Modeling Sequences/Sets: Transformers

Prof. Adriana Kovashka
University of Pittsburgh
March 25, 2021
Plan for this lecture

• Background
  – Context prediction, unsupervised learning

• Transformer models
  – Self-attention
  – Adapting self-attention for sequential data
  – The transformer architecture, encoder/decoder
  – Pre-training, BERT

• Transformers beyond language
Additional resources

• Learning about transformers on your own?
  • Key recommended resource:
    • http://nlp.seas.harvard.edu/2018/04/03/attention.html
    • The Annotated Transformer by Sasha Rush
    • Jupyter Notebook using PyTorch that explains everything!
  • The Illustrated Transformer
    • http://jalammar.github.io/illustrated-transformer/
  • Attention visualizer
    • https://github.com/jessevig/bertviz

Adapted from Christopher Manning
How do we represent the meaning of a word?

Definition: meaning (Webster dictionary)

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.
- the idea that is expressed in a work of writing, art, etc.

Commonest linguistic way of thinking of meaning:

\[ \text{signifier (symbol)} \leftrightarrow \text{signified (idea or thing)} \]

= denotational semantics
How do we have usable meaning in a computer?

Common solution: Use e.g. WordNet, a thesaurus containing lists of synonym sets and hypernyms (“is a” relationships).

e.g. synonym sets containing “good”:

```python
from nltk.corpus import wordnet as wn
poses = { 'n': 'noun', 'v': 'verb', 's': 'adj (s)', 'a': 'adj', 'r': 'adv'}
for synset in wn.synsets("good"):
    print("{}": {}.format(poses[synset.pos()],
        ", ".join([l.name() for l in synset.lemmas()])))
```

noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good adj:
good
adj (sat): estimable, good, honorable, respectable adj (sat):
beneficial, good
adj (sat): good
adj (sat): good, just, upright
...
adverb: well, good
adverb: thoroughly, soundly, good

e.g. hypernyms of “panda”:

```python
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(panda.closure(hyper))

[Synset('procyonid.n.01'),
 Synset('carnivore.n.01'),
 Synset('placental.n.01'),
 Synset('mammal.n.01'),
 Synset('vertebrate.n.01'),
 Synset('chordate.n.01'),
 Synset('animal.n.01'),
 Synset('organism.n.01'),
 Synset('living_thing.n.01'),
 Synset('whole.n.02'),
 Synset('object.n.01'),
 Synset('physical_entity.n.01'),
 Synset('entity.n.01')]
```
Problems with resources like WordNet

- Great as a resource but missing nuance
  - e.g. “proficient” is listed as a synonym for “good”. This is only correct in some contexts.

- Missing new meanings of words
  - e.g., wicked, badass, nifty, wizard, genius, ninja, bombest
  - Impossible to keep up-to-date!

- Subjective

- Requires human labor to create and adapt

- Can’t compute accurate word similarity
Representing words as discrete symbols

In traditional NLP, we regard words as discrete symbols: hotel, conference, motel - a localist representation

Words can be represented by one-hot vectors:

\[
\text{motel} = [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0]
\]
\[
\text{hotel} = [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]
\]

Vector dimension = number of words in vocab (e.g. 500,000)
Problem with words as discrete symbols

Example: in web search, if user searches for “Seattle motel”, we would like to match documents containing “Seattle hotel”.

But:

motel = [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
hotel = [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]

These two vectors are orthogonal.
There is no natural notion of similarity for one-hot vectors!

Solution:
• Could try to rely on WordNet’s list of synonyms to get similarity?
  • But it is well-known to fail badly: incompleteness, etc.
• Instead: learn to encode similarity in the vectors themselves
Representing words by their context

- **Distributional semantics**: A word’s meaning is given by the words that frequently appear close-by
  - “*You shall know a word by the company it keeps*” (J. R. Firth 1957)
  - One of the most successful ideas of modern statistical NLP!
- When a word \( w \) appears in a text, its **context** is the set of words that appear nearby (within a fixed-size window).
- Use the many contexts of \( w \) to build up a representation of \( w \)

...government debt problems turning into crises as happened in 2009...
...saying that Europe needs unified regulation to replace the hodgepodge...
...India has just given its banking system a shot in the arm...

These context words will represent *banking*
Stanford University is located in__________, California.
I put ___ fork down on the table.
The woman walked across the street, checking for traffic over ____shoulder.
I went to the ocean to see the fish, turtles, seals, and _____.

John Hewitt
Overall, the value I got from the two hours watching it was the sum total of the popcorn and the drink.

The movie was____.
Iroh went into the kitchen to make some tea.
Standing next to Iroh, Zuko pondered his destiny.
Zuko left the______.
What can we learn from reconstructing the input?

