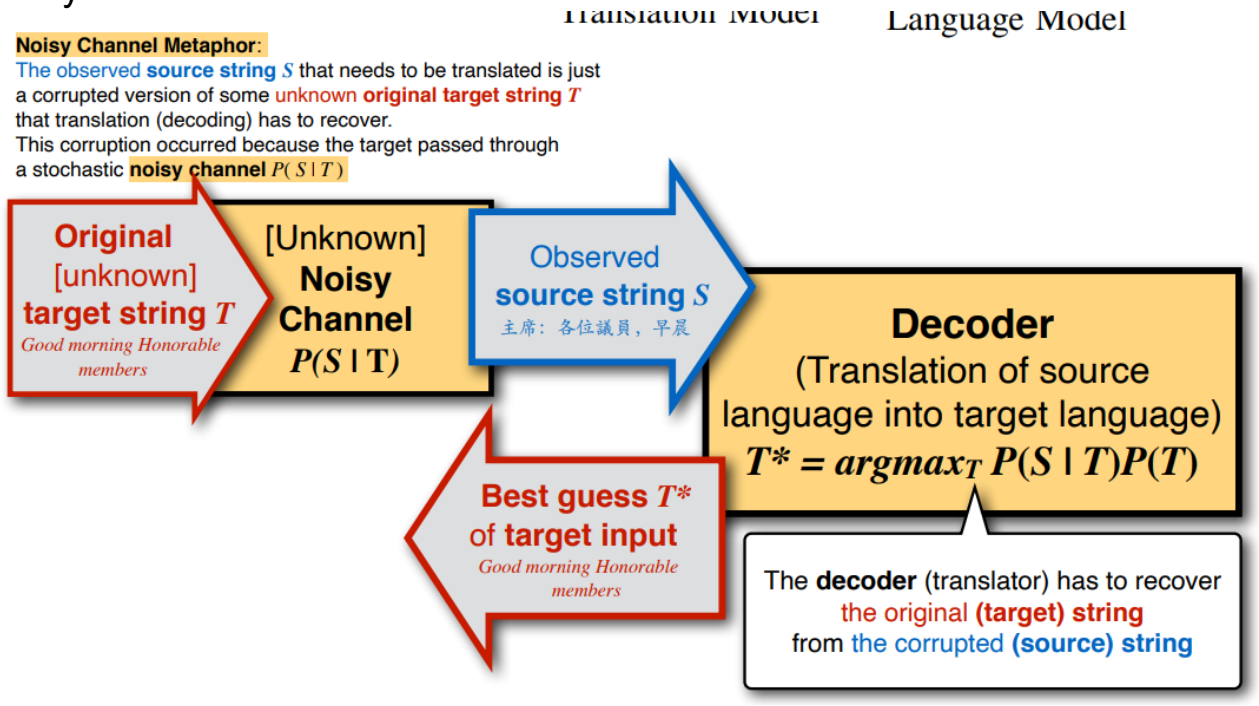
  Motion events have two properties:
  - **manner** of motion (*swimming*)
  - **direction** of motion (*across* the lake)

  Languages express either the manner with a verb and the direction with a 'satellite' or vice versa (L. Talmy):
  English (satellite-framed): *He* [*swam*]$_{\text{MANNER}}$ **[*across*]$_{\text{DIR}}$** *the lake*
  French (verb-framed): *Il a* **[*traversé*]$_{\text{DIR}}$** *le lac* **[*à la nage*]$_{\text{MANNER}}$**

$$\text{Target}^* = \text{argmax}_{\text{Target}} \underbrace{P(\text{Source} \mid \text{Target})}_{\text{Translation Model}} \underbrace{P(\text{Target})}_{\text{Language Model}}$$

So, we want to compute isy Channel Metaphor: :

Briefly:

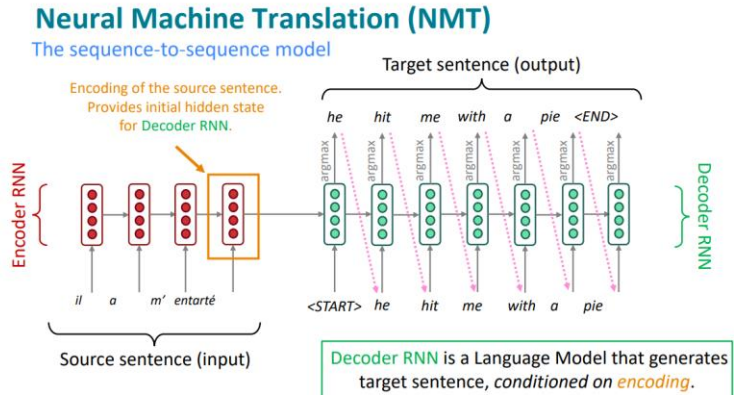


Let's break it down:

- The translation model P(S|T) is intended to capture the faithfulness of the translation. (the noisy channel) => we need to score S, and don't need it to generate a grammatical S, so it is a simple model.

- P(S|T) needs to be trained on a parallel corpus.
- P(T) is intended to capture the fluency of the translation.
- P(T) can be trained on a (very large) monolingual corpus.
- Decoding in SMT: Impose strong independence assumptions in model, use dynamic programming for globally optimal solutions (e.g. Viterbi algorithm) instead of enumerating every possible y and calculate the probability which is really expensive

- Neural MT: => Need a big parallel corpus for training



Different methods for decoding:

1. Greedy decoding: generate (“decode”) the target sentence by taking argmax on each step of the decoder => take most probable word on each step.
   Stop: until the model produces an <END> token.
   Problem of this method: there is not any way to undo decisions
2. Exhaustive search decoding: on each step t of the decoder, we’re tracking V^t possible partial translations, where V is vocab size, this O(V^T) complexity is too expensive! => this is trying all possible sequences y.
3. Beam search decoding: On each step of decoder, keep track of the k most probable partial translations (which we call hypotheses) => k is the beam size!

