Advanced Methods in Text Analytics

Transfer Learning

What is Transfer Learning?

- "Acquiring knowledge from one task or domain, and then applying it (*transferring* it) to solve a new task." Jurafsky et al. 2025
- **Common approach** follows **two-steps** (Howard and Ruder, 2018)**:**
  1. **Pre-training** a model on some tasks that "allows" model to learn rich representations
  2. **Fine-tuning** the learned representations (weights) in the context of a new downstream application
- Why is it called **pre**-training?
  - Train model for general use *before* using it on specific applications
- **Fine-tuning** refers to **further updating weights** of pre-trained model
  - Downstream task usually implies using a **downstream model** in combination with the pre-trained model
  - Updating weights **may result in "catastrophic forgetting"**
- Focus this week: pre-training tasks and architectures
  - Next week: fine-tuning methods and applications

But First, a Word on Tokenization

- Recall introduction lecture: **tokenization is important!**
  - Should we tokenize words? What are words?
  - Should we tokenize characters? What meaning do they encode?
- Nice sweet spot: tokenize subwords
  - E.g. *unlikeliest -> un-likely-est*
  - Each a morpheme, i.e. a meaning-bearing unit
- **Byte-Pair Encoding** (Sennrich et al. 2016): very common subword tokenizer, roughly as follows
  1. Starting vocabulary *V* is set of characters
  2. Find most common two-character subword, create token for it, add to *V*
  3. Keep finding common and longer subwords until *k* new tokens are created
- Thus, can handle unknown words by breaking them into characters
  - Variants exist, e.g. WordPiece, SentencePiece
- Tokenization is the focus of an entire future lecture

Outline

1. **Pre-Training**
   1. Transformer-based language models
   2. Masked Language Models
   3. Causal (or Autoregressive) Language Models

2. **Fine-Tuning**

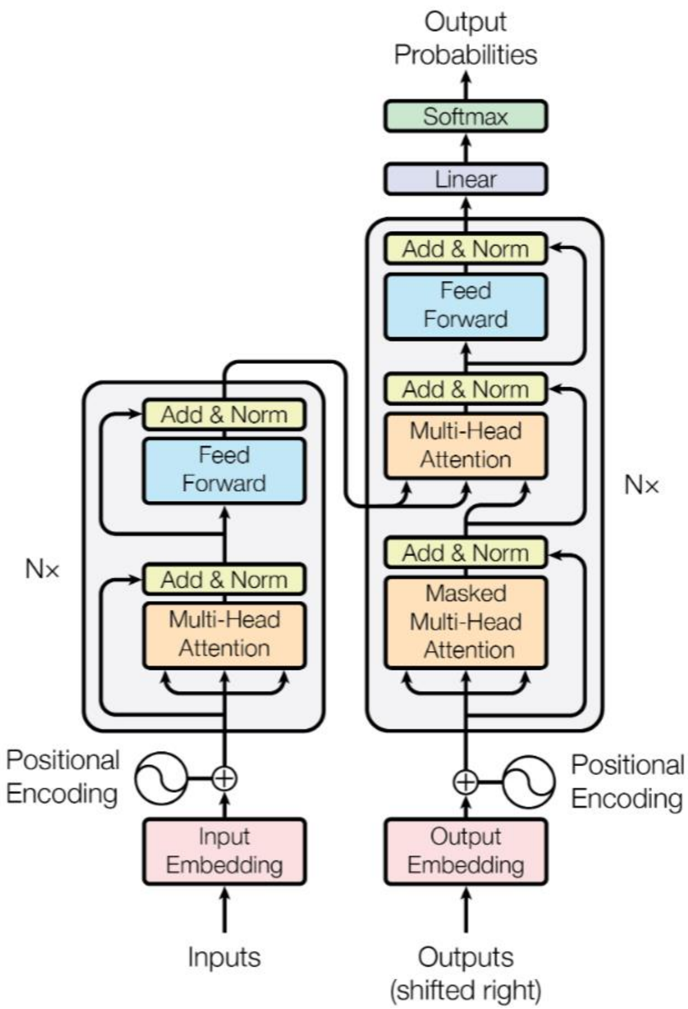
Pre-Training

Pre-Training Methods

- Take **skip-gram** as an **example of pre-training**
  - **Goal:** learn useful (static) representations
  - **Task:** predict whether two words are likely to be in context
- In other words, the **actual goal is not necessarily solving the task**
- There are **different tasks for pre-training**
  - Each a **different training objective**
  - Multi-task training possible, e.g. objective is linear combination of tasks
- **Most common** pre-training **tasks/approaches**
  - **Masked Language Models** (MLMs)
  - **Causal (or Autoregressive) Language Models** (CLMs)
- Different task -> usually different architecture and objective
  - MLM: **BERT**, encoder-only transformer
  - CLM: **GPTs**, decoder-only transformer
  - Other variants exist, all based on the transformer architecture
- First: transformer-based LMs, then: MLMs and CLMs

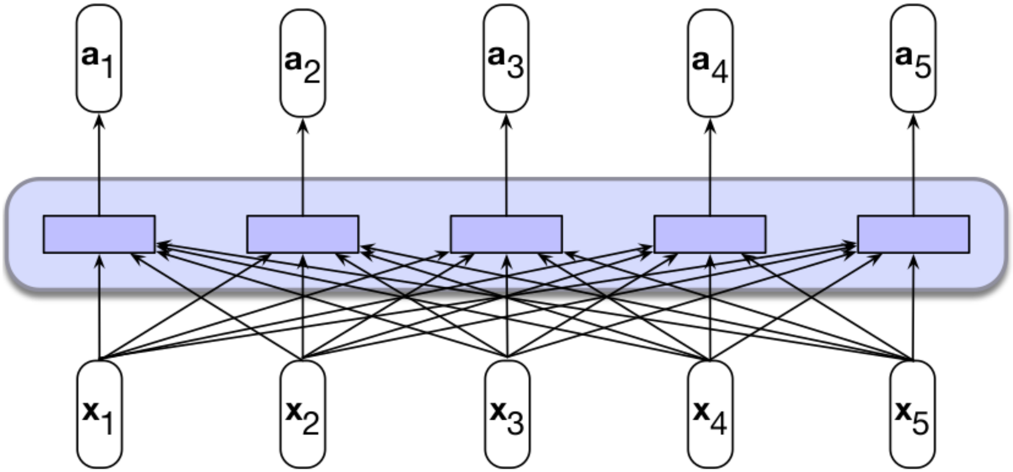
Encoder-Only and Decoder-Only

- Recall the transformer architecture
  - Encoder-decoder
- Some models use only the encoder
  - **Input:** tokenized input sequence
  - **Output:** contextualized input tokens
- Other models use only the decoder
  - **Input:** start symbol, previous outputs
  - **Output:** new generated word
- Both can be stacked



Recap: Self-Attention Layer (1)

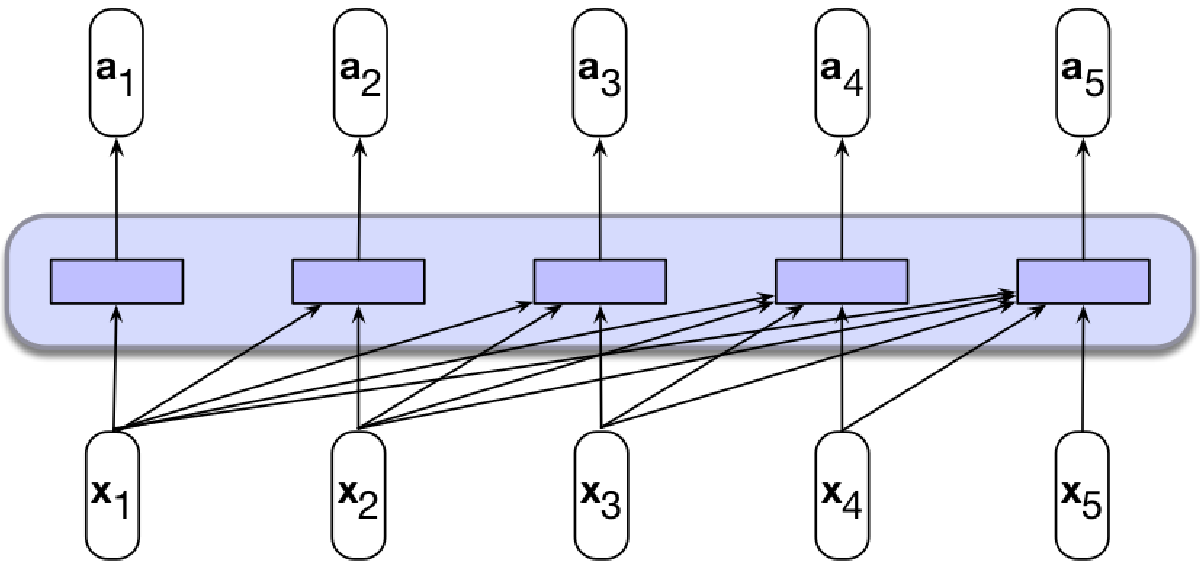
- **Input:** sequence of *n* tokens
- **Output:** sequence of *n contextualized* tokens (here, **bidirectional**)



- Layer is parameterized by $\boldsymbol{W}^Q$, $\boldsymbol{W}^K$ and $\boldsymbol{W}^V$
  - Each a transformation for using tokens as: queries, keys, values
- Thus, **same words gets different representations based on context**
  - “They **broke** in tears when they heard they got the new **flat**.”
  - “They **broke** the world record by 2 seconds **flat**.”

Recap: Self-Attention Layer (2)

- **Self-attention can also be causal**
    - Token at time step *t* attends only to previous inputs (< *t*)

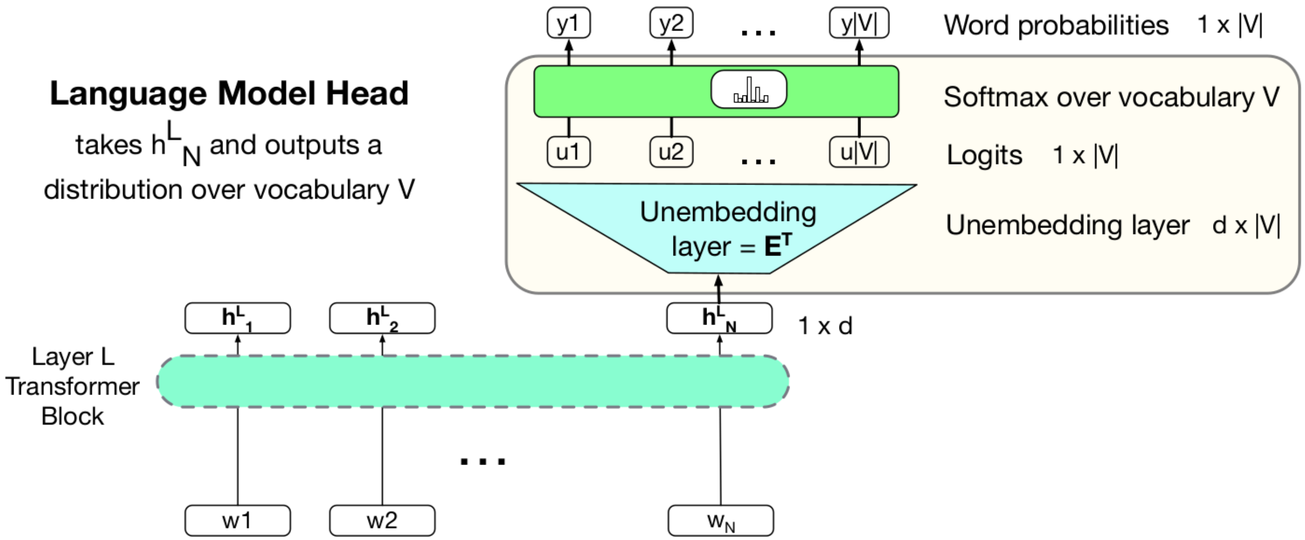


- Input and output still the same as any self-attention layer
    - But **"context"** of each output embedding is **only inputs from previous steps**
    - Recall that without positional embeddings, there is no "time"

The Language Modeling Head (1)

- A **transformer block/layer** typically outputs token representations
  - Transformer block/layer = self-attention + MLP
- **Output embeddings are same size as input embeddings**
  - Can either be a sequence in encoder-only models
  - Or a single token per time-step in decoder-only models
- How do we turn these outputs into a language model?
  - Each representation fed to a "classification head" (**a downstream model**)
  - **Classification head:** softmax *layer* that projects to space of possible classes
  - **Language modeling head:** projects to vocabulary space
  - I.e. same mechanism used for language modeling with RNNs
- If the **task** is to **predict next word:**
  - Feed *contextualized* representation of input at time step *t* to LM head to predict word at time step *t + 1*
- Let's visualize this!