I was thinking about the sequence that goes
1, 1, 2, 3, 5, 8, 13, 21, ____
Word vectors

We will build a dense vector for each word, chosen so that it is similar to vectors of words that appear in similar contexts.

\[
\text{banking} = \begin{pmatrix}
0.286 \\
0.792 \\
-0.177 \\
-0.107 \\
0.109 \\
-0.542 \\
0.349 \\
0.271
\end{pmatrix}
\]

Note: word vectors are sometimes called word embeddings or word representations. They are a distributed representation.
Word meaning as a neural word vector - visualization

\[
\text{expect} = \begin{pmatrix}
0.286 \\
0.792 \\
-0.177 \\
-0.107 \\
0.109 \\
-0.542 \\
0.349 \\
0.271 \\
0.487
\end{pmatrix}
\]
Word2Vec Overview

Word2vec (Mikolov et al. 2013) is a framework for learning word vectors

Idea:

• We have a large corpus of text
• Every word in a fixed vocabulary is represented by a vector
• Go through each position $t$ in the text, which has a center word $c$ and context (“outside”) words $o$
• Use the similarity of the word vectors for $c$ and $o$ to calculate the probability of $o$ given $c$ (or vice versa)
• Keep adjusting the word vectors to maximize this probability
Word2Vec Overview

- Example windows and process for computing $P(w_{t+j} | w_t)$
Word2Vec Overview

- Example windows and process for computing $P(w_{t+j} \mid w_t)$
Word2Vec: objective function

For each position $t = 1, \ldots, T$, predict context words within a window of fixed size $m$, given center word $w_j$.

The objective function is the (average) negative log likelihood:

$$ J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log P(w_{t+j} \mid w_t; \theta) $$

Minimizing objective function $\iff$ Maximizing predictive accuracy
Word2Vec: objective function

- We want to minimize the objective function:

\[
J(\theta) = -\frac{1}{T} \log L(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log P(w_{t+j} | w_t; \theta)
\]

- **Question:** How to calculate \( P(w_{t+j}|w_t; \theta) \)?

- **Answer:** We will use two vectors per word \( w \):
  - \( v_w \) when \( w \) is a center word
  - \( u_w \) when \( w \) is a context word

- Then for a center word \( c \) and a context word \( o \):

\[
P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}
\]
**Word2Vec: prediction function**

The softmax function maps arbitrary values to a probability distribution $p_i$

- "max" because amplifies probability of largest $x_i$
- "soft" because still assigns some probability to smaller $x_i$
- Frequently used in Deep Learning

Exponentiation makes anything positive

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Dot product compares similarity of $o$ and $c$.

$$u^T v = u \cdot v = \sum_{i=1}^{n} u_i v_i$$

Larger dot product = larger probability

Normalize over entire vocabulary to give probability distribution

- This is an example of the **softmax function** $\mathbb{R}^n \rightarrow \mathbb{R}^n$

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i$$
ELMo: Embeddings from Language Models


- Breakout version of word token vectors or contextual word vectors
- Learn word token vectors using long contexts not context windows (here, whole sentence, could be longer)
- Learn a deep Bi-NLM and use all its layers in prediction
ELMo: Embeddings from Language Models

- Train a bidirectional LM
- Aim at performant but not overly large LM:
  - Use 2 biLSTM layers
  - Use character CNN to build initial word representation
  - Use 4096 dim hidden/cell LSTM states with 512 dim projections to next input
  - Use a residual connection
  - Tie parameters of token input and output (softmax) and tie these between forward and backward LMs

Adapted from Christopher Manning
ELMo used in a sequence tagger

ELMo results: Great for all tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>Previous SOTA</th>
<th>Our baseline</th>
<th>ELMo + baseline</th>
<th>Increase (absolute/relative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQuAD</td>
<td>Liu et al. (2017)</td>
<td>84.4</td>
<td>81.1</td>
<td>85.8</td>
</tr>
<tr>
<td>SNLI</td>
<td>Chen et al. (2017)</td>
<td>88.6</td>
<td>88.0</td>
<td>88.7 ± 0.17</td>
</tr>
<tr>
<td>SRL</td>
<td>He et al. (2017)</td>
<td>81.7</td>
<td>81.4</td>
<td>84.6</td>
</tr>
<tr>
<td>Coref</td>
<td>Lee et al. (2017)</td>
<td>67.2</td>
<td>67.2</td>
<td>70.4</td>
</tr>
<tr>
<td>NER</td>
<td>Peters et al. (2017)</td>
<td>91.93 ± 0.19</td>
<td>90.15</td>
<td>92.22 ± 0.10</td>
</tr>
<tr>
<td>SST-5</td>
<td>McCann et al. (2017)</td>
<td>53.7</td>
<td>51.4</td>
<td>54.7 ± 0.5</td>
</tr>
</tbody>
</table>

Christopher Manning
ELMo: Roles of different layers

- The two biLSTM NLM layers have differentiated uses/meanings
  - Lower layer is better for lower-level syntax, etc.
    - Part-of-speech tagging, syntactic dependencies, NER
  - Higher layer is better for higher-level semantics
    - Sentiment, Semantic role labeling, question answering, SNLI

Adapted from Christopher Manning
Issues with recurrent models: Linear interaction distance

• O(sequence length) steps for distant word pairs to interact means:
  • Hard to learn long-distance dependencies (because gradient problems!)
  • Linear order of words is “baked in”; not necessarily the right way to think about sentences...