A hypothesis $y_1, \ldots, y_t$ has a score which is its log probability:

$$\text{score}(y_1, \ldots, y_t) = \log P_{\text{LM}}(y_1, \ldots, y_t | x) = \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$$

- Scores are all negative, and higher score is better
- We search for high-scoring hypotheses, tracking top *k* on each step

Beam search is not guaranteed to find optimal solution but is efficient.

Stop:

- different hypotheses may produce <END> tokens on different timesteps => when a hypothesis produces <END>, that hypothesis is complete. So, place it aside and continue exploring other hypotheses via beam search
- continue until reach timestep T (where T is some pre-defined cutoff)
- or continue until have at least n completed hypotheses (where n is pre-defined cutoff) => how to select the best hypotheses between all completed ones?

  Normalize by length. Use this to select top one instead:

  $$\frac{1}{t} \sum_{i=1}^{t} \log P_{\text{LM}}(y_i | y_1, \ldots, y_{i-1}, x)$$

  we normalized as different hypothesizes have different length and this will affect their score!

Advantages of NMT in comparison with SMT:

1. Better performance
2. More fluent
3. Better use of context
4. Better use of phrase similarities
5. A single neural network to be optimized end-to-end => no subcomponents to be individually optimized
6. Requires much less human engineering effort => no feature engineering and same method for all language pairs

Disadvantages of NMT in comparison with SMT:

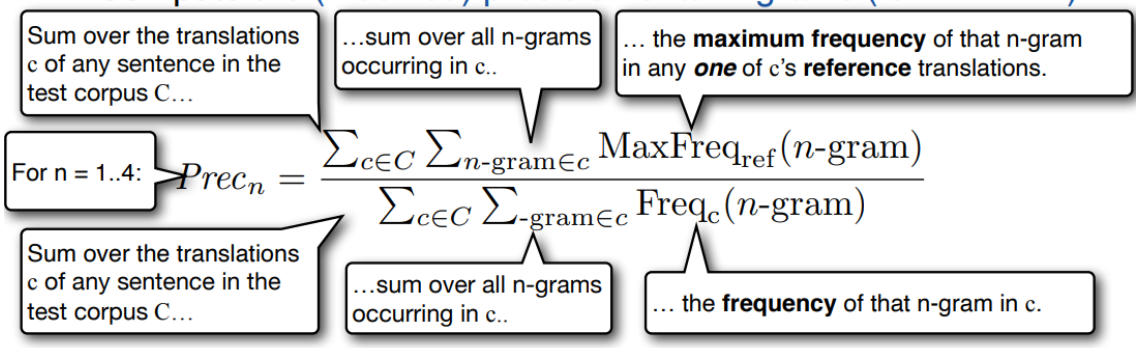
1. less interpretable => Hard to debug

2. difficult to control => can't easily specify rules or guidelines for translation
3. Safety concerns

Evaluate NMT:

1. BLEU compares the machine-written translation to one or several human-written translation(s), and computes a similarity score based on:
    - n-gram precision (usually for 1, 2, 3 and 4-grams) => How many n-grams in the candidate translation occur also in one of the reference translation:

Compute the (modified) precision of all *n*-grams (for *n* = 1…4)



    - plus a penalty for too-short system translations

Penalize short candidate translations by a **brevity penalty** BP

c = length (number of words) of the whole candidate translation corpus

r = Pick for each candidate the reference translation that is closest in length; sum up these lengths.

Brevity penalty $BP = \exp(1-c/r)$ for $c \leq r$; $BP = 1$ for $c > r$

(BP ranges from e for c=0 to 1 for c=r)

The BLEU score is the geometric mean of the modified n-gram precision (for n=1..4), weighted by a brevity penalty BP:

$$\textbf{BLEU} = BP \times \exp\left(\frac{1}{N}\sum_{n=1}^{N} \log Prec_n\right)$$

Geometric mean for $a_1, \ldots, a_N > 0$ = N-th root of $\prod_{n=1}^{N} a_n$

$$\sqrt[N]{\prod_{n=1}^{N} a_n} = \left(\prod_{n=1}^{N} a_n\right)^{\frac{1}{N}} = \exp\left(\frac{1}{N}\sum_{n=1}^{N} \log a_n\right)$$

- overall:

2. Human evaluation:
    - ask human raters to judge the fluency and the adequacy of the translation (e.g. on a scale of 1 to 5)
    - give rater the sentence with one word replaced by blank. Ask rater to guess the missing word in the blank => correlated with fluency is accuracy on cloze task.
    - Can you use the translation to perform some task (e.g. answer multiple-choice questions about the text) => similar to adequacy is informativeness.

Problems in MT:

1. Out-of-vocabulary words
2. Domain mismatch between train and test data
3. Maintaining context over longer text
4. Low-resource language pairs
5. Failures to accurately capture sentence meaning
6. Pronoun (or zero pronoun) resolution errors
7. Morphological agreement errors
8. NMT picks up biases in training data
9. Uninterpretable systems do strange things

Application 2: (Question Answering and Information Retrieval)

HCI (Human-Computer Interaction): => unlike search engine, we want exact answer not doc!

1. Request NL (not keywords)
2. Request in SPARQL
3. Response in XML
4. Response in NL (not document)

QA models' steps:

1. User asks a question
2. Question analysis => classification and extraction, extended keywords, named entity recognition
3. Search of documents => which documents contain the answer
4. Extraction of answer => process candidate documents and extract the answer

Questions can be:

1. Factoid: simple like When, where, how many, how much, who, what:

   سؤالاتی هستند که به دنبال اطلاعات واقعی خاص یا پاسخ های مجزا هستند. این سؤالات معمولاً پاسخ‌های واضح و مشخصی دارند که اغلب کوتاه هستند و مستقیماً از یک زمینه یا منبع دانش قابل استخراج هستند.