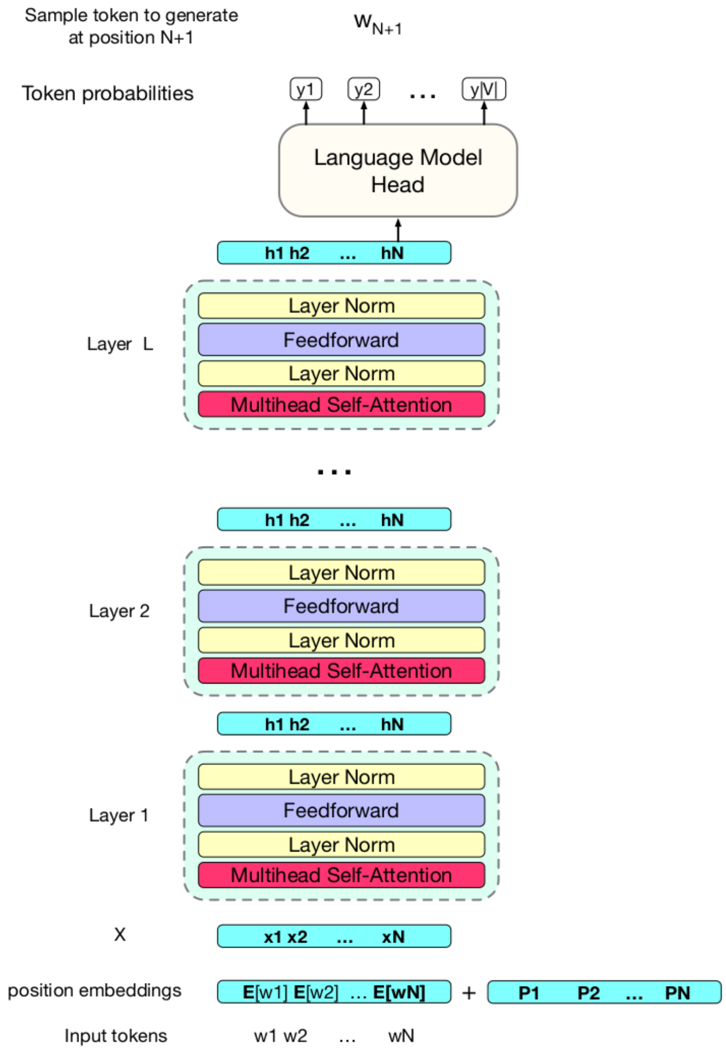
The Language Modeling Head (2)



- $\boldsymbol{h}^L{}_N \in R^d$ is output embedding of block *L* at time step *N*
- Unembedding layer $\boldsymbol{U} \in R^{d \times |V|}$ projects from *d* to *|V|* (vocabulary space)
  - Output is logits, used as input to softmax *function* (no weights)
- Note that ***U*** is same size as initial (static) embedding layer $\boldsymbol{E} \in R^{|V| \times d}$
  - Hence, we usually set to be $\boldsymbol{U} = \boldsymbol{E}^T$ (weight tying, Press et al. 2017)
  - **Weight tying:** use same matrix as ***E*** and ***U***, known to improve performance

Transformer-Based LMs

- All together:
  - (Static) input embeddings $\boldsymbol{E}[w_i]$
  - Positional embeddings $\boldsymbol{p}_i$ (added)
  - *L* transformer layers (blocks)
  - Language model head
- For each **transformer layer/block**
  - **Input:** sequence of embeddings of size *d*
  - **Output:** sequence of embeddings of size *d*
- For **language model head**
  - **Input:** single embedding of size *d*
  - **Output:** probability of each token in vocab. *V*
- **Input to downstream model depends on task**
  - E.g. for sequence classification, we need to represent entire input sequence
  - For sequence labelling, each token representation is classified



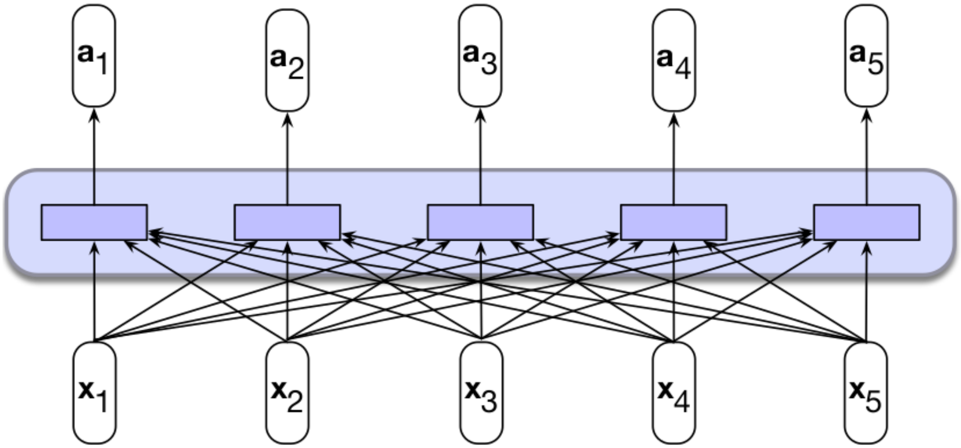
Masked Language Models

Masked Language Models (1)

- Using information about the future in a sequence is often not suitable
  - E.g. in text generation, we use past generated words to generate new ones
  - We don’t want to train a model to use information that isn't available at inference time
- But **often the entire input sequence is available during inference**!
  - E.g. in machine translation we get an entire document to translate
  - In dialogue scenarios, we get entire input from user at each turn
- And **looking at subsequent tokens can be useful for certain tasks**
  - Tasks that require labeling each sequence element (token-level classification)
  - E.g. part-of-speech tagging (adjectives come *before* nouns in English)
  - Generally, we obtain richer representations by using more context
- With transformers, **we control that in the self-attention layers**
  - If we can't look at future tokens: **causal self-attention**
  - If we can look at future tokens: **bidirectional self-attention**

Masked Language Models (2)

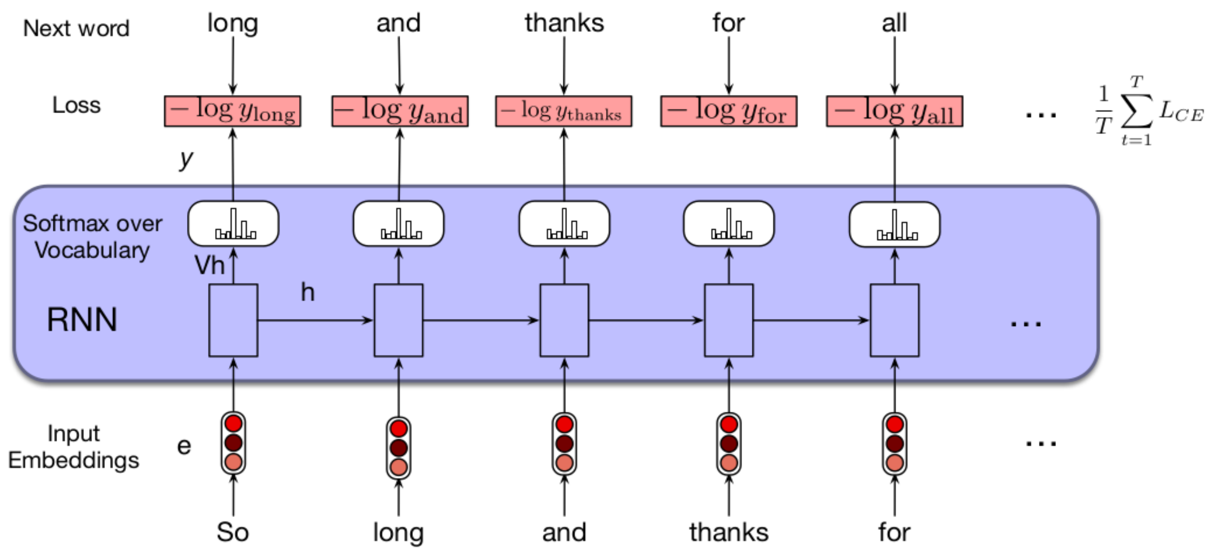
- Contextualized representations from **bidirectional self-attention layers** consider **context from both sides of each input word**



- **Masked language modeling:** predicting masked words in input sequence
  - **Masked words:** hidden, usually replaced with [MASK] token
  - **Architecture:** we use *bidirectional* transformer encoders to train models to predict *masked* words in the input sequence
- What exactly does *masking* mean? The training approach should clarify!

Training Masked Language Models (1)

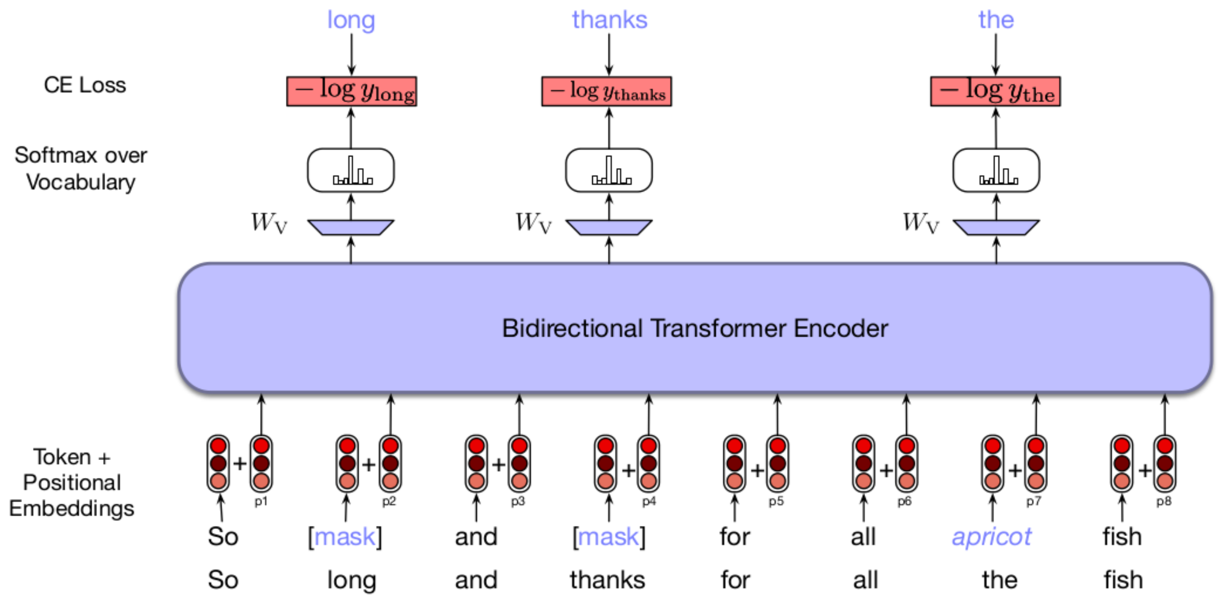
- Recall training RNNs
  - We predict the next word given previous ones
  - The loss corresponding to each input word is measured against next word in input sequence, final loss is average over all tokens



- **Can we do that here?**
  - **No**, bidirectional self-attention looks at next word!
  - Task of predicting next word therefore trivial

Training Masked Language Models (2)

- Instead, **cloze task**, i.e. fill-in the blank ([Taylor, 1953](#))
  - “Turn ____ homework in.”
  - Many examples from same sentence: “Turn your __________ in.”