The chef who ... was

Info of chef has gone through O(sequence length) many layers!

Adapted from John Hewitt
Issues with recurrent models: Lack of parallelizability

- Forward and backward passes have $O(\text{sequence length})$ unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can’t be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!

Numbers indicate min # of steps before a state can be computed
If not recurrence, then what? How about word windows?

- **Word window models aggregate local contexts**
  - Also known as 1D convolution
  - Number of unparallelizable operations not tied to sequence length!

Numbers indicate min # of steps before a state can be computed

Adapted from John Hewitt
If not recurrence, then what? How about word windows?

- **Word window models aggregate local contexts**
- What about long-distance dependencies?
  - Stacking word window layers allows interaction between farther words
  - But if your sequences are too long, you’ll just ignore long-distance context

---

Red states indicate those “visible” to $h_k$

Too far from $h_k$ to be considered

Adapted from John Hewitt
If not recurrence, then what? How about attention?

- **Attention** treats each word’s representation as a **query** to access and incorporate information from **a set of values**.
  - We saw attention from the **decoder** to the **encoder**; today we’ll think about attention **within a single sentence**.
  - If **attention** gives us access to any state... maybe we can just use attention and don’t need the RNN?
- Number of unparallelizable operations not tied to sequence length.
- All words interact at every layer!

![Diagram](image)

All words attend to all words in previous layer; most arrows here are omitted.

Adapted from John Hewitt
**Self-Attention**

- Attention operates on **queries**, **keys**, and **values**.
  - We have some **queries** $q_1, q_2, \ldots, q_T$. Each query is $q_i \in \mathbb{R}^d$
  - We have some **keys** $k_1, k_2, \ldots, k_T$. Each key is $k_i \in \mathbb{R}^d$
  - We have some **values** $v_1, v_2, \ldots, v_T$. Each value is $v_i \in \mathbb{R}^d$
- In **self-attention**, the queries, keys, and values are drawn from the same source.
  - For example, if the output of the previous layer is $x_1, \ldots, x_T$, (one vec per word) we could let $v_i = k_i = q_i = x_i$ (that is, use the same vectors for all of them!)
- The (dot product) self-attention operation is as follows:

$$
\begin{align*}
  e_{ij} &= q_i^T k_j \\
  \alpha_i &= \frac{\exp(e_{ij})}{\sum_j \exp(e_{ij})} \\
  \text{output } i &= \sum_j \alpha_{ij} v_j
\end{align*}
$$

- Compute **key-query** affinities
- Compute attention weights from affinities (softmax)
- Compute outputs as weighted sum of **values**

The number of queries can differ from the number of keys and values in practice.
Self-Attention

• In the diagram at the right, we have stacked self-attention blocks, like we might stack LSTM layers.

• Can self-attention be a drop-in replacement for recurrence?

• No. It has a few issues, which we’ll go through.

• First, self-attention is an operation on sets. It has no inherent notion of order.

Self-attention doesn’t know the order of its inputs.
Barriers and solutions for Self-Attention as a building block

**Barriers**
- Doesn’t have an inherent notion of order!

**Solutions**

John Hewitt
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
  \[ p_i \in \mathbb{R}^d, \text{ for } i \in \{1,2, ..., T\} \text{ are position vectors} \]

- Don’t worry about what the \( p_i \) are made of yet!
- Easy to incorporate this info into our self-attention block: just add the \( p_i \) to our inputs!
- Let \( v_i', k_i', q_i' \) be our old values, keys, and queries.

\[
\begin{align*}
  v_i &= v_i' + p_i \\
  q_i &= q_i' + p_i \\
  k_i &= k_i' + p_i
\end{align*}
\]

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:

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

- 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/
Barriers and solutions for Self-Attention as a building block

**Barriers**
- Doesn’t have an inherent notion of order!
- No nonlinearities for deep learning! It’s all just weighted averages

**Solutions**
- Add position representations to the inputs

John Hewitt
Adding nonlinearities in self-attention

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors

- Easy fix: add a **feed-forward network** to post-process each output vector.

\[ m_i = MLP(output_i) = W_2 \cdot \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2 \]

Intuition: the FF network processes the result of attention
# Barriers and solutions for Self-Attention as a building block

**Barriers**

- Doesn’t have an inherent notion of order!
- No nonlinearities for deep learning magic! It’s all just weighted averages
- Need to ensure we don’t “look at the future” when predicting a sequence
  - Like in machine translation
  - Or language modeling

**Solutions**

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output
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 $-\infty$.

\[
e_{ij} = \begin{cases} q^T k_j, & j < i \\ -\infty, & j \geq i \end{cases}
\]

![Masking the future in self-attention](image)

We can look at these (not greyed out) words

For encoding these words

```
START The chef who
-\infty -\infty -\infty -\infty
-\infty -\infty -\infty -\infty
-\infty -\infty -\infty -\infty
-\infty -\infty -\infty -\infty
```

John Hewitt
**Barriers**

- Doesn’t have an inherent notion of order!
- No nonlinearities for deep learning magic! It’s all just weighted averages
- Need to ensure we don’t “look at the future” when predicting a sequence
  - Like in machine translation
  - Or language modeling

**Solutions**

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output.
- Mask out the future by artificially setting attention weights to 0!
Necessities for a self-attention building block:

- **Self-attention:**
  - the basis of the method.

