Natural Language Processing

AI51701/CSE71001

Lecture 14

11/7/2023

Instructor: Taehwan Kim

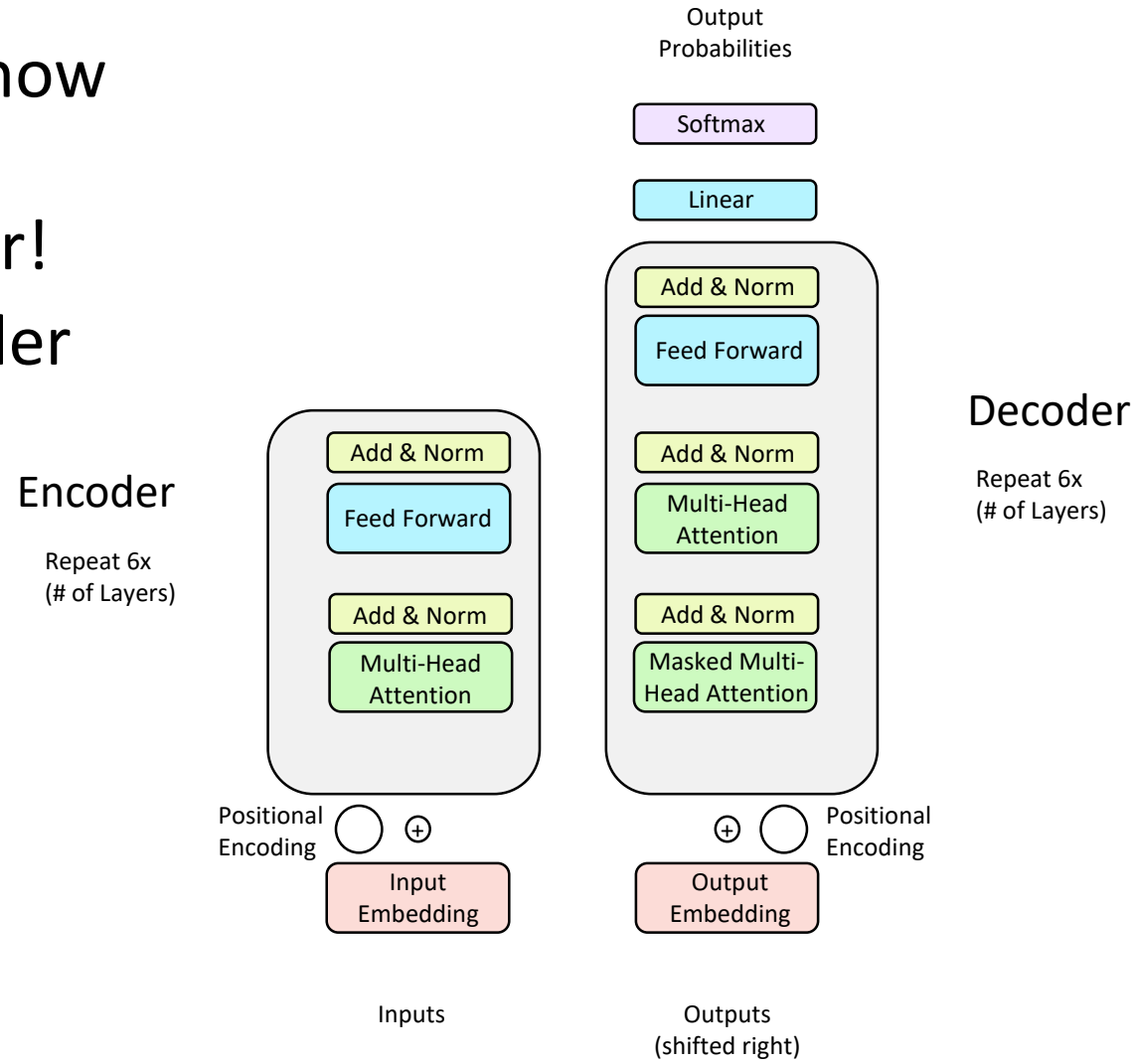
Announcement

- Assignment 2 was released
  - Due: Nov. 12 at 11:59pm
  - For data downloading and splits, please check announcement emails from TA

Transformers

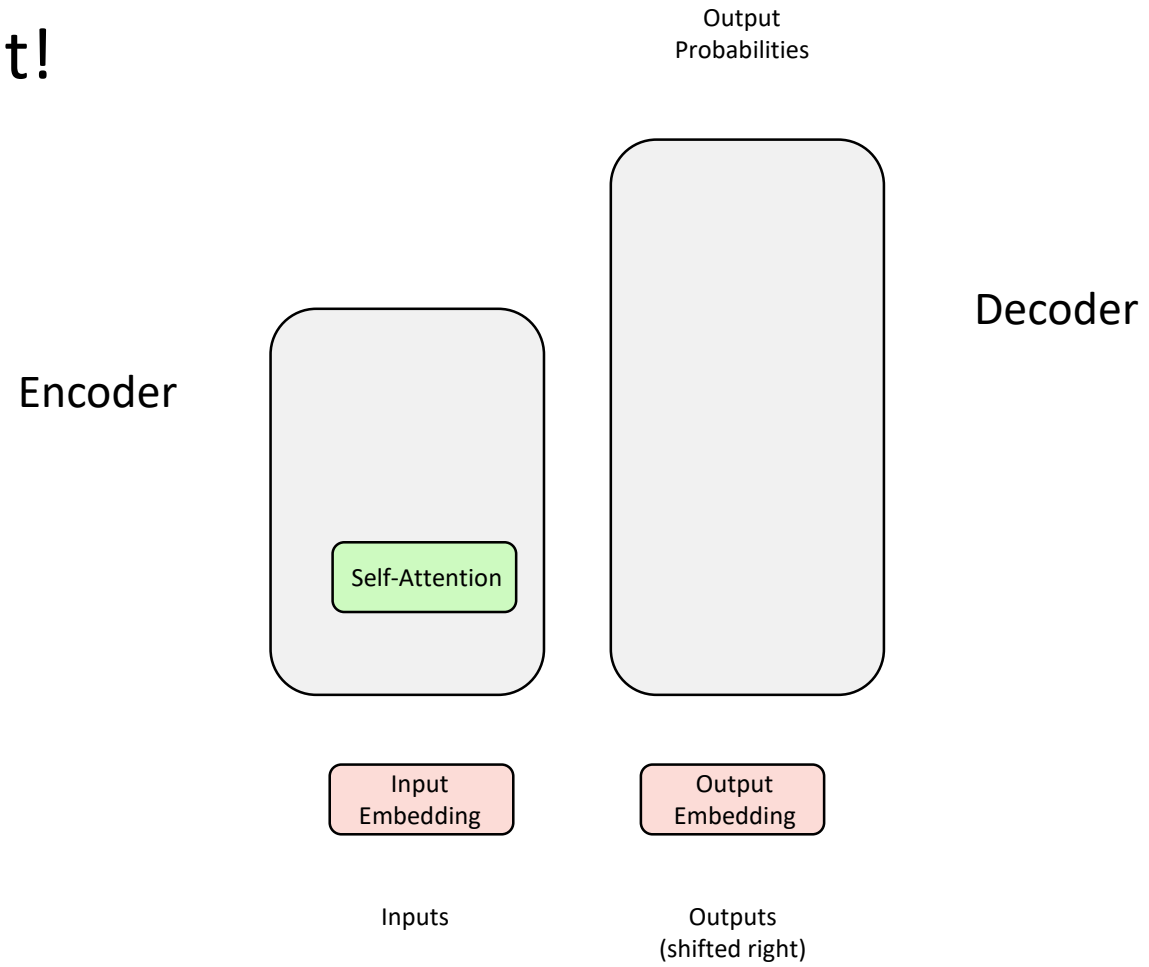
The Transformer Encoder-Decoder [Vaswani et al., 2017]

- In this section, you will learn exactly how the Transformer architecture works:
  - First, we will talk about the Encoder!
  - Next, we will go through the Decoder (which is quite similar)!



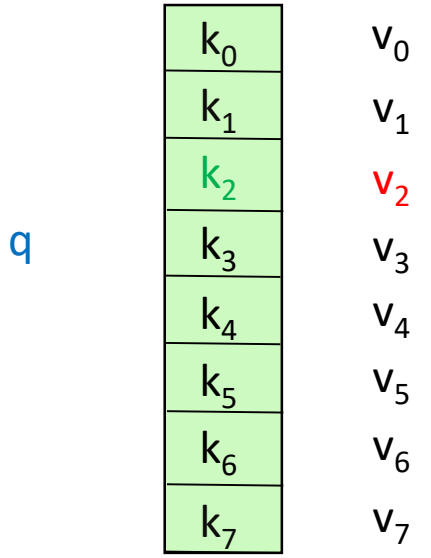
Encoder: Self-Attention

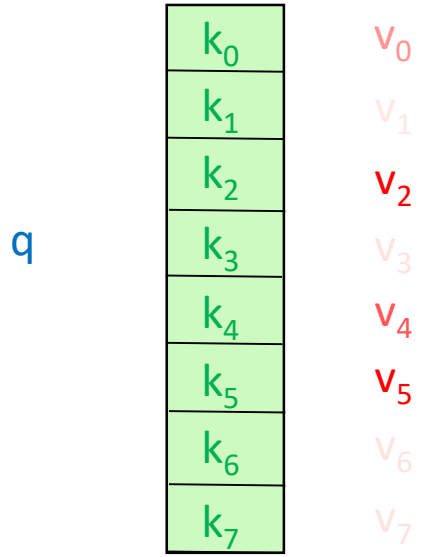
- Self-Attention is the core building block of Transformer, so let's first focus on that!



Intuition for Attention Mechanism

- Let's think of attention as a "fuzzy" or approximate hashtable:
  - To look up a value, we compare a query against keys in a table.
  - In a hashtable (shown on the bottom left):
    - Each **query** (hash) maps to exactly one **key-value** pair.
  - In (self-)attention (shown on the bottom right):
    - Each **query** matches each **key** to varying degrees.
    - We return a sum of **values** weighted by the **query**-**key** match.





Recipe for Self-Attention in the Transformer Encoder

- Step 1: For each word x_i, calculate its **query**, **key**, and **value**.

$$q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i$$

- Step 2: Calculate attention score between **query and keys.**

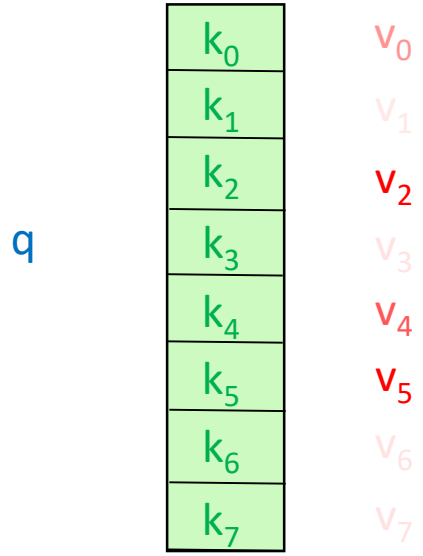
$$e_{ij} = q_i \cdot k_j$$

- Step 3: Take the softmax to normalize attention scores.

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(e_{ij})}{\sum_k exp(e_{ik})}$$

- Step 4: Take a weighted sum of **values**.

$$Output_i = \sum_j \alpha_{ij} v_j$$



Recipe for (Vectorized) Self-Attention in the Transformer Encoder

- Step 1: With embeddings stacked in X, calculate queries, keys, and values.

$$Q = XW^Q \quad K = XW^K \quad V = XW^V$$

- Step 2: Calculate attention score between **query and keys.**

$$E = QK^T$$

- Step 3: Take the softmax to normalize attention scores.

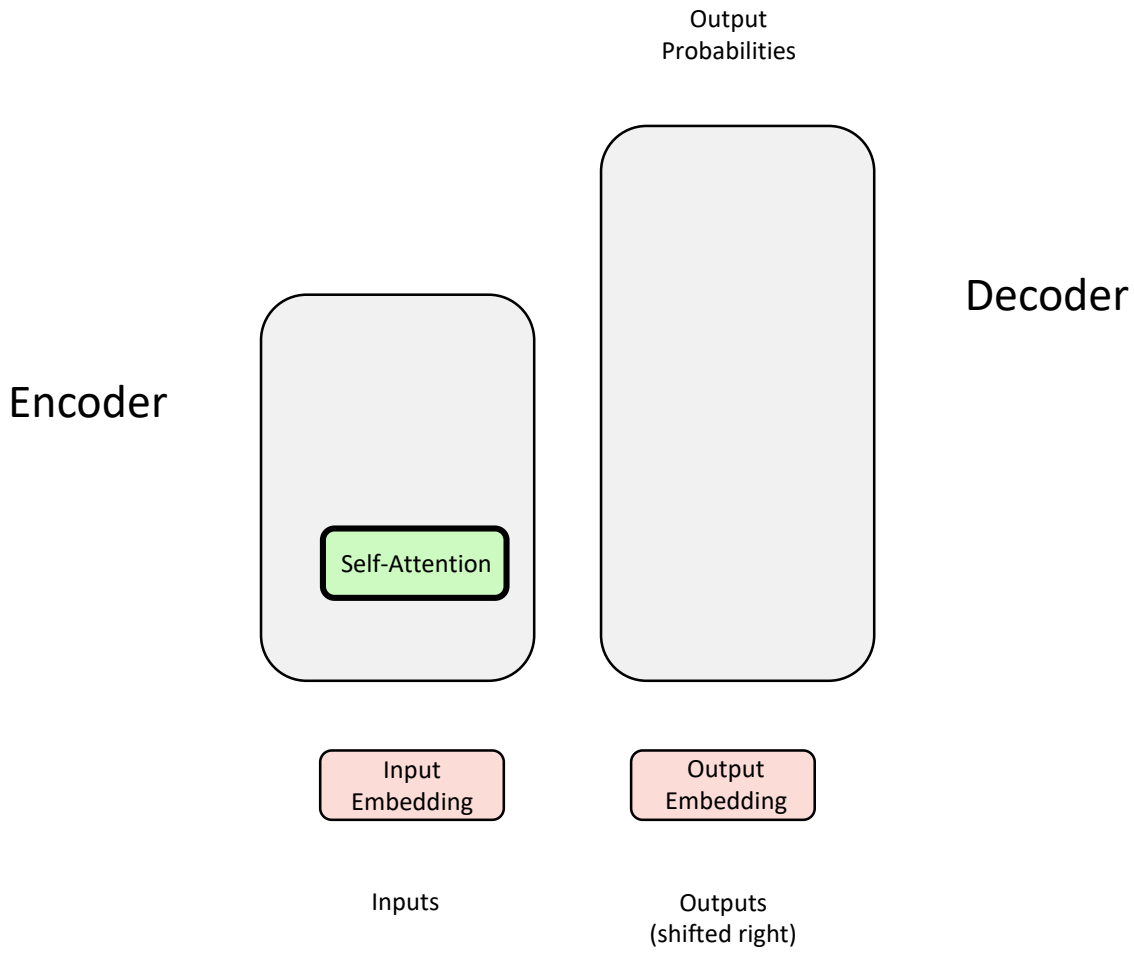
$$A = softmax(E)$$

- Step 4: Take a weighted sum of **values**.

$$Output = AV$$

$$Output = softmax\left(QK^T\right)V$$

What We Have So Far: (Encoder) Self-Attention!