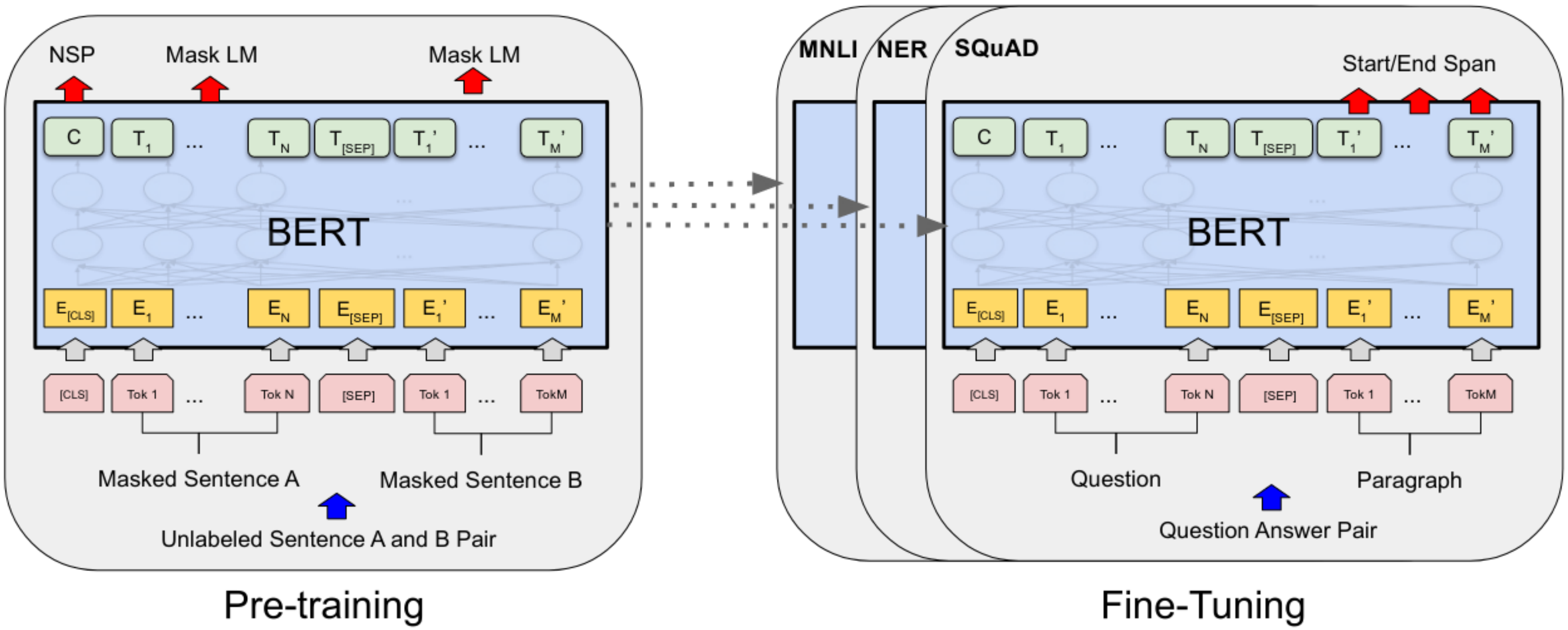


- **Loss averaged over masked tokens only**, all tokens "attended" to
  - Note: **self-supervision**, predict parts of input using rest of input
  - More details by discussing quintessential MLM: **BERT**

The BERT Model (1)

- **BERT: B**idirectional **E**ncoder **R**epresentations from **T**ransformers
- Seminal work by [Devlin et al. 2018]
  - A clear success case is transfer learning
  - **Revolution:** jump in state-of-the-art performance in several NLP tasks
  - Defined masked language modeling as discussed today
- **Goal:** learn representations of language
  - Not predict masked words, i.e. not the task from training objective
  - Thus, similar to word embeddings, but now using transformers
- Their **general architecture**:
  - **Encoder-only** transformer
  - 12 stacked layers of transformer encoders, each with 12 attention heads
  - Hidden layers of size 728
  - Max input length: 512 tokens (recall: attention cost quadratic in this length)
- Overall: **100M parameters**

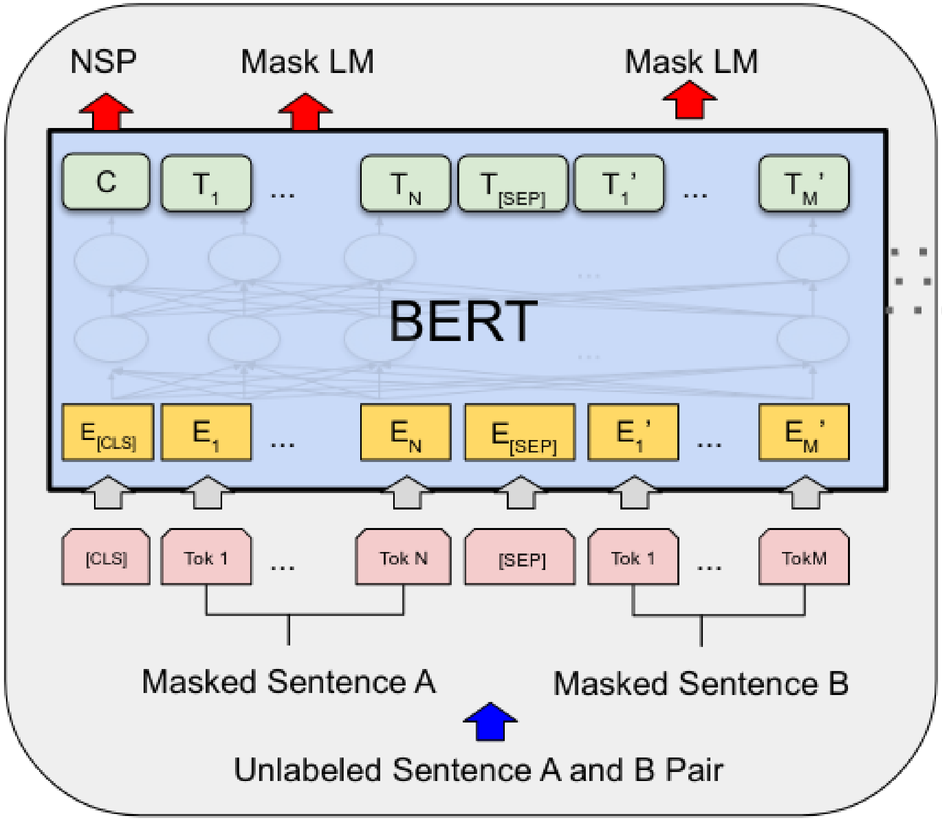
The BERT Model (2)



- Pre-training -> Fine-tuning framework
  - **Pre-training** designed to include both token level and sentence level tasks
  - **Fine-tuning** required minimal changes to pre-training architecture
- Let's discuss some of these important design choices

Pre-Training BERT (1)

- Two level representations
  - Both in same input sequence
- **Token level**
  - E_i inputs, T_i outputs
- **Sentence level**
  - **[CLS]**, **[SEP]** inputs
  - **C**, $\mathbf{T_{[SEP]}}$ outputs
- Thus, model "aware" of sentence pairs
  - Useful for downstream tasks
  - E.g. (question, answer) pairs
- **[CLS] token** meant for sequence classification
  - Final representation C used as sequence representation
- **[SEP] token** separates two input sentences



Pre-Training BERT (2)

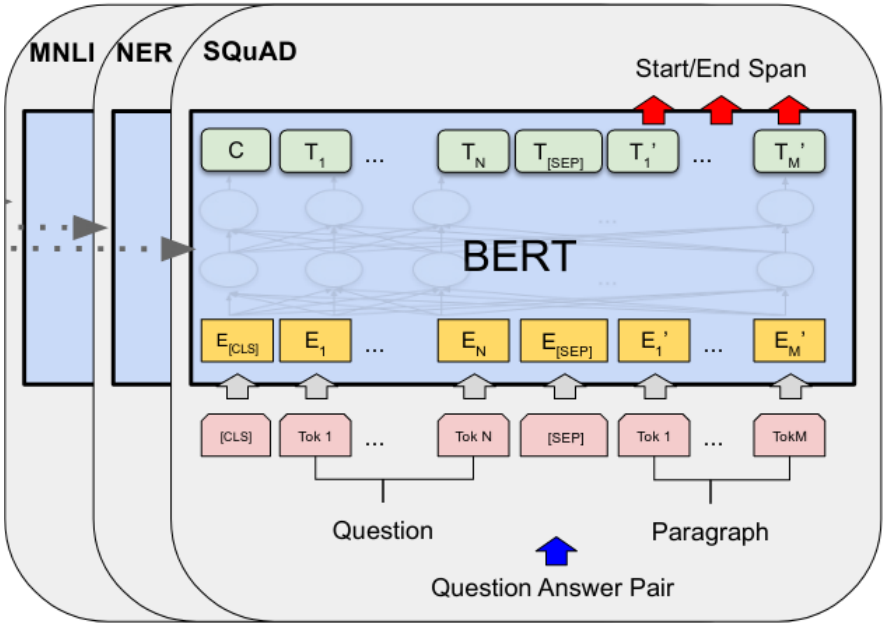
- They (pre-)trained with **two self-supervised tasks**
  1. Masked language modeling (MLM), i.e. fill-in the blank
  2. Next sentence prediction (NSP), e.g. for QA, natural language inference
- **For MLM training objective**:
  - Replaced 15% of input tokens in training corpus
    - 80% of that time replaced with [MASK] token
    - 10% of that time replaced with another random token
    - 10% left unchanged
  - Output token T_i fed to softmax for classification, original word as target
  - Why all of this? Because [MASK] token never seen at inference time!
    - So, model does should not rely on this signal to make such predictions
- For **next sentence prediction objective**:
  - Binary classification: next sentence or not? ([CLS] token to softmax layer)
  - 50% of examples were subsequent sentences in training corpus (positives)
  - 50% of examples had 2nd sentence be a randomly chosen one (negatives)

Pre-Training BERT (3)

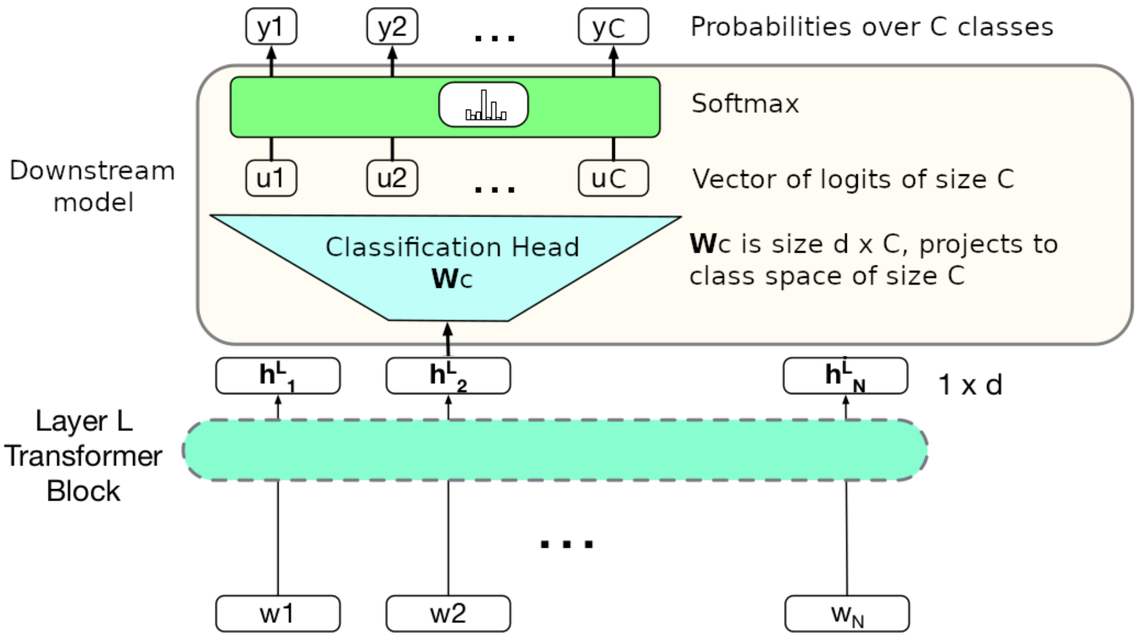
- **Final loss:** combined loss of both MLM and NSP training objectives
- **Training corpus**
  - BooksCorpus: 800M words
  - English Wikipedia: 2.5B words
  - (Much less data than used for training large language models today)
- **Important:** training data must be full documents, not shuffled sentences (often a valid approach)
  - Why?
- **Tokenization:** *WordPiece* subword embeddings
  - Vocabulary size: 30K
  - Recall: subwords is nice balance between words and characters
- It took 40 epochs to converge model during training
  - With the hardware at the time (64 Google TPUs), that was 4 days

Fine-Tuning BERT (1)

- **Straightforward:** representations computed by BERT used as input for downstream model
  - Note: requires using entire model, not just its weights as word2vec
- Can handle **tasks** where **input is single sequence**, e.g. sequence labeling (NER) and sequence classification, or **pairs of sequences**, e.g. QA (SQuAD) or NLI (MNLI)
- Given input data, e.g. sequence to classify**:**
  - **Sequence** fed as **input** to BERT
  - **BERT output** passed to classification head (each token for sequence labeling, CLS token for sequence classification)
- Let's visualize this!