- **Position representations:**
  - Specify the sequence order, since self-attention is an unordered function of its inputs.

- **Nonlinearities:**
  - At the output of the self-attention block
  - Frequently implemented as a simple feed-forward network.

- **Masking:**
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from “leaking” to the past.

- That’s it! But this is not the **Transformer** model we’ve been hearing about.
Transformer Overview

Attention is all you need. 2017. Aswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin


- Non-recurrent sequence-to-sequence encoder-decoder model
- Task: machine translation with parallel corpus
- Predict each translated word
- Final cost/error function is standard cross-entropy error on top of a softmax classifier

This and related figures from paper ↑

Christopher Manning
Transformer Encoder

- For encoder, at each block, we use the same Q, K and V from the previous layer

- Blocks are repeated 6 times (in vertical stack)
Transformer Decoder

- 2 sublayer changes in decoder
- Masked decoder self-attention on previously generated outputs:

  ![Diagram of Transformer Decoder with masked self-attention]

  ![Diagram of Transformer Decoder showing encoder-decoder attention]

- Encoder-Decoder Attention, where queries come from previous decoder layer and keys and values come from output of encoder

  ![Diagram of Transformer Decoder with encoder-decoder attention]

Blocks repeated 6 times also
The Transformer Encoder-Decoder

[Vaswani et al., 2017]

Another look at the Transformer Encoder and Decoder Blocks at a high level
Next, let’s look at the Transformer Encoder and Decoder Blocks

What’s left in a Transformer Encoder Block that we haven’t covered?

1. **Key-query-value attention**: How do we get the $k, q, v$ vectors from a single word embedding?
2. **Multi-headed attention**: Attend to multiple places in a single layer!
3. **Tricks to help with training!**
   1. Residual connections
   2. Layer normalization
   3. Scaling the dot product
   4. These tricks **don’t improve** what the model is able to do; they help improve the training process
The Transformer Encoder: Dot-Product Attention

• Inputs: a query q and a set of key-value (k-v) pairs to an output
• Query, keys, values, and output are all vectors

• Output is weighted sum of values, where
• Weight of each value is computed by an inner product of query and corresponding key
• Queries and keys have same dimensionality $d_k$, value have $d_v$

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

Christopher Manning
The Transformer Encoder: Dot-Product Attention – Matrix notation

- When we have multiple queries $q$, we stack them in a matrix $Q$:

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

- Becomes:

$$A(Q, K, V) = \text{softmax}(QK^T)V$$

$$[|Q| \times d_k] \times [d_k \times |K|] \times [|K| \times d_v]$$

softmax row-wise $= [|Q| \times d_v]$
The Transformer Encoder: Key-Query-Value Attention

- We saw that self-attention is when keys, queries, and values come from the same source. The Transformer does this in a particular way:
  - Let $x_1, \ldots, x_T$ be input vectors to the Transformer encoder; $x_i \in \mathbb{R}^d$

- Then keys, queries, values are:
  - $k_i = K x_i$, where $K \in \mathbb{R}^{d \times d}$ is the key matrix.
  - $q_i = Q x_i$, where $Q \in \mathbb{R}^{d \times d}$ is the query matrix.
  - $v_i = V x_i$, where $V \in \mathbb{R}^{d \times d}$ is the value matrix.

- These matrices allow different aspects of the $x$ vectors to be used/emphasized in each of the three roles.
Let’s look at how key-query-value attention is computed, in matrices.

- Let $X = [x_1; \ldots; x_T] \in \mathbb{R}^{T \times d}$ be the concatenation of input vectors.
- First, note that $XK \in \mathbb{R}^{T \times d}$, $XQ \in \mathbb{R}^{T \times d}$, $XV \in \mathbb{R}^{T \times d}$.
- The output is defined as $\text{output} = \text{softmax}(XQ(XK)\top) \times XV$.

First, take the query-key dot products in one matrix multiplication: $XQ(XK)$

Next, softmax, and compute the weighted average with another matrix multiplication.

All pairs of attention scores!
What if we want to look in multiple places in the sentence at once?

• For word $i$, self-attention “looks” where $x^TQ^TKx_j$ 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 $Q_P, K_P, V_P \in \mathbb{R}^{d \times \frac{d}{h}}$, where $h$ is the number of attention heads, and $P$ ranges from 1 to $h$.

