BERT and Friends

CS 780/880 Natural Language Processing Lecture 20

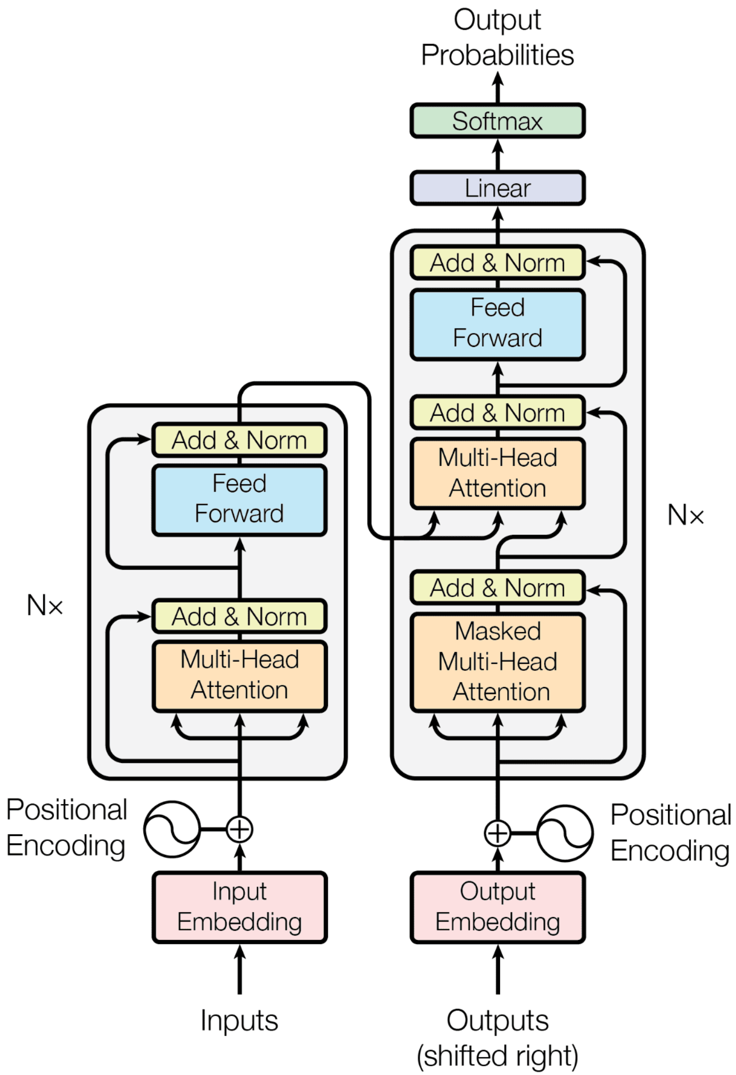
Samuel Carton, University of New Hampshire

Last lecture

Transformer architecture

- Many layers
    - **Self-attention**
    - Feed-forward
    - Residuals
- Encoder-decoder
    - Encoder nonrecurrent
    - Decoder recurrent
- Positional encodings

Pretrained transformer (BERT) is a good starting point for fine-tuning!



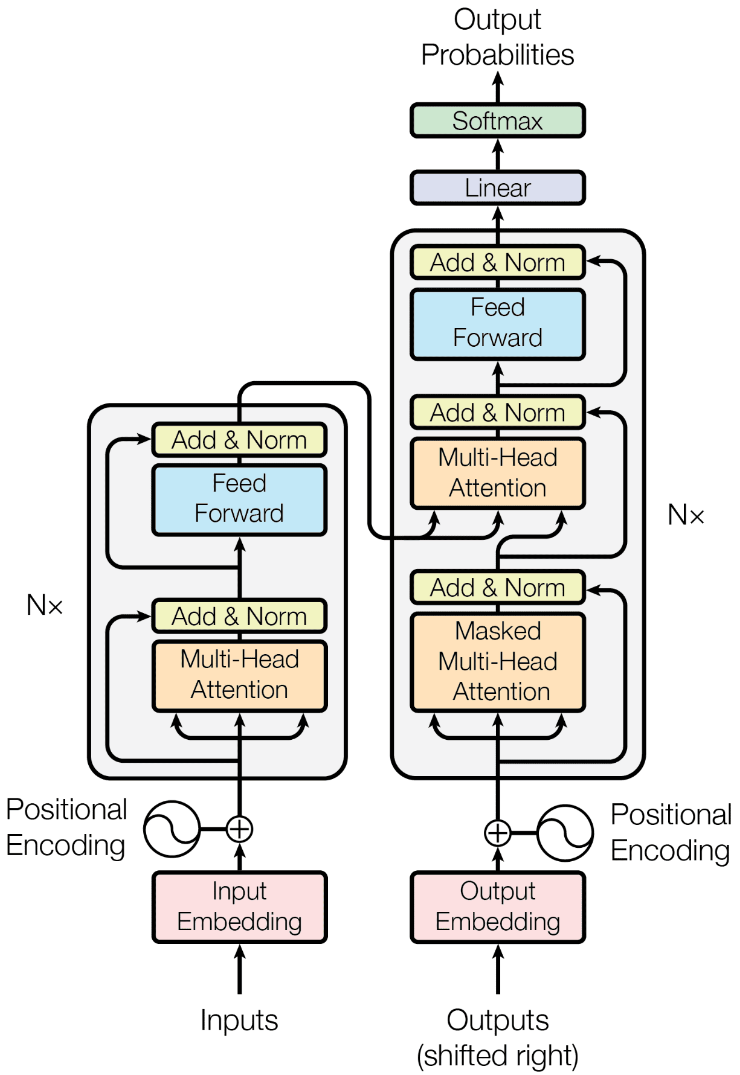
Last lecture

Transformer architecture

- Many layers
  - **Self-attention**
  - Feed-forward
  - Residuals
- Encoder-decoder
  - Encoder nonrecurrent
  - Decoder recurrent
- Positional encodings

? ? ? ?

Pretrained transformer is a good starting point for fine-tuning!



Pretrained transformers

Every current well-known large language model is a transformer that has been extensively **pretrained** on a large corpus of text, with some language modeling objective

- BERT, RoBERTa, T5, GPT-X, etc.

