Natural Language Processing with Deep Learning

CS224N/Ling284

Anna Goldie
Lecture 10: Pretraining

Slides coauthored with John Hewitt
Breaking (Transformer) News!

AlphaCode (a pre-trained Transformer-based code generation model) achieved a top 54.3% rating on Codeforces programming competitions!

[Li et al., 2022]
More Breaking (Transformer) News!

Pre-Trained Transformer-Based theorem prover sets new state-of-the-art (41.2% vs. 29.3%) on a collection of challenging math Olympiad questions (miniF2F)!

```latex
\begin{verbatim}
problem 1

adapted from AMC12 2000 problem 5
prove that if |x - 2| = p, where x < 2, then x - p = 2 - 2p.

\begin{verbatim}
theorem amc12_2000_p5 -- : theorem name
(x p : R) -- : the statement we want
(h0 : x < 2) -- : to prove
(h1 : abs (x - 2) = p) :
  x - p = 2 - 2 * p ::= begin -- : formal proof starts here
  -- this first tactic requires that the prover invent
  -- the term: `abs (x - 2) = -(x - 2)`.
  have h2 : abs (x - 2) = -(x - 2), { apply abs_of_neg,
      linarith,
  },
  rw h1 at h2,
  -- at this stage the remaining goal to prove is:
  -- `x - p = 2 - 2 * p` knowing that `p = -(x - 2)`.
  linarith,
end
\end{verbatim}
\end{verbatim}

[Polu et al., 2022]
```
Lecture Plan

1. Quick review of Transformer model
2. Brief note on subword modeling
3. Motivating model pretraining from word embeddings
4. Model pretraining three ways
   1. Decoders
   2. Encoders
   3. Encoder-Decoders
5. Very large models and in-context learning

Reminders:

Assignment 5 is out today! It covers Lecture 9 (Tuesday) and Lecture 10 (Today)!
Hugging Face Transformers Tutorial Session on Friday 1:30-2:30pm (Thornton 102 and recorded)!
The Transformer Encoder-Decoder [Vaswani et al., 2017]

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

[diagram of the Transformer Encoder-Decoder model with arrows indicating flows between layers and blocks, labeled with 'word embeddings', 'position representations', 'Transformer Encoder', and 'Transformer Decoder']

[Input sequence] [predictions!] [decoder attends to encoder states] [output sequence]
The Transformer Encoder-Decoder [Vaswani et al., 2017]

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

[diagram showing the Transformer Encoder and Decoder blocks, including Residual + LayerNorm, Feed-Forward, Residual + LayerNorm, Multi-Head Attention, Word Embeddings, Position Representations, and an indication that the decoder attends to the encoder states and predictions are made.]
The Transformer Encoder-Decoder [Vaswani et al., 2017]

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

Transformers have revolutionized natural language processing by enabling efficient and effective attention mechanisms. Unlike recurrent neural networks (RNNs), which process data sequentially, transformers use self-attention mechanisms to capture dependencies across the entire input sequence.

In a transformer model, inputs are first encoded using word embeddings and position representations. Then, these encoded inputs are passed through multiple layers of self-attention and cross-attention mechanisms. Self-attention allows the model to focus on relevant parts of the input sequence, while cross-attention allows the model to understand how the input relates to the output sequence.

The output from these attention mechanisms is then passed through feed-forward networks and residual layers, followed by layer normalization. This process is repeated several times within each block, allowing the model to learn complex relationships in the data.

Finally, the output from the last layer is passed through a softmax function to generate predictions for the next word or sequence element. The predictions are then fed back into the model to refine the output, creating a feedback loop between the encoder and decoder blocks.

This architecture has been highly successful in a variety of natural language processing tasks, including machine translation, text summarization, and question answering.
Lecture Plan

1. Quick review of Transformer model
2. Brief note on subword modeling
3. Motivating model pretraining from word embeddings
4. Model pretraining three ways
   1. Decoders
   2. Encoders
   3. Encoder-Decoders
5. Very large models and in-context learning
Word structure and subword models

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>→ hat</td>
<td></td>
</tr>
<tr>
<td>learn</td>
<td>→ learn</td>
<td></td>
</tr>
<tr>
<td>taaaaasty</td>
<td>→ UNK</td>
<td></td>
</tr>
<tr>
<td>laern</td>
<td>→ UNK</td>
<td></td>
</tr>
<tr>
<td>Transformerify</td>
<td>→ UNK</td>
<td></td>
</tr>
</tbody>
</table>
Word structure and subword models

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.