• Each attention head performs attention independently:
  • $output_P = \text{softmax}(XQ_PK_P^TX^T) \ast XV_P$, where $output_P \in \mathbb{R}^{d/h}$
  • Then the outputs of all the heads are combined!
    • $output = Y[output_1; \ldots; output_h]$, where $Y \in \mathbb{R}^{d \times d}$

• Each head gets to “look” at different things, and construct value vectors differently.
The Transformer Encoder: Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
  - For word $i$, self-attention “looks” where $x^T Q^T K x_j$ 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, $Q_P, K_P, V_P \in \mathbb{R}^{d \times \frac{d}{h}}$, where $h$ is the number of attention heads, and $P$ ranges from 1 to $h$.

**Single-head attention**
(just the query matrix)

$$X Q = X Q$$

**Multi-head attention**
(just two heads here)

$$X Q_1 Q_2 = X Q_1 X Q_2$$

*Same amount of computation as single-head self-attention!*

John Hewitt
Attention visualization in layer 5

- Words start to pay attention to other words in sensible ways
In 5th layer. Isolated attentions from just the word ‘its’ for attention heads 5 and 6. Note that the attentions are very sharp for this word.
Parallel attention heads

I kicked the ball

Who did what? To whom?
**The Transformer Encoder:**

**Residual connections** [He et al., 2016]

- **Residual connections** are a trick to help models train better.
  - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where $i$ represents the layer)

\[
X^{(i-1)} \rightarrow \text{Layer} \rightarrow X^{(i)}
\]

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

\[
X^{(i-1)} \rightarrow \text{Layer} \rightarrow X^{(i)}
\]

- Residual connections are thought to make the loss landscape considerably smoother (thus easier training!)

[Loss landscape visualization, Li et al., 2018, on a ResNet]
The Transformer Encoder: 
Layer normalization [Ba et al., 2016]

- **Layer normalization** is a trick to help models train faster.
- Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation **within each layer**.
  - LayerNorm’s success may be due to its normalizing gradients [Xu et al., 2019]
- Let \( x \in \mathbb{R}^d \) be an individual (word) vector in the model.
- Let \( \mu = \frac{1}{d} \sum_{j=1}^{d} x_j \); this is the mean; \( \mu \in \mathbb{R} \).
  - Let \( \sigma = \sqrt{\frac{1}{d} \sum_{j=1}^{d} (x_j - \mu)^2} \); this is the standard deviation; \( \sigma \in \mathbb{R} \).
  - Let \( \gamma \in \mathbb{R}^d \) and \( \beta \in \mathbb{R}^d \) be learned “gain” and “bias” parameters. (Can omit!)
  - Then layer normalization computes:
    \[
    \text{output} = \frac{x - \mu}{\sqrt{\sigma + \epsilon}} \ast \gamma + \beta
    \]

Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias
The Transformer Encoder: Scaled Dot Product [Vaswani et al., 2017]

- "Scaled Dot Product" attention is a final variation to aid in Transformer training.
- When dimensionality $d$ becomes large, dot products between vectors become large, inputs to the softmax function can be large, making gradients small.

- Instead of the self-attention function we’ve seen:

$$\text{output}_P = \text{softmax}(X Q P K^T X^T) \ast X V_P$$

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

$$\text{output}_P = \text{softmax}\left(\frac{X Q P K^T X^T}{\sqrt{d/h}}\right) \ast X V_P$$
The Transformer Encoder-Decoder
[Vaswani et al., 2017]

Looking back at the whole model, zooming in on an Encoder block:

[diagram showing the flow of information between the Transformer Encoder and Decoder blocks, with Word Embeddings and Position Representations]

[Input sequence] + [Output sequence]
The Transformer Encoder-Decoder
[Vaswani et al., 2017]

Looking back at the whole model, zooming in on an Encoder block:

- Transformer Encoder
  - Residual + LayerNorm
  - Feed-Forward
  - Residual + LayerNorm
  - Multi-Head Attention

[Input sequence]

[Output sequence]
The Transformer Encoder-Decoder

[John Hewitt]

Looking back at the whole model, zooming in on a Decoder block:

Transformer Encoder

Transformer Encoder

Word Embeddings + Position Representations

[input sequence]

Transformer Decoder

Residual + LayerNorm

Feed-Forward

Residual + LayerNorm

Multi-Head Cross-Attention

Residual + LayerNorm

Masked Multi-Head Self-Attention

Word Embeddings + Position Representations

[output sequence]
The Transformer Encoder-Decoder

[Vaswani et al., 2017]

The only new part is attention from decoder to encoder. Like we saw in previous lecture!
The Transformer Decoder: Cross-attention (details)

- We saw 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; $x_i \in \mathbb{R}^d$
- Let $z_1, \ldots, z_T$ be input vectors from the Transformer decoder, $z_i \in \mathbb{R}^d$
- Then keys and values are drawn from the encoder (like a memory):
  - $k_i = Kh_i, v_i = Vh_i$.
- And the queries are drawn from the decoder, $q_i = Qz_i$. 
Let’s look at how cross-attention is computed, in matrices.