2. Non-factoid: hard like Why, How:

   این سؤالات ممکن است نیاز به درک، تفسیر یا استدلال عمیق تری داشته باشد تا پاسخی رضایت بخش ارائه شود. این سوالات معمولا غیر واقعی اغلب شامل نظر، قضاوت، توضیح یا حدس و گمان هستند.

AskMSR QAS steps:

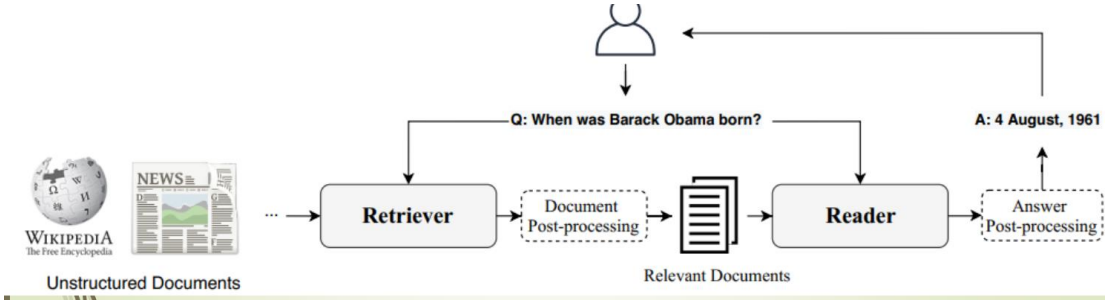
1. Asking question
2. Rewrite query
3. Use search engine

4. Collect summaries, extract N-Grams
5. Filter N-Grams
6. Tile N-Grams
7. N-Best answers

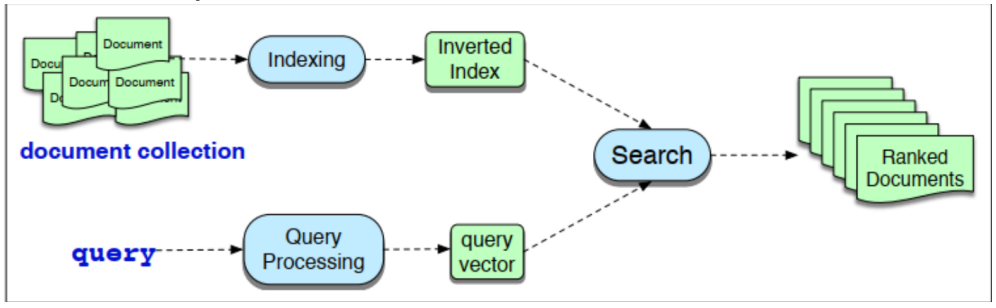
QAS can be:

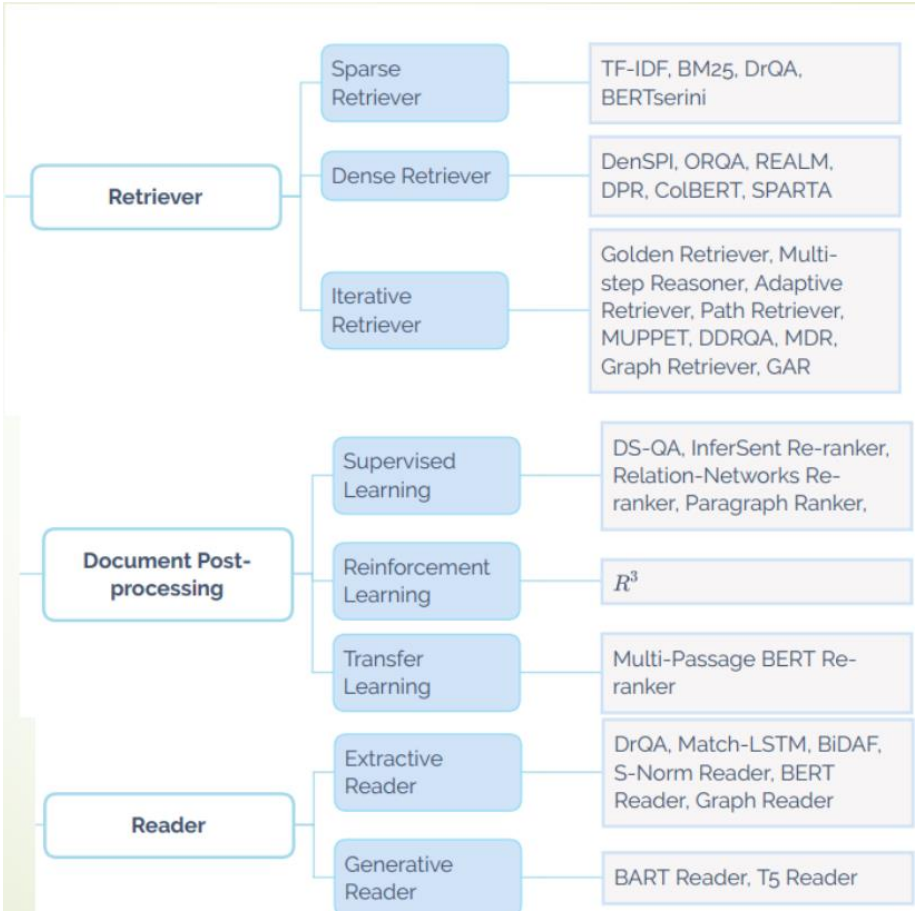
1. Open-domain:

این سیستم ها برای پاسخگویی به سؤالات از طیف گسترده ای از موضوعات بدون هیچ محدودیتی طراحی شده اند. همچنین این سیستم‌ها معمولاً از مدل‌های زبانی در مقیاس بزرگ استفاده می‌کنند که بر روی مجموعه متنی متنوعی از اینترنت آموزش داده شده‌اند. آنها متکی به درک زمینه سؤال و ایجاد پاسخ مرتبط بر اساس اطلاعات موجود، بدون محدود شدن به یک دامنه یا پایگاه داده خاص هستند.