Fine-Tuning BERT (2)



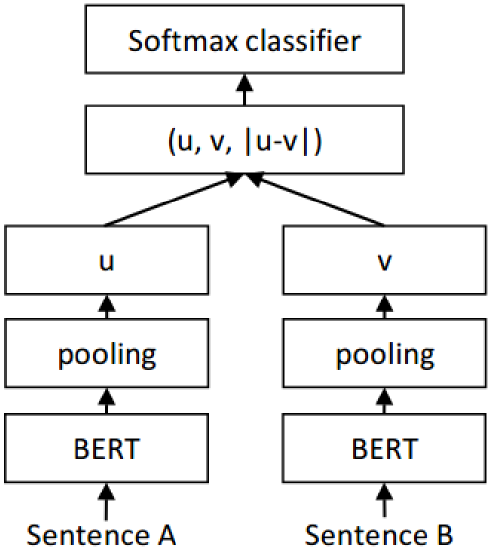
- Here's an example for sequence labeling, i.e. we classify each token $\boldsymbol{h}^L_i$
  - Downstream model is linear projection $\boldsymbol{W}_C$ to classification space C
  - During fine-tuning, $\boldsymbol{W}_C$ + all model parameters were updated
  - Fine-tuning model often took 1 hour (doable compared to pre-training)
- For sequence classification task, CLS token used as downstream input

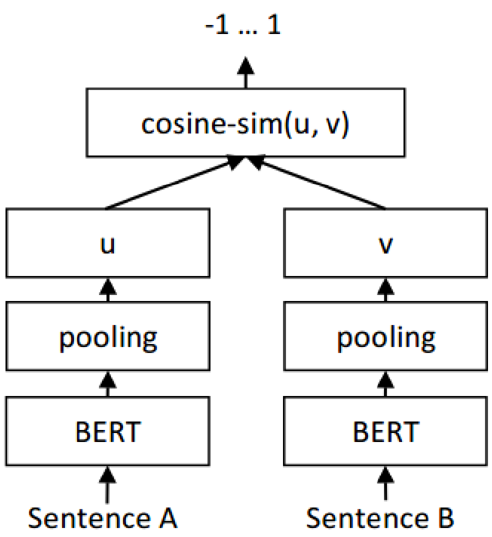
Variations of BERT (1)

- BERT's architecture was highly influential
  - Spawned several variants of the original model
- **RoBERTa** (Liu et al. 2019)
  - More than a variant, the result of a **reproduction study**
  - They **reimplemented**, **retrained**, did more **ablation**
  - **Proposed changes** such as:
    - (1) **Dynamic masking**, i.e. sampling tokens for masking for each input sequence during training (as opposed to once before entire training run)
    - (2) **Training without NSP** loss (no more sentence pairs)
- **SpanBERT** (Joshi et al. 2020)
  - **Masked entire subsequences** instead of individual tokens (i.e. a span)
  - Marked masked span with special tokens (start-of-span, end-of-span)
  - Model trained on **additional objective** Span Boundary Objective (SBO)
  - They outperformed BERT in span-based tasks, e.g. span-based QA
  - This model was concurrent with RoBERTa

Variations of BERT (2)

- Sentence-BERT ([Reimers et al. 2019]): better **sentence embeddings**
- Siamese network architecture (here, two BERTs with shared weights)
  - **Pooling** BERT output vectors (MEAN/MAX) to get sentence embedding
  - **Training** (left): Feed sentence vectors to classification/regression head
  - **Inference** (right): for regression, cosine similarity between sentences





- Results: better sentence embeddings vs [CLS] token, other methods

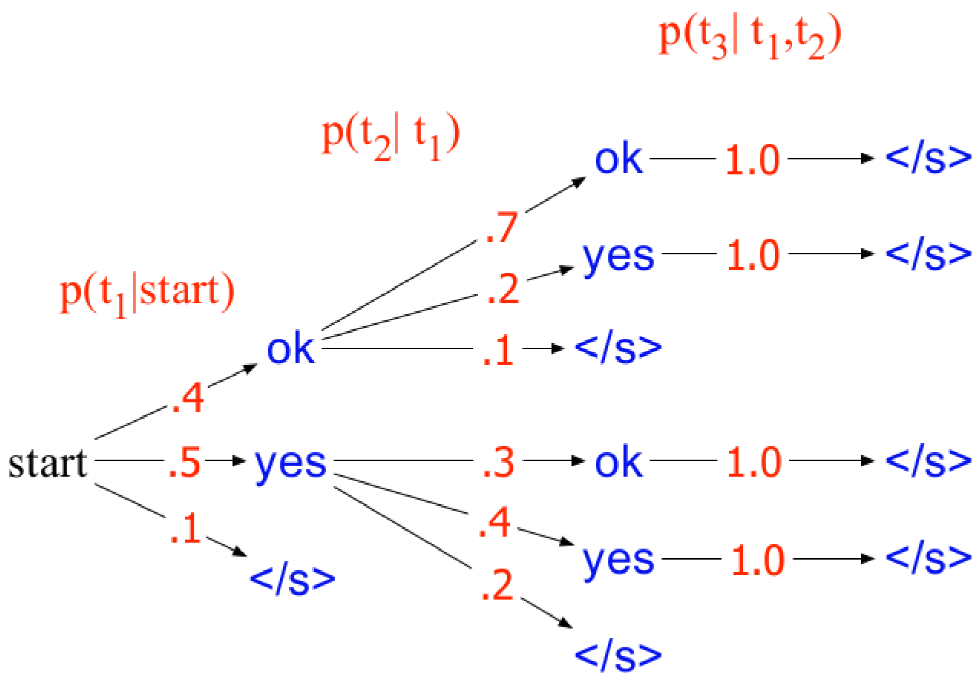
Causal Language Models

Autoregressive Text Generation

- **Recall autoregressive generation:** word generated at each time-step is conditioned on word generated at previous time-step
- **Concrete steps:**
  1. Feed starting symbol <*s*>, model produces softmax distribution over vocabulary
  2. Sample next word w_i from softmax distribution (using some sampling method)
  3. Add sampled word to previous input sequence, i.e. "<*s*> w_i", feed to model
  4. Model produces new softmax distribution, sample next word w_{i+1} again
  5. Repeat steps 3-4 with latest sampled word until symbol </*s*> is sampled
- **Sampling methods important part of generation**, as seen in tutorials
  - **Greedy decoding:** sample most probable word
  - **Top-k:** sample from top *k* words
  - **Top-p:** sample from words that make up top *p* probability mass
- More involved sampling methods exist
  - Before we look at causal language models, let's have a quick look at a very common and less simple sampling method in more detail

Beam Search (1)

- Say we have a pre-trained LM that can produce distributions for next word prediction
- Assume further our vocabulary is *V = {yes, ok, </s>}*
- In the following example, the **most probable sequence** is *"ok ok </s>"*
  - But **greedy sampling would fail** to predict it, it would first choose "*yes*"



Beam Search (2)

- The previous example illustrates that when we want the **most probable sequence**, shortsighted approaches may not be suitable
  - Shortsighted: only considering the probability of the next word
- **Beam Search:**
  - **Iteratively sample top *k*** most probable words (i.e. also greedy)
  - The **score of each possible word** is based on the probabilities of the previously sampled words, i.e. **consider the probability of the sequence**
- Score of sequence of y_i words given some input x, e.g. in translation, is:

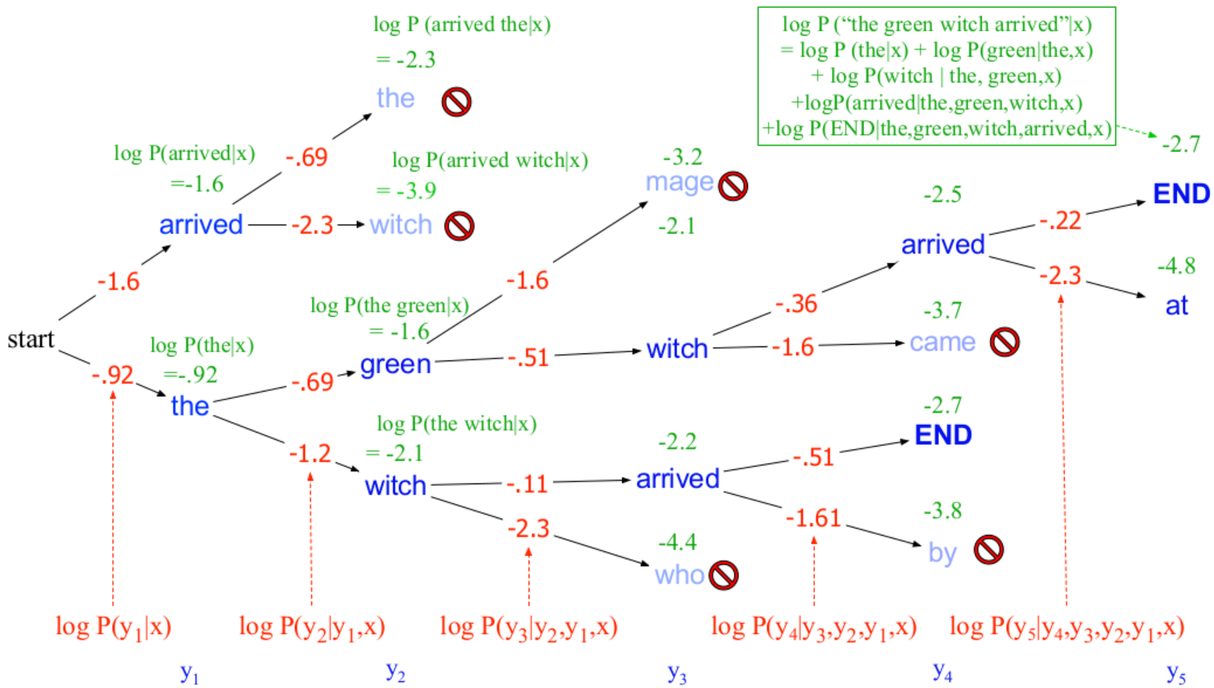
$$
\begin{aligned}
score(y) &= \log P(y|x) \\
&= \log\left(P(y_1|x)P(y_2|y_1,x)P(y_3|y_1,y_2,x)\ldots P(y_t|y_1,\ldots,y_{t-1},x)\right) \\
&= \sum_{i=1}^{t} \log P(y_i|y_1,\ldots,y_{i-1},x)
\end{aligned}
$$

- We normalize by sequence length to avoid favoring shorter sequences:

$$
score(y) = \log P(y|x) = \frac{1}{t}\sum_{i=1}^{t} \log P(y_i|y_1,\ldots,y_{i-1},x)
$$

Beam Search (3)

- At each time step:
  1. Select top *k* probably words from distribution over vocabulary
  2. Predict next word with each of the *k* top words, i.e. *k x V* probabilities
  3. Each next word is scored by its probability times the probability of its path
  4. We pick the top *k* most probable paths, and continue the process