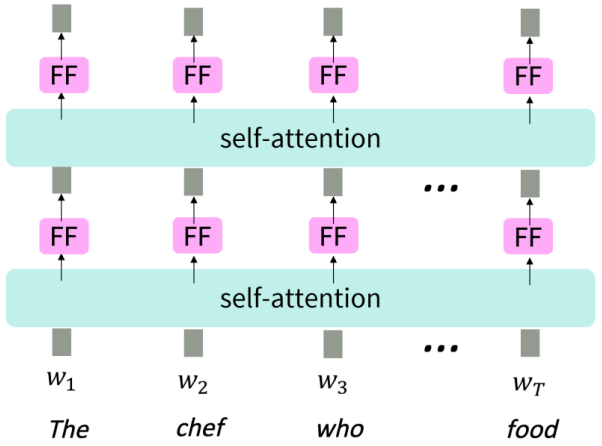


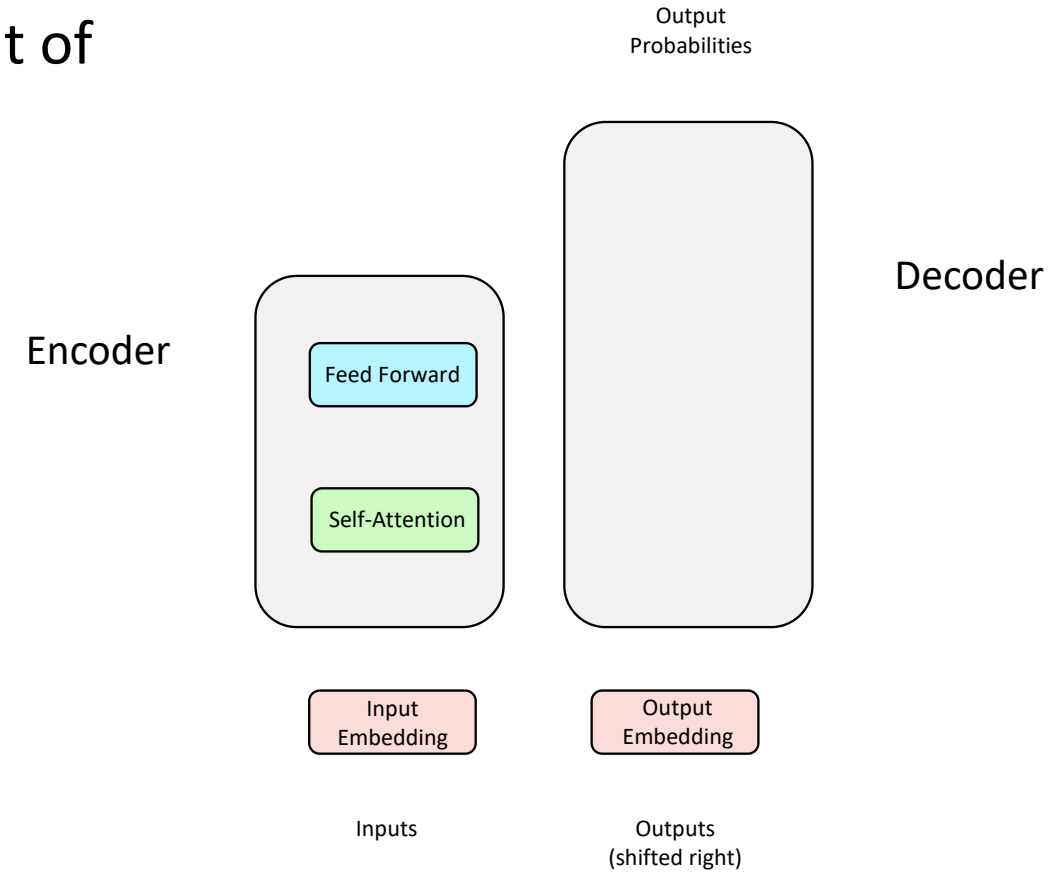
But attention isn't quite all you need!

- **Problem**: Since there are no element-wise non-linearities, self- attention is simply performing a re-averaging of the value vectors.
- **Easy fix:** Apply a feedforward layer to the output of attention, providing non-linear activation (and additional expressive power).

Equation for Feed Forward Layer

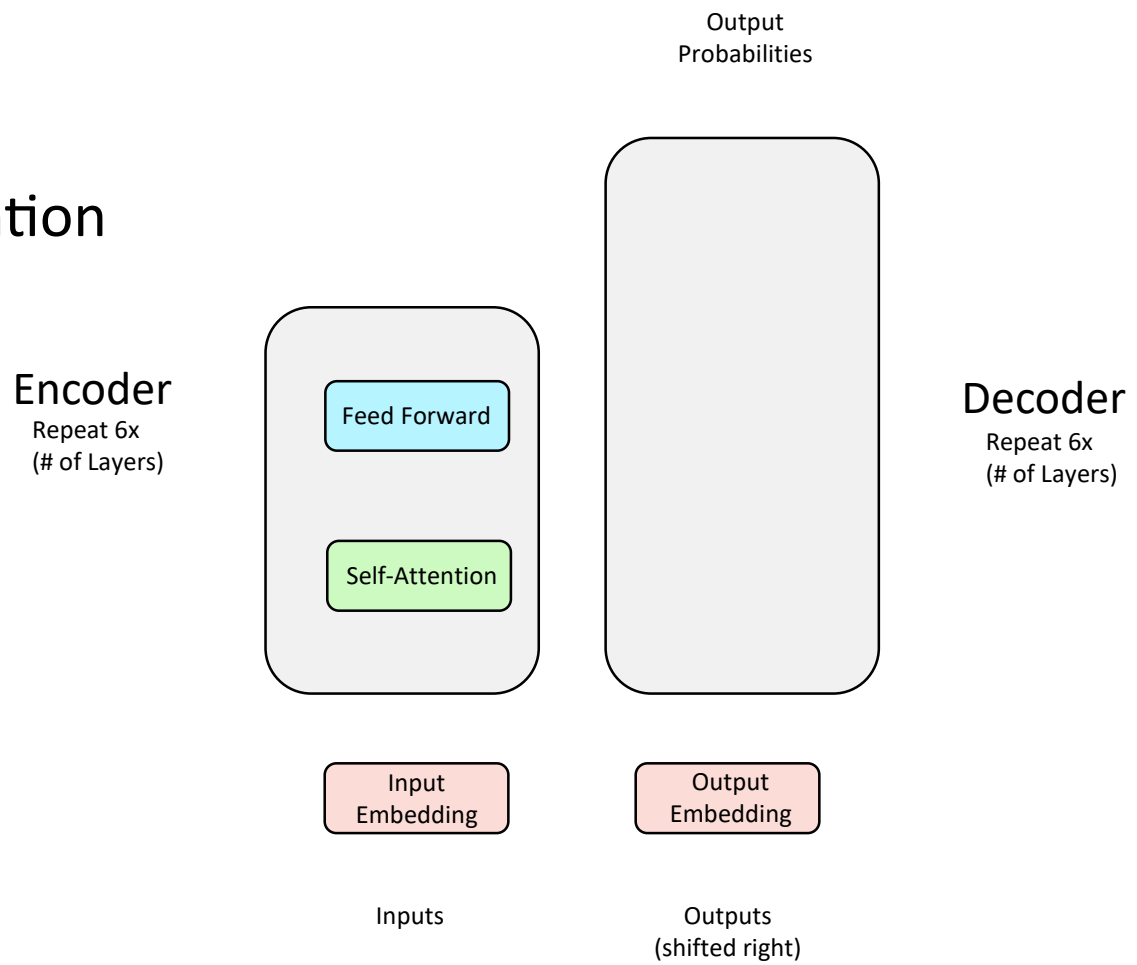
$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2$$





But how do we make this work for deep networks?

- ❑ Training Trick #1: Residual Connections
- ❑ Training Trick #2: LayerNorm
- ❑ Training Trick #3: Scaled Dot Product Attention

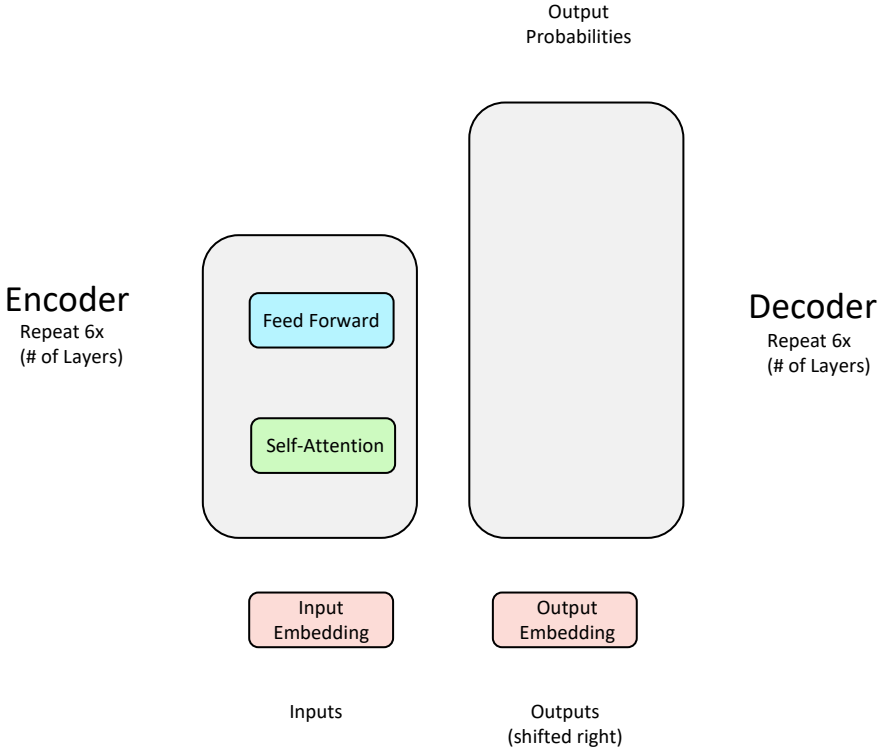


Training Trick #1: Residual Connections [He et al., 2016]

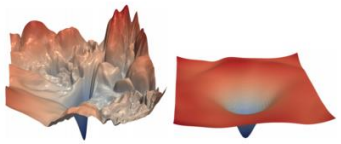
- Residual connections are a simple but powerful technique from computer vision.
- Deep networks are surprisingly bad at learning the identity function!
- Therefore, directly passing "raw" embeddings to the next layer can actually be very helpful!

$$x_\ell = F(x_{\ell-1}) + x_{\ell-1}$$

- This prevents the network from "forgetting" or distorting important information as it is processed by many layers.



Residual connections are also thought to smooth the loss landscape and make training easier!



[no residuals] [residuals]

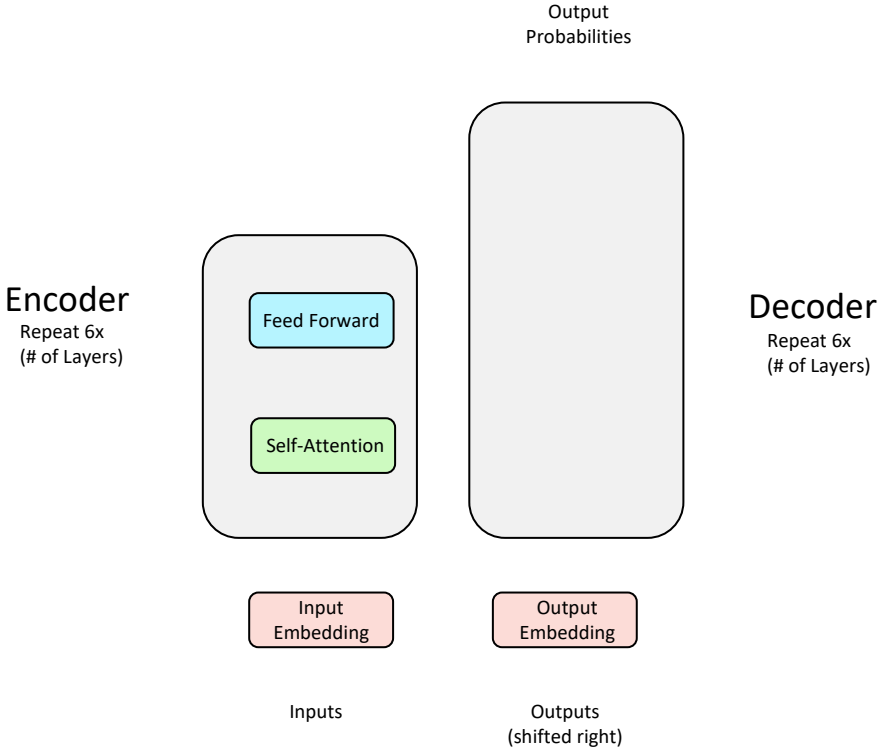
[Loss landscape visualization, Li et al., 2018, on a ResNet]

Training Trick #2: Layer Normalization [Ba et al., 2016]

- **Problem**: Difficult to train the parameters of a given layer because its input from the layer beneath keeps shifting.
- **Solution**: Reduce uninformative variation by **normalizing** to zero mean and standard deviation of one within each **layer**.

Mean: $\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$ Standard Deviation: $\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$

$$x^{\ell\prime} = \frac{x^\ell - \mu^\ell}{\sigma^\ell + \epsilon}$$

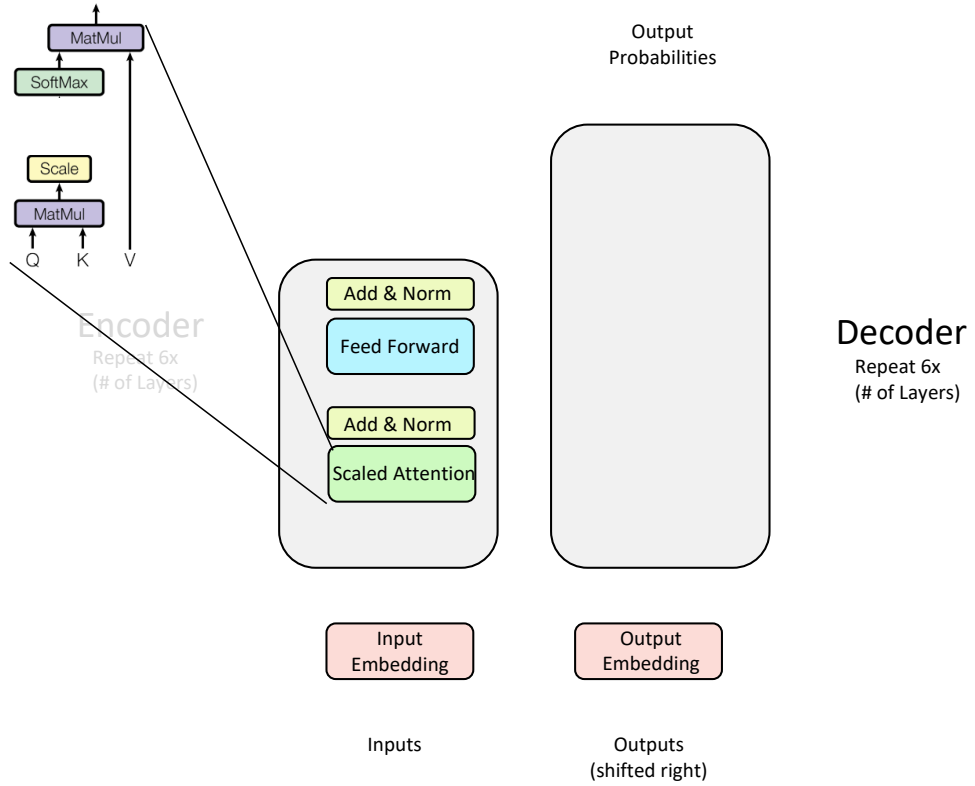


Training Trick #3: Scaled Dot Product Attention

- After LayerNorm, the mean and variance of vector elements is 0 and 1, respectively. (Yay!)
- However, the dot product still tends to take on extreme values, as its variance scales with dimensionality d_k

Quick Statistics Review:

- Mean of sum = sum of means = $d_k * 0 = 0$
- Variance of sum = sum of variances = $d_k * 1 = d_k$
- To set the variance to 1, simply divide by $\sqrt{d_k}$!



Updated Self-Attention Equation:

$$Output = softmax\left(QK^T / \sqrt{d_k}\right)V$$

Fixing the first self-attention problem: sequence order

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$p_i \in \mathbb{R}^d$, for $i \in \{1,2, \ldots, T\}$ are position vectors

- Don't worry about what the pi are made of yet!
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!
- Let $\tilde{v}_i, \tilde{k}_i, \tilde{q}_i$ be our old values, keys, and queries.

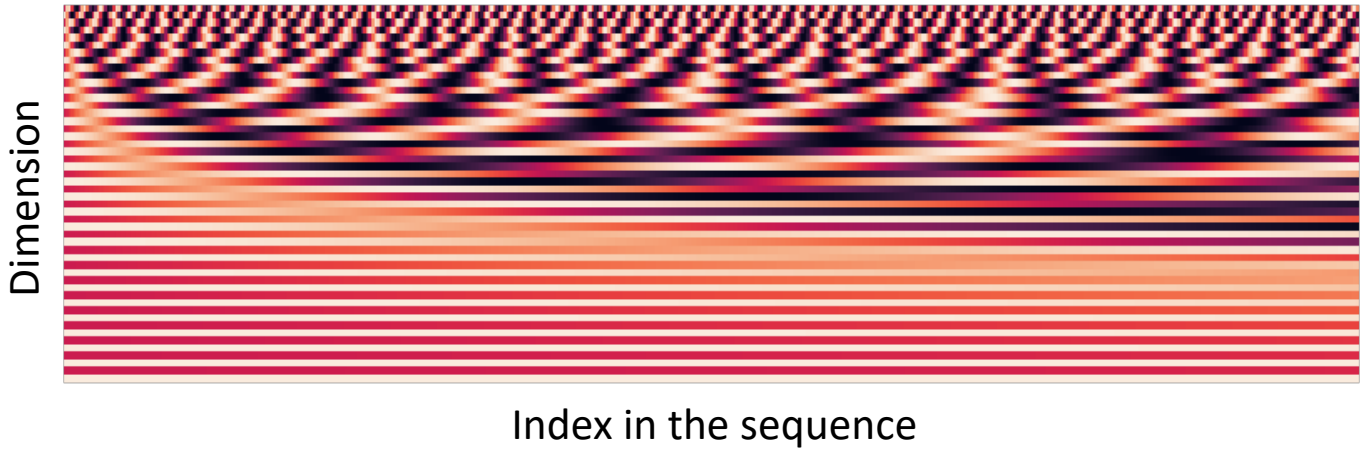
$$v_i = \tilde{v}_i + p_i$$
$$q_i = \tilde{q}_i + p_i$$
$$k_i = \tilde{k}_i + p_i$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add…

Position representation vectors through sinusoids

- **Sinusoidal position representations**: concatenate sinusoidal functions of varying periods:

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

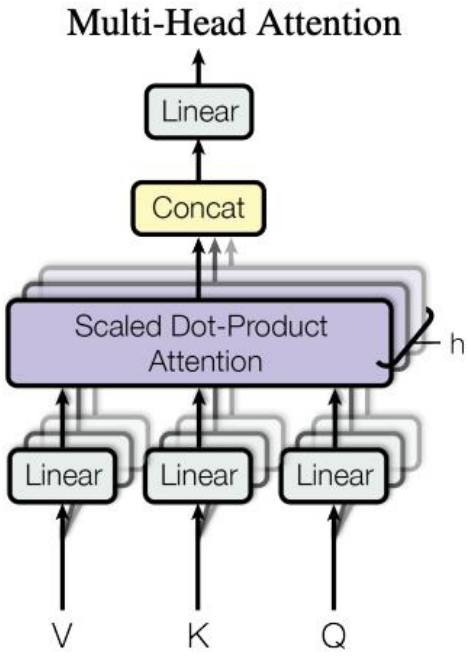


- Pros:
  - Periodicity indicates that maybe "absolute position" isn't as important
  - Maybe can extrapolate to longer sequences as periods restart
- Cons:
  - Not learnable; also the extrapolation doesn't really work

Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

Multi-Headed Self-Attention: k heads are better than 1!