In Retrieve part:

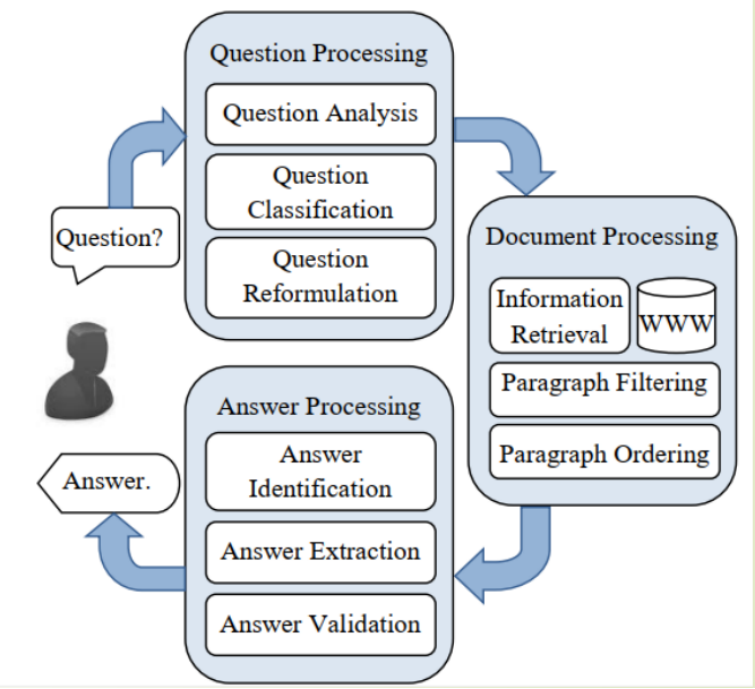




2. Closed-domain:

این نوع از سیستم ها برای پاسخ به سؤالات در یک دامنه یا حوزه موضوعی خاص طراحی شده اند. معمولاً بر داده های ساختاریافته یا پایگاه های دانش از پیش تعریف شده مرتبط با حوزه خاص متکی هستند. اغلب در برنامه های کاربردی تخصصی استفاده می شوند که در آن اطلاعات به خوبی تعریف شده و ساختار یافته است، مانند پشتیبانی مشتری.

QAS be like:



In QP => Keyword, convert to Machine

Understandable

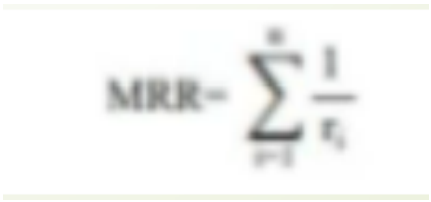
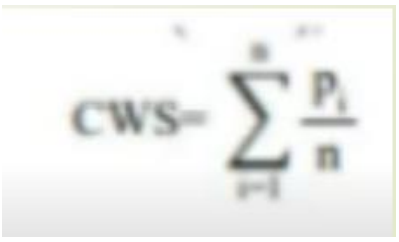
In DP => Filtering and ordering

In AP => Validation

Target: achieve Human understanding

Evaluation Metrics of Answers:

1. Relevance (of answer)
2. Correctness (of answer)
3. Conciseness (shortness of answer)
4. Completeness
5. Justification:
    - Precision => number of correct answers /number of questions answered
    - Recall => number of correct answers /number of questions to be answered
    - F-measure => 2* (Precision * recall) /Precision +recall (harmonic mean)

- MRR (mean reciprocal rank) 
- Confidence Weighted Score 

Application 3: (HMM)

Markov Chains:

- The probabilities of sequences of random variables, states, each of which can take on values from some set
- Markov assumption: if we want to predict the future in the sequence, all that matters is the current state => like a bigram language model

$$P(q_i = a | q_1 \ldots q_{i-1}) = P(q_i = a | q_{i-1})$$

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
$A = a_{11} a_{12} \ldots a_{N1} \ldots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$
$\pi = \pi_1, \pi_2, \ldots, \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{N} \pi_i = 1$

Hidden Markov Model:

- A hidden Markov model (HMM) allows us to talk about both observed events (like words that we see in the input) and hidden events (like part-of-speech tags) that we think of as causal factors in our probabilistic model

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
$A = a_{11} \ldots a_{ij} \ldots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{N} a_{ij} = 1 \quad \forall i$
$B = b_i(o_t)$	a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probability of an observation o_t (drawn from a vocabulary $V = v_1, v_2, ..., v_V$) being generated from a state q_i
$\pi = \pi_1, \pi_2, ..., \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

- HMM's assumptions:
    1. Markov assumption:

$$P(q_i|q_1 \ldots q_{i-1}) = P(q_i|q_{i-1})$$

    2. Output Independence:

$$P(o_i|q_1 \ldots q_i, \ldots, q_T, o_1, \ldots, o_i, \ldots, o_T) = P(o_i|q_i)$$

- each hidden state produces only a single observation. Thus, the sequence of hidden states and the sequence of observations have the same length.
- Three important computations:
    1. Likelihood: Given an HMM, lambda = (A, B) and an observation sequence O, determine the likelihood P(O|lambda):
        1) $$P(O,Q) = P(O|Q) \times P(Q) = \prod_{i=1}^{T} P(o_i|q_i) \times \prod_{i=1}^{T} P(q_i|q_{i-1})$$

        For an HMM with N hidden states and an observation sequence of T observations, there are N^T possible hidden sequences!
        2) Foreword Algorithm: using dynamic programming so that the possible hidden sequence will be N^2 * T:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t) \qquad \text{(A.12)}$$