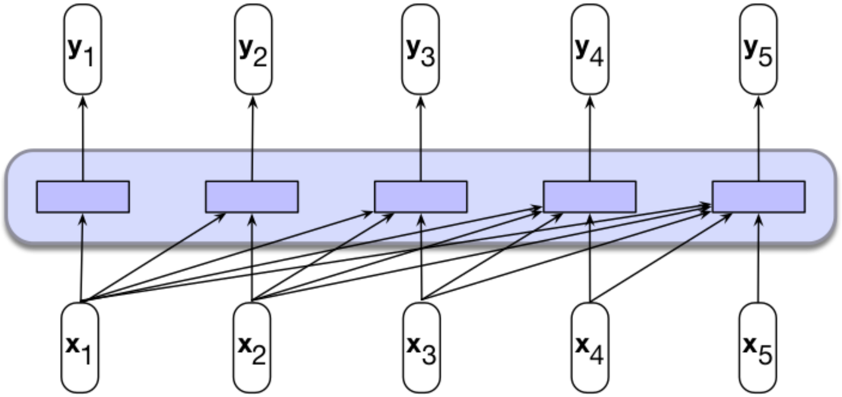


Beam Search (4)

- **Beam search is widely used** in text-generation applications:
  - E.g. machine translation, summarization
  - Generally applications where we compare different output sequences
- However, it has **shortcomings**
  - Cost intensive
  - Size of beam, i.e. value of *k*, still greedy, can miss better solutions
  - Sampled sequences often similar to one another (Li and Jurafsky, 2016)
- Variants have been proposed, e.g. diverse beam search
  - Divides sampled sequences into different sampling groups
  - Promotes diversity between sequences in different sampling groups
- Still, **beam search remains a very common sampling method**

Causal Language Models

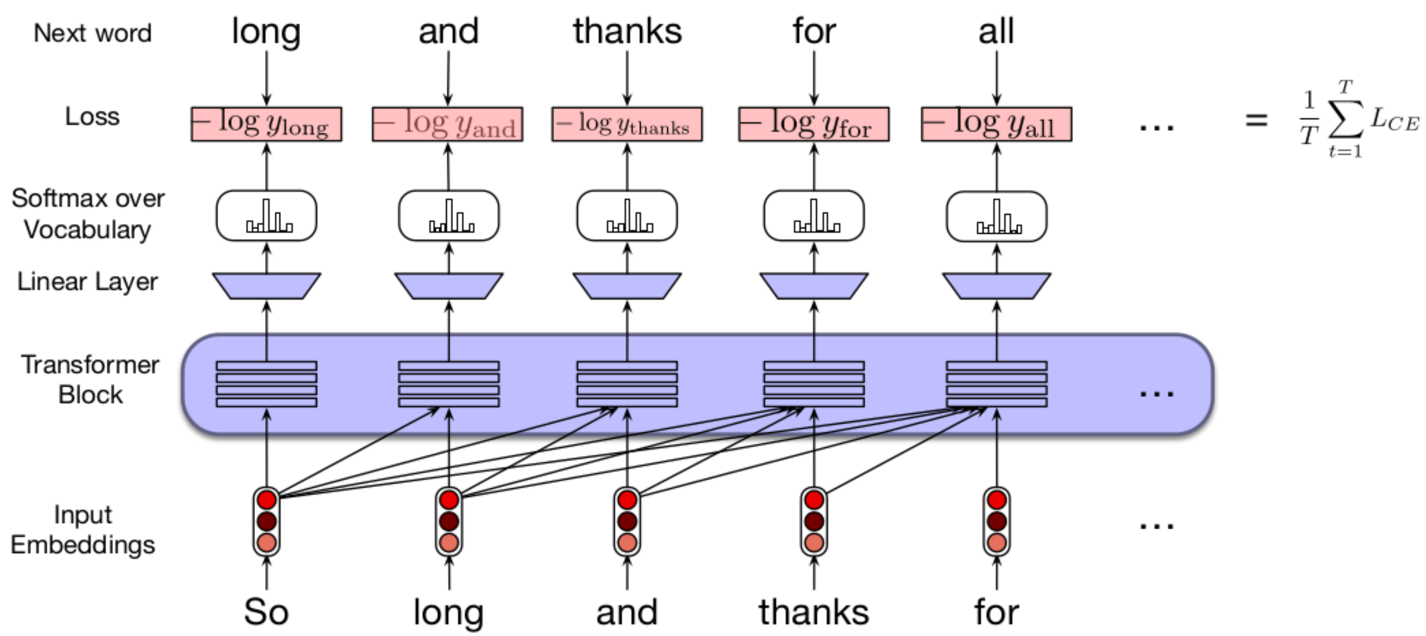
- **Causal language modeling:** given sequence of words, predict next word
  - **Context window/prompt:** given input sequence words
  - **Architecture:** we use causal self-attention in transformers (only look back)



- Text autoregressively generated with CLMs generally based on:
  - **Model predictions**, i.e. softmax over *V*
  - **Size of context window** (quadratic cost in self-attention using transformers)
  - **Sampling strategy/heuristic** (beam search a common one)
- Let's see how to train a CLM using transformers.
  - Then we'll go over some seminal transformer-based CLMs

Training Causal Language Models

- **Teacher forcing** does apply here:
  - Construct training sequences from given corpus in self-supervised manner
  - Given sequence, *force* each subsequence as input, next word as target
  - Compute loss between model's prediction and target word
  - Average loss over all predictions in given sequence
  - **Note:** computing loss for each word is independent, parallelizable



Generative Pre-Trained Transformer

- That is what GPT stands for (Radford et al. 2018)
- Published at time when pre-training & fine-tuning framework was new
  - Unclear what tasks to pre-train on
  - Unclear how to use learned representations in downstream models
  - Some of this still unclear, but pragmatic success with some architectures, e.g. BERT, GPT
- GPT: **autoregressive language modeling** is a **useful pre-training task**
- **Goal:** "…learn universal representations that transfer with little adaptation to a wide range of tasks."
  - So, transfer learning, important to avoid task-specific customization
- Two-step process:
  - **Pre-train** on large corpus in self-supervised manner
  - **Fine-tune** model on downstream tasks in supervised manner
  - Supervised downstream tasks need not be related to pre-training corpus (again, transfer learning)

GPT Architecture, Training, Fine-Tuning

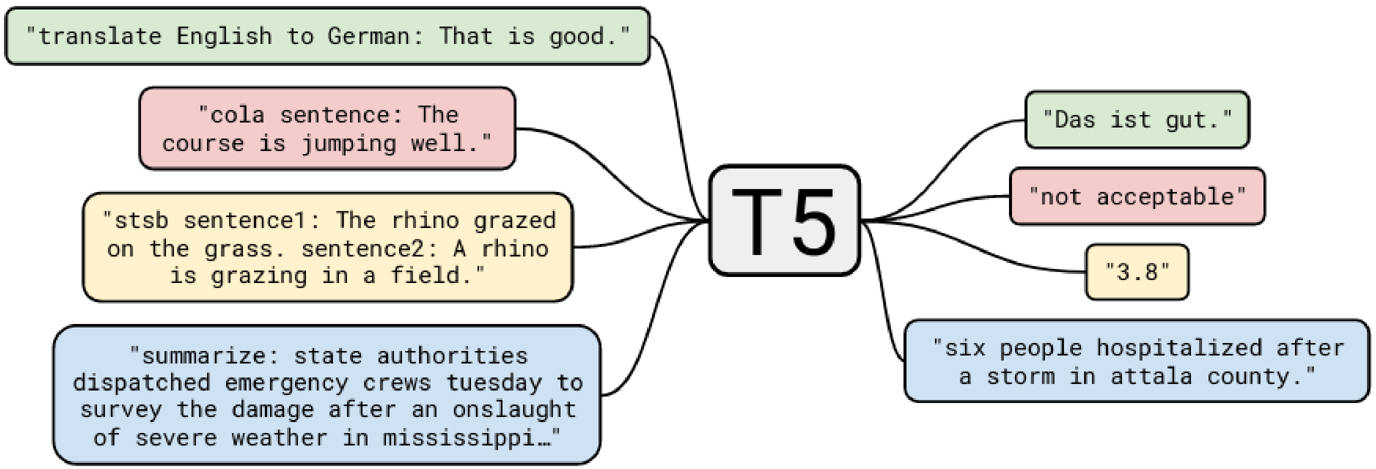
- Their **general architecture** (**117M parameters**):
  - **Decoder-only** transformer (i.e. no self-attention layer attending to encoder)
  - 12 layers of transformer decoders, each with 12 attention heads
  - Hidden layers of size 728, position-independent MLP of size 3072
  - Max input length: 512 tokens (attention cost quadratic in this length)
- So, architecture **quite similar to BERT**, except decoder-only
  - In fact, BERT heavily borrowed from GPT, as claimed by the BERT authors
- **Pre-Training**
  - **Self-supervised** causal language modeling on **BooksCorpus**
  - **Tokenization:** byte pair encoding (vocabulary size: 40K)
- **Fine-tuning**
  - Task inputs fed to pre-trained model to produce final representation $\boldsymbol{h}^{L}_{N}$, i.e. representation of token N in layer L, $\boldsymbol{h}^{L}_{N}$ then fed to classification head
  - They fine-tuned *all parameters* using the classification objective + the CLM objective

GPT Variants

- **GPT-2** (Radford et al. 2019): mostly same architecture as GPT-1
  - Moved around/added more layer normalization (more in LLMs lecture)
  - Initialized weights passed by residual layers (scaled them down)
  - Vocabulary expanded to 50K
  - Context window expanded from 512 to 1024 (again, quadratic cost)
  - Largest variant: **1.5B parameters**
  - **Their language model was able to do** NLP tasks such as **QA**, **machine translation** and **summarization** without training/tuning for it (**zero-shot**)
- **GPT-3** (Brown et al. 2020): same as GPT-2
  - Used sparse attention, cost from $O(n^2)$ to $O(n\sqrt{n})$ where n is input length
  - **Corpus:** Common Crawl dataset (1 trillion words)
  - Largest variant: **175B parameters**
  - Achieved **great performance in few-shot tasks described in prompts**
- We'll discuss these **zero-shot/few-shot** settings in future **LLMs lecture**.

Other Transformer-Based Models

- **T5** (Raffel et al. 2020): explored different training objectives, datasets
  - **Encoder-decoder** architecture, so cross attention from decoder to encoder!
  - They cast all tasks as text-to-text for this purpose, e.g. *stsb* is sentence similarity, model trained to generate one of possible classes



- **Electra** (Clark et al. 2020): **different task**, predict whether word was replaced randomly or not, so binary classification, not masking
  - More efficient training, competitive with larger models on some datasets

Which Architecture is Best?

- **No golden rule**
  - Encoder-decoder better?
  - Encoder-only?
  - Decoder-only?
- What about all **other design decisions?**
  - Which positional encodings?
  - Which tokenization approach?
- **Generally unclear**
  - Pragmatic evidence exists, e.g. studies like RoBERTa, T5
  - But results change as training/models/data change or scale up

The BERT Legacy

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

- **Large language models (LLMs)** currently hugely successful
  - They are almost exclusively transformer-based **decoder-only** models
  - I.e. they generate tokens one step at a time
- However, **for many applications, encoder-only models are arguably more suitable**
  - E.g. classification tasks where input is token/sequence representation, retrieval where input is query/document representation
  - Evidence of this: BERT-like models are some of the most popular models in HuggingFace Hub (vast repository of pre-trained models)
- Recently, BERT has received a "modern touch"
  - Mostly an update after all the lessons from pre-training decoder-only LLMs
- ModernBERT (2024): introduces improvements mostly for speed
  - E.g. different positional embeddings, alternating attention (not all layers attend to entire input sequence)
- NeoBERT (2025): concurrent with ModernBERT, similar improvements

Summary: Pre-Training

- **Transfer learning:**
  1. **Pre-training:** train model on some task(s) that "allows" model to learn rich representations, usually from large amounts of text. Representations then useful in downstream applications
  2. **Fine-tuning:** update learned representations (weights) in context of new downstream application, usually using new downstream model/component, e.g. a classification head
- **Pre-training** now **mostly done with transformer-based models**
- Different kinds of **pre-training objectives**, two dominant ones
  - Masked language modeling (**MLM**)
  - Causal language modeling (**CLM**)
- Different kinds of **architectures**
  - **Encoder-only**, typically used for training MLMs
  - **Decoder-only**, typically used for training CLMs
  - **Encoder-decoder**, cross-attention (decoder also attends to encoder outputs)

Fine-Tuning

What is Fine-Tuning?