- **High-Level Idea:** Let's perform self-attention multiple times in parallel and combine the results.



[Vaswani et al. 2017]

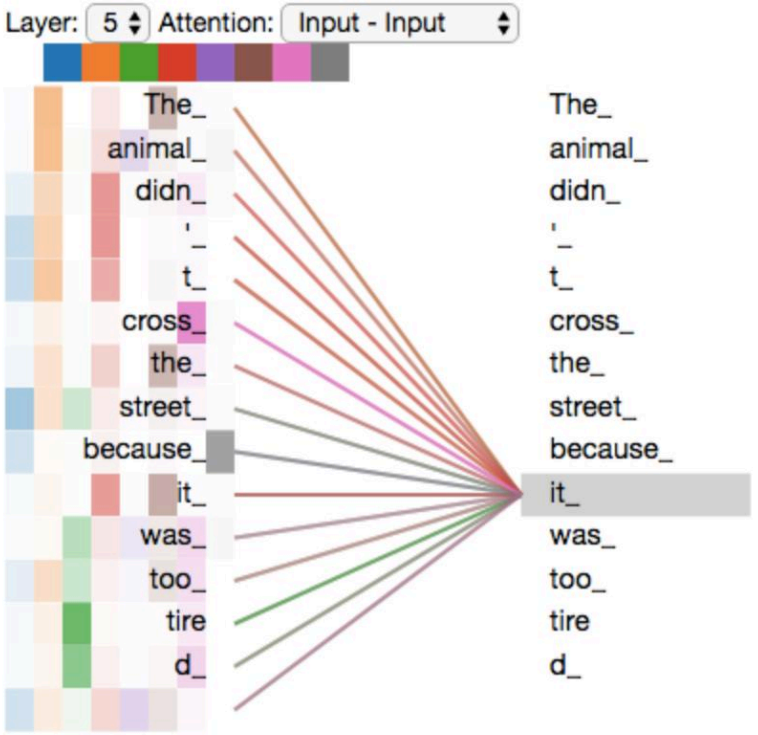
Wizards of the Coast, Artist: Todd Lockwood

The Transformer Encoder: Multi-headed Self-Attention

- What if we want to look in multiple places in the sentence at once?
  - For word i, self-attention "looks" where $x_i^T Q^T K x_j$ is high, but maybe we want to focus on different j for different reasons?

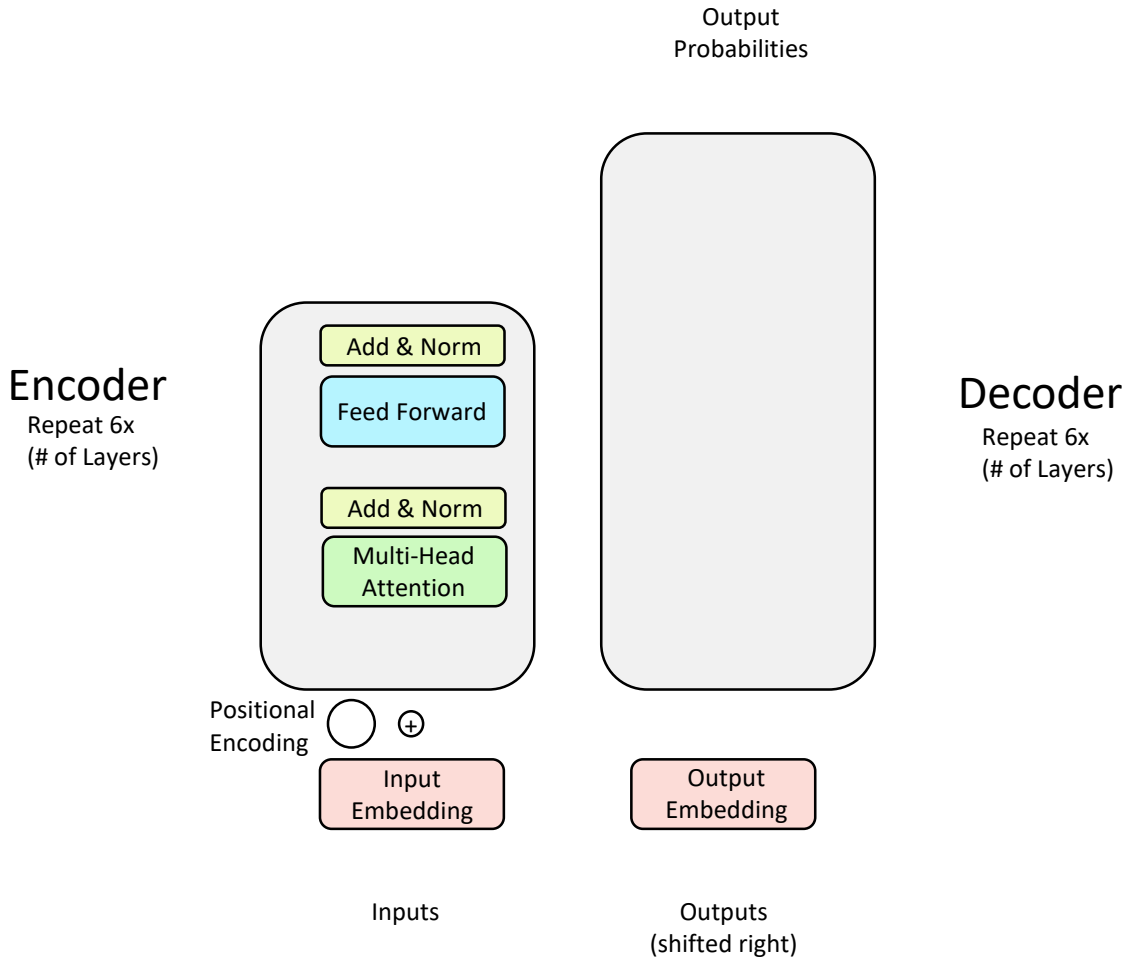
- We'll define **multiple attention "heads"** through multiple Q,K,V matrices.

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



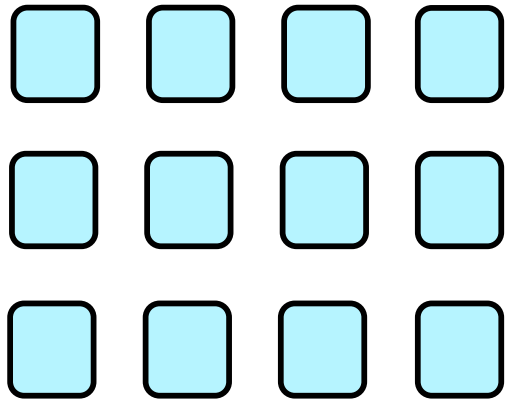
Credit to https://jalammar.github.io/illustrated-transformer/

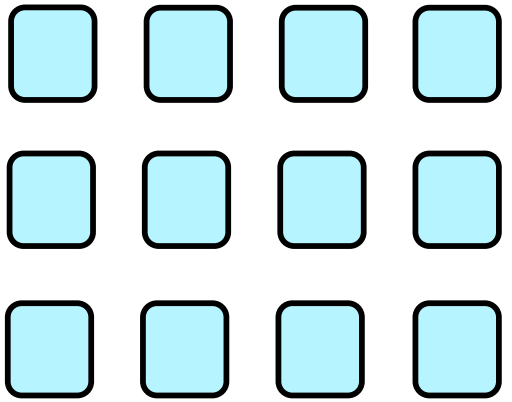
We've completed the Encoder! Time for the Decoder...



Decoder: Masked Multi-Head Self-Attention

- **Problem:** How do we keep the decoder from "cheating"? If we have a language modeling objective, can't the network just look ahead and "see" the answer?
- **Solution:** Masked Multi-Head Attention. At a high-level, we hide (mask) information about future tokens from the model.