The three factors that are multiplied in Eq. A.12 in extending the previous paths to compute the forward probability at time t are

$\alpha_{t-1}(i)$	the **previous forward path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

The whole algorithm will be:

1. Initialization:

$$\alpha_1(j) = \pi_j b_j(o_1) \quad 1 \le j \le N$$

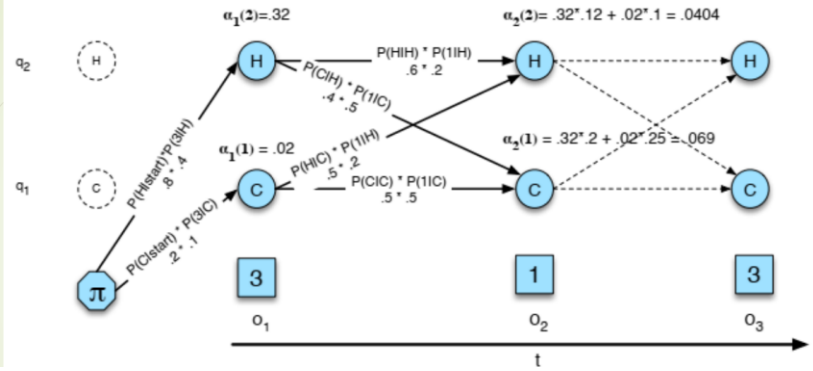
2. Recursion:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(o_t); \quad 1 \le j \le N, 1 < t \le T$$

3. Termination:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

Example:



2. Decoding: Given as input an HMM, lambda = (A, B) and a sequence of observations O = o1, o2, …., oT, find the most probable sequence of states Q = q1q2q3 …qT:

1) Naïve: For each possible hidden state sequence (HHH, HHC, HCH, etc.), we could run the forward algorithm and compute the likelihood of the observation sequence given that hidden state sequence.
2) Viterbi algorithm: using dynamic programming => like minimum edit distance:

$$v_t(j) = \max_{1 \le i \le N-1} v_{t-1}(i)\, a_{ij}\, b_j(o_t)$$

$v_{t-1}(i)$	the **previous Viterbi path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

The whole algorithm will be:

1. **Initialization:**

$$v_1(j) = \pi_j b_j(o_1) \qquad 1 \le j \le N$$
$$bt_1(j) = 0 \qquad 1 \le j \le N$$

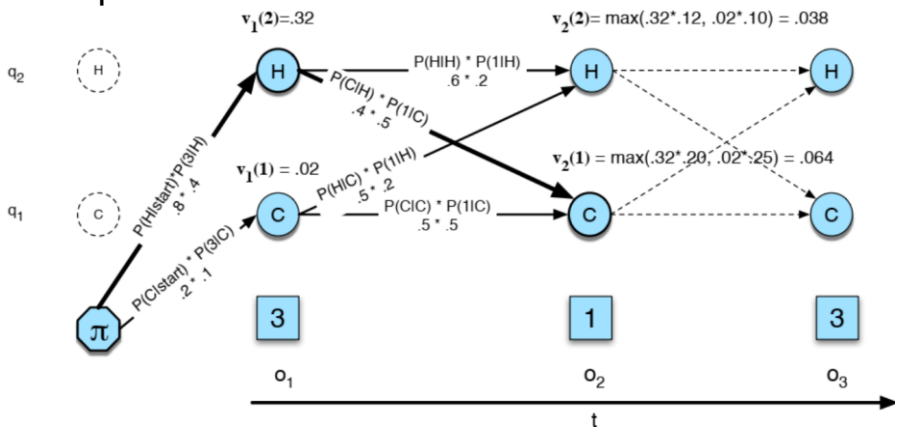
2. **Recursion**

$$v_t(j) = \max_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t); \quad 1 \le j \le N, 1 < t \le T$$
$$bt_t(j) = \operatorname{argmax}_{i=1}^{N} v_{t-1}(i)\, a_{ij}\, b_j(o_t); \quad 1 \le j \le N, 1 < t \le$$

3. **Termination:**

$$\text{The best score:} \quad P* = \max_{i=1}^{N} v_T(i)$$
$$\text{The start of backtrace:} \quad q_T* = \operatorname{argmax}_{i=1}^{N} v_T(i)$$

Example:



3. Learning: Given an observation sequence O and the set of possible states in the HMM, learn the HMM parameters A and B. => using Foreword-backward, or BaumBaum-Welch Welch algorithm:

The **backward** probability β is the probability of seeing the observations from time $t+1$ to the end, given that we are in state i at time t (and given the automaton λ): $\beta_t(i) = P(o_{t+1};o_{t+2} \ldots o_T \mid q_t = i;\lambda)$

We also use **forward** probability α ($P(O \mid \lambda)$)

Backward:

1. **Initialization:**

$$\beta_T(i) = 1, \quad 1 \le i \le N$$

2. **Recursion**

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij}\, b_j(o_{t+1})\, \beta_{t+1}(j), \quad 1 \le i \le N, 1 \le t < T$$

3. **Termination:**

$$P(O|\lambda) = \sum_{j=1}^{N} \pi_j\, b_j(o_1)\, \beta_1(j)$$

$$\xi_t(i,j) = P(q_t = i, q_{t+1} = j | O, \lambda)$$

$$\text{not-quite-}\xi_t(i,j) = P(q_t = i, q_{t+1} = j, O|\lambda)$$

$$P(X|Y,Z) = \frac{P(X,Y|Z)}{P(Y|Z)}$$

it will be:

$$\text{not-quite-}\xi_t(i,j) = \alpha_t(i)\, a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)$$

$$P(O|\lambda) = \sum_{j=1}^{N} \alpha_t(j)\beta_t(j)$$

$$\xi_t(i,j) = \frac{\alpha_t(i)\, a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{\sum_{j=1}^{N} \alpha_t(j)\beta_t(j)}$$