- **Traditionally, updating weights** of pre-trained model when training on **new** downstream task
  - Resulting **contextualized representations *tuned* to** the **new task**
  - **New** = not used/seen during pre-training
- Downstream task usually implies additional **downstream model**
  - Downstream model usually comes with its separate weights
  - These weights *must be trained* (e.g. **classification head** in image below)

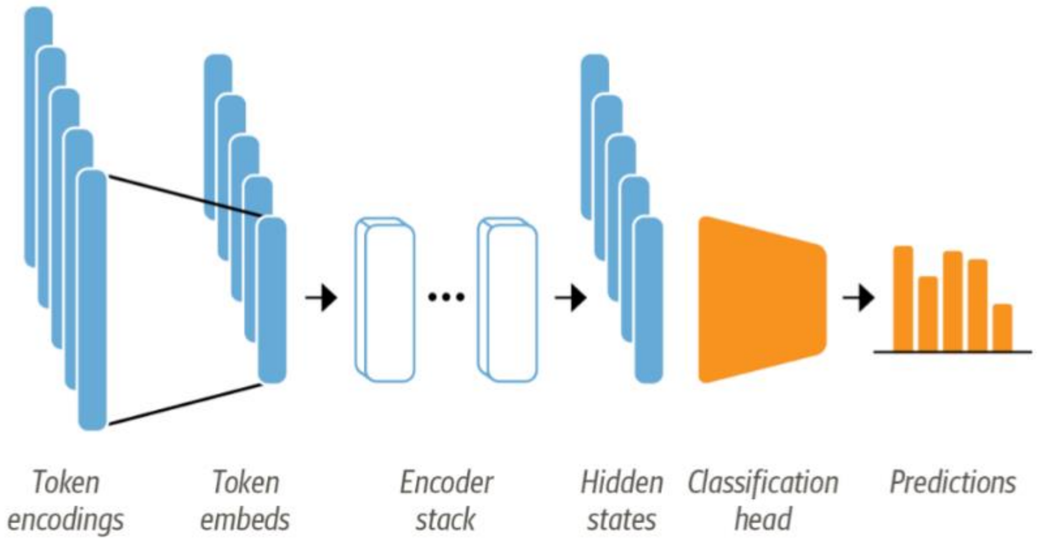
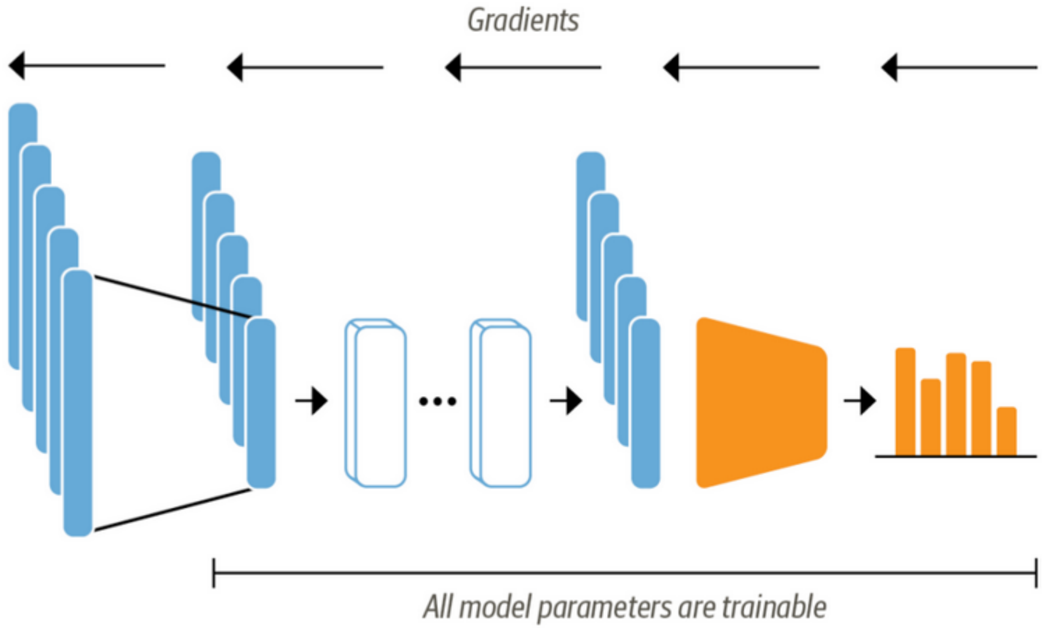


Image source

To Update or Not to Update (1)

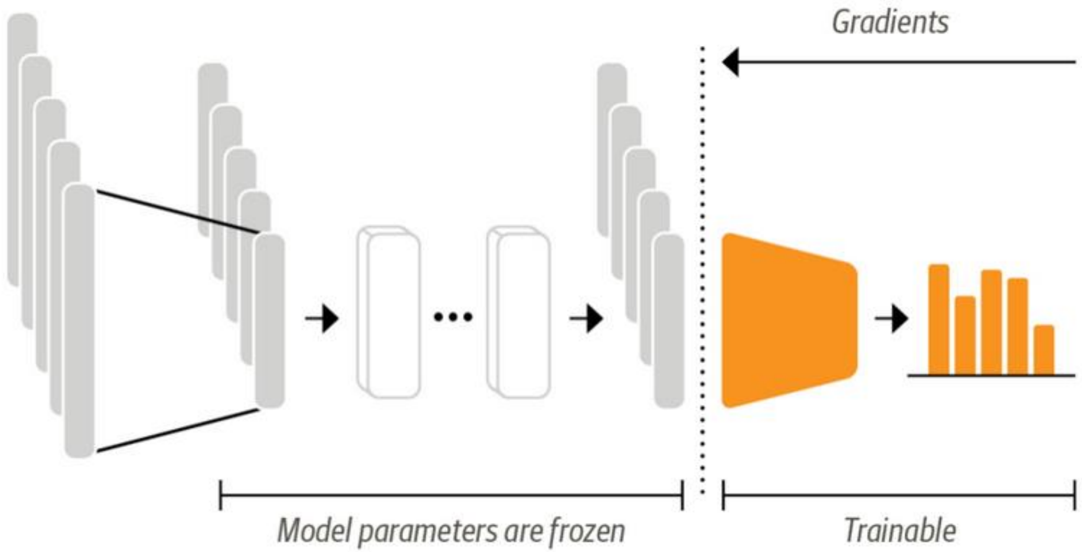
- Updating pre-trained weights **may result in “catastrophic forgetting”**
  - Encoded information during pre-training lost/replaced



- Pre-training can be very expensive
  - E.g. META AI's Llama3 CLM (400B parameters) trained with 16.000 GPUs
  - Catastrophic forgetting can potentially undo some/much of that effort
  - Understanding how specific knowledge is stored in weights is open question

To Update or Not to Update (2)

- **“Freezing”** (some or all) pre-trained weights means not updating them
  - Still referred to as fine-tuning, because we use a downstream model



- Definition unclear, so **relevant questions when discussing fine-tuning:**
  - Freezing or not freezing weights? Same or different training objective?
  - Which downstream model? How does it interact with pre-trained model?
  - How are downstream examples constructed/fed to model?
- This section: downstream tasks, basic/parameter efficient fine-tuning

Types of Downstream Tasks

- Two main types:
  - **Sequence classification:** predict label for entire input sequence
  - **Sequence labeling:** predict label for each input token
- **In each case, we need to**:
  1. Get general information encoded in pre-trained model
  2. Use this information with downstream model designed for downstream application

1. **How to get information encoded in pre-trained model?**
   - Depends on downstream task
   - Generally, we use some of the representations in the model, e.g. tokens in any (usually last) encoder layers
   - **No golden rules**, open question is what each layer encodes, usability for different tasks, e.g. multilingual transformers
2. **How to use pre-trained information with downstream models?**
   - Again, depends on application (let’s see some cases)

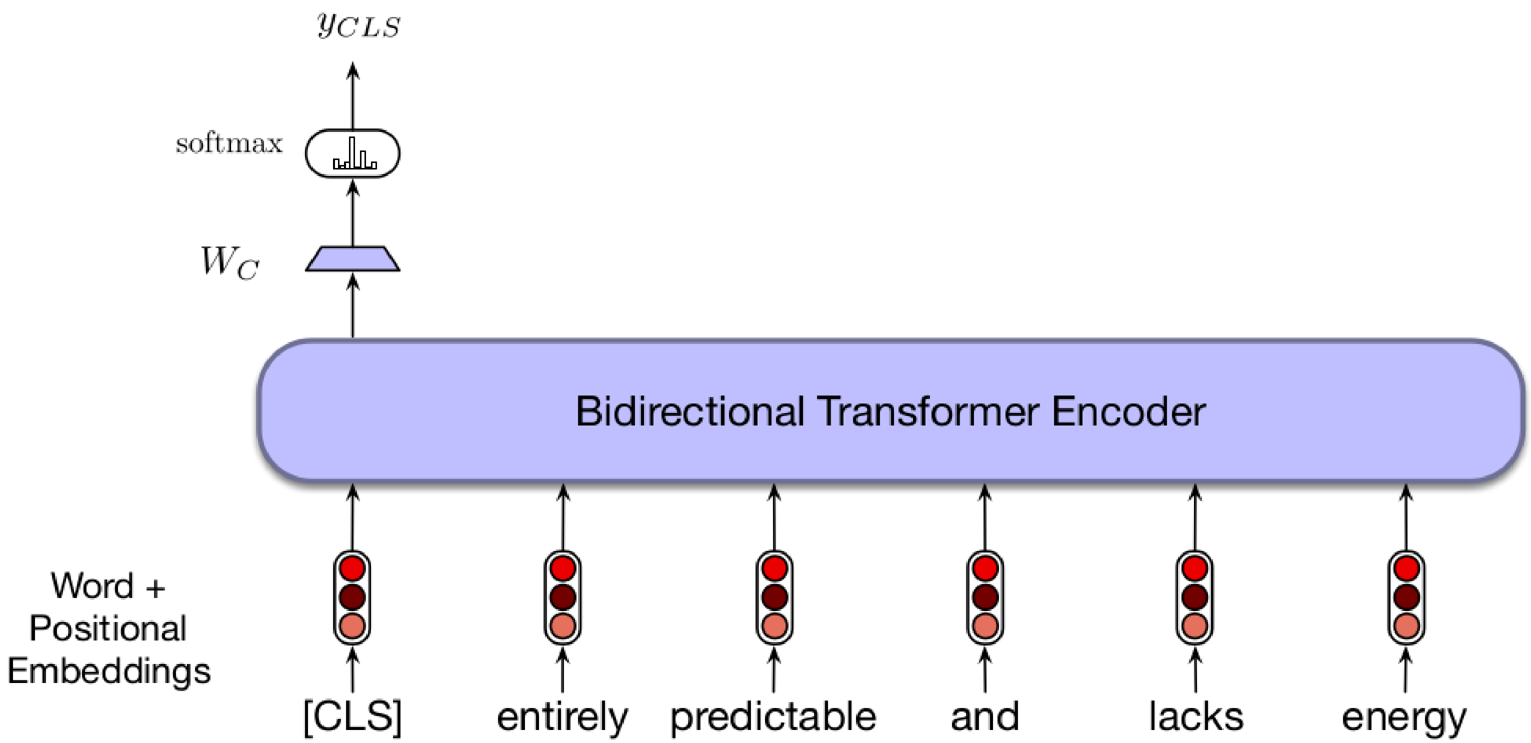
Sequence Classification (1)

- **Example:** Say we are interested in **sentiment classification**
  - **Three labels:** positive, negative, neutral
- **We have:**
  - A good mount of input sequences and corresponding sentiment labels
  - A pre-trained transformer LM
- **How can we use these resources to solve this task?**
  - In other words, how do we use this pre-trained model to solve this task?
  - Is it clear why we think this is a good idea in the first place?
- Usually: **we get single sentence representation from model**
  - Then feed this representation to a classifier that predicts 1 of three possible labels.
  - Downstream classifier usually referred to as **classification head**
- How can we get a sentence representation?
  - And what types of architecture can our classifier head have?