The difference between different models is mostly just:

- Training objective
- Use of encoder only, decoder only, or both
- Model size
- Training set size & composition
- Dataset preprocessing
- Minor architecture differences

…which seems like a lot, but it’s still pretty remarkable that the underlying model is mostly the same (Transformer)

Well-known models

There's a few key models that are in wide use:

- BERT
- RoBERTa
- XLNet
- DistilBERT
- T5
- GPT family

bert-base-uncased Updated Nov 16, 2022 • ↓ 44.8M • ♡ 706	jonatasgrosman/wav2vec2-large-xlsr-53-english Updated 17 days ago • ↓ 43M • ♡ 62
Davlan/distilbert-base-multilingual-cased-ner-hrl Updated Jun 27, 2022 • ↓ 29.4M • ♡ 22	gpt2 Updated Dec 16, 2022 • ↓ 19.8M • ♡ 866
xlm-roberta-base Updated 4 days ago • ↓ 19M • ♡ 236	openai/clip-vit-large-patch14 Updated Oct 4, 2022 • ↓ 10.6M • ♡ 313
microsoft/layoutlmv3-base Updated Dec 13, 2022 • ↓ 9.11M • ♡ 114	distilbert-base-uncased Updated Nov 16, 2022 • ↓ 8.85M • ♡ 170
distilroberta-base Updated Nov 16, 2022 • ↓ 7.87M • ♡ 55	roberta-base Updated Mar 6 • ↓ 7.25M • ♡ 146
t5-base Updated 5 days ago • ↓ 5.97M • ♡ 178	openai/clip-vit-base-patch32 Updated Oct 4, 2022 • ↓ 5.93M • ♡ 152
bert-base-cased Updated Nov 16, 2022 • ↓ 5.93M • ♡ 91	xlm-roberta-large Updated 5 days ago • ↓ 5.75M • ♡ 125

Most can be downloaded at https://huggingface.co/models

BERT

Bidirectional **E**ncoder **R**epresentations from **T**ransformers

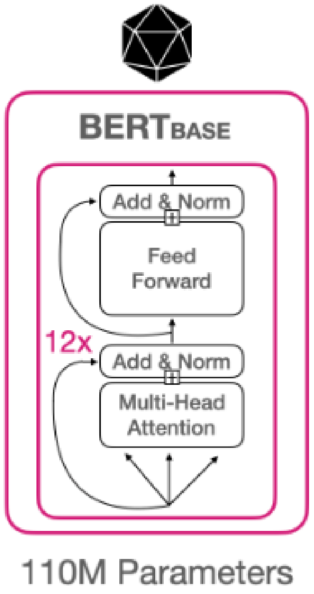
Encoder-only model

Bert-base:

- 12 layers, 12 heads per layer
- 110 million parameters

Two pretraining objectives:

- **Masked language modeling (Mask-LM)**
- **Next sentence prediction (NSP)**



Bert: Pre-training of deep bidirectional transformers for language understanding

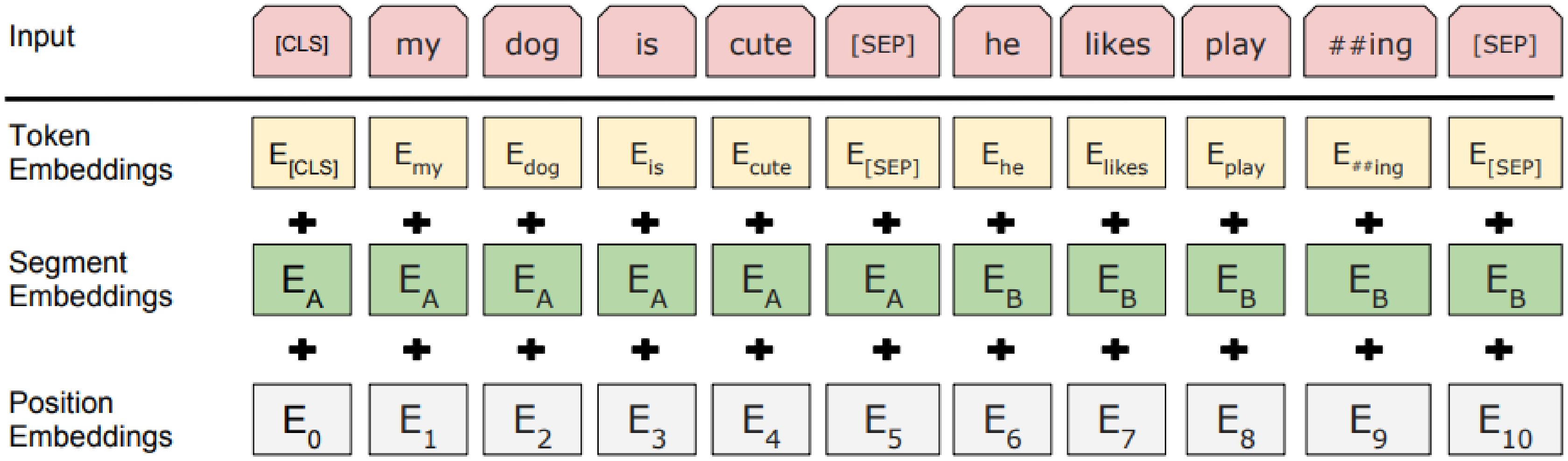
J Devlin, MW Chang, K Lee, K Toutanova - arXiv preprint arXiv …, 2018 - arxiv.org

We introduce a new language representation model called BERT, which stands for Bidirectional Encoder Representations from Transformers. Unlike recent language representation models, BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and …

BERT encoder

The BERT encoder:

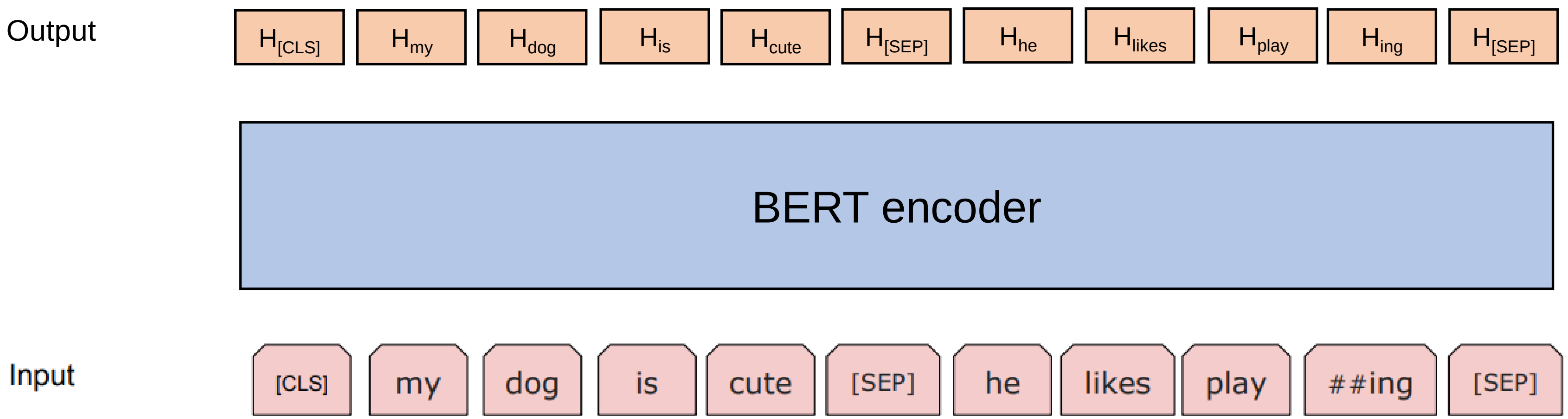
1. Takes in wordpieces
2. With [CLS] at the beginning and [SEP] between sentences
3. Adds positional and segment ID (0 or 1) embeddings
4. Outputs a hidden state vector for each wordpiece (including [CLS] and [SEP]s)



BERT encoder

The BERT encoder:

1. Takes in wordpieces
2. With [CLS] at the beginning and [SEP] between sentences
3. Adds positional and segment ID (0 or 1) embeddings
4. Outputs a hidden state vector for each wordpiece (including [CLS] and [SEP]s)



BERT pretraining

Mask-LM: Randomly mask 15% of tokens and try to predict them from H_{token}

NSP: Randomly sample correct/incorrect sentence pairs, try to predict which is correct from $H_{[CLS]}$

A term used for this overall approach is **denoising autoencoding**

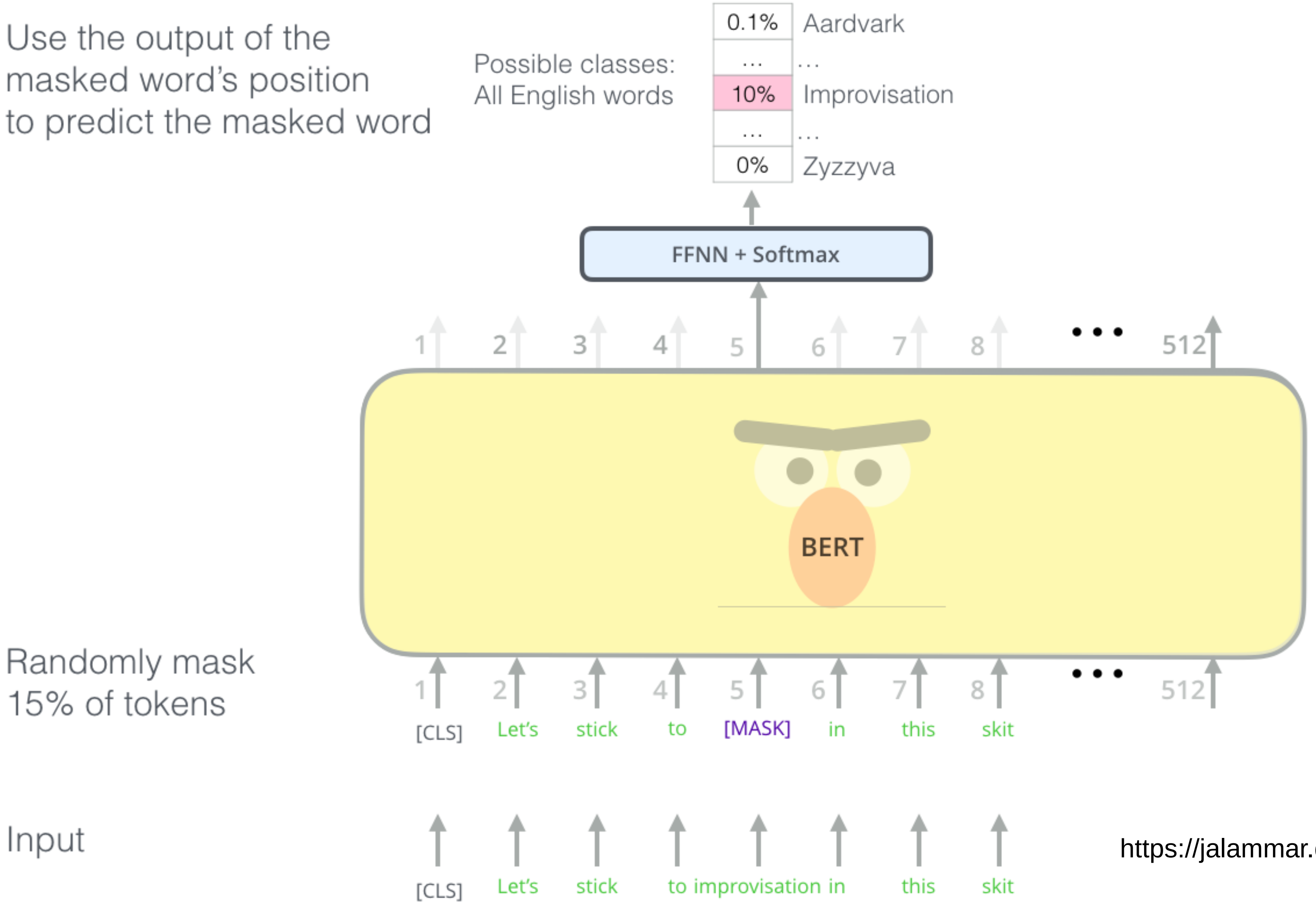
- "Denoising" because it tries to correct missing tokens
- "Autoencoding" because it tries to encode unlabeled text to a vector representation

Pretraining corpus:

- BooksCorpus (800M words)
- English Wikipedia (2,500M words)

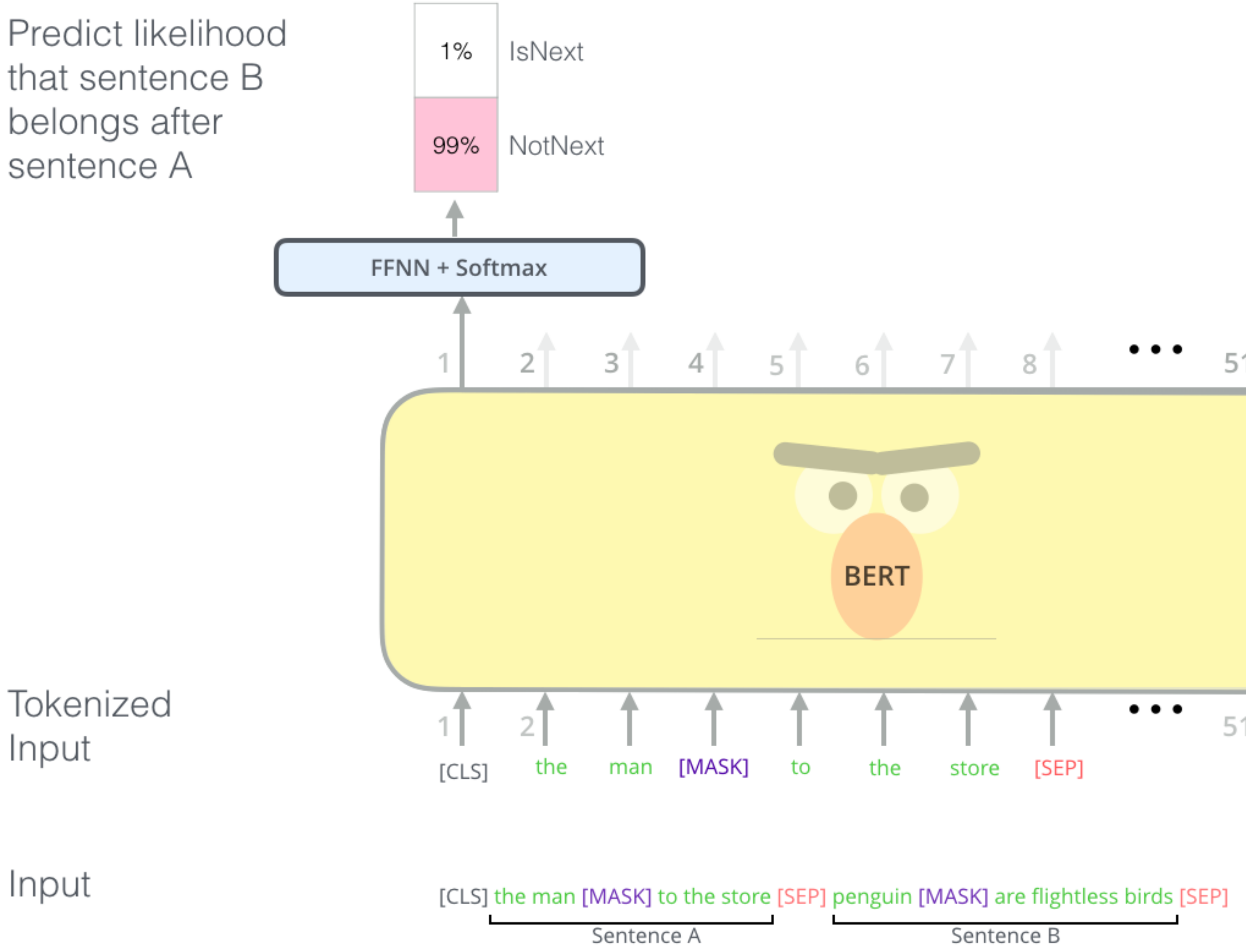
$BERT_{LARGE}$ is also available (24 layers, 16 heads per layer, 340M params)