- Let $H = [h_1; ...; h_T] \in \mathbb{R}^{T \times d}$ be the concatenation of encoder vectors.
- Let $Z = [z_1; ...; z_T] \in \mathbb{R}^{T \times d}$ be the concatenation of decoder vectors.
- The output is defined as $\text{output} = \text{softmax}(ZQ(HK)) \times HV$.

First, take the query-key dot products in one matrix multiplication: $ZQ(HK)$

Next, softmax, and compute the weighted average with another matrix multiplication.

All pairs of attention scores!
Great Results with Transformers

Next, document generation!

<table>
<thead>
<tr>
<th>Model</th>
<th>Test perplexity</th>
<th>ROUGE-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq2seq-attention, $L = 500$</td>
<td>5.04952</td>
<td>12.7</td>
</tr>
<tr>
<td>Transformer-ED, $L = 500$</td>
<td>2.46645</td>
<td>34.2</td>
</tr>
<tr>
<td>Transformer-D, $L = 4000$</td>
<td>2.22216</td>
<td>33.6</td>
</tr>
<tr>
<td>Transformer-DMCA, no MoE-layer, $L = 11000$</td>
<td>2.05159</td>
<td>36.2</td>
</tr>
<tr>
<td>Transformer-DMCA, MoE-128, $L = 11000$</td>
<td>1.92871</td>
<td>37.9</td>
</tr>
<tr>
<td>Transformer-DMCA, MoE-256, $L = 7500$</td>
<td>1.90325</td>
<td>38.8</td>
</tr>
</tbody>
</table>

The old standard

Transformers all the way down.

[Liu et al., 2018]; WikiSum dataset
What would we like to fix about the Transformer?

- **Quadratic compute in self-attention:**
  - Computing all pairs of interactions means our computation grows \textit{quadratically} with the sequence length!
  - For recurrent models, it only grew linearly!
- **Position representations:**
  - Are simple absolute indices the best we can do to represent position?
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]
One of the benefits of self-attention over recurrence was that it’s highly parallelizable.

However, its total number of operations grows as $O(T^2d)$, where $T$ is the sequence length, and $d$ is the dimensionality.

Think of $d$ as around 1,000.

- So, for a single (shortish) sentence, $T \leq 30; T^2 \leq 900$.
- In practice, we set a bound like $T = 512$.
- **But what if we’d like $T \geq 10,000$?** For example, to work on long documents?
Recent work on improving on quadratic self-attention cost

- Considerable recent work has gone into the question, *Can we build models like Transformers without paying the $O(T^2)$ all-pairs self-attention cost?*
- For example, **Linformer** [Wang et al., 2020]

Key idea: map the sequence length dimension to a lower-dimensional space for values, keys
Recent work on improving on quadratic self-attention cost

- Considerable recent work has gone into the question, *Can we build models like Transformers without paying the $O(T^2)$ all-pairs self-attention cost?*
- For example, **BigBird** [Zaheer et al., 2021]

Key idea: replace all-pairs interactions with a family of other interactions, **like local windows, looking at everything**, and **random interactions**.

(a) Random attention  
(b) Window attention  
(c) Global Attention  
(d) BIGBIRD
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 train the model to reconstruct those parts.

• This has been exceptionally effective at building strong:
  • representations of language
  • parameter initializations for strong NLP models.

Adapted from John Hewitt
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.
- Save the network parameters.
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!

---

John Hewitt
Capturing meaning via context: What kinds of things does pretraining learn?

There’s increasing evidence that pretrained models learn a wide variety of things about the statistical properties of language:

- Stanford University is located in__________, California. [Trivia]
- 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]
- Iroh went into the kitchen to make some tea. Standing next to Iroh, Zuko pondered his destiny. Zuko left the______. [some reasoning – this is harder]
- I was thinking about the sequence that goes 1, 1, 2, 3, 5, 8, 13, 21,_____[some basic arithmetic; they don’t learn the Fibonacci sequence]
- Models also learn – and can exacerbate racism, sexism, all manner of bad biases.
- More on all this in the interpretability lecture!

Adapted from John Hewitt
The neural architecture influences the type of pretraining, and natural use cases.

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

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

**Encoder-Decoder**
- Good parts of decoders and encoders?
- What’s the best way to pretrain them?
Pretraining decoders

When using language model pretrained decoders, we can ignore that they were trained to model $p(w_t|w_{1:t-1})$.

We can finetune them by training a classifier on the last word’s hidden state.

$$h_1,\ldots, h_T = \text{Decoder}(w_1, \ldots, w_T)$$

$$y \sim Aw_T + b$$

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)

\[
\begin{align*}
    h_1, \ldots, h_T &= \text{Decoder}(w_1, \ldots, w_T) \\
    w_t &\sim A w_{t-1} + b
\end{align*}
\]

Where $A, b$ were pretrained in the language model!

[Note how the linear layer has been pretrained.]
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.
- The acronym “GPT” never showed up in the original paper; it could stand for “Generative PreTraining” or “Generative Pretrained Transformer”
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*  \(\text{entailment}\)

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.