<table>
<thead>
<tr>
<th>Conjugation of ambia</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Form</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Intensive</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Positive form</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Imperfective</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Perfective</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Negative</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Katalamba</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Punjabi</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| **Non-finite forms**  |   |   |   |
| **Plural**            |   |   |   |

| **Past**              |   |   |   |
| **Positive**          |   |   |   |
| **Negative**          |   |   |   |

| **Future**            |   |   |   |
| **Positive**          |   |   |   |
| **Negative**          |   |   |   |

| **Subjunctive**       |   |   |   |
| **Present**           |   |   |   |
| **Past Conditional**  |   |   |   |
| **Conditional to Past** |   |   |   |

Here’s a small fraction of the conjugations for ambia – to tell.
The byte-pair encoding algorithm

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 pair of adjacent characters “a,b”; add subword “ab” to the vocab.
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.

[Sennrich et al., 2016, Wu et al., 2016]
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>taaaasty</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>

Common words

Variations

misspellings

novel items
Outline

1. Quick review of Transformer models
2. Brief note on subword modeling
3. Motivating model pretraining from word embeddings
4. Model pretraining three ways
   1. Decoders
   2. Encoders
   3. Encoder-Decoders
5. Very large models and in-context learning
Motivating word meaning and context

Recall the adage we mentioned at the beginning of the course:

“You shall know a word by the company it keeps” (J. R. Firth 1957: 11)

This quote is a summary of distributional semantics, and motivated word2vec. But:

“... the complete meaning of a word is always contextual, and no study of meaning apart from a complete context can be taken seriously.” (J. R. Firth 1935)

Consider I record the record: the two instances of record mean different things.

[Thanks to Yoav Goldberg on Twitter for pointing out the 1935 Firth quote.]
Where we were: pretrained word embeddings

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

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

[Recall, movie gets the same word embedding, no matter what sentence it shows up in]
Where we’re going: **pretraining whole models**

In modern NLP:

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

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

![Diagram showing pretraining](image)

[This model has learned how to represent entire sentences through pretraining]
What can we learn from reconstructing the input?

Stanford University is located in __________, California.
What can we learn from reconstructing the input?

I put ___ fork down on the table.
The woman walked across the street, checking for traffic over ___ shoulder.
What can we learn from reconstructing the input?

I went to the ocean to see the fish, turtles, seals, and _____. 
Overall, the value I got from the two hours watching it was the sum total of the popcorn and the drink.

The movie was ___.

What can we learn from reconstructing the input?
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, ____
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!

Decoder (Transformer, LSTM, ++ )

Iroh goes to make tasty tea END

Step 2: Finetune (on your task)
Not many labels; adapt to the task!

Decoder (Transformer, LSTM, ++ )

... the movie was ...

Lecture Plan

1. A brief note on subword modeling
2. Motivating model pretraining from word embeddings
3. Model pretraining three ways
   1. Decoders
   2. Encoders
   3. Encoder-Decoders
4. Very large models and in-context learning
Pretraining for three types of architectures

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
- **Examples**: GPT-2, GPT-3, LaMDA

**Encoders**
- Gets bidirectional context – can condition on future!
- Wait, how do we pretrain them?
- **Examples**: BERT and its many variants, e.g. RoBERTa

**Encoder-Decoders**
- Good parts of decoders and encoders?
- What’s the best way to pretrain them?
- **Examples**: Transformer, T5, Meena
Pretraining for three types of architectures

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
  - **Examples:** GPT-2, GPT-3, LaMDA

- **Encoders**
  - Gets bidirectional context – can condition on future!
  - Wait, how do we pretrain them?
  - **Examples:** BERT and its many variants, e.g. RoBERTa

- **Encoder-Decoders**
  - Good parts of decoders and encoders?
  - What’s the best way to pretrain them?
  - **Examples:** Transformer, T5, Meena
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.

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

$$y \sim Ah_T + b$$

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

Gradients backpropagate through the whole network.

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

It’s natural to pretrain decoders as language models and then use them as generators, finetuning their $p_\theta(w_t | w_{1:t-1})$!

This is helpful in tasks **where the output is a sequence** with a vocabulary like that at pretraining time!

- Dialogue (context=dialogue history)
- Summarization (context=document)

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

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

[Note how the linear layer has been pretrained.]
Generative Pretrained Transformer (GPT) [Radford et al., 2018]

2018’s GPT was a big success in pretraining a decoder!