Masked language modeling



https://jalammar.github.io/illustrated-bert/

Next-sentence prediction



https://jalammar.github.io/illustrated-bert/

Deep contextualized representations

A key thing about these models is that they produce **deep contextualized representations** of their input

- A single vector that represents the whole sequence ($H_{[CLS]}$) or an individual token (H_{token})
- The vector reflects the **context** surrounding that token.
  - So H_{jerk} will be different for “You are a jerk.” versus “I like jerk chicken”
  - Compare and contrast to word vectors

With large scale pretraining, we have models which can produce useful representations of input, which we can then fine-tune to do specific things

- Kind of like teaching someone English before trying to teach them to grade papers

RoBERTa

Essentially a refinement/exploration of BERT

- Same architecture & training data
  - Also encoder-only
- **Ditches NSP**
- Does "dynamic" mask-LM
- Improved performance on NLP benchmarks

Comparable size to $\text{BERT}_{\text{LARGE}}$

- 24 layers, 16 heads per layer, 355M params total

Probably a better default choice than BERT, if you have the GPU memory

Roberta: A robustly optimized bert pretraining approach
Y Liu, M Ott, N Goyal, J Du, M Joshi, D Chen… - arXiv preprint arXiv …, 2019 - arxiv.org
… configuration **RoBERTa** for **Robustly optimized BERT approach**. Specifically, **RoBERTa** is … (eg, the **pretraining** objective), we begin by training **RoBERTa** following the BERTLARGE …
Save Cite Cited by 6469 Related articles All 5 versions

	MNLI	QNLI	QQP	RTE	SST	MRPC	CoLA	STS	WNLI	Avg
Single-task single models on dev										
$\text{BERT}_{\text{LARGE}}$	86.6/-	92.3	91.3	70.4	93.2	88.0	60.6	90.0	-	-
$\text{XLNet}_{\text{LARGE}}$	89.8/-	93.9	91.8	83.8	95.6	89.2	63.6	91.8	-	-
RoBERTa	**90.2/90.2**	**94.7**	**92.2**	**86.6**	**96.4**	**90.9**	**68.0**	**92.4**	**91.3**	-

XLNet

Another competitor of BERT that occasionally shows up in the literature

Also uses **only** the encoder

Xlnet: Generalized autoregressive pretraining for language understanding
Z Yang, Z Dai, Y Yang, J Carbonell… - Advances in neural …, 2019 - proceedings.neurips.cc
With the capability of modeling bidirectional contexts, denoising autoencoding based pretraining like BERT achieves better performance than pretraining approaches based on autoregressive language modeling. However, relying on corrupting the input with masks, BERT neglects dependency between the masked positions and suffers from a pretrain-finetune discrepancy. In light of these pros and cons, we propose XLNet, a generalized autoregressive pretraining method that (1) enables learning bidirectional contexts by …
☆ Save 99 Cite Cited by 6635 Related articles All 17 versions

Pretrains using a variant of autoregressive language modeling called **permutation language modeling**

Comparison with BERT

- Same size
- Additional training data:
    - ClueWeb
    - Common Crawl
- Broadly improved performance

Autoregressive language modeling

Basic idea: train the model to be most likely to reproduce the training data

We've learned this before (a couple times), but this is alternative terminology.

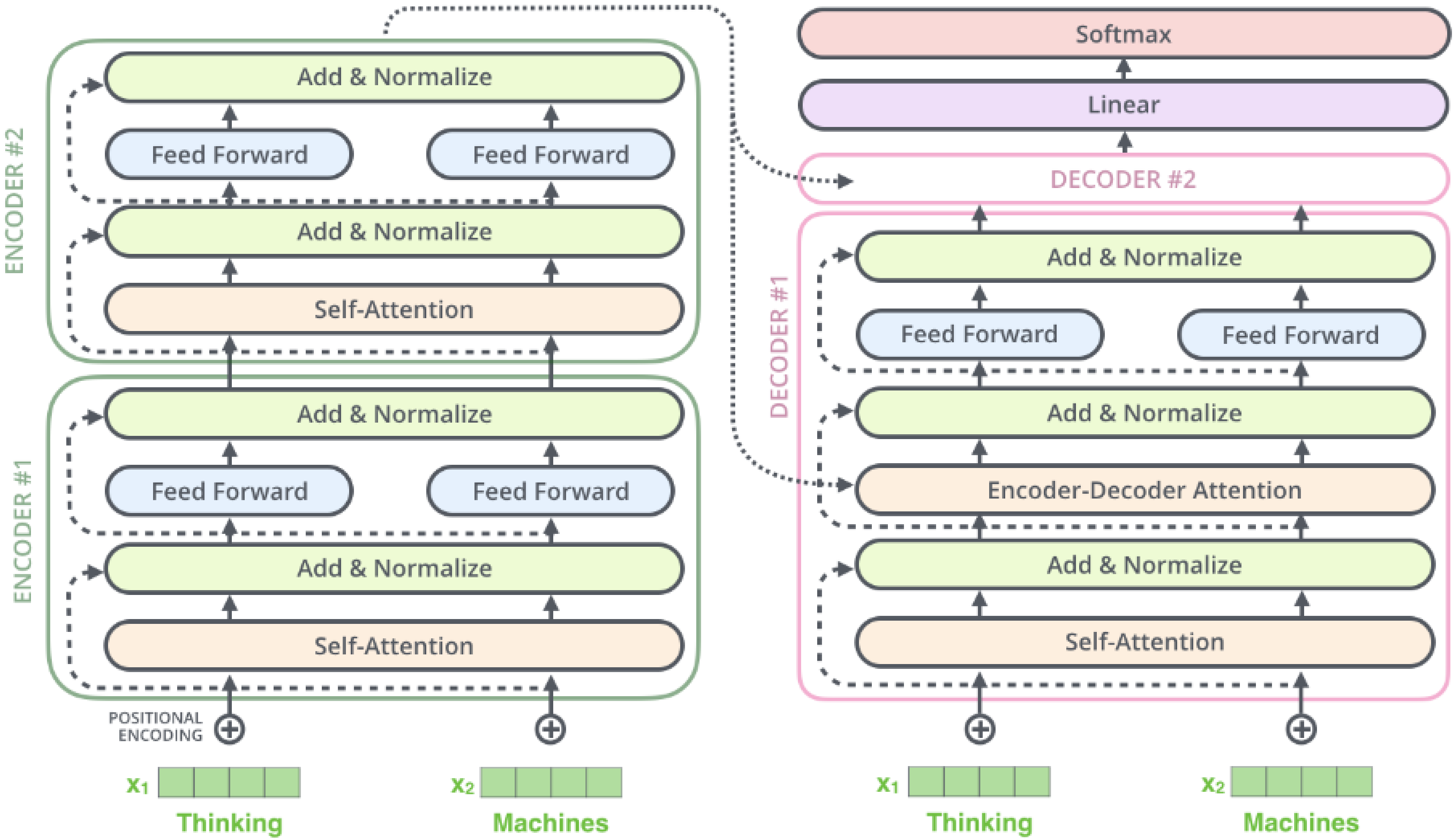
Based on a **forward factorization** of the text where each x_t is dependent on $\{x_0 \ldots x_{t-1}\}$, so we can factorize the overall likeliness of x as a sum of log-probabilities of each individual x_t:

$$\max_{\theta} \quad \log p_\theta(\mathbf{x}) = \sum_{t=1}^{T} \log p_\theta(x_t \mid \mathbf{x}_{<t}) = \sum_{t=1}^{T} \log \frac{\exp\left(h_\theta(\mathbf{x}_{1:t-1})^\top e(x_t)\right)}{\sum_{x'} \exp\left(h_\theta(\mathbf{x}_{1:t-1})^\top e(x')\right)},$$

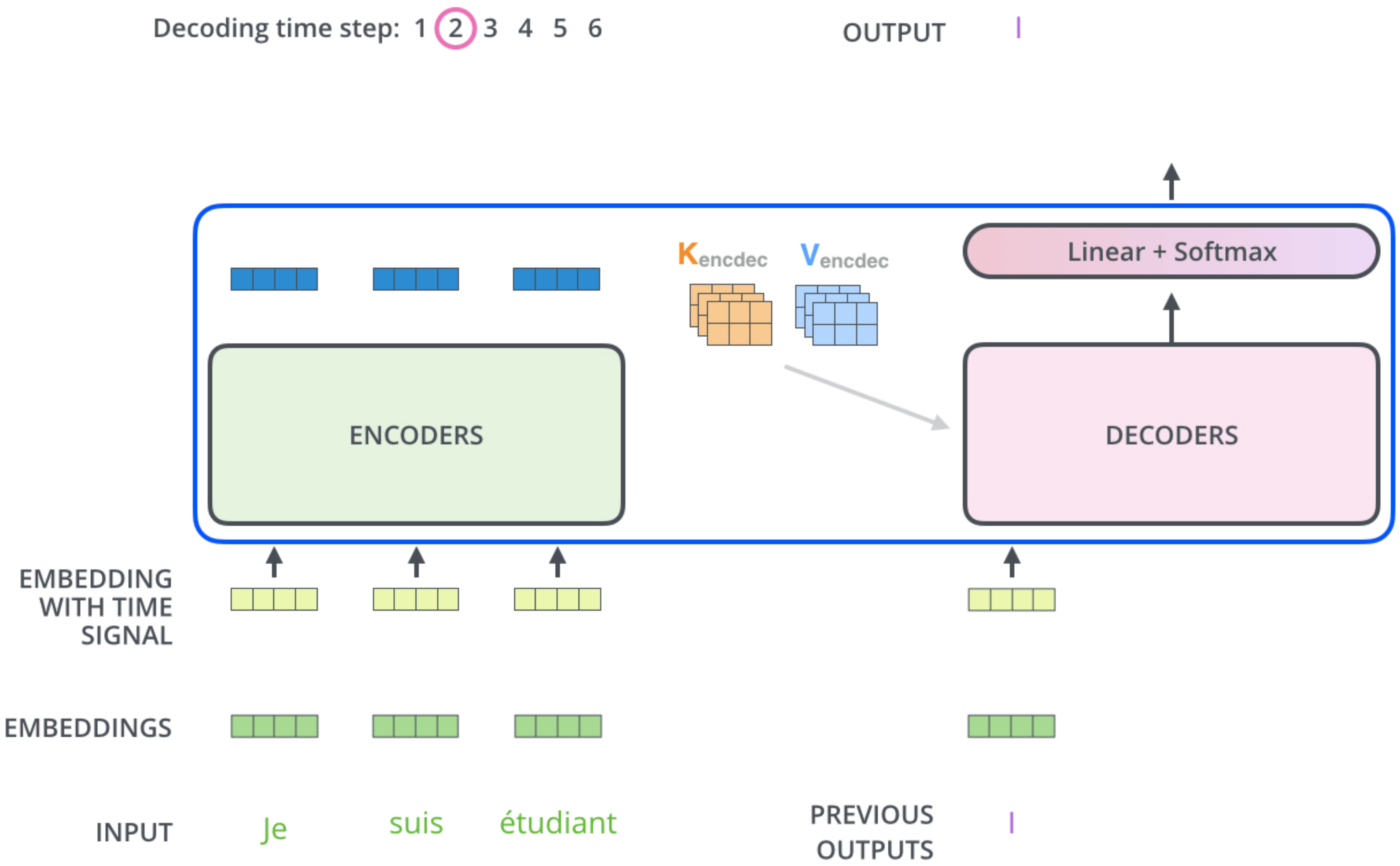
Several options for exactly how to do this:

- **Teacher forcing**: each $x_{<t}$ is drawn from the true data
- **Naïve autoregression**: each $x_{<t}$ is the one generated by the model

Autoregressive language modeling



Autoregressive decoding



XLNET: Permutation language modeling

Rather than only optimizing for token likelihood in forward factorization, XLNet optimizes for every possible permutation of the text

So not just $P(X_3 \mid X_1, X_2)$, but also $P(X_3)$, $P(X_3|X_4)$, $P(X_3|X_1, X_4, X_2)$, etc..

But doesn't use decoder!