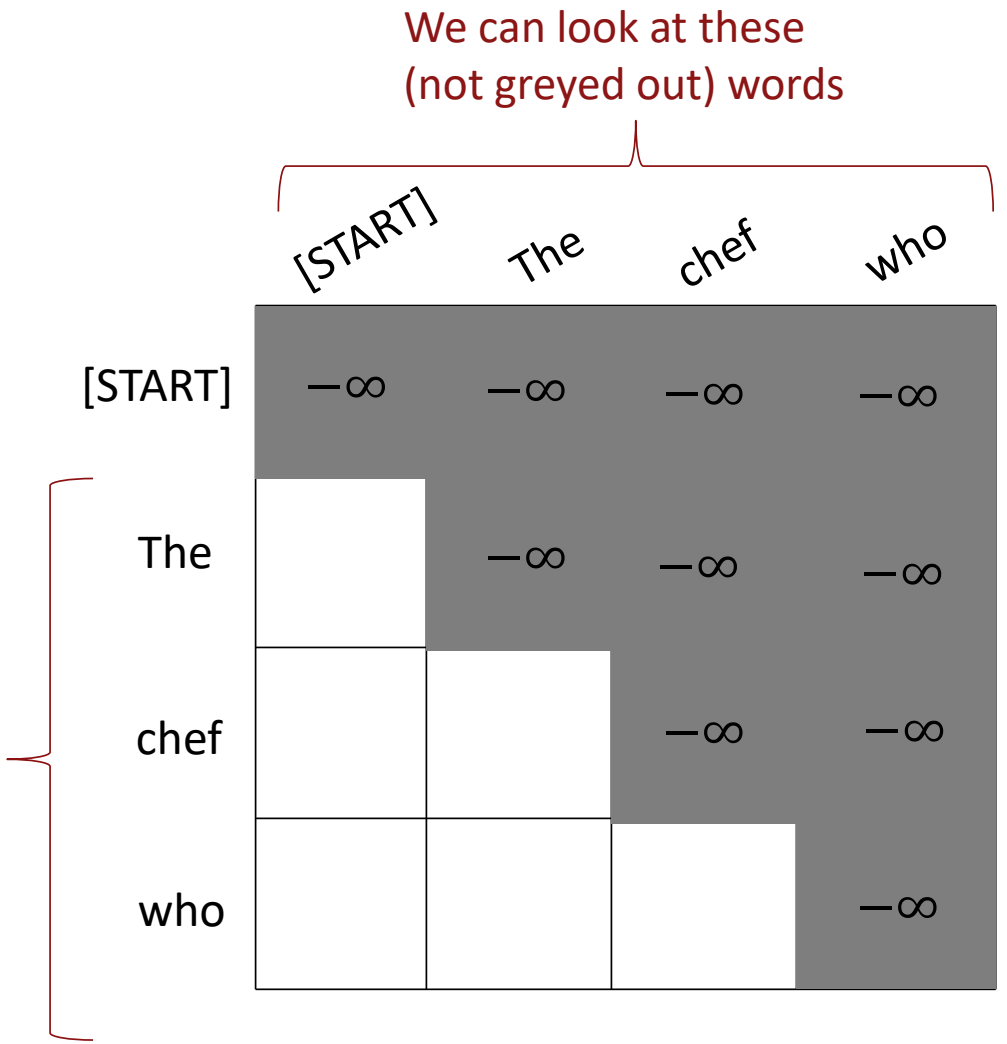


Transformer-Based
Encoder-Decoder Model

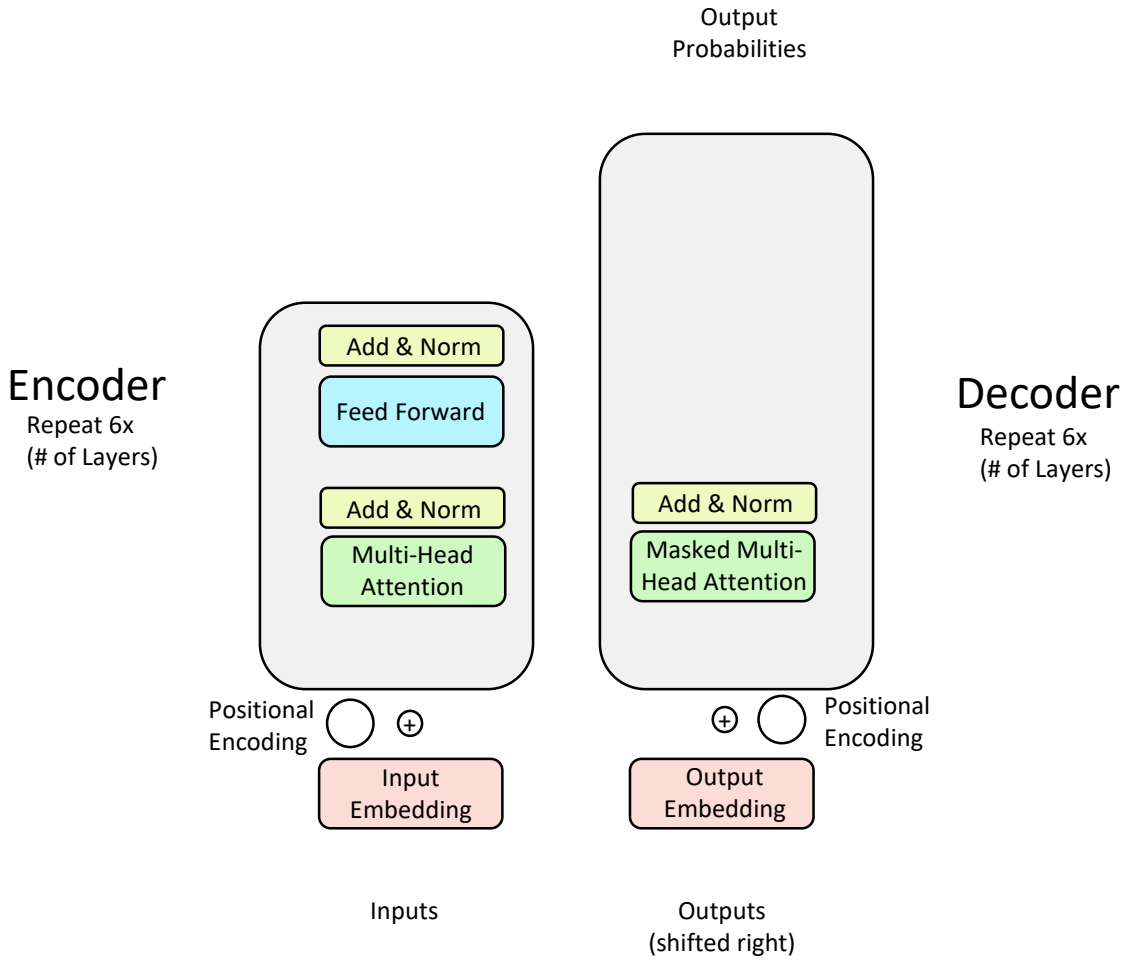
Masking the future in self-attention

- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

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

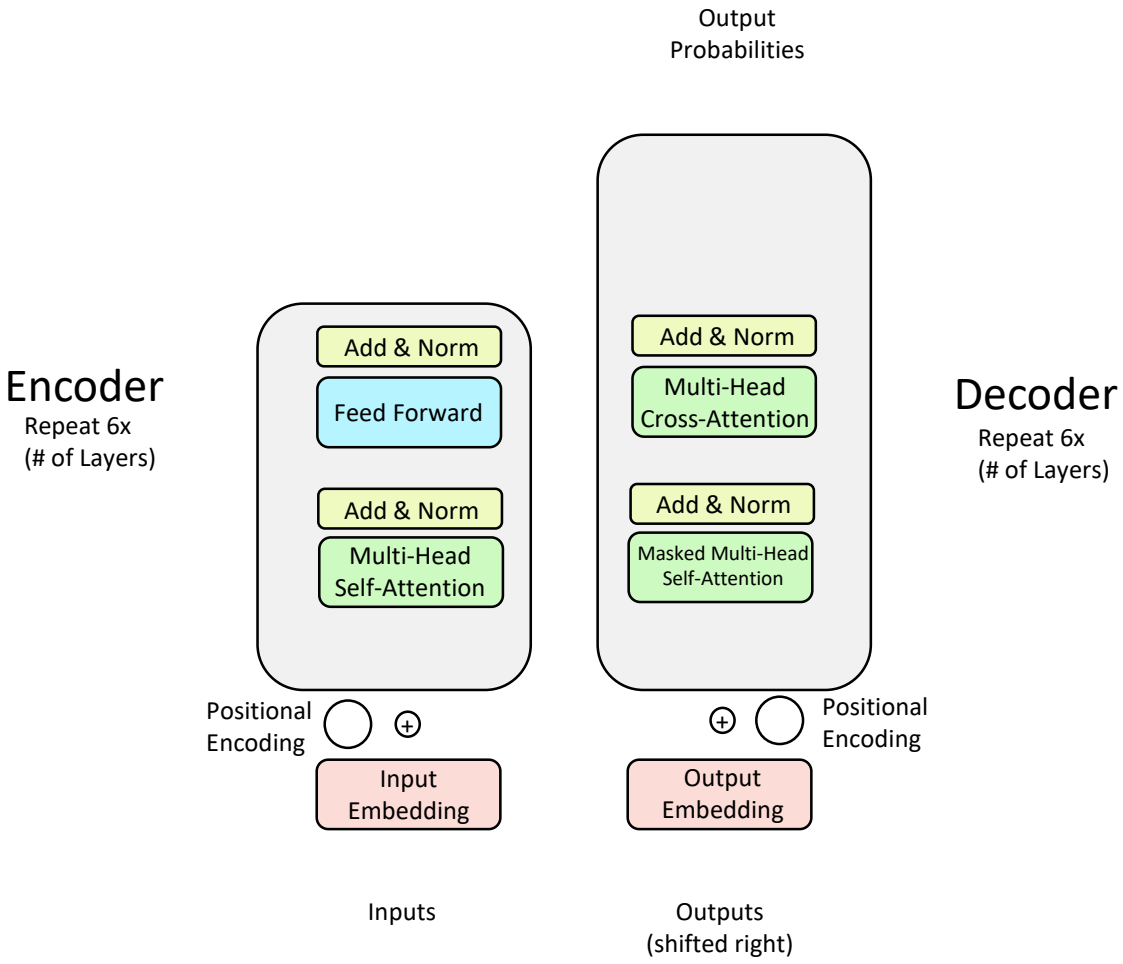


Decoder: Masked Multi-Headed Self-Attention

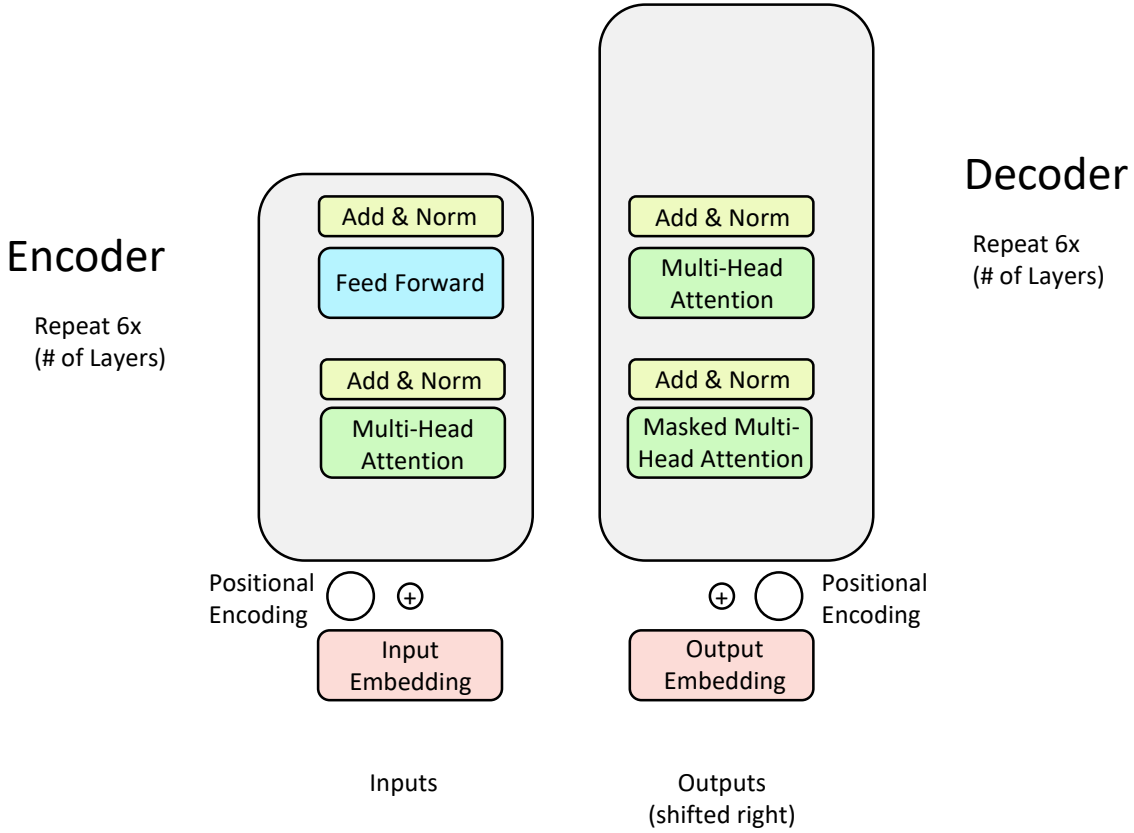


Encoder-Decoder Attention

- We saw that self-attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like what we saw last week.
- Let $h_1, \ldots, h_T$ be **output** vectors **from** the Transformer **encoder**; $x_i \in \mathrm{R}^d$
- Let $z_1, \ldots, z_T$ be input vectors from the Transformer **decoder**, $z_i \in \mathrm{R}^d$
- Then keys and values are drawn from the **encoder** (like a memory):
  - $k_i = K\mathrm{h}_i$, $v_i = V\mathrm{h}_i$
- And the queries are drawn from the **decoder**, $q_i = Qz_i$.

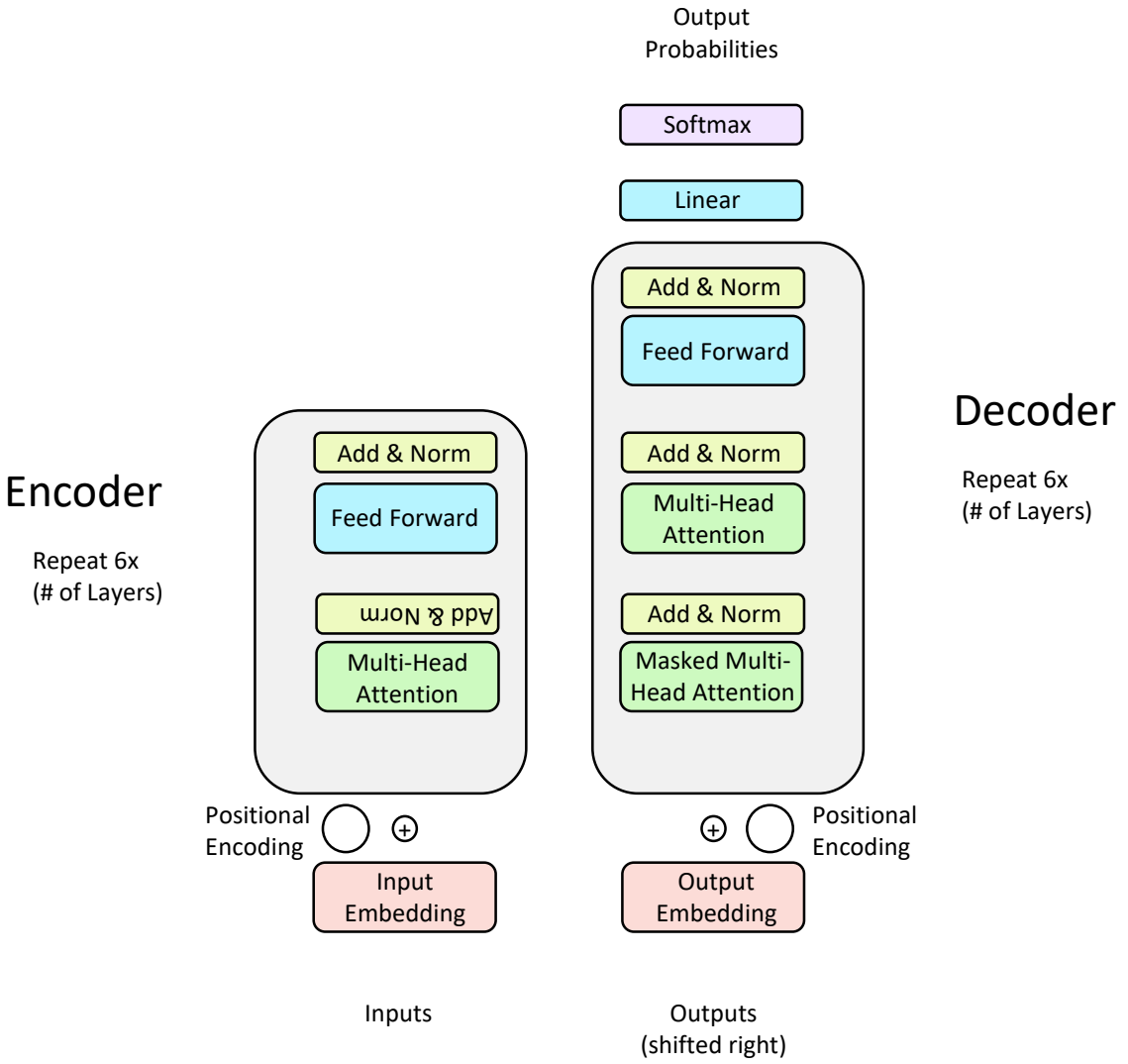


Decoder: Finishing touches!

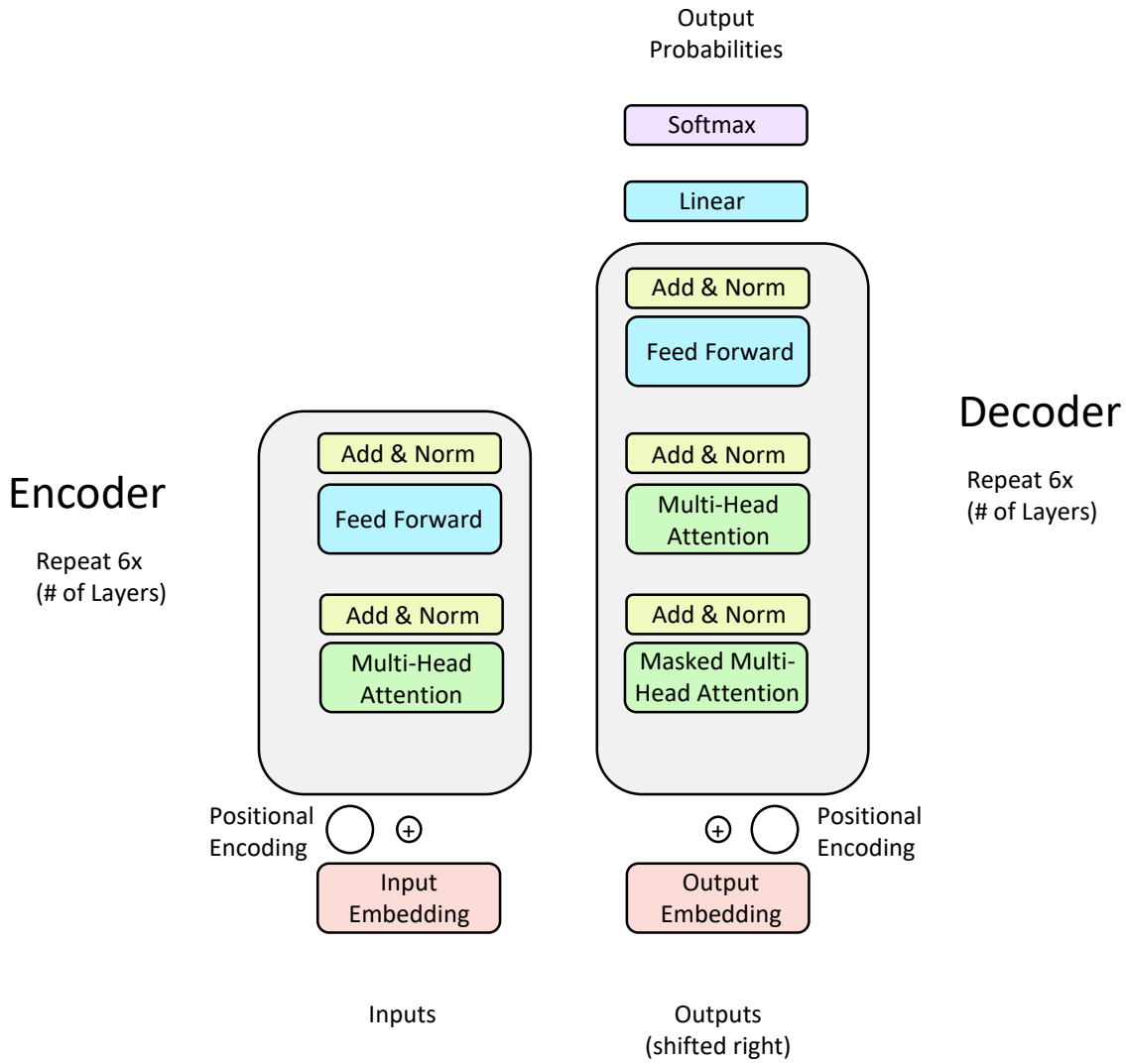


Decoder: Finishing touches!

- ❑ Add a feed forward layer (with residual connections and layer norm)
- ❑ Add a final linear layer to project the embeddings into a much longer vector of length vocab size (logits)
- ❑ Add a final softmax to generate a probability distribution of possible next words!



Recap of Transformer Architecture



Motivating model pretraining from word embeddings

From Stanford 224n course

Motivating model pretraining from word embeddings

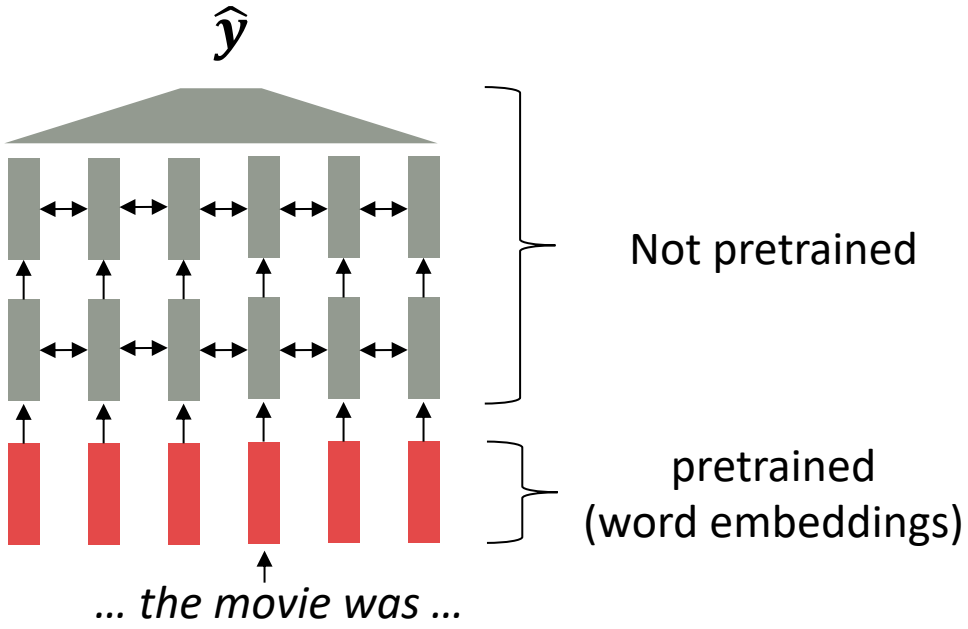
- Recall that for the following corpus:
  - A bodle of ***tesgüino*** is on the table
  - Everybody likes ***tesgüino***
  - ***Tesgüino*** makes you drunk
  - We make ***tesgüino*** out of corn.

- From context, we can guess ***tesgüino*** is an alcoholic beverage like beer
- Intuition: two words are similar if they have similar word contexts
- This example motivates distributional models such as word2vec
- But consider *I* ***record*** *the* ***record***: the two instances of ***record*** mean different things.

Where we were: **pretrained word embeddings**

- Circa 2017:
  - Start with pretrained word embeddings (no context!)
  - Learn how to incorporate context in an LSTM or Transformer while training on the task.

- **Some issues to think about**:
  - The training data we have for our **downstream task** (like question answering)
  - must be sufficient to teach all contextual aspects of language.
  - Most of the parameters in our network are randomly initialized!