- Transformer decoder with 12 layers.
- 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers.
- Byte-pair encoding with 40,000 merges
- Trained on BooksCorpus: over 7000 unique books.
  - Contains long spans of contiguous text, for learning long-distance dependencies.
How do we format inputs to our decoder for **finetuning tasks**?

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></td>
<td></td>
<td></td>
<td></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>
Examining the Effect of Pretraining in GPT [Radford et al., 2018]

As more layers are transferred, performance improves on RACE (a large-scale reading comprehension dataset) and MultiNLI.

Zero-shot performance of Transformer vs. LSTM as a function of the # of pre-training updates.
Increasingly convincing generations (GPT2) [Radford et al., 2018]

We mentioned how pretrained decoders can be used in their capacities as language models. GPT-2, a larger version of GPT 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.
Pretraining for three types of architectures

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
- **Examples:** GPT-2, GPT-3, LaMDA

**Encoders**
- Gets bidirectional context – can condition on future!
- Wait, how do we pretrain them?
- **Examples:** BERT and its many variants, e.g. RoBERTa

**Encoder-Decoder**
- Good parts of decoders and encoders?
- What’s the best way to pretrain them?
- **Examples:** Transformer, T5, Meena
Pretraining encoders: what pretraining objective to use?

So far, we’ve looked at language model pretraining. But encoders get bidirectional context, so we can’t do language modeling!

Idea: replace some fraction of words in the input with a special [MASK] token; predict these words.

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

\[ \text{Devlin et al., 2018} \]
Devlin et al., 2018 proposed the “Masked LM” objective, open-sourced their model as the tensor2tensor library, and released the weights of their pretrained Transformer (BERT).

Some more details about Masked LM for BERT:

- Predict a random 15% of (sub)word tokens.
  - Replace input word with [MASK] 80% of the time
  - Replace input word with a random token 10% of the time
  - Leave input word unchanged 10% of the time (but still predict it!)
- Why? Doesn’t let the model get complacent and not build strong representations of non-masked words. (No masks are seen at fine-tuning time!)
BERT: Bidirectional Encoder Representations from Transformers

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

[Devlin et al., 2018]
BERT: Bidirectional Encoder Representations from Transformers

- The pretraining input to BERT was two separate contiguous chunks of text:

<table>
<thead>
<tr>
<th>Input</th>
<th>[CLS]</th>
<th>my</th>
<th>dog</th>
<th>is</th>
<th>cute</th>
<th>[SEP]</th>
<th>he</th>
<th>likes</th>
<th>play</th>
<th>#ing</th>
<th>[SEP]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token Embeddings</td>
<td>$E_{[CLS]}$</td>
<td>$E_{my}$</td>
<td>$E_{dog}$</td>
<td>$E_{is}$</td>
<td>$E_{cute}$</td>
<td>$E_{[SEP]}$</td>
<td>$E_{he}$</td>
<td>$E_{likes}$</td>
<td>$E_{play}$</td>
<td>$E_{#ing}$</td>
<td>$E_{[SEP]}$</td>
</tr>
<tr>
<td>Segment Embeddings</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_A$</td>
<td>$E_B$</td>
<td>$E_B$</td>
<td>$E_B$</td>
<td>$E_B$</td>
</tr>
<tr>
<td>Position Embeddings</td>
<td>$E_0$</td>
<td>$E_1$</td>
<td>$E_2$</td>
<td>$E_3$</td>
<td>$E_4$</td>
<td>$E_5$</td>
<td>$E_6$</td>
<td>$E_7$</td>
<td>$E_8$</td>
<td>$E_9$</td>
<td>$E_{10}$</td>
</tr>
</tbody>
</table>

- 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]
BERT: Bidirectional Encoder Representations from Transformers

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.”
BERT: Bidirectional Encoder Representations from Transformers

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

- **QQP**: Quora Question Pairs (detect paraphrase questions)
- **QNLI**: natural language inference over question answering data
- **SST-2**: sentiment analysis
- **CoLA**: corpus of linguistic acceptability (detect whether sentences are grammatical.)
- **STS-B**: semantic textual similarity
- **MRPC**: microsoft paraphrase corpus
- **RTE**: a small natural language inference corpus

![Table of Results](image)

Note that BERT$_{BASE}$ was chosen to have the same number of parameters as OpenAI GPT. [Devlin et al., 2018]
Limitations of pretrained encoders

Those results looked great! Why not use pretrained encoders for everything?