- Instead manipulates model attention and positional encodings to erase and reorder tokens from input

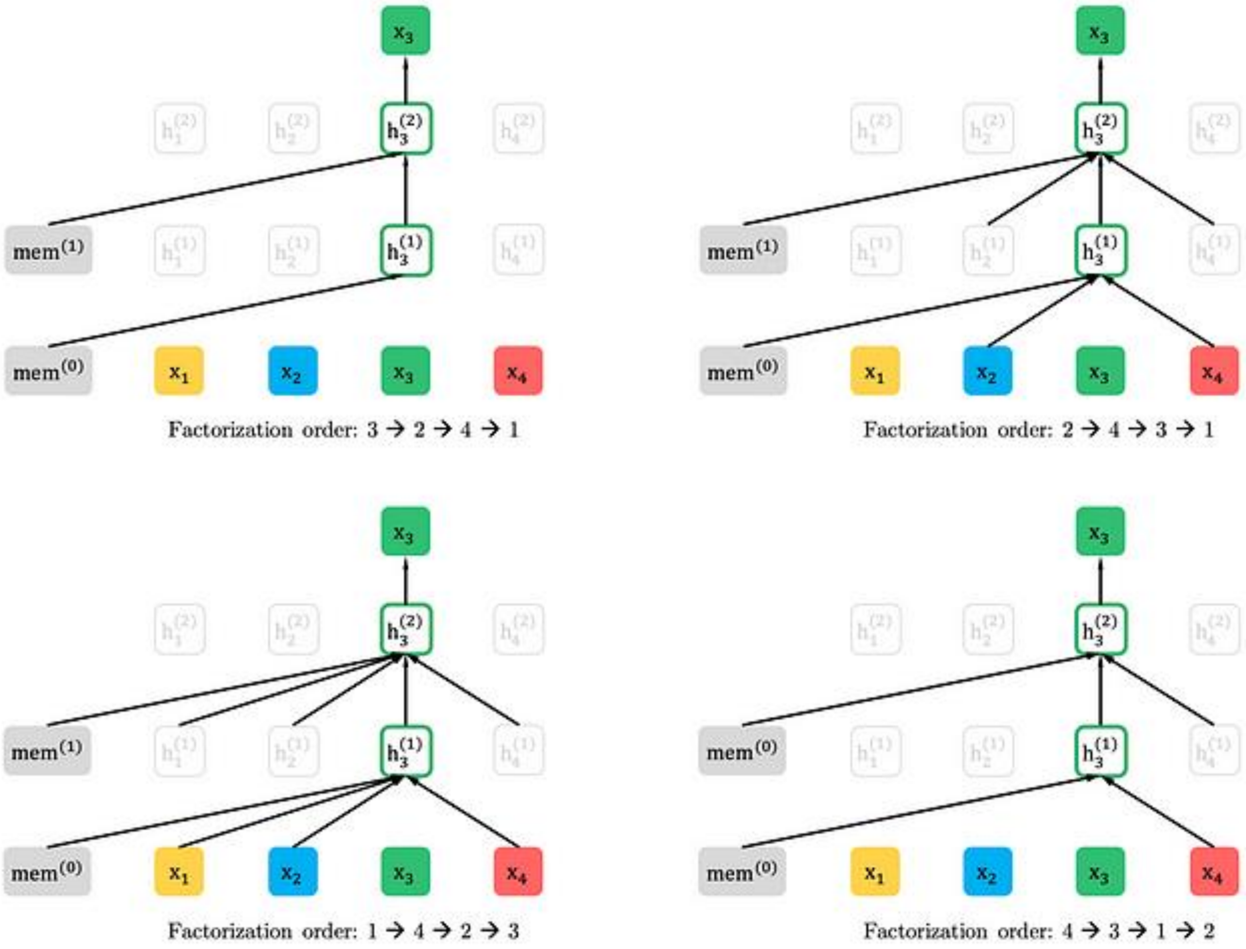


Figure 1: Illustration of the permutation language modeling objective for predicting x_3 given the same input sequence $\mathbf{x}$ but with different factorization orders.

DistilBERT

A version of BERT that has been reduced in size from BERT by a process called **knowledge distillation**

DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter
V Sanh, L Debut, J Chaumond, T Wolf - arXiv preprint arXiv:1910.01108, 2019 - arxiv.org
As Transfer Learning from large-scale pre-trained models becomes more prevalent in Natural Language Processing (NLP), operating these large models in on-the-edge and/or under constrained computational training or inference budgets remains challenging. In this work, we propose a method to pre-train a smaller general-purpose language representation model, called DistilBERT, which can then be fine-tuned with good performances on a wide range of tasks like its larger counterparts. While most prior work investigated the use of ...
Save Cite Cited by 3319 Related articles All 4 versions

Knowledge distillation:

- Big (trained) teacher model and small student model
- Train student model to emulate teacher model
- Different from regular training because teacher model produces nonzero probabilities over other possible classes, which is richer training data than 1's and 0's
    - Kind of like explaining that a shape in a CT scan is a tumor, but also looks like a cyst, rather than "it's just a tumor and not a cyst"

97% of the performance of BERT, but 40% smaller and 60% faster

- 6 layers, 12 heads per layer, 66M parameters

T5

Important model. Really the first big improvement from the BERT variants.

Uses the full encoder-decoder apparatus of the Transformer architecture

Does a combination of unsupervised language modeling and supervised text-to-text modeling

Fine-tuned T5 is still pretty close to SoTA for many NLP tasks

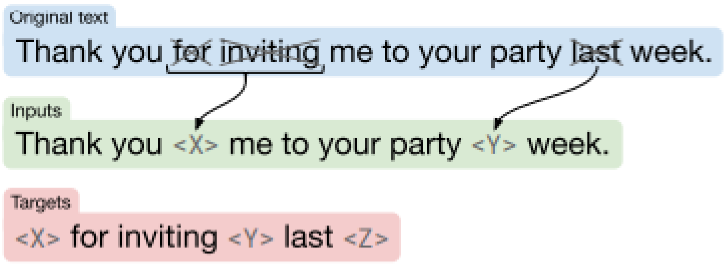
Exploring the **limits** of **transfer learning** with a **unified** text-to-text transformer
C Raffel, N Shazeer, A Roberts, K Lee… - … of Machine **Learning** …, 2020 - dl.acm.org
… The effectiveness of **transfer learning** has given rise to a diversity of approaches, … In this paper, we **explore** the landscape of **transfer learning** techniques for NLP by introducing a **unified** …

T5 pretraining—unsupervised

Creates a big, cleaned-up unsupervised training corpus:
“Colossal Clean Crawled Corpus”: cleaned-up version of Common Crawl

Uses variant of masked-LM objective from BERT, mapping corrupted text to true text