<table>
<thead>
<tr>
<th>Method</th>
<th>MNLI-m</th>
<th>MNLI-mm</th>
<th>SNLI</th>
<th>SciTail</th>
<th>QNLI</th>
<th>RTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESIM + ELMo [44] (5x)</td>
<td>-</td>
<td>-</td>
<td>89.3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CAFE [58] (5x)</td>
<td>80.2</td>
<td>79.0</td>
<td>89.3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Stochastic Answer Network [35] (3x)</td>
<td>80.6</td>
<td>80.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CAFE [58]</td>
<td>78.7</td>
<td>77.9</td>
<td>88.5</td>
<td>83.3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GenSen [64]</td>
<td>71.4</td>
<td>71.3</td>
<td>-</td>
<td>-</td>
<td>82.3</td>
<td>59.2</td>
</tr>
<tr>
<td>Multi-task BiLSTM + Attn [64]</td>
<td>72.2</td>
<td>72.1</td>
<td>-</td>
<td>-</td>
<td>82.1</td>
<td>61.7</td>
</tr>
<tr>
<td>Finetuned Transformer LM (ours)</td>
<td><strong>82.1</strong></td>
<td><strong>81.4</strong></td>
<td><strong>89.9</strong></td>
<td><strong>88.3</strong></td>
<td><strong>88.1</strong></td>
<td><strong>56.0</strong></td>
</tr>
</tbody>
</table>
We mentioned how pretrained decoders can be used in their capacities as language models. GPT-2, a larger version of GPT trained on more data, 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.
Let’s take a look at the assumptions we’ve made about a language’s vocabulary.

We assume a fixed vocab of tens of thousands of words, built from the training set. All *novel* words seen at test time are mapped to a single UNK.

<table>
<thead>
<tr>
<th>word</th>
<th>vocab mapping</th>
<th>embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>hat</td>
<td>pizza (index)</td>
<td></td>
</tr>
<tr>
<td>learn</td>
<td>tasty (index)</td>
<td></td>
</tr>
<tr>
<td>taaaaasty</td>
<td>UNK (index)</td>
<td></td>
</tr>
<tr>
<td>laern</td>
<td>UNK (index)</td>
<td></td>
</tr>
<tr>
<td>Transformerify</td>
<td>UNK (index)</td>
<td></td>
</tr>
</tbody>
</table>
Finite vocabulary assumptions make even less sense in many languages.

- Many languages exhibit complex morphology, or word structure.
  - The effect is more word types, each occurring fewer times.

Example: Swahili verbs can have hundreds of conjugations, each encoding a wide variety of information. (Tense, mood, definiteness, negation, information about the object, ++)

Here’s a small fraction of the conjugations for ambia – to tell.

---

Aside: Word structure and subword models
Subword modeling in NLP encompasses a wide range of methods for reasoning about structure below the word level. (Parts of words, characters, bytes.)

- The dominant modern paradigm is to learn a vocabulary of parts of words (subword tokens).
- At training and testing time, each word is split into a sequence of known subwords.

**Byte-pair encoding** is a simple, effective strategy for defining a subword vocabulary.

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

Originally used in NLP for machine translation; now a similar method (WordPiece) is used in pretrained models.

[Source: Sennrich et al., 2016, Wu et al., 2016]

Aside: The byte-pair encoding algorithm
Aside: Word structure and subword models

Common words end up being a part of the subword vocabulary, while rarer words are split into (sometimes intuitive, sometimes not) components.

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

<table>
<thead>
<tr>
<th>word</th>
<th>vocab mapping</th>
<th>embedding</th>
</tr>
</thead>
<tbody>
<tr>
<td>hat</td>
<td>hat</td>
<td></td>
</tr>
<tr>
<td>learn</td>
<td>learn</td>
<td></td>
</tr>
<tr>
<td>taaaaasty</td>
<td>taa## aaa## sty</td>
<td></td>
</tr>
<tr>
<td>laern</td>
<td>la## ern##</td>
<td></td>
</tr>
<tr>
<td>Transformerify</td>
<td>Transformer## ify</td>
<td></td>
</tr>
</tbody>
</table>

John Hewitt
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.

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

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

[Devlin et al., 2018]
Devlin et al., 2018 proposed the “Masked LM” objective and released the weights of a pretrained Transformer, a model they labeled 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]
Mask out $k\%$ of the input words, and then predict the masked words.
- They always use $k = 15\%$.

```
store          gallon
↑               ↑
the man went to the [MASK] to buy a [MASK] of milk
```

- Too little masking: Too expensive to train.
- Too much masking: Not enough context.
BERT: Bidirectional Encoder Representations from Transformers

- Additional task: Next sentence prediction
- To learn *relationships* between sentences, predict whether Sentence B is actual sentence that proceeds Sentence A, or a random sentence

---

**Sentence A** = The man went to the store.
**Sentence B** = He bought a gallon of milk.
**Label** = IsNextSentence

---

**Sentence A** = The man went to the store.
**Sentence B** = Penguins are flightless.
**Label** = NotNextSentence

Adapted from Christopher Manning
The pretraining input to BERT was two separate contiguous chunks of text:

In addition to masked input reconstruction, 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**: small natural language inference corpus

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

[Devlin et al., 2018]
Extensions of BERT

You’ll see a lot of BERT variants like RoBERTa, SpanBERT, and others...