If your task involves generating sequences, consider using a pretrained decoder; BERT and other pretrained encoders don’t naturally lead to nice autoregressive (1-word-at-a-time) generation methods.

![Diagram](image)
Extensions of BERT

You’ll see a lot of BERT variants like RoBERTa, SpanBERT, +++

Some generally accepted improvements to the BERT pretraining formula:
• RoBERTa: mainly just train BERT for longer and remove next sentence prediction!
• SpanBERT: masking contiguous spans of words makes a harder, more useful pretraining task

[BERT]

[SpanBERT]
Extensions of BERT

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_{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 for three types of architectures

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
- **Examples:** GPT-2, GPT-3, LaMDA

**Encoders**
- Gets bidirectional context – can condition on future!
- Wait, how do we pretrain them?
- **Examples:** BERT and its many variants, e.g. RoBERTa

**Encoder-Decoders**
- Good parts of decoders and encoders?
- What’s the best way to pretrain them?
- **Examples:** Transformer, T5, Meena
Pretraining encoder-decoders: what pretraining objective to use?

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

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

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

[Raffel et al., 2018]
Pretraining encoder-decoders: what pretraining objective to use?

What Raffel et al., 2018 found to work best was **span corruption**. Their model: **T5**.

Replace different-length spans from the input with unique placeholders; decode out the spans that were removed!

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

[Raffel et al., 2018]
Pretraining encoder-decoders: what pretraining objective to use?

Raffel et al., 2018 found encoder-decoders to work better than decoders for their tasks, and span corruption (denoising) to work better than language modeling.
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>Model</th>
<th>NQ</th>
<th>WQ</th>
<th>TQA dev</th>
<th>TQA test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Karpukhin et al. (2020)</td>
<td>41.5</td>
<td>42.4</td>
<td>57.9</td>
<td></td>
</tr>
<tr>
<td>T5.1.1-Base</td>
<td>25.7</td>
<td>28.2</td>
<td>24.2</td>
<td>30.6</td>
</tr>
<tr>
<td>T5.1.1-Large</td>
<td>27.3</td>
<td>29.5</td>
<td>28.5</td>
<td>37.2</td>
</tr>
<tr>
<td>T5.1.1-XL</td>
<td>29.5</td>
<td>32.4</td>
<td>36.0</td>
<td>45.1</td>
</tr>
<tr>
<td>T5.1.1-XXL</td>
<td>32.8</td>
<td>35.6</td>
<td>42.9</td>
<td>52.5</td>
</tr>
<tr>
<td>T5.1.1-XXL + SSM</td>
<td>35.2</td>
<td>42.8</td>
<td>51.9</td>
<td>61.6</td>
</tr>
</tbody>
</table>

220 million params
770 million params
3 billion params
11 billion params

[Raffel et al., 2018]
Outline

1. Prelude: A brief note on subword modeling
2. Motivating model pretraining from word embeddings
3. Model pretraining three ways
   1. Decoders
   2. Encoders
   3. Encoder-Decoders
4. Very large models and in-context learning
GPT-3, In-context learning, and very large models

So far, we’ve interacted with pretrained models in two ways:

• Sample from the distributions they define (maybe providing a prompt)
• Fine-tune them on a task we care about, and then take their predictions.

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

GPT-3 is the canonical example of this. The largest T5 model had 11 billion parameters. **GPT-3 has 175 billion parameters.**
GPT-3, In-context learning, and very large models

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

The in-context examples seem to specify the task to be performed, and the conditional distribution mocks performing the task to a certain extent.

**Input (prefix within a single Transformer decoder context):**

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

**Output (conditional generations):**

loutre...”
GPT-3, In-context learning, and very large models

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

• We learned about GPT-X, BERT, T5 and other large pre-trained language models
• Emergent in-context learning is not yet well-understood!
• “Small” models like BERT have become general tools in a wide range of settings.
• Many issues left to explore!
  • Bias, toxicity, and fairness (Guest Lecturer: Maarten Sap)
  • Retrieval Augmented Language Models + Knowledge (Guest Lecturer: Kelvin Guu)
  • Scaling Laws (Guest Lecturer: Jared Kaplan)
• Assignment 5 out today! It covers material from Tuesday’s and today’s lectures.
• Hugging Face Transformers Tutorial Session on Friday 1:30-2:30pm (Thornton 102 and recorded)!