Sequence Classification (2)

- How to get a **sentence representation** from our **transformer**?
  - **[CLS] token** if available, e.g. in BERT
  - **Mean pooling** over (contextualized) output tokens (as in T5)
- What types of **architectures** can we use **for** our **classifier head**?
  - No general restrictions
  - Simple/common approach? **Softmax layer** (linear layer + softmax function)
- In the case of our sentiment classification task
  - Let $\boldsymbol{y} \in \boldsymbol{R}^d$ be the sentence embedding we use, e.g. CLS token
  - Then, our classifier is $\boldsymbol{y}_{CLS} = Softmax(\boldsymbol{W}_C\, \boldsymbol{y})$, parameterized by $\boldsymbol{W}_C \in \boldsymbol{R}^{3xd}$
- But the classifier can be move involved
  - E.g. fully-connected FNN with as many layers/architecture you want
  - As usual, the more parameters, the more data required for training
- Let's visualize this!

Sequence Classification (3)



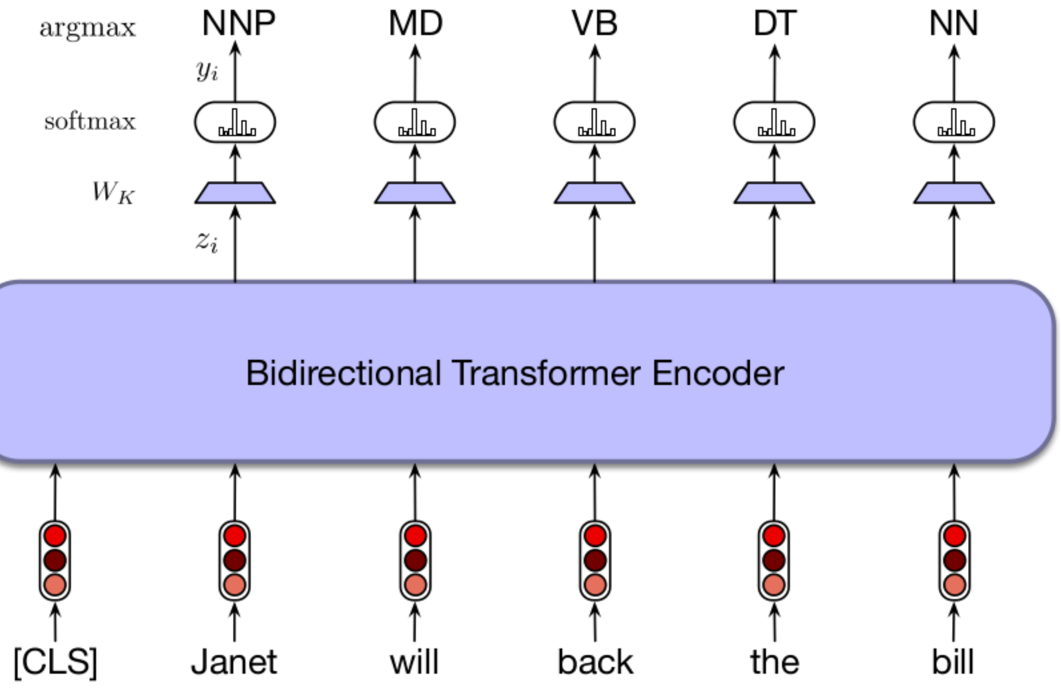
- We assume an architecture where [CLS] token is available
  - Recall: **we need labeled data** for our task to train $\boldsymbol{W_C}$ (not pre-trained), i.e. no more self-supervision, but loss is still typically cross entropy

Pair-Wise Sequence Classification

- **Some problems** involve **classifying two sentences**
  - Logical entailment: does one sentence entail the other?
  - Paraphrase detection: is one sentence paraphrasing the other?
- Common such task: **natural language inference (NLI)**
  - Given two sentences, identify relation between the two
  - E.g. entails, contradicts, neutral
- How can we **fine-tune** a **pre-trained transformer for NLI?**
  - Feed single representation for entire input sequence to classifier head
  - Input sequence includes pair of sentences, [SEP] token
  - Any other way?
- Again, **no golden rules**!
  - E.g. feed each sentence separately, concatenate representations for classifier input
  - Different approaches may have different impact on pre-trained weights during fine-tuning

Sequence Labeling

- **Example:** Say we are interested in **part-of-speech (POS) tagging**
  - Each input token gets **one of *k* labels**, e.g. verb, noun, modifier, etc.
- **How do we fine-tune a classifier for this approach?**
  - Classifier head fed each output token for prediction, i.e. $\boldsymbol{y_i = Softmax(W_K z_i)}$ where $\boldsymbol{z_i}$ is contextualized representation of token *i*



How Feasible is Fine-Tuning?

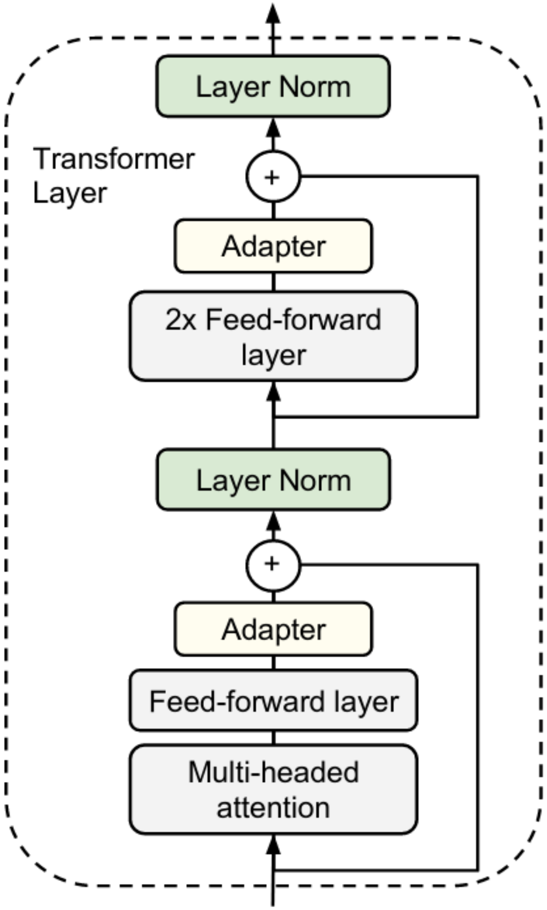
- **Pre-training** a (large) language model is **expensive**
  - BERT (2018) had 100M parameters, LLMs today in the billions/trillions
- **Is fine-tuning LMs less costly than pre-training?**
  - Not necessarily, still tuning the entire pre-trained model
  - E.g. fine-tuning with same objective -> continued pre-training
  - *Store copy of tuned model for each task you tune it for!*
  - But, usually requires less time/data due to pre-trained "bootstrap"
- **Do you need to fine-tune entire pre-trained LM?**
  - Perhaps just train a few layers, perhaps the last layers only?
  - Recall concept of hierarchical representations
  - We can choose what to freeze!
- That is the basic idea behind **parameter efficient fine-tuning (PEFT)**
  - This name used in NLP community, relatively new
  - Similar approaches already used in Computer Vision before that

Parameter Efficient Fine-Tuning

- What if you have **small amount of data**?
  - Or **not enough compute power** to continue training a large transformer
  - Lots of weights -> lots of data for training, lots of compute power
- **PEFT:** fine-tune small subset of pre-trained weights
- **Advantages**
  - Requires less data, compute power
  - Less prone to overfitting to whatever data you have
- **Disadvantages**
  - Usually worse performance than standard fine-tuning
- Different approaches suggested in recent years
  - Let’s go over some of the most popular ones

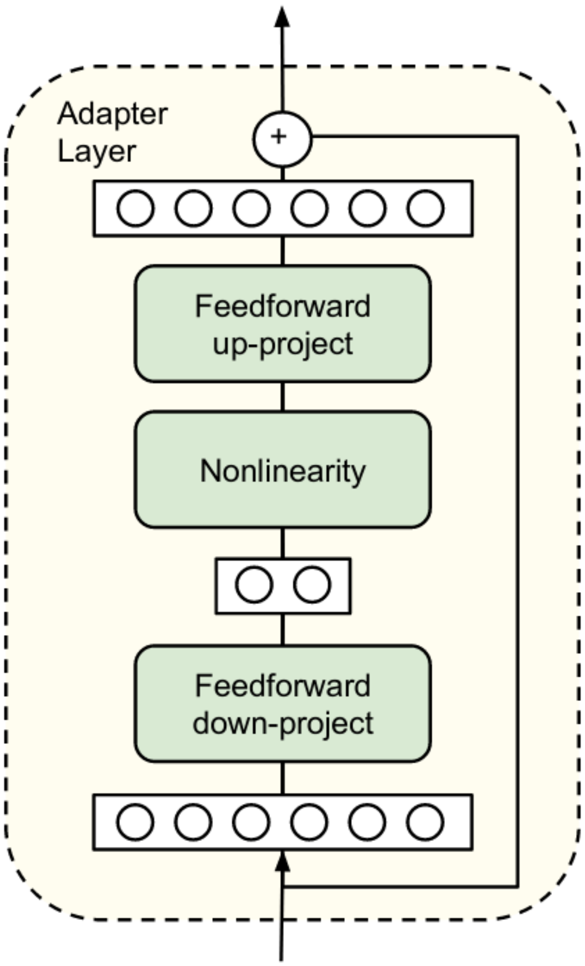
Adapters (1)

- Used by Rebuffi et al. for computer vision tasks in 2017
- Houlsby et al. (2019) proposed it in NLP, applied it to transformers
- **Main idea:** insert adapter modules in transformer block, tune adapter weights only during fine-tuning
- **Adapter layer:** tunable layer added to transformer block, originally two, one after self-attention layer and another after projection layer



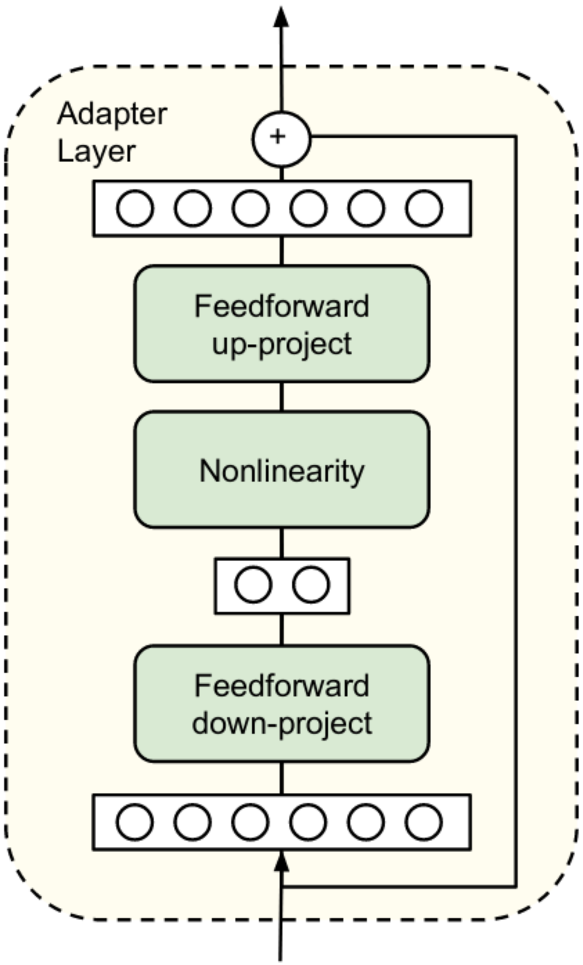
Adapters (2)

- **Adapter layer should not have many parameters**
  - Thus, easier/less expensive to tune
  - Originally, between 0.5 and 8% of pre-trained model
- **Components in adapter layer**
  - Projection layer down to *bottleneck* dimension *m*
  - Non-linearity
  - Projection up to original input size *d*
- **# parameters:** $2md + m + d$
  - Two projections + output biases
- **Bottleneck size:** controls trade-off between model performance and parameter efficiency