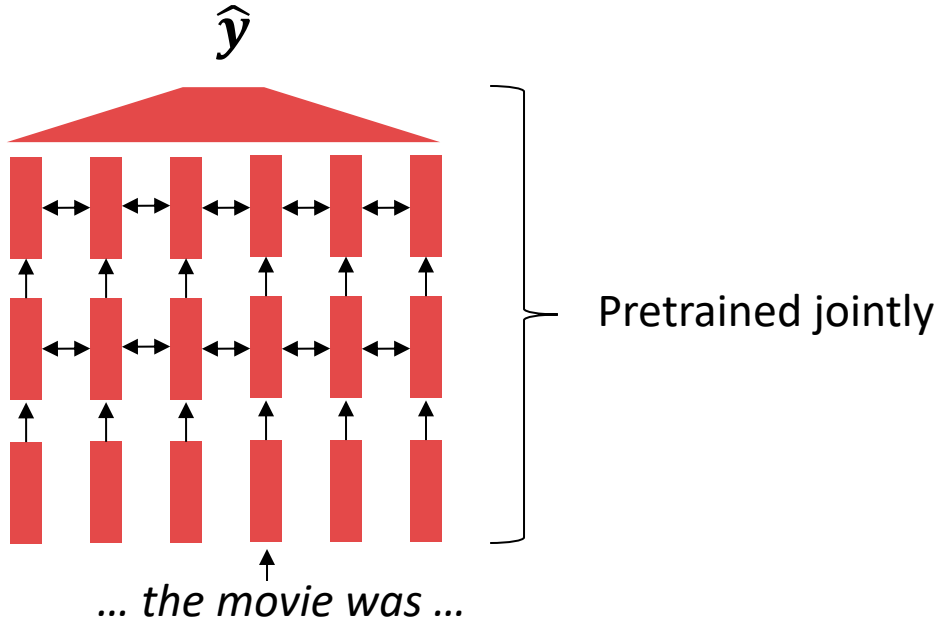


[Recall, *movie* gets the same word embedding, no matter what sentence it shows up in]

Where we're going: **pretraining whole models**

- In modern NLP:
  - All (or almost all) parameters in NLP networks are initialized via **pretraining**.
  - Pretraining methods hide parts of the input from the model, and then train the model to reconstruct those parts.

- This has been exceptionally effective at building strong:
  - **representations of language**
  - **parameter initializations** for strong NLP models.
  - **probability distributions** over language that we can sample from



[This model has learned how to represent entire sentences through pretraining]

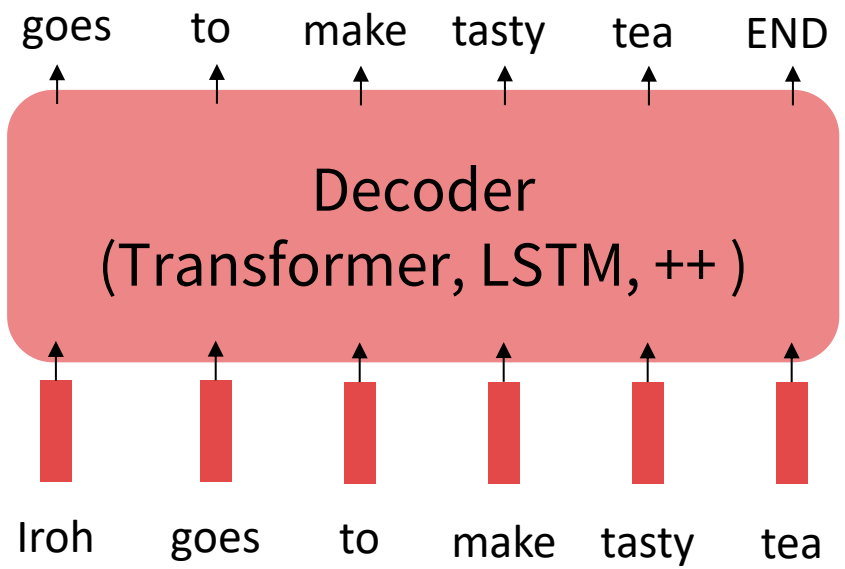
What can we learn from reconstructing the input?

- I put ___ fork down on the table. [syntax]
- The woman walked across the street,
  checking for traffic over ___ shoulder. [coreference]
- I went to the ocean to see the fish, turtles, seals, and _____. [lexical semantics/topic]
- Overall, the value I got from the two hours watching it was the sum total of the popcorn and the drink. The movie was ___. [sentiment]
- I was thinking about the sequence that goes
  1, 1, 2, 3, 5, 8, 13, 21, ____ [some basic arithmetic]

Pretraining through language modeling (Dai and Le, 2015)

- Recall the **language modeling** task:
  - Model $p_\theta(w_t|w_{1:t-1})$, the probability distribution over words given their past contexts.
  - There's lots of data for this! (In English.)

- **Pretraining through language modeling**:
  - Train a neural network to perform language modeling on a large amount of text.

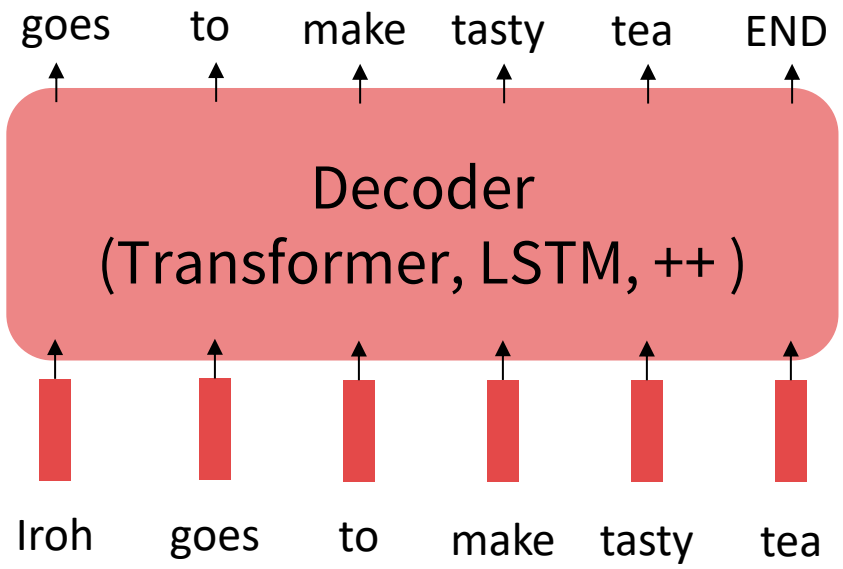


The Pretraining / Finetuning Paradigm

❑ Pretraining can improve NLP applications by serving as parameter initialization.

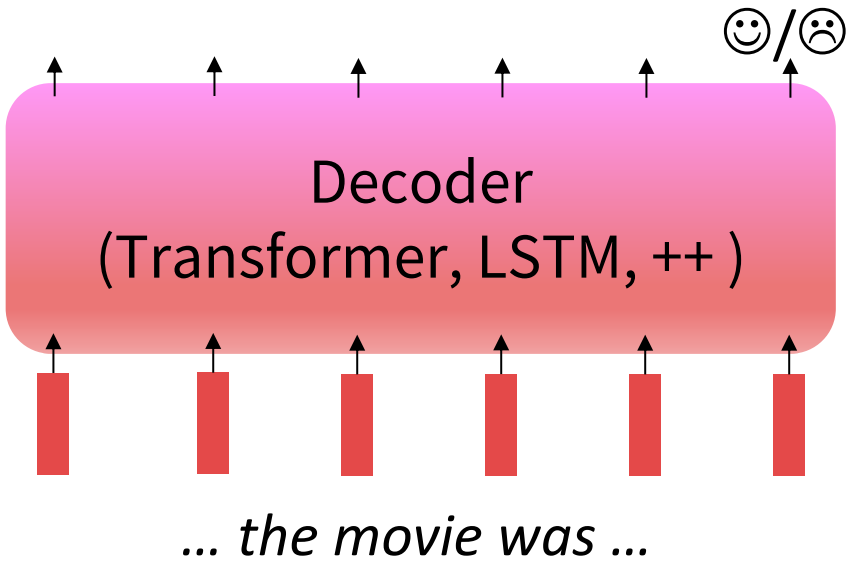
Step 1: Pretrain (on language modeling)

Lots of text; learn general things!



Step 2: Finetune (on your task)

Not many labels; adapt to the task!

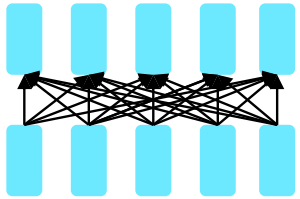


Stochastic gradient descent and pretrain/finetune

- Why should pretraining and finetuning help, from a "training neural nets" perspective?
  - Consider, provides parameters θ' by approximating $\min_\theta L_{\text{pretrain}}(\theta)$
    - (The pretraining loss.)
  - Then, finetuning approximates $\min_\theta L_{\text{finetune}}(\theta)$, starting at θ'.
    - (The finetuning loss)
  - The pretraining may matter because stochastic gradient descent sticks (relatively) close to θ' during finetuning.
    - So, maybe the finetuning local minima near θ' tend to generalize well!
    - And/or, maybe the gradients of finetuning loss near θ' propagate nicely!

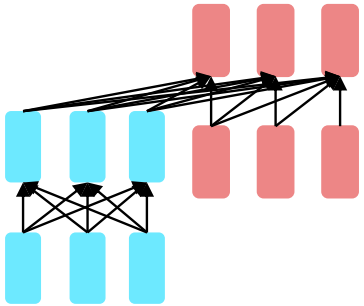
Pretraining for three types of architectures

❑ The neural architecture influences the type of pretraining, and natural use cases.



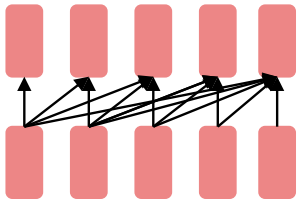
Encoders

- Gets bidirectional context – can condition on future!
- Wait, how do we pretrain them?



Encoder-Decoders

- Good parts of decoders and encoders?
- What's the best way to pretrain them?

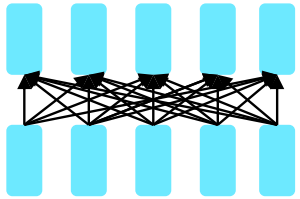


Decoders

- Language models! What we've seen so far.
- Nice to generate from; can't condition on future words

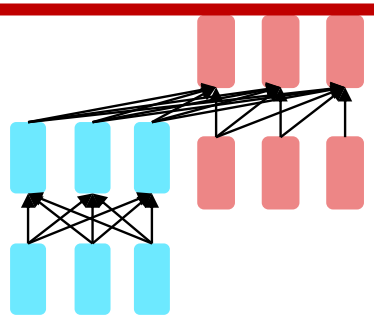
Pretraining for three types of architectures

❑ The neural architecture influences the type of pretraining, and natural use cases.



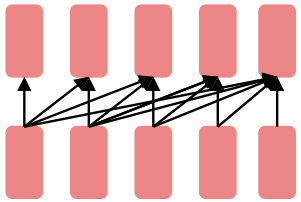
Encoders

- Gets bidirectional context – can condition on future!
- Wait, how do we pretrain them?



Encoder-Decoders

- Good parts of decoders and encoders?
- What's the best way to pretrain them?

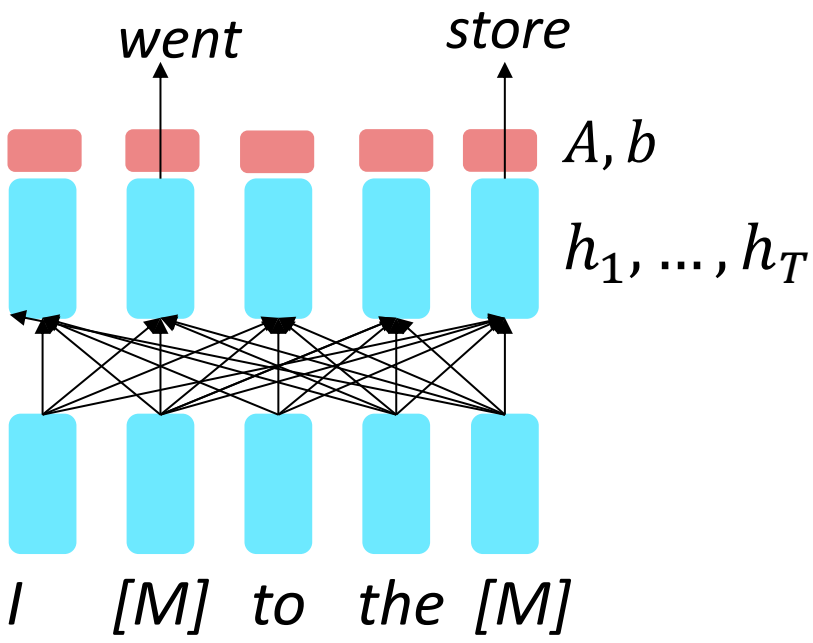


Decoders

- Language models! What we've seen so far.
- Nice to generate from; can't condition on future words

Pretraining encoders: what pretraining objective to use?

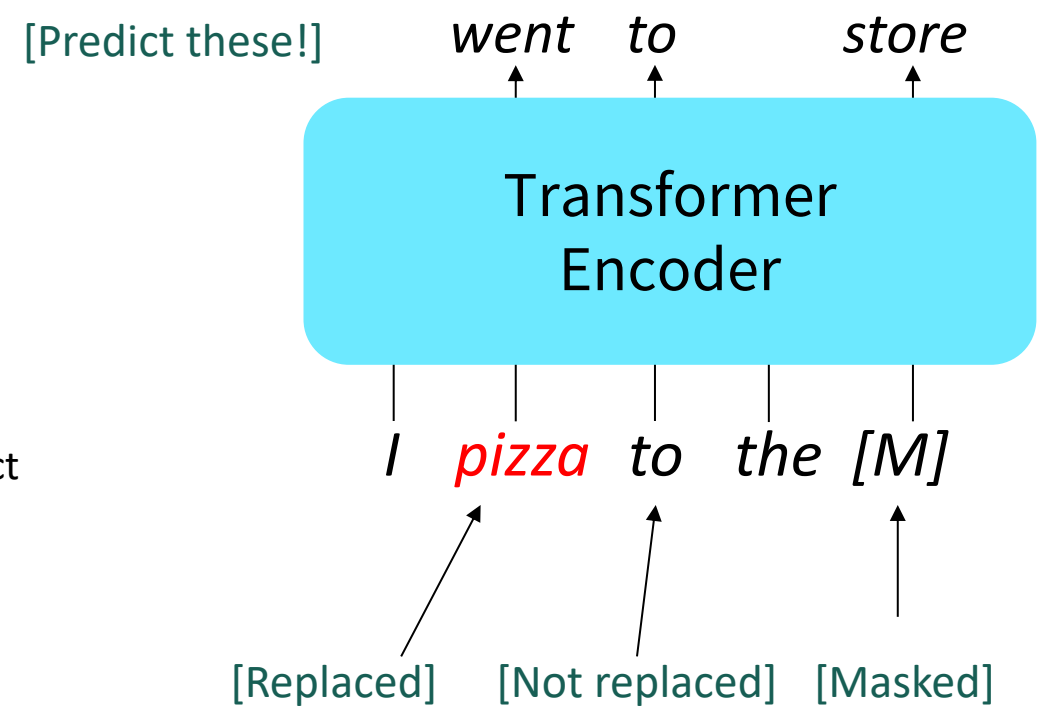
- So far, we've looked at language model pretraining. But **encoders get bidirectional context**, so we can't do language modeling!
- **Idea**: replace some fraction of words in the input with a special [MASK] token; predict these words.
- Only add loss terms from words that are "masked out." If $\tilde{x}$ is the masked version of x, we're learning $p_\theta(x|\tilde{x})$. Called **Masked LM**.