Here, in the E-step, we compute the expected state occupancy count lambda and the expected state transition count epsilon from the earlier A and B probabilities. In the M-step, we use them to recompute new A (all alphas) and B (all betas) probabilities.

function FORWARD-BACKWARD(*observations* of len T, *output vocabulary* V, *hidden state set* Q) **returns** $HMM=(A,B)$

initialize A and B
iterate until convergence
E-step

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{\alpha_T(q_F)} \quad \forall\, t \text{ and } j$$

$$\xi_t(i,j) = \frac{\alpha_t(i)\, a_{ij} b_j(o_{t+1})\beta_{t+1}(j)}{\alpha_T(q_F)} \quad \forall\, t,\ i, \text{ and } j$$

M-step

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1}\sum_{k=1}^{N} \xi_t(i,k)}$$

$$\hat{b}_j(v_k) = \frac{\sum_{t=1 s.t.\ O_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

return A, B

Application 4: (Automatic Speech Recognition (ASR) and Text-to-Speech)

Digitizing Sound Wave:

1. Sampling: measure its amplitude at a particular time; the sampling rate is the number of samples taken per second.
   Nyquist frequency: The maximum frequency wave that can be measured is one whose frequency is half the sample rate
2. Quantization: => All values that are closer together than the minimum granularity (the quantum size) are represented identically. We refer to each sample at time index n in the digitized, quantized waveform as x[n]

- The data conversation between two people ca be stored individually or in the same file. If we want to store individually, there are two methods:
    1) Compression format:

$$F(x) = \frac{\operatorname{sgn}(x)\log(1+\mu|x|)}{\log(1+\mu)} \quad -1 \le x \le 1$$

mui is 255 and x is the intensity of the signal.
    2) Linear: values are generally referred to as linear PCM values (PCM stands for pulse code modulation)

- Windowing: extract spectral features from a small window of speech that characterizes part of a particular phoneme (واج)
stationary = the signal inside of the window
We extract this roughly stationary portion of speech by using a window which is non-zero inside a region and zero elsewhere:

$$y[n] = w[n]s[n]$$

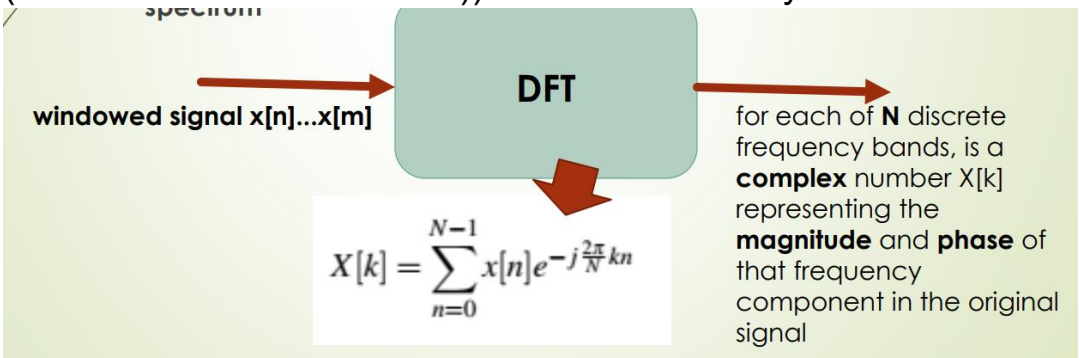
where y[n] is called frame
Three parameters of the window:
    1) window size or frame size of the window: its width in milliseconds
    2) frame stride (also called shift or offset) between successive windows
    3) shape of the window:
        - Rectangular: abruptly cuts off the signal at its boundaries, which creates problems when we do Fourier analysis.

$$rectangular \qquad w[n] = \begin{cases} 1 & 0 \le n \le L-1 \\ 0 & \text{otherwise} \end{cases}$$

        - Hamming: which shrinks the values of the signal toward zero at the window boundaries, avoiding discontinuities.

$$Hamming \qquad w[n] = \begin{cases} 0.54 - 0.46\cos\left(\frac{2\pi n}{L}\right) & 0 \le n \le L-1 \\ 0 & \text{otherwise} \end{cases}$$

- Spectral information: how much energy the signal contains at different frequency bands => by plotting the magnitude against the frequency, we can visualize the spectrum. (computing DFT (Discrete Fourier Transform)). We can do this by FFT.