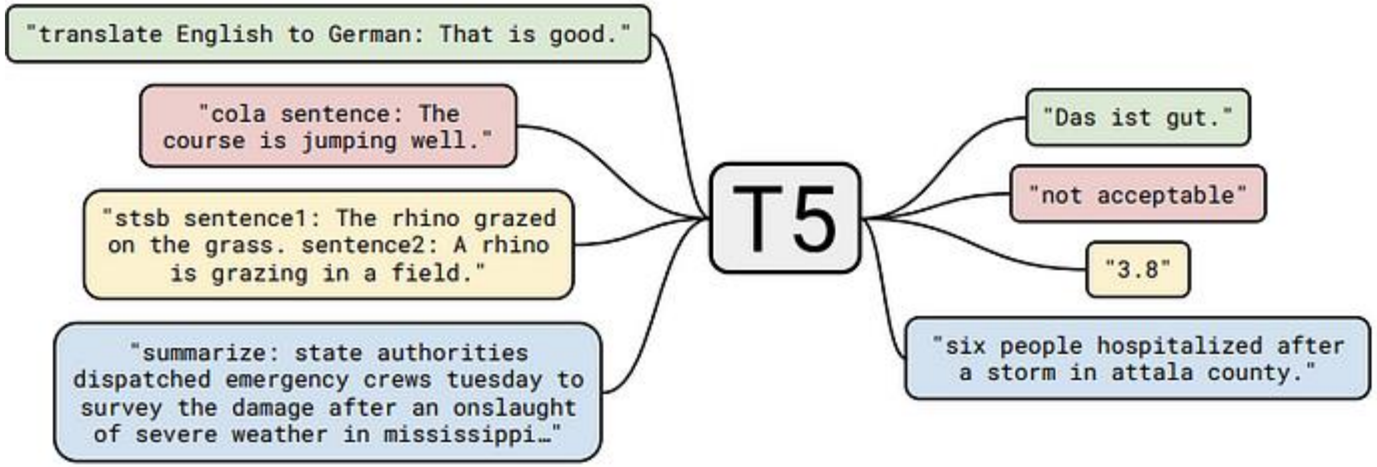
- Can mask out contiguous sequences of tokens at once
- Uses teacher-forcing to train decoder



T5 pretraining—supervised

Also converts a diverse set of supervised learning datasets into text-to-text tasks, and trains on them

- GLUE and SuperGLUE



T5

Most common model is T5-11b (11 billion parameters), but smaller variants also exist

Fine-tuned T5-11b is still pretty competitive in NLP benchmarks

Leaderboard Version: **2.0**

	Rank	Name	Model	URL	Score	BoolQ	CB	COPA	MultiRC	ReCoRD	RTE	WiC	WSC	AX-b	AX-g
	1	JDExplore d-team	Vega v2		91.3	90.5	98.6/99.2	99.4	88.2/62.4	94.4/93.9	96.0	77.4	98.6	-0.4	100.0/50.0
+	2	Liam Fedus	ST-MoE-32B		91.2	92.4	96.9/98.0	99.2	89.6/65.8	95.1/94.4	93.5	77.7	96.6	72.3	96.1/94.1
	3	Microsoft Alexander v-team	Turing NLR v5		90.9	92.0	95.9/97.6	98.2	88.4/63.0	96.4/95.9	94.1	77.1	97.3	67.8	93.3/95.5
	4	ERNIE Team - Baidu	ERNIE 3.0		90.6	91.0	98.6/99.2	97.4	88.6/63.2	94.7/94.2	92.6	77.4	97.3	68.6	92.7/94.7
	5	Yi Tay	PaLM 540B		90.4	91.9	94.4/96.0	99.0	88.7/63.6	94.2/93.3	94.1	77.4	95.9	72.9	95.5/90.4
+	6	Zirui Wang	T5 + UDG, Single Model (Google Brain)		90.4	91.4	95.8/97.6	98.0	88.3/63.0	94.2/93.5	93.0	77.9	96.6	69.1	92.7/91.9
+	7	DeBERTa Team - Microsoft	DeBERTa / TuringNLRv4		90.3	90.4	95.7/97.6	98.4	88.2/63.7	94.5/94.1	93.2	77.5	95.9	66.7	93.3/93.8
	8	SuperGLUE Human Baselines	SuperGLUE Human Baselines		89.8	89.0	95.8/98.9	100.0	81.8/51.9	91.7/91.3	93.6	80.0	100.0	76.6	99.3/99.7
+	9	T5 Team - Google	T5		89.3	91.2	93.9/96.8	94.8	88.1/63.3	94.1/93.4	92.5	76.9	93.8	65.6	92.7/91.9

https://super.gluebenchmark.com/leaderboard

GPT-1

Precursor model to GPT-2, GPT-3, and GPT-4

Decoder-only. Does not encode entire input sequence—rather, just encodes input sequence token-by-token

Trained using standard autoregressive language modeling objective

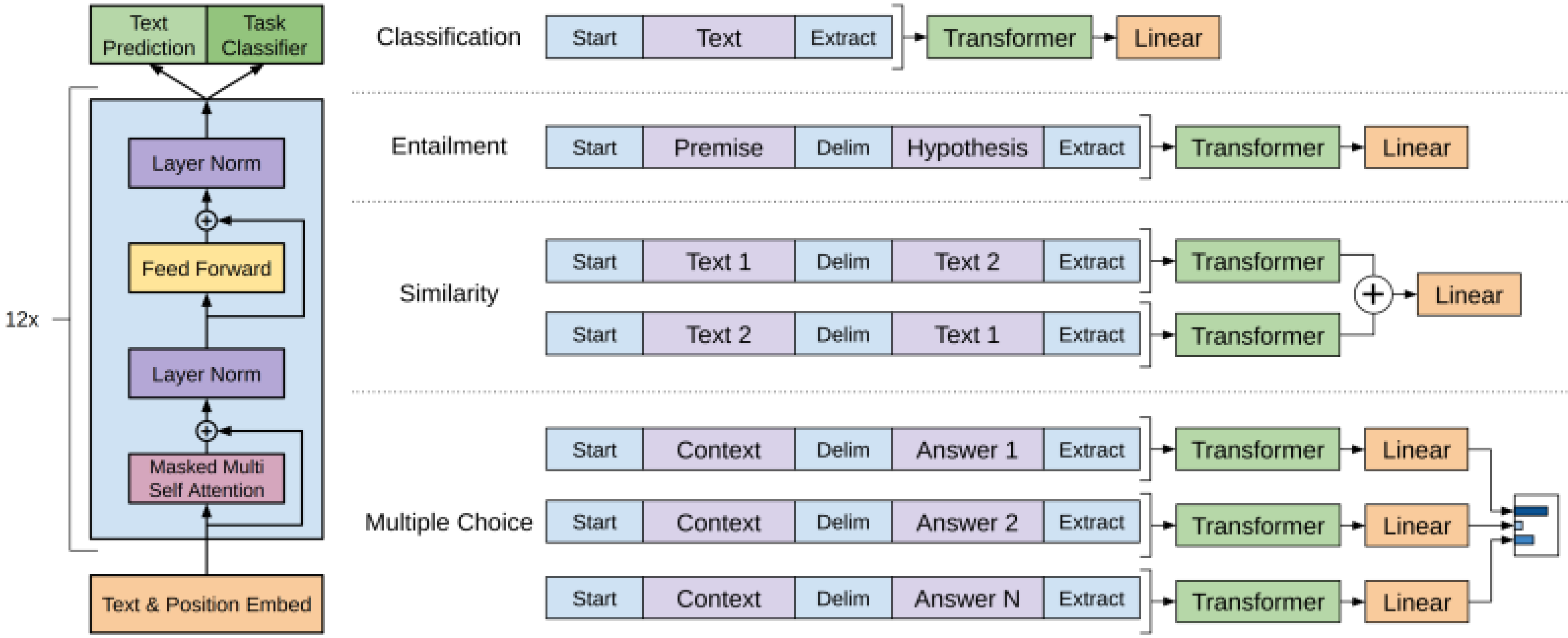
- Uses teacher forcing (I believe)

Trained on BooksCorpus dataset

12 layers, 12 heads per layer, 120M parameters

[PDF] Improving language understanding by generative pre-training
A Radford, K Narasimhan, T Salimans, I Sutskever - 2018 - cs.ubc.ca
Natural language understanding comprises a wide range of diverse tasks such as textual entailment, question answering, semantic similarity assessment, and document classification. Although large unlabeled text corpora are abundant, labeled data for learning these specific tasks is scarce, making it challenging for discriminatively trained models to perform adequately. We demonstrate that large gains on these tasks can be realized by generative pre-training of a language model on a diverse corpus of unlabeled text, followed …

GPT-1 fine-tuning



GPT-2,3,4

All larger and more capable versions of GPT-1

Same model with slight modifications

Same or larger datasets

GPT-2: 1.5B parameters
GPT-3: 175B parameters
GPT-4: ???????????