(Devlin et al., 2018)

BERT: Bidirectional Encoder Representations from Transformers

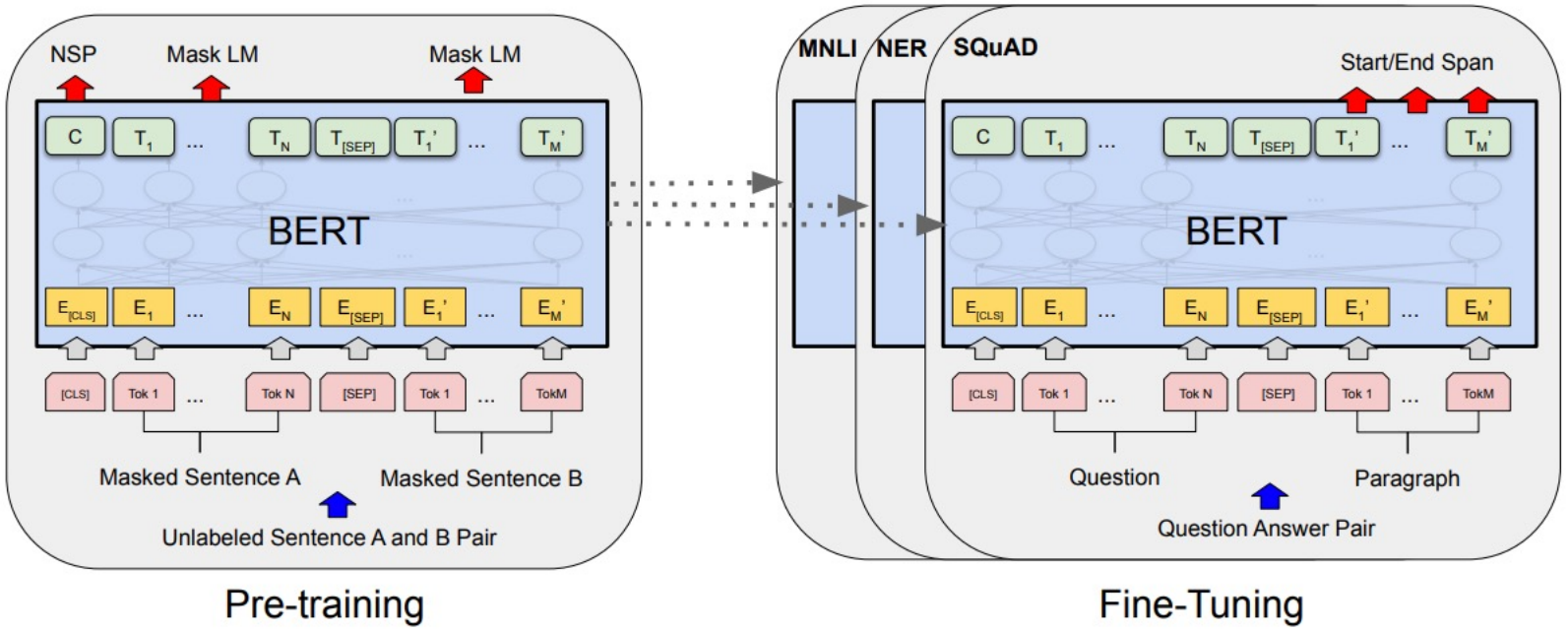
- Devlin et al., 2018 proposed the "Masked LM" objective, and **released the weights of their pretrained Transformer (BERT)**.
- Some more details about Masked LM for BERT:
  - Predict a random 15% of (sub)word tokens.
    - Replace input word with [MASK] 80% of the time
    - Replace input word with a random token 10% of the time
    - Leave input word unchanged 10% of the time (but still predict it!)
- Why? Doesn't let the model get complacent and not build strong representations of non-masked words. (No masks are seen at fine-tuning time!)



(Devlin et al., 2018)

BERT: Bidirectional Encoder Representations from Transformers

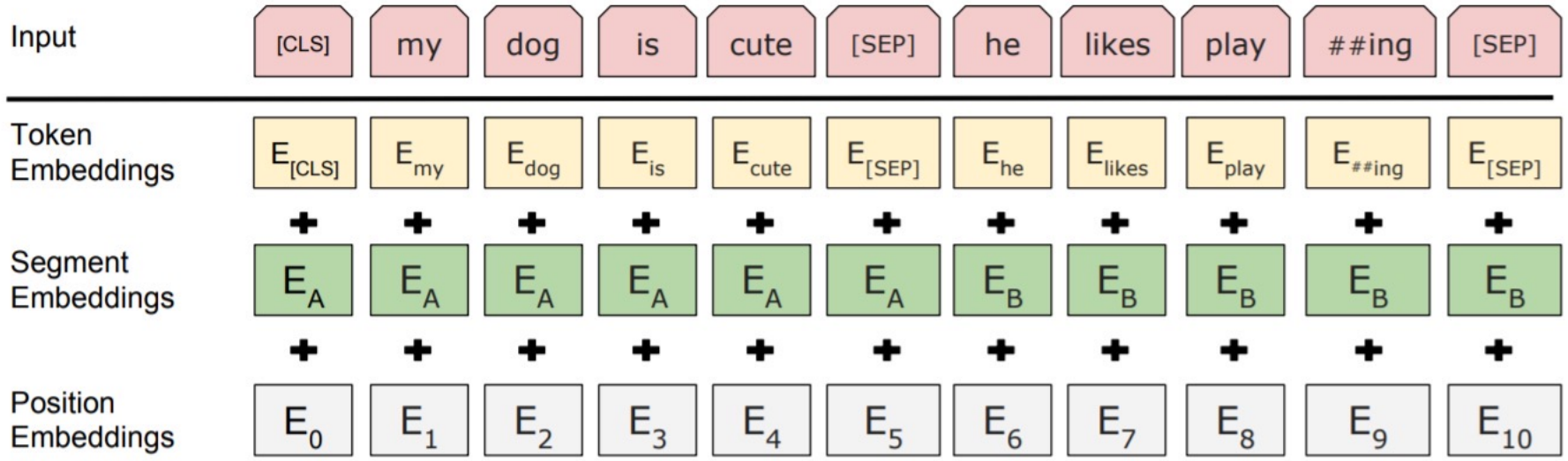
- **Unified Architecture**: As shown below, there are minimal differences between the pre-training architecture and the fine-tuned version for each downstream task.



(Devlin et al., 2018)

BERT: Bidirectional Encoder Representations from Transformers

- 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.
  - Later work has argued this "next sentence prediction" is not necessary.

(Devlin et al., 2018 , Liu et al., 2019)

Details about BERT

- Two models were released:
  - BERT-base: 12 layers, 768-dim hidden states, 12 attention heads, 110 million params.
  - BERT-large: 24 layers, 1024-dim hidden states, 16 attention heads, 340 million params.
- Trained on:
  - BooksCorpus (800 million words)
  - English Wikipedia (2,500 million words)
- Pretraining is expensive and impractical on a single GPU.
  - BERT was pretrained with 64 TPU chips for a total of 4 days.
  - (TPUs are special tensor operation acceleration hardware)
- Finetuning is practical and common on a single GPU
  - "Pretrain once, finetune many times."

(Devlin et al., 2018)

BERT: Bidirectional Encoder Representations from Transformers

- BERT was massively popular and hugely versatile; finetuning BERT led to new state-of- the-art results on a broad range of tasks.
  - QQP: Quora Question Pairs (detect paraphrase questions)
  - QNLI: natural language inference over question answering data
  - SST-2: sentiment analysis
  - CoLA: corpus of linguistic acceptability (detect whether sentences are grammatical.)
  - STS-B: semantic textual similarity
  - MRPC: microsoft paraphrase corpus
  - RTE: a small natural language inference corpus

BERT: Bidirectional Encoder Representations from Transformers

- BERT was massively popular and hugely versatile; finetuning BERT led to new state-of- the-art results on a broad range of tasks.

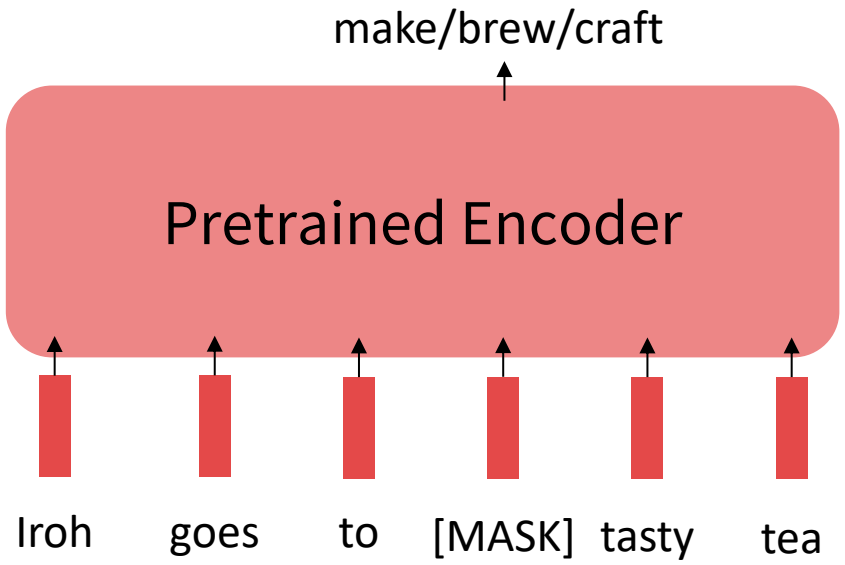
System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	**Average** -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

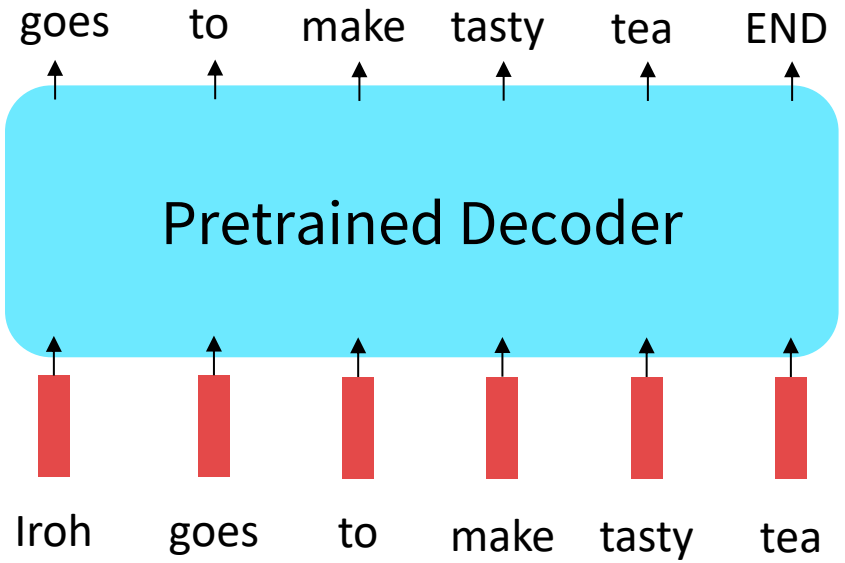
Note that $\text{BERT}_{\text{BASE}}$ was chosen to have the same number of parameters as OpenAI GPT.

(Devlin et al., 2018)

Limitations of pretrained encoders

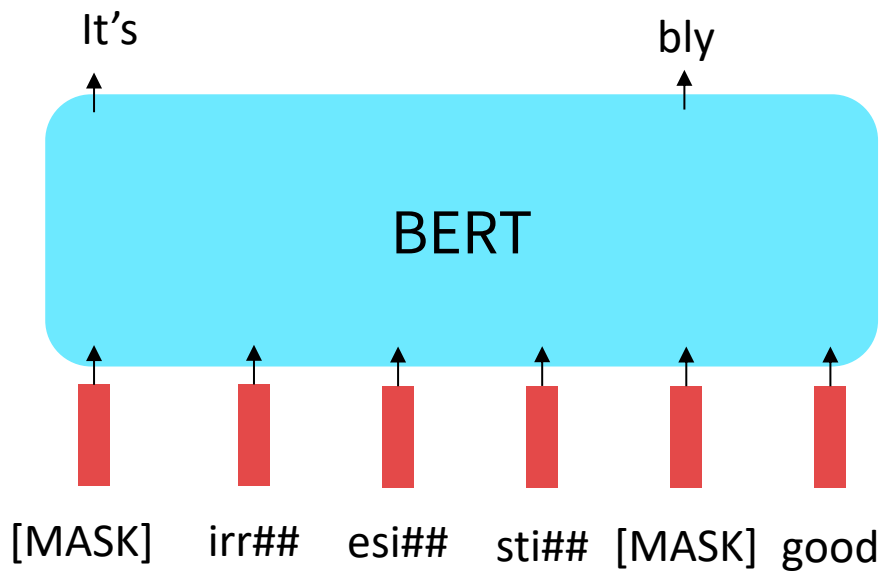
- Those results looked great! Why not used pretrained encoders for everything?
- If your task involves generating sequences, consider using a pretrained decoder; BERT and other pretrained encoders don't naturally lead to nice autoregressive (1-word-at-a-time) generation methods.

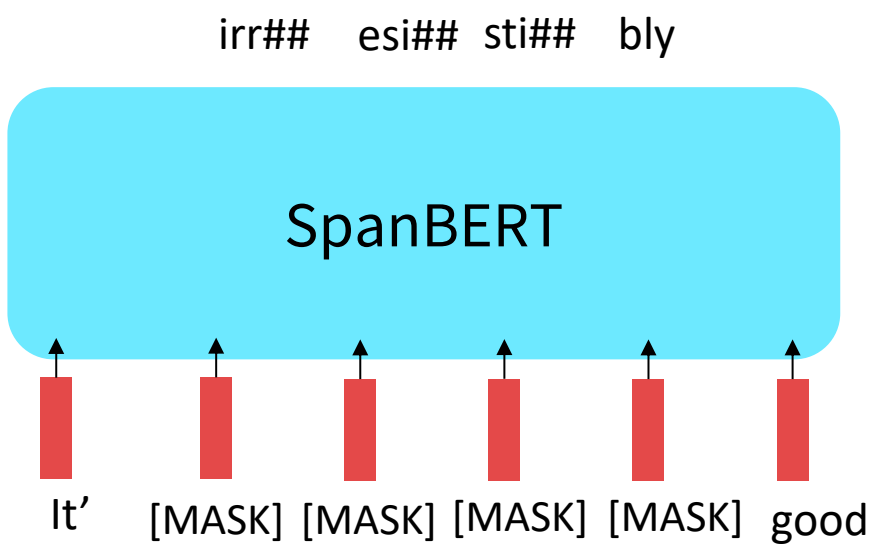




Extensions of BERT

- You'll see a lot of BERT variants like RoBERTa, SpanBERT, +++
- Some generally accepted improvements to the BERT pretraining formula:
  - RoBERTa: mainly just train BERT for longer and remove next sentence prediction!
  - SpanBERT: masking contiguous spans of words makes a harder, more useful pretraining task





(Liu et al., 2019, Joshi et al., 2020)

Extensions of BERT

- A takeaway from the RoBERTa paper: more compute, more data can improve pretraining even when not changing the underlying Transformer encoder.

Model	data	bsz	steps	SQuAD (v1.1/2.0)	MNLI-m	SST-2
RoBERTa						
with BOOKS + WIKI	16GB	8K	100K	93.6/87.3	89.0	95.3
+ additional data (§3.2)	160GB	8K	100K	94.0/87.7	89.3	95.6
+ pretrain longer	160GB	8K	300K	94.4/88.7	90.0	96.1
+ pretrain even longer	160GB	8K	500K	**94.6/89.4**	**90.2**	**96.4**
$\text{BERT}_{\text{LARGE}}$						
with BOOKS + WIKI	13GB	256	1M	90.9/81.8	86.6	93.7

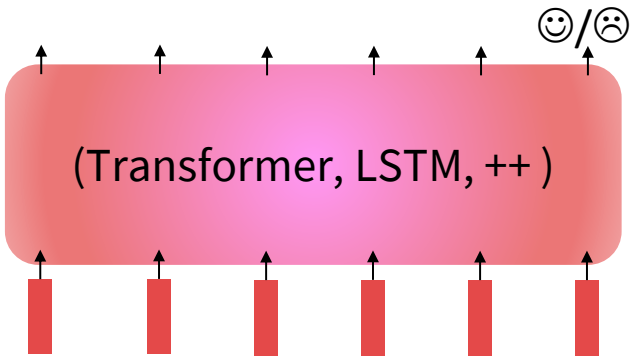
(Liu et al., 2019, Joshi et al., 2020)

Full Finetuning vs. Parameter-Efficient Finetuning

- ❑ Finetuning every parameter in a pretrained model works well, but is memory-intensive.
- ❑ But **lightweight** finetuning methods adapt pretrained models in a constrained way.
- ❑ Leads to less overfitting and/or more efficient finetuning and inference.