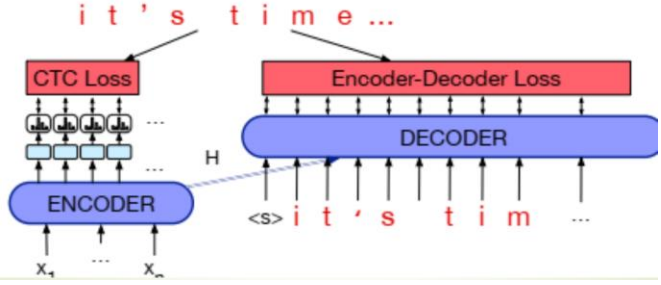


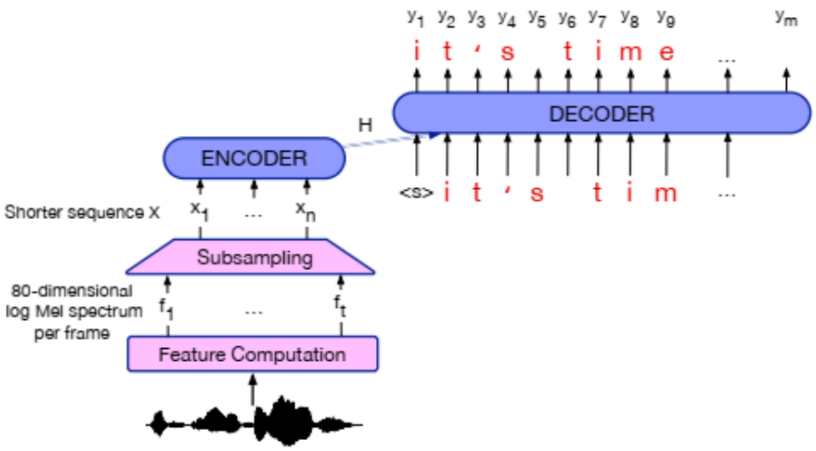
- Mel Filter Bank: Human hearing is not equally sensitive at all frequency bands, so bias toward low frequencies helps human recognition. (we have very fine resolution at low frequencies, and less resolution at high frequencies) => information in low frequencies is crucial for distinguishing vowels or nasals but information in high frequencies is less crucial for successful recognition.

$$mel(f) = 1127 \ln(1 + \frac{f}{700})$$

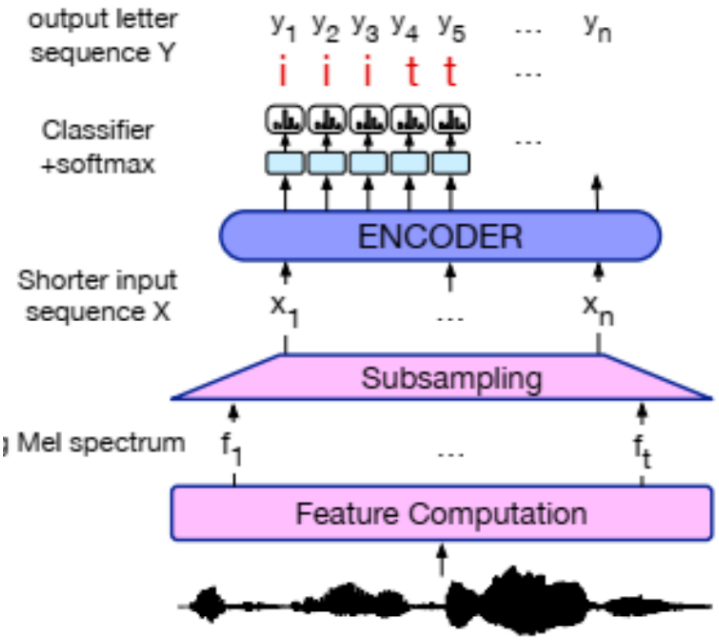
encoder-decoder speech recognizer:

- Feature extraction (Mel spectrum) and the subsampling (shorter sequence of X) will be the input of the encoder
- Output of the encoder will be one of the inputs of the decoder
- For encoder part we can use CTC.

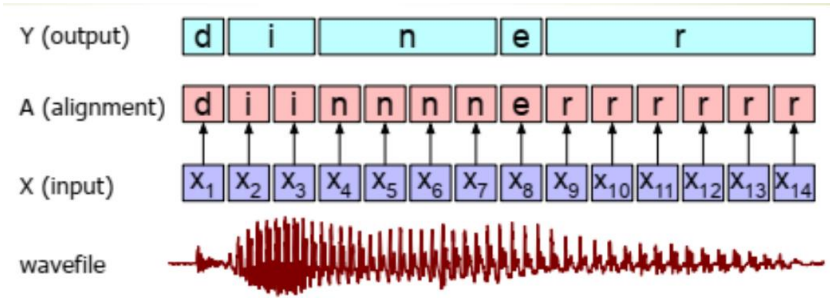




Connectionist Temporal Classification (CTC):



- doesn't handle double letters



- use negative log-likelihood loss with a special CTC loss function

ASR Evaluation: Word Error Rate

$$\text{Word Error Rate} = 100 \times \frac{\text{Insertions} + \text{Substitutions} + \text{Deletions}}{\text{Total Words in Correct Transcript}}$$

Text-to-speech (TTS) systems:

1. preprocessing: text normalization preprocessing for handling non-standard words: numbers, monetary amounts, dates, and other concepts that are verbalized differently than they are spelled using its semiotic class:

semiotic class	examples	verbalization
abbreviations	**gov't**, *N.Y., mph*	government
acronyms read as letters	**GPU**, *D.C., PC, UN, IBM*	G P U
cardinal numbers	**12**, *45, 1/2, 0.6*	twelve
ordinal numbers	*May* **7**, *3rd, Bill Gates III*	seventh
numbers read as digits	*Room* **101**	one oh one
times	*3.20,* **11:45**	eleven forty five
dates	**28/02** *(or in US, 2/28)*	February twenty eighth
years	**1999**, *80s, 1900s, 2045*	nineteen ninety nine
money	**$3.45**, *€250, $200K*	three dollars forty five
money in tr/m/billions	**$3.45 billion**	three point four five billion dollars
percentage	**75%** *3.4%*	seventy five percent

   Normalizing may depend on morphological properties (gender) and can be done:
   1) rule: using 1-Tokenization (regular expressions) or 2-verbalization
   2) encoder-decoder model
2. encoder-decoder model for spectrogram prediction: maps from strings of letters to mel spectrographs: sequences of mel spectral values over time.
3. Vocoder: maps from mel spectrograms to waveforms (vocoding)

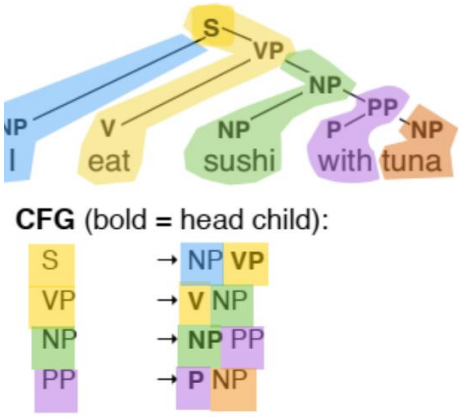
TTS Evaluation:

1. Mean opinion score (MOS): a rating of how good the synthesized utterances are, usually on a scale from 1–5.
2. AB test: compare two systems by playing the same sentence synthesized by two different systems (an A and a B system). The human listeners choose which of the two utterances they like better. We do this for say 50 sentences (presented in random order) and compare the number of sentences preferred for each system.