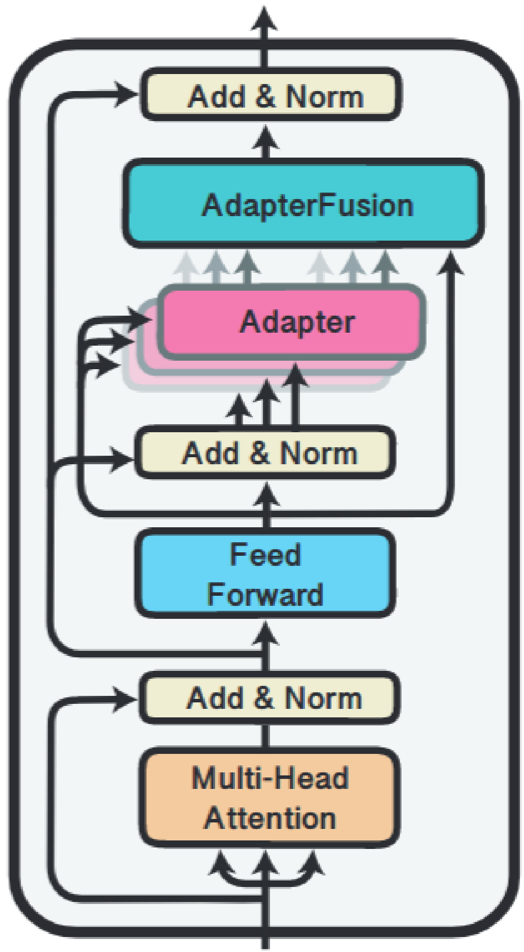
Adapters (3)

- **Important:** initialization should result in (close to) identity function
  - Why?
- Training needs to ***continue*** in fine-tuning process
  - We start with a working forward pass!
  - Random *init* would break that function!
- **Residual connection inside adapter layer serves this purpose**
  - Initialize adapter weights to zero
  - Then you get identity
- **Recall residual connections** add input back to output
  - Let $y = f(x)$ be the operator.
  - With residual connection, $y = f(x) + x$
  - Thus operator is identity if $f(x) = 0$



Adapters (4)

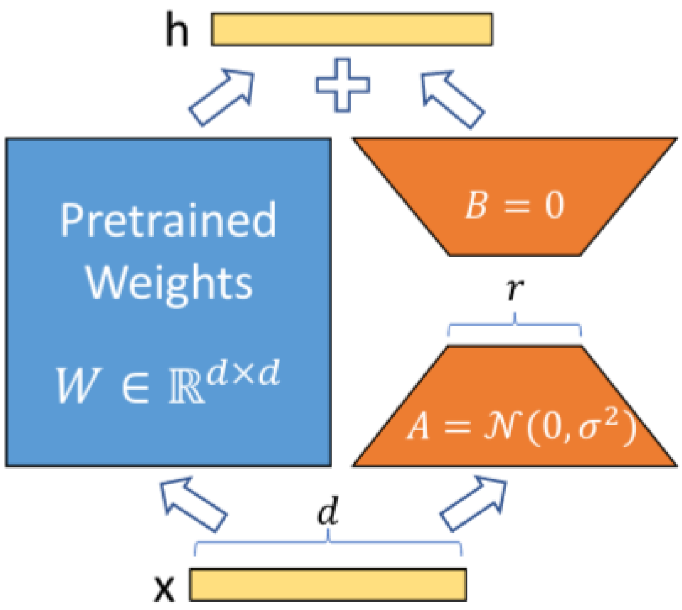
- Other architectures possible
- Bapna et al. (2019) use only single adapter layer after projection layer
- Pfeifer et al. (2021) proposed to **combine several task-specific adapters**
  - Task-specific adapters combined with **attention mechanism**
  - Allows classifier to use **information from several different tasks in non-destructive manner**
  - No catastrophic forgetting compared to fine-tuning same adapter on new task



Low-Rank Adaptation (1)

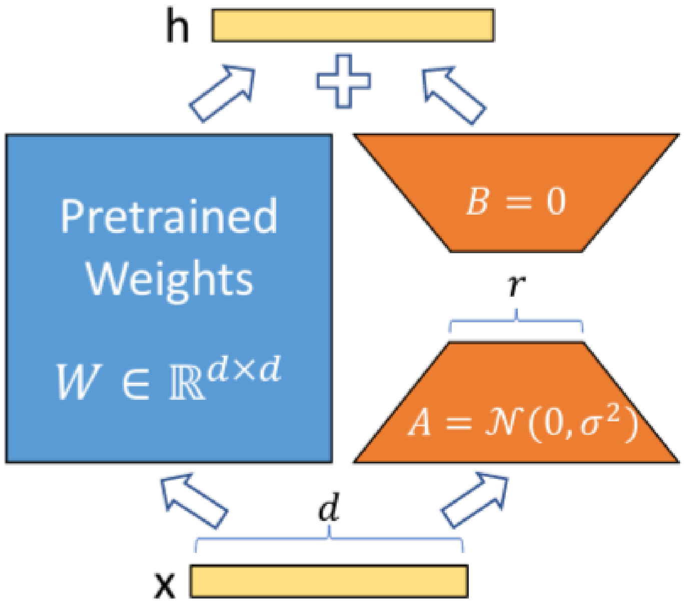
- **LoRA:** Low-rank adaptation ([Hu et al. 2021](#))
- Similar to adapters, adds trainable modules to transformer block
- Specifically, **two projection layers parallel** to projection layer in transformer
- Perspective: updated weight $\boldsymbol{W'}$ during fine-tuning is $\boldsymbol{W'} = \boldsymbol{W_O} + \Delta \boldsymbol{W_O}$
  - Simulate this effect by freezing $\boldsymbol{W_O}$, adding parallel projections $\boldsymbol{AB}$
  - Thus, $\boldsymbol{W'} = \boldsymbol{W_O} + \boldsymbol{AB}$
- ***A*** projects down to small size

  Thus, ***AB*** is low-rank factorization of $\Delta \boldsymbol{W_O}$
- As with adapters, initialization must be identity
  - They achieve this by setting **B** = **0**, so that $\boldsymbol{AB} = \boldsymbol{0}$



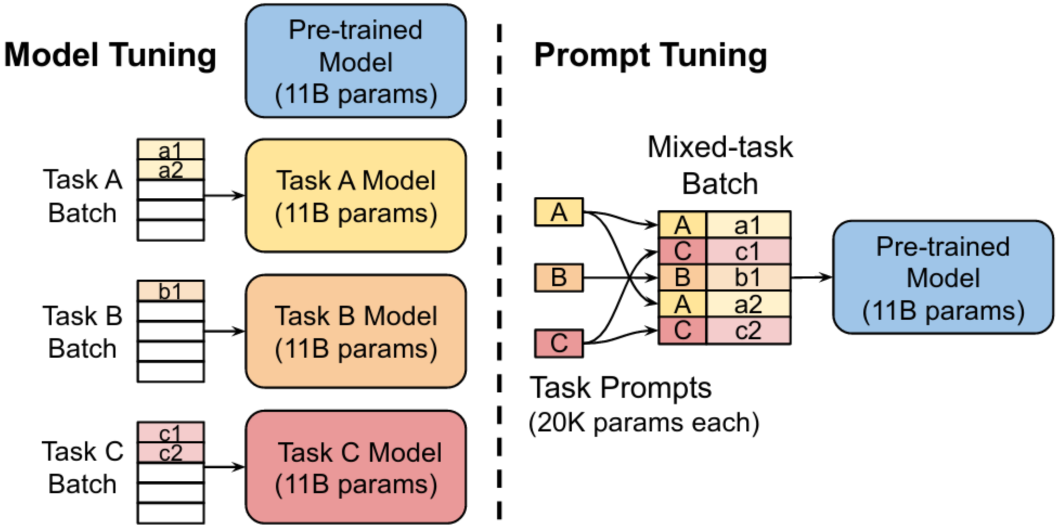
Low-Rank Adaptation (2)

- Downside to *all adapters* in general?
  - (Slightly) higher inference costs
  - We add more parameters!
- LoRA designed to prevent this
  - Once adapter is learned, original $\boldsymbol{W_O}$ can be updated by *merging back* **AB**
  - That is, final weight $\boldsymbol{W} = \boldsymbol{W_O} + \boldsymbol{AB}$
  - Thus, **no additional inference costs**
  - But catastrophic forgetting possible
- Generally, **adapters have trade-offs**
  - **CON:** increased inference cost
  - **PRO:** prevent catastrophic forgetting, original model frozen, adapters can always be removed (gives rise to modular deep learning)



Prompt Tuning (1)

- PEFT method, different from adapters, proposed by Lester el al. (2021)
- **GPT-3:** performs well on new tasks using **in-context learning**
  - In-context learning: describe task and provide few examples in prompt
  - **Advantage?** No fine-tuning! Same frozen model used for different tasks!
  - **Problem?** Approach sensitive to prompt, gives rise to *prompt design*
- **Main idea:** prepend task-specific tokens to input sequences (prompt)
  - Then, during fine-tuning, only new task-specific tokens are updated
  - Each task A, B, C gets new task token **A**, **B**, **C**
  - Prepend task examples with task tokens, e.g. prepend **A** to a1, a2
  - Tune only task tokens
  - At inference, use new tokens in same way



Prompt Tuning (2)

- As with T5 model, they cast tasks as text-to-text
  - **Normally:** downstream classification is $p(y|\boldsymbol{X})$ where $\boldsymbol{X}$ is input sequence (prompt) and y is single class label, e.g. *0* or *1* if binary
  - **Text-to-text:** we have $p(\boldsymbol{Y}|\boldsymbol{X})$, where $\boldsymbol{Y}$ is generated output sequence that represent class label, e.g. "neutral" for sentiment analysis
  - **Fine-tuning:** $p_{\boldsymbol{\Theta}}(\boldsymbol{Y}|\boldsymbol{X})$ where $\boldsymbol{\Theta}$ is pre-trained model parameters we tune
- **Prompting:** prepend tokens $\boldsymbol{P}$ to input sequence
  - Normally, $\boldsymbol{P}$ made up of known tokens from pre-trained embedding layer
  - E.g. $\boldsymbol{P}$ = "Summarize the following article into a single sentence"
  - Performance sensitive to chosen prompt, may require prompt design
- **Prompt tuning:** add fixed prompt **P** per task using special new tokens
  - $\boldsymbol{\Theta}_P$ = *n*ew tokens added to vocabulary
  - Then: $p_{\Theta;\Theta P}(\boldsymbol{Y}|[\boldsymbol{P},\boldsymbol{X}])$ but we only tune $\boldsymbol{\Theta}_P$
  - Fixed prompt removes need for prompt design
- Similar approaches: prefix tuning by [Li et al. (2021)](), P-Tuning by [Liu et al. (2022)]()

Another Perspective on Tasks

- We can classify downstream tasks as follows:
  - Sequence classification
  - Sequence labeling
  - Etc.
- We can also **classify tasks based on available data, usually:**
  - **Few shot:** few examples available
  - **Zero shot:** no examples available (just text description)
- These definitions are not so consistent in the literature
  - More details in future LLM lecture

Summary: Fine-Tuning

- Difficult to define
  - Generally, using pre-trained model PT on new task not seen during training
  - May or may not imply updating pre-trained model parameters $\boldsymbol{\Theta}_{PT}$
  - Usually includes use of downstream model DM, e.g. classification head
  - Parameters of downstream model $\boldsymbol{\Theta}_{DM}$ are definitely trained
- **Relevant questions when discussing fine-tuning:**
  - Freezing or not freezing weights? Same or different training objective?
  - Which downstream model? How does it interact with pre-trained model?
  - How are downstream examples constructed/fed to model?
- **Parameter Efficient Fine-Tuning** (PEFT)
  - Updating all model parameters $\boldsymbol{\Theta}_{PT}$ can be prohibitely expensive
  - PEFT: tune only subset of parameters of $\boldsymbol{\Theta}_{PT}$, or new added parameters $\boldsymbol{\Theta}_{FT}$
- Most common PEFT approaches: **adapters, LoRA**

References

- Speech and Language Processing, Jurafsky et al., 2024
  - Chapters 10 and 11
- Natural Language Processing: A Machine Learning Perspective, Zhang et al., 2021
  - Chapter 17
- References linked in corresponding slides