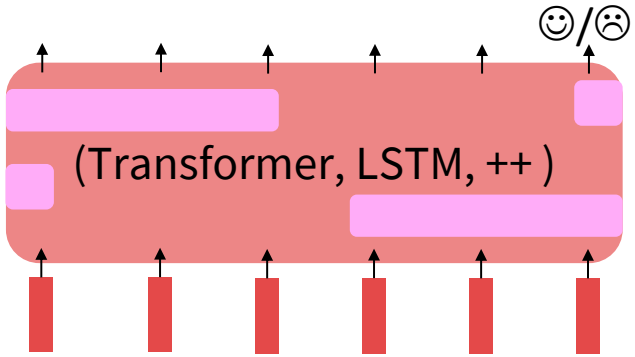
Full Finetuning

Adapt all parameters



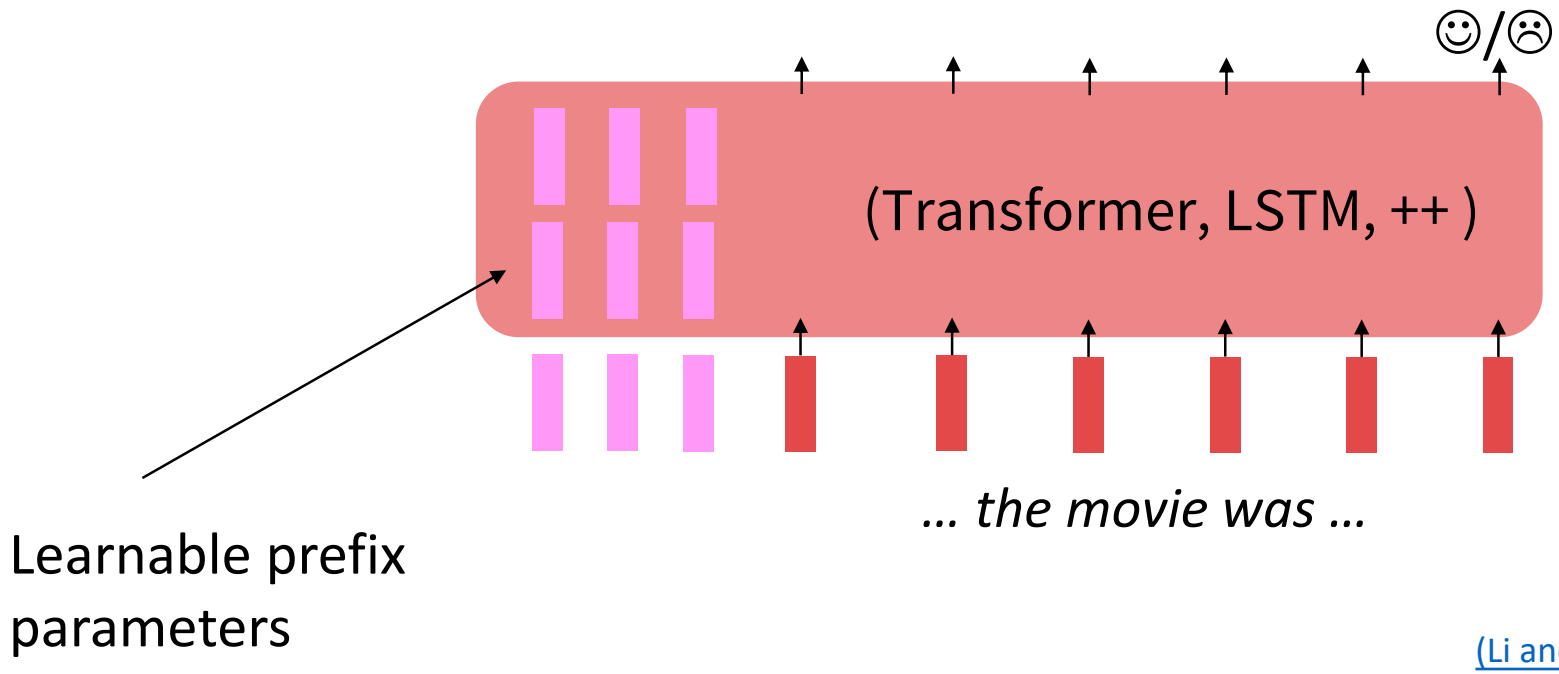
Lightweight Finetuning

Train a few existing or new parameters



Parameter-Efficient Finetuning: Prefix-Tuning, Prompt tuning

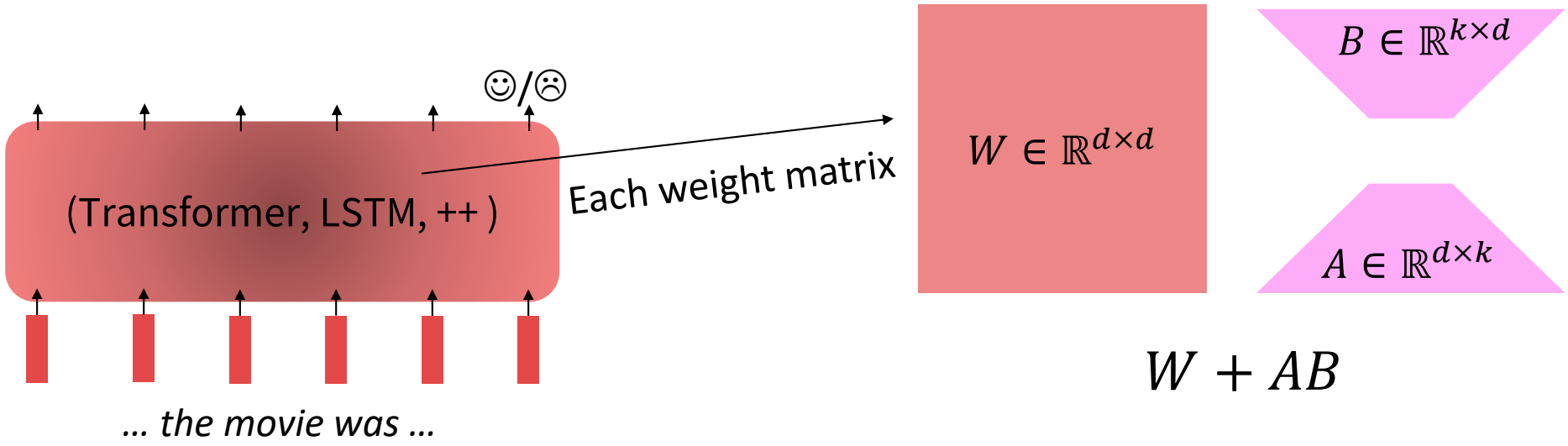
- Prefix-Tuning adds a **prefix** of parameters, and **freezes all pretrained parameters**.
- The prefix is processed by the model just like real words would be.
- Advantage: each element of a batch at inference could run a different tuned model.



(Li and Liang, 2021; Lester et al., 2021)

Parameter-Efficient Finetuning: Low-Rank Adaptation

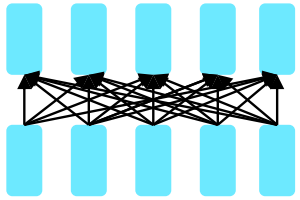
- Low-Rank Adaptation Learns a low-rank "diff" between the pretrained and finetuned weight matrices.
- Easier to learn than prefix-tuning.



(Hu et al., 2021)

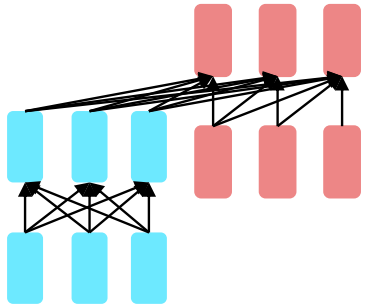
Pretraining for three types of architectures

❑ The neural architecture influences the type of pretraining, and natural use cases.



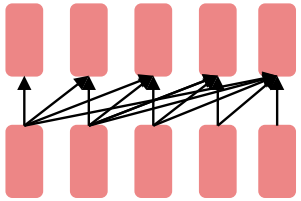
Encoders

- Gets bidirectional context – can condition on future!
- Wait, how do we pretrain them?



Encoder-Decoders

- Good parts of decoders and encoders?
- What's the best way to pretrain them?



Decoders

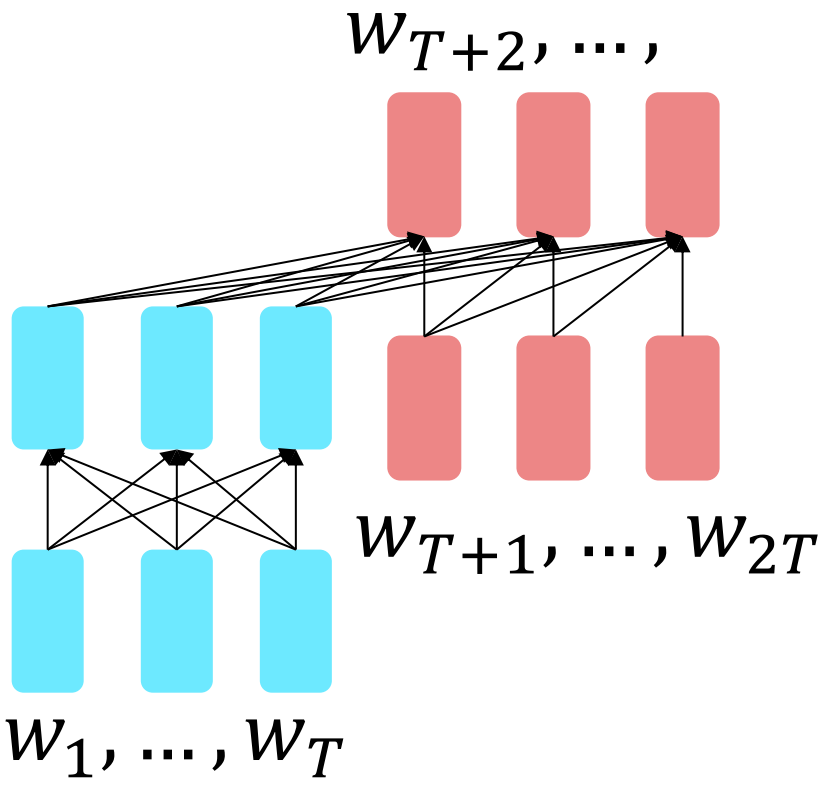
- Language models! What we've seen so far.
- Nice to generate from; can't condition on future words

Pretraining encoder-decoders: what pretraining objective to use?

- For **encoder-decoders**, we could do something like **language modeling**, but where a prefix of every input is provided to the encoder and is not predicted.

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

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



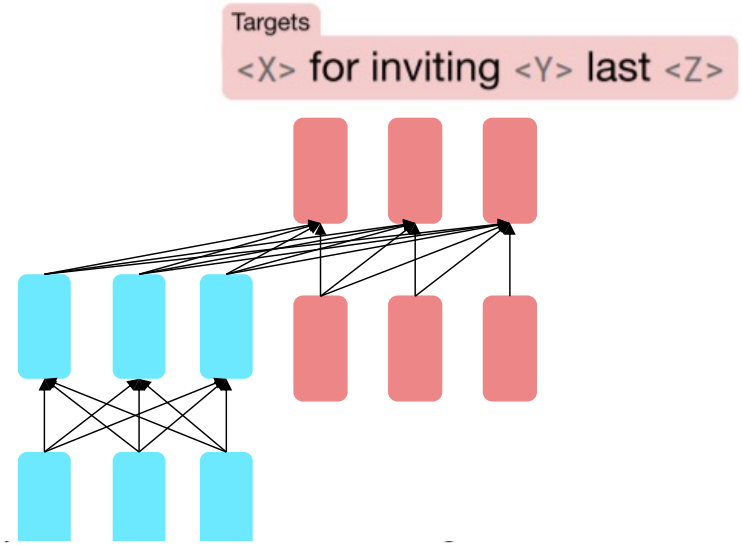
(Raffel et al., 2018)

Pretraining encoder-decoders: what pretraining objective to use?

- What Raffel et al., 2018 found to work best was **span corruption**. Their model: **T5**.
- Replace different-length spans from the input with unique placeholders; decode out the spans that were removed!

Original text

Thank you ~~for inviting~~ me to your party ~~last~~ week.



- This is implemented in text preprocessing: it's still an objective that looks like **language modeling** at the decoder side.

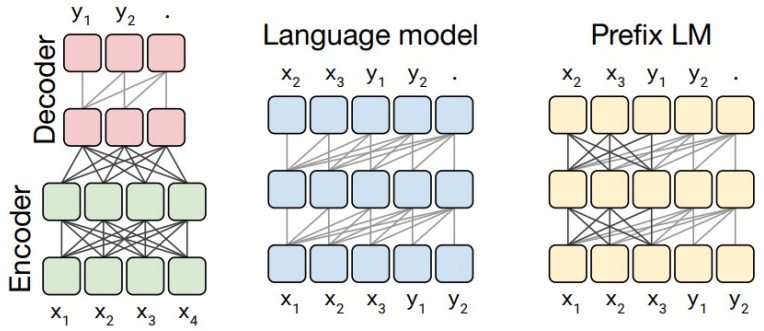
Inputs

Thank you <X> me to your party <Y> week.

(Raffel et al., 2018)

Pretraining encoder-decoders: what pretraining objective to use?

- Raffel et al., 2018 found encoder-decoders to work better than decoders for their tasks, and span corruption (denoising) to work better than language modeling.

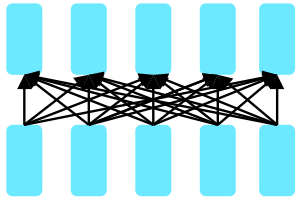


Architecture	Objective	Params	Cost	GLUE	CNNDM	SQuAD	SGLUE	EnDe	EnFr	EnRo
★ Encoder-decoder	Denoising	$2P$	M	**83.28**	**19.24**	**80.88**	**71.36**	**26.98**	**39.82**	**27.65**
Enc-dec, shared	Denoising	P	M	82.81	18.78	**80.63**	**70.73**	26.72	39.03	**27.46**
Enc-dec, 6 layers	Denoising	P	$M/2$	80.88	18.97	77.59	68.42	26.38	38.40	26.95
Language model	Denoising	P	M	74.70	17.93	61.14	55.02	25.09	35.28	25.86
Prefix LM	Denoising	P	M	81.82	18.61	78.94	68.11	26.43	37.98	27.39
Encoder-decoder	LM	$2P$	M	79.56	18.59	76.02	64.29	26.27	39.17	26.86
Enc-dec, shared	LM	P	M	79.60	18.13	76.35	63.50	26.62	39.17	27.05
Enc-dec, 6 layers	LM	P	$M/2$	78.67	18.26	75.32	64.06	26.13	38.42	26.89
Language model	LM	P	M	73.78	17.54	53.81	56.51	25.23	34.31	25.38
Prefix LM	LM	P	M	79.68	17.84	76.87	64.86	26.28	37.51	26.76

(Raffel et al., 2018)

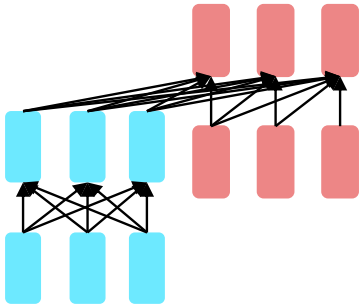
Pretraining for three types of architectures

❑ The neural architecture influences the type of pretraining, and natural use cases.



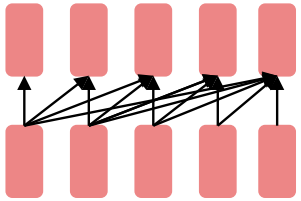
Encoders

- Gets bidirectional context – can condition on future!
- Wait, how do we pretrain them?



Encoder-Decoders

- Good parts of decoders and encoders?
- What's the best way to pretrain them?



Decoders

- Language models! What we've seen so far.
- Nice to generate from; can't condition on future words

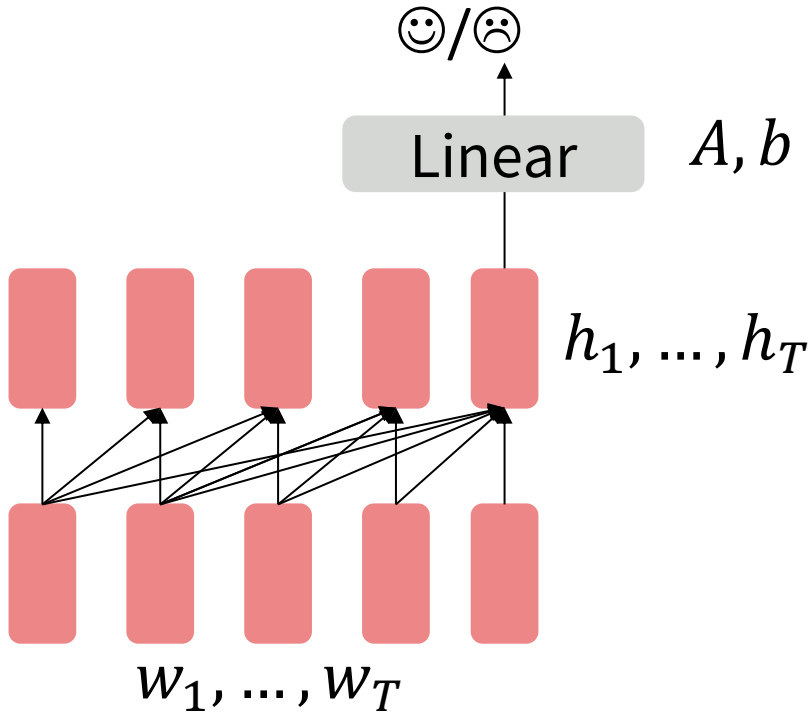
Pretraining decoders

- When using language model pretrained decoders, we can ignore that they were trained to model $p_\theta(w_t|w_{1:t-1})$
- We can finetune them by training a classifier on the last word's hidden state.

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

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

- Gradients backpropagate through the whole network.