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

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

[Liu et al., 2019; Joshi et al., 2020]
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!

This is implemented in text preprocessing: it’s still an objective that looks like language modeling at the decoder side.
A fascinating property of T5: it can be finetuned to answer a wide range of questions, retrieving knowledge from its parameters.

NQ: Natural Questions
WQ: WebQuestions
TQA: Trivia QA

All “open-domain” versions

<table>
<thead>
<tr>
<th></th>
<th>NQ</th>
<th>WQ</th>
<th>TQA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>dev</td>
</tr>
<tr>
<td>Karpukhin et al. (2020)</td>
<td>41.5</td>
<td>42.4</td>
<td>57.9</td>
</tr>
<tr>
<td>T5.1.1-Base</td>
<td>25.7</td>
<td>28.2</td>
<td>24.2</td>
</tr>
<tr>
<td>T5.1.1-Large</td>
<td>27.3</td>
<td>29.5</td>
<td>28.5</td>
</tr>
<tr>
<td>T5.1.1-XL</td>
<td>29.5</td>
<td>32.4</td>
<td>36.0</td>
</tr>
<tr>
<td>T5.1.1-XXL</td>
<td>32.8</td>
<td>35.6</td>
<td>42.9</td>
</tr>
<tr>
<td>T5.1.1-XXL + SSM</td>
<td>35.2</td>
<td>42.8</td>
<td>51.9</td>
</tr>
</tbody>
</table>

[220 million params]
[770 million params]
[3 billion params]
[11 billion params]

[Raffel et al., 2018]
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 take their predictions.

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

Dosovitskiy, ICLR 2021, https://github.com/google-research/vision_transformer

https://www.youtube.com/watch?v=TrdevFK_am4
Figure 1: Overview of the proposed UNITER model (best viewed in color), consisting of an Image Embedder, a Text Embedder and a multi-layer self-attention Transformer, learned through three pre-training tasks.
Cross-modal transformers

Figure 1: Our ViLBERT model consists of two parallel streams for visual (green) and linguistic (purple) processing that interact through novel co-attentinal transformer layers. This structure allows for variable depths for each modality and enables sparse interaction through co-attention. Dashed boxes with multiplier subscripts denote repeated blocks of layers.
Cross-modal transformers

Figure 1: The LXMERT model for learning vision-and-language cross-modality representations. ‘Self’ and ‘Cross’ are abbreviations for self-attention sub-layers and cross-attention sub-layers, respectively. ‘FF’ denotes a feed-forward sub-layer.
# Visual Commonsense Reasoning leaderboard

![Image of people and pancakes](https://visualcommonsense.com/leaderboard/)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Model</th>
<th>Q-&gt;A</th>
<th>QA-&gt;R</th>
<th>Q-&gt;AR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Human Performance, University of Washington (Zellers et al. '18)</td>
<td>91.0</td>
<td>93.0</td>
<td>85.0</td>
</tr>
<tr>
<td>2</td>
<td>UNITER-large (ensemble), MS D365 AI</td>
<td>79.8</td>
<td>83.4</td>
<td>66.8</td>
</tr>
<tr>
<td></td>
<td><a href="https://arxiv.org/abs/1909.11740">Link</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>UNITER-large (single model), MS D365 AI</td>
<td>77.3</td>
<td>80.8</td>
<td>62.8</td>
</tr>
<tr>
<td></td>
<td><a href="https://arxiv.org/abs/1909.11740">Link</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>VI-LBERT (ensemble of 10 models), Georgia Tech &amp; Facebook AI Research</td>
<td>76.4</td>
<td>78.0</td>
<td>59.8</td>
</tr>
<tr>
<td></td>
<td><a href="https://arxiv.org/abs/1903.02265">Link</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>VI-LBERT (single model), MSRA &amp; USTC</td>
<td>75.8</td>
<td>78.4</td>
<td>59.7</td>
</tr>
<tr>
<td></td>
<td><a href="https://arxiv.org/abs/1908.08530">Link</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>VI-LBERT (ensemble of 5 models), Georgia Tech &amp; Facebook AI Research</td>
<td>75.7</td>
<td>77.5</td>
<td>58.8</td>
</tr>
<tr>
<td></td>
<td><a href="https://arxiv.org/abs/1903.02265">Link</a></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cost of training

ULMfit
Jan 2018
Training: 1 GPU day

GPT
June 2018
Training
240 GPU days

BERT
Oct 2018
Training
256 TPU days
~320–560 GPU days

GPT-2
Feb 2019
Training
~2048 TPU v3 days according to a reddit thread

Christopher Manning