Application 5: (Dependency Parsing)

- Dependencies form an acyclic graph over the words in a sentence
- Each word has only one parent
- The (non-root) nodes of a dependency tree are the words/tokens in the sentence which has exactly one incoming edge (from its parent).
- The root node of a dependency tree is a special token ROOT which has exactly one child (the head of the sentence).
- The edges of a dependency tree are dependencies from a head (parent) to a dependent (child).
- A complete dependency tree (parse) for a sentence includes all tokens in the sentence

CFG:

- capture only nested dependencies
- The dependency graph is a tree which does not cross
- 

Different types of dependencies:

1. Head-argument: arguments may be obligatory but occur once. => the head cannot replace the construction. *eat sushi*
2. Head-modifier: modifiers are optional and can occur more than once! => the head can replace the entire construction. *fresh sushi*
3. Head-specifier: between function words (prepositions, determiners) and the arguments. *the sushi*
4. Coordination: unclear where the head is. *sushi and sashimi*

Two types of parsers:

1. Transition-based (shift-reduced): read the sentence incrementally, word by word.
    - Transitions:
        - Shift: read the next word
        - Reduce: attach one word to another
    - Model: predict the next action
    - Return: a single, projective (no crossing edges), dependency tree
    - ```
      function DEPENDENCYPARSE(words) returns dependency tree

        state ← {[root], [words], [] }  ; initial configuration
        while state not final
          t ← ORACLE(state)         ; choose a transition operator to apply
          state ← APPLY(t, state)   ; apply it, creating a new state
        return state
      ```
    - The parser processes the sentence $S = w_0 w_1 ... w_n$ from left to right ("**incremental parsing**")

      The parser uses **three data structures**:

      σ: a **stack** of partially processed words $w_i \in T_S$

      β: a **buffer** of remaining input words $w_i \in T_S$

      A: a **set of dependency arcs** $(w_i, \ell, w_j) \in T_S \times L \times T_S$

      w_0 is a special ROOT token.

      $T_S = \{w_0, w_1, ..., w_n\}$ are the tokens in the input sentence

      L is a predefined set of dependency relation labels

      (w_i, ℓ, w_j) is a dependency with label ℓ from head w_i to w_j
    - data structures of the parser and their definition
      ```

The stack σ is a list of partially processed words

We can *shift* the top word of β onto the top of σ (grow the stack by one) or remove one of the top two words from σ by attaching it to the other top word (this *reduces* the stack by one element)

σ|w_j or σ|w_iw_j: w_j is the topmost word on the stack.
σ|w_iw_j: w_i is the second top word on the stack.

The buffer β is the remaining input words

We read words from β (left-to-right) and push ('*shift*') them onto σ

w|β: w is on top of the buffer. w is the next word to be shifted onto σ

The set of arcs A defines the current tree.

We add a new arc to A by attaching the first word on top of the stack to the second word on top of the stack or vice versa

-
- Actions:

In any configuration (σ, β, A), take one of these actions:

1) **Shift** w_k from buffer β to stack σ:

Shift: $(\sigma, w_k|\beta, A) \Rightarrow (\sigma|w_k, \beta, A)$

2) Add a ***leftwards*** **dependency arc with label** ℓ from w_j to w_i:

LeftArc-ℓ: $(\sigma|w_iw_j, \beta, A) \Rightarrow (\sigma|w_j, \beta, A \cup \{(w_j, \ell, w_i)\})$

w_i (2nd on stack) is a dependent (with label ℓ) of head w_j (1st on stack)
w_i is removed from the stack σ: only do this if w_i has no further children to be attached

3) Add a ***rightwards*** **dependency arc with label** ℓ from w_i to w_j:

Right Arc-ℓ: $(\sigma|w_iw_j, w_k|\beta, A) \Rightarrow (\sigma|w_i, \beta, A \cup \{(w_i, \ell, w_j)\})$

w_j (1st on stack) is a dependent (with label ℓ) of head w_i (2nd on stack).
w_i is removed from the stack σ: only do this if w_i has no further children to be attached

In the configuration $(\sigma|w_iw_j, w_k|\beta, A)$:

w_i and w_j are the top two elements of the stack.
Each may already have some dependents of their own
w_k (top of the buffer) does not have any dependents yet
w_i (2nd on stack) precedes w_j (top of stack): $i < j$
w_j (top of stack) precedes w_k (top of buffer): $j < k$

We start in the **initial configuration** ($[w_0]$, $[w_1, \ldots, w_n]$, {})
(**Root token**, **Input Sentence**, **No tree**)

- Initial state:

In the initial configuration, we can only **push w_1 onto the stack.**

- Termination:

We want to end in a **terminal configuration** ($[w_0]$, $[]$, A)
(**Root token**, **Empty buffer**, **Complete tree**)

In a terminal configuration, we have **read all of the input words** (empty buffer) and we have **attached all input words**.
(The root w_0 is the only token that can't get attached to any other word)

We can only reach $(\sigma|w_j, w_k|\beta, A)$ if all words w_l with $j < l < k$ have already been attached to their parent w_m with $j \leq m < k$

2. Graph-based: consider all words in the sentence at once.
    - Use minimum spanning tree algorithm to find the best tree
    - Model: score each dependency edge
    - Return: the top k trees, including non-projective ones