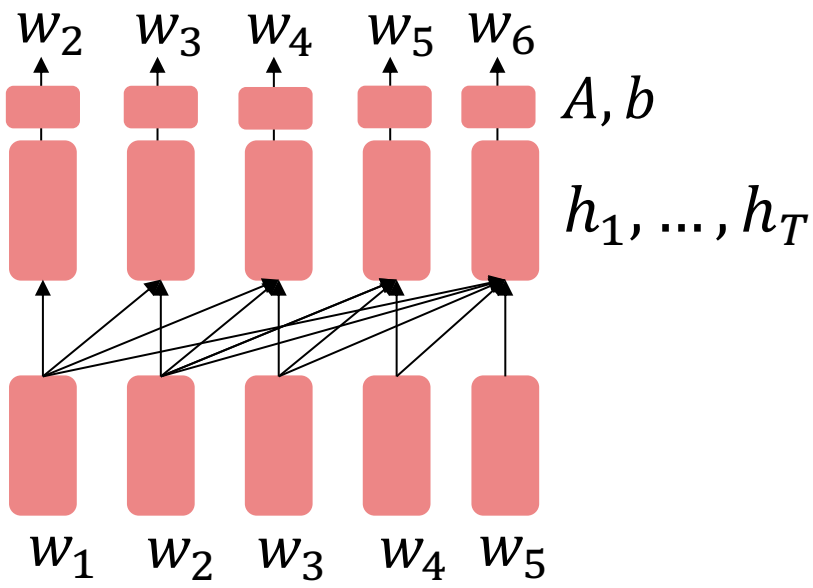
[Note how the linear layer hasn't been pretrained and must be learned from scratch.]

Pretraining decoders

- It's natural to pretrain decoders as language models and then use them as generators, finetuning their $p_\theta(w_t|w_{1:t-1})$
- This is helpful in tasks **where the output is a sequence** with a vocabulary like that at pretraining time!
  - Dialogue (context=dialogue history)
  - Summarization (context=document)

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

where A, b were pretrained in the language model!



[Note how the linear layer has been pretrained.]

Generative Pretrained Transformer (GPT) (Radford et al., 2018)

- 2018's GPT was a big success in pretraining a decoder!
  - Transformer decoder with 12 layers.
  - 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers.
  - Byte-pair encoding with 40,000 merges
  - Trained on BooksCorpus: over 7000 unique books.
    - Contains long spans of contiguous text, for learning long-distance dependencies.

Generative Pretrained Transformer (GPT) (Radford et al., 2018)

- How do we format inputs to our decoder for finetuning tasks?
- Natural Language Inference: Label pairs of sentences as entailing/contradictory/neutral

  Premise: *The man is in the doorway*

  Hypothesis: *The person is near the door*

  } **entailment**

- Radford et al., 2018 evaluate on natural language inference.
- Here's roughly how the input was formatted, as a sequence of tokens for the decoder.

  [START] *The man is in the doorway* [DELIM] *The person is near the door* [EXTRACT]

- The linear classifier is applied to the representation of the [EXTRACT] token.

Generative Pretrained Transformer (GPT) (Radford et al., 2018)

- GPT results on various *natural language inference* datasets.

Method	MNLI-m	MNLI-mm	SNLI	SciTail	QNLI	RTE
ESIM + ELMo [44] (5x)	-	-	89.3	-	-	-
CAFE [58] (5x)	80.2	79.0	89.3	-	-	-
Stochastic Answer Network [35] (3x)	80.6	80.1	-	-	-	-
CAFE [58]	78.7	77.9	88.5	83.3		
GenSen [64]	71.4	71.3	-	-	82.3	59.2
Multi-task BiLSTM + Attn [64]	72.2	72.1	-	-	82.1	**61.7**
Finetuned Transformer LM (ours)	**82.1**	**81.4**	**89.9**	**88.3**	**88.1**	56.0

Generative Pretrained Transformer (GPT) (Radford et al., 2018)

- GPT results on various *question answering and commonsense reasoning* datasets.

Method	Story Cloze	RACE-m	RACE-h	RACE
val-LS-skip [55]	76.5	-	-	-
Hidden Coherence Model [7]	77.6	-	-	-
Dynamic Fusion Net [67] (9x)	-	55.6	49.4	51.2
BiAttention MRU [59] (9x)	-	60.2	50.3	53.3
Finetuned Transformer LM (ours)	**86.5**	**62.9**	**57.4**	**59.0**

Increasingly convincing generations (GPT2) (Radford et al., 2018)

- We mentioned how pretrained decoders can be used in their capacities as language models.
- GPT-2, a larger version of GPT (48 layers) trained on more data (~ x10), was shown to produce relatively convincing samples of natural language.

Context (human-written): In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.
GPT-2: The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science. Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

GPT-3, In-context learning, and very large models

- So far, we've interacted with pretrained models in two ways:
  - Sample from the distributions they define (maybe providing a prompt)
  - Fine-tune them on a task we care about, and then take their predictions.

- Emergent behavior: Very large language models seem to perform some kind of learning **without gradient steps** simply from examples you provide within their contexts.
- GPT-3 is the canonical example of this. The largest T5 model had 11 billion parameters. **GPT-3 has 175 billion parameters**.

GPT-3, In-context learning, and very large models

- Very large language models seem to perform some kind of learning **without gradient steps** simply from examples you provide within their contexts.

- The in-context examples seem to specify the task to be performed, and the conditional distribution mocks performing the task to a certain extent.
  - Input (prefix within a single Transformer decoder context):

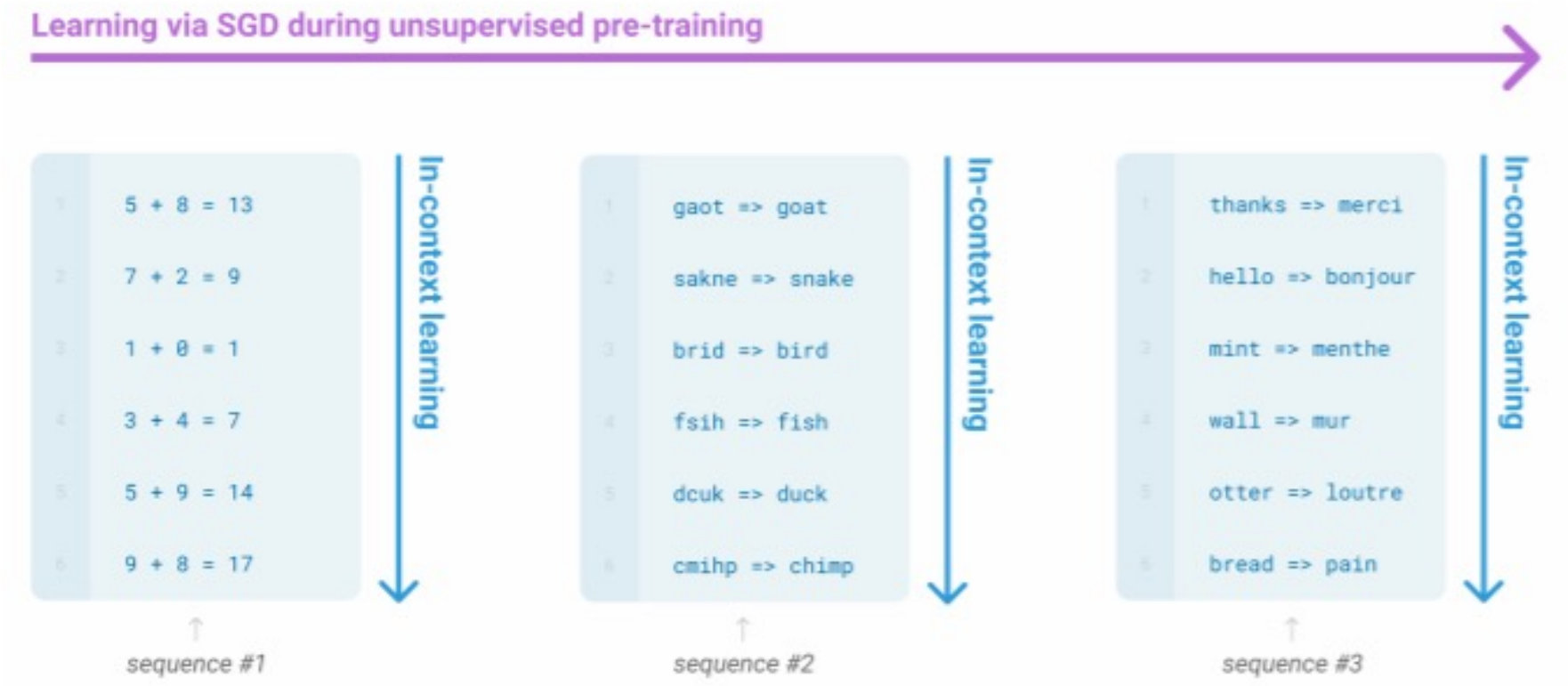
    “ thanks -> merci
    hello -> bonjour
    mint -> menthe
    otter -> ”

  - Output (conditional generations):

    loutre…”

GPT-3, In-context learning, and very large models

- Very large language models seem to perform some kind of learning **without gradient steps** simply from examples you provide within their contexts.



Scaling Efficiency: how do we best use our compute

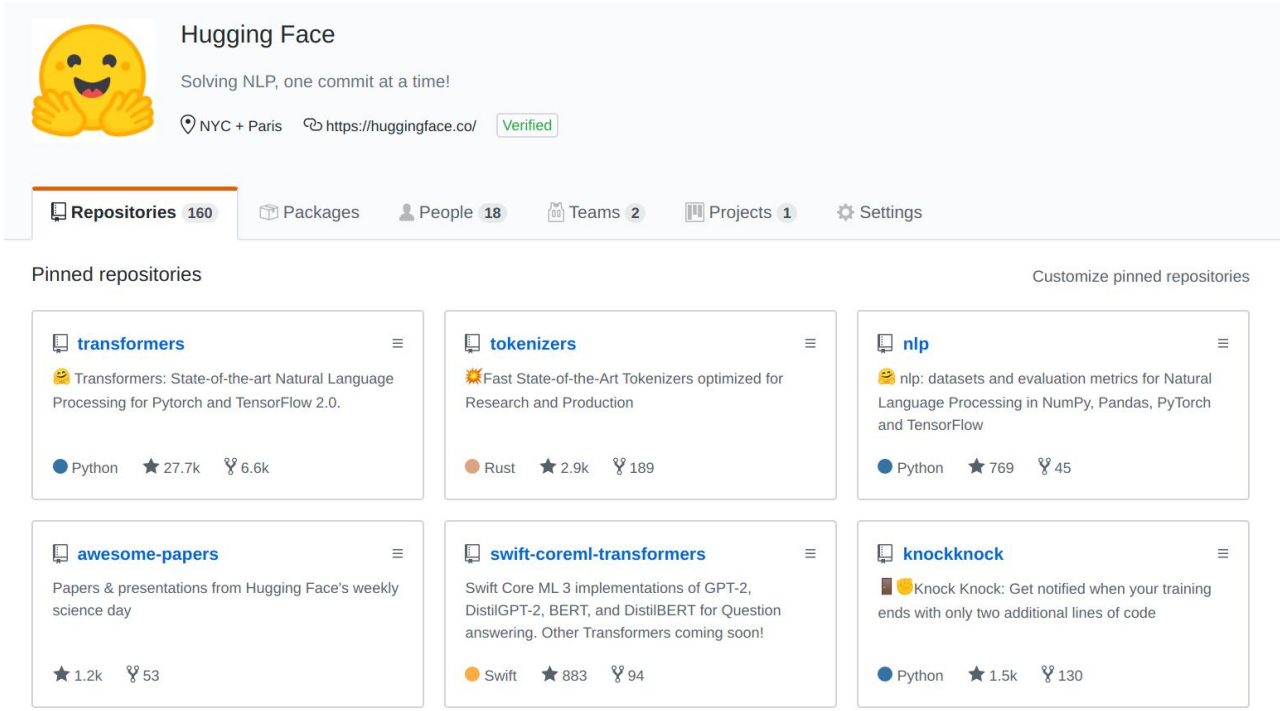
- GPT-3 was **175B parameters** and trained on **300B** tokens of text.
- Roughly, the cost of training a large transformer scales as **parameters*tokens**
- Did OpenAI strike the right parameter-token data to get the best model? No.

Model	Size (# Parameters)	Training Tokens
LaMDA (Thoppilan et al., 2022)	137 Billion	168 Billion
GPT-3 (Brown et al., 2020)	175 Billion	300 Billion
Jurassic (Lieber et al., 2021)	178 Billion	300 Billion
Gopher (Rae et al., 2021)	280 Billion	300 Billion
MT-NLG 530B (Smith et al., 2022)	530 Billion	270 Billion
Chinchilla	70 Billion	1.4 Trillion

This 70B parameter model is better than the much larger other models!

Hugging Face: a quick tutorial

- Most popular open source NLP library
  - 1,000+ Research paper mentions
  - Used in production by 1000+ companies



Hugging Face: a quick tutorial

- The most basic object in the Transformers library is the **pipeline()** function.
  - It connects a model with its necessary preprocessing and postprocessing steps, allowing us to directly input any text and get an intelligible answer

- Sentiment-analysis

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")
classifier("I've been waiting for a HuggingFace course my whole life.")
```

```
[{'label': 'POSITIVE', 'score': 0.9598047137260437}]
```

Hugging Face: a quick tutorial

- Zero-shot classification
  - we need to classify texts that haven't been labelled
  - it use a model trained for NLI task (contradiction, neutral, and entailment)
  - for list of candidate labels, each sequence/label pair is fed through the model as a premise/hypothesis pair, and it gets out the logits for these three categories for each label.

```
from transformers import pipeline

classifier = pipeline("zero-shot-classification")
classifier(
    "This is a course about the Transformers library",
    candidate_labels=["education", "politics", "business"],
)
```

```
{'sequence': 'This is a course about the Transformers library',
 'labels': ['education', 'business', 'politics'],
 'scores': [0.8445963859558105, 0.111976258456707, 0.043427448719739914]}
```

Hugging Face: a quick tutorial

- Text generation
  - you provide a prompt and the model will auto-complete it by generating the remaining text.

```
from transformers import pipeline

generator = pipeline("text-generation")
generator("In this course, we will teach you how to")
```

```
[{'generated_text': 'In this course, we will teach you how to understand and use '
                    'data flow and data interchange when handling user data. We '
                    'will be working with one or more of the most commonly used '
                    'data flows — data flows of various types, as seen by the '
                    'HTTP'}]
```

Hugging Face: a quick tutorial

- Using different models in a pipeline
  - so far, we have use default models but you can also choose a particular model from the Hub to use in a pipeline for a specific task

```
from transformers import pipeline

generator = pipeline("text-generation", model="distilgpt2")
generator(
    "In this course, we will teach you how to",
    max_length=30,
    num_return_sequences=2,
)
```

```
[{'generated_text': 'In this course, we will teach you how to manipulate the world and '
                    'move your mental and physical capabilities to your advantage.'},
 {'generated_text': 'In this course, we will teach you how to become an expert and '
                    'practice realtime, and with a hands on experience on both real '
                    'time and real'}]
```

Hugging Face: a quick tutorial

- Named entity recognition
  - Named entity recognition (NER) is a task where the model has to find which parts of the input text correspond to entities such as persons, locations, or organizations.

```
from transformers import pipeline

ner = pipeline("ner", grouped_entities=True)
ner("My name is Sylvain and I work at Hugging Face in Brooklyn.")
```

```
[{'entity_group': 'PER', 'score': 0.99816, 'word': 'Sylvain', 'start': 11, 'end': 18},
 {'entity_group': 'ORG', 'score': 0.97960, 'word': 'Hugging Face', 'start': 33, 'end': 45},
 {'entity_group': 'LOC', 'score': 0.99321, 'word': 'Brooklyn', 'start': 49, 'end': 57}
]
```

Hugging Face: a quick tutorial

- Question answering

```
from transformers import pipeline

question_answerer = pipeline("question-answering")
question_answerer(
    question="Where do I work?",
    context="My name is Sylvain and I work at Hugging Face in Brooklyn",
)
```

```
{'score': 0.6385916471481323, 'start': 33, 'end': 45, 'answer': 'Hugging Face'}
```

Hugging Face: a quick tutorial

- Summarization
  - Summarization is the task of reducing a text into a shorter text while keeping all (or most) of the important aspects referenced in the text.

```
from transformers import pipeline

summarizer = pipeline("summarization")
summarizer(
    """
    America has changed dramatically during recent years. Not only has the number of
    graduates in traditional engineering disciplines such as mechanical, civil,
    electrical, chemical, and aeronautical engineering declined, but in most of
    the premier American universities engineering curricula now concentrate on
    suffers an increasingly serious decline in the number of engineering graduates
    and a lack of well-educated engineers.
"""
)
```

```
[{'summary_text': ' America has changed dramatically during recent years . The '
                  'number of engineering graduates in the U.S. has declined in '
```

Hugging Face: a quick tutorial

- Translation
  - You can either specify a language pair in the task name or a model to use

```
from transformers import pipeline

translator = pipeline("translation", model="Helsinki-NLP/opus-mt-fr-en")
translator("Ce cours est produit par Hugging Face.")
```

```
[{'translation_text': 'This course is produced by Hugging Face.'}]
```

Hugging Face: a quick tutorial

- Currently available pipelines
  - See https://huggingface.co/docs/transformers/main_classes/pipelines

- Also provides dataset library and tokenizer library
  - For how to use, see https://huggingface.co/course/chapter5/1?fw=pt and https://huggingface.co/course/chapter6/1?fw=pt

- Fine-tuning a pretrained model
  - For how to do so, see https://huggingface.co/course/chapter3/1?fw=pt