How to choose?

For general-purpose NLP fine-tuning, use the biggest model you can train:

- T5-11b → RoBERTa-Large → BERT-base → DistilBERT

For text generation, GPT-2 or GPT-Neo

For specific domains, try to find domain-specific versions of models

- E.g. MatSciBERT for materials-science specific NLP tasks
- Hugging Face has a nice search interface

Important to try multiple models

Classification with BERT

I showed you last class how to build a classifier around a BERT model.

Brief review:

- BERT tokenizer will do tokenization, batching and padding for you. Noice.
- The output layer should be built on top of the BERT's output for the [CLS] token, which gets added to the front of the sequence by the tokenizer

BERT tokenizer

```python
from transformers import BertTokenizerFast

# This command goes out onto the Hugging Face website and downloads the tokenizer
# associated with the pretrained bert-base-uncased model

# We'll talk later about how this pretraining works, but the long story short is
# that this thing will do all the preprocessing we need for us.
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
```

```
Downloading (…)okenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 1.26kB/s]
Downloading (…)solve/main/vocab.txt: 100% 232k/232k [00:00<00:00, 591kB/s]
Downloading (…)/main/tokenizer.json: 100% 466k/466k [00:00<00:00, 800kB/s]
Downloading (…)lve/main/config.json: 100% 570/570 [00:00<00:00, 42.3kB/s]
```

BERT tokenizer

```
# But you can tell it to return PyTorch tensors instead
tokenized_pt = tokenizer.encode_plus('The tokenizer has lots of functionality.', return_tensors='pt')
from pprint import pprint
pprint(tokenized_pt)
```

```
{'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]),
 'input_ids': tensor([[  101,  1996, 19204, 17629,  2038,  7167,  1997, 15380,  1012,   102]]),
 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])}
```

```
# One thing to note is that transformer-based models operate on wordpieces, not words
# Also note how it inserts a [CLS] token at the beginning and a [SEP] token at the end
print(tokenizer.convert_ids_to_tokens(tokenized['input_ids']))
```

```
['[CLS]', 'the', 'token', '##izer', 'has', 'lots', 'of', 'functionality', '.', '[SEP]']
```

BERT tokenizer

```python
# If we give it a list of texts, it will return a batch of results (and do padding!)
texts = ['This is the first sentence.',
         'This may be the second sentence, I really do not know.',
         'I never learned to count.']

tokenized_batch = tokenizer.batch_encode_plus(texts, return_tensors='pt', padding=True)
pprint(tokenized_batch)
```

```
{'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]]),
 'input_ids': tensor([[ 101, 2023, 2003, 1996, 2034, 6251, 1012,  102,    0,    0,    0,    0,
            0,    0,    0],
        [ 101, 2023, 2089, 2022, 1996, 2117, 6251, 1010, 1045, 2428, 2079, 2025,
         2113, 1012,  102],
        [ 101, 1045, 2196, 4342, 2000, 4175, 1012,  102,    0,    0,    0,    0,
            0,    0,    0]]),
 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])}
```

SST 2 dataset

```
display(dev_df)
```

	sentence	label
0	it 's a charming and often affecting journey .	1
1	unflinchingly bleak and desperate	0
2	allows us to hope that nolan is poised to emba...	1
3	the acting , costumes , music , cinematography...	1
4	it 's slow -- very , very slow .	0
...	...	...
867	has all the depth of a wading pool .	0
868	a movie with a real anarchic flair .	1
869	a subject like this should inspire reaction in...	0
870	... is an arthritic attempt at directing by ca...	0
871	looking aristocratic , luminous yet careworn i...	1

872 rows × 2 columns

SST-2 Dataset

```python
import torch
from torch.utils.data import Dataset

# With it being easy to generate batches of tokenized texts, it's actually easier
# not to do the tokenization beforehand, and just store texts
# It's a little bit slow though, so if you found this to be bottleneck
# you'd want to pre-tokenize everything and then batch/pad as necessary
class SST2TransformerDataset(Dataset):
    def __init__(self,
                 labels=None,
                 texts=None):

        self.y = torch.tensor(labels,dtype=torch.int64)
        self.texts = texts

    def __len__(self):
        return self.y.shape[0]

    def __getitem__(self, idx):
        rdict = {
            'y': self.y[idx],
            'text': self.texts[idx]
        }
        return rdict
```

SST-2 DataLoader

```python
# And then for our custom collate function we can just make use of that batch_encode_plus method
from typing import List, Dict, Union
from torch.utils.data import DataLoader

def SST2_transformer_collate(batch:List[Dict[str, Union[torch.Tensor,str]]]):
  y_batch = torch.tensor([example['y'] for example in batch])

  # We'll just reuse the tokenizer we created earlier, since it doesn't change
  tokenized_batch = tokenizer.batch_encode_plus([example['text'] for example in batch],
                                                  return_tensors='pt',
                                                  padding=True,
                                                  max_length=512,
                                                  truncation=True)

  return {
      'y':y_batch,
      'input_ids':tokenized_batch['input_ids'],
      'attention_mask':tokenized_batch['attention_mask']
  }

batch_size = 16
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, collate_fn = SST2_transformer_collate, shuffle=True)
dev_dataloader = DataLoader(dev_dataset, batch_size=batch_size, collate_fn = SST2_transformer_collate, shuffle=False)
```

BERT classifier model

```
class BertClassifier(pl.LightningModule):
  def __init__(self,
               learning_rate:float,
               num_classes:int,
               freeze_bert:bool=False,
               **kwargs):
    super().__init__(**kwargs)

    # Like with the LSTM, we'll define a central BERT we're gonna use
    # Again, this will download this from Hugging Face in the background
    self.bert = BertModel.from_pretrained('bert-base-uncased')

    # If we want to speed up training, we can freeze the BERT module and train
    # just the output layer. This will hurt accuracy though.
    if freeze_bert:
      for param in self.bert.parameters():
        param.requires_grad = False

    # Then the only other thing we need is an output layer, whose input size will
    # be the BERT's output size (768), which can can find as follows:
    self.output_layer = torch.nn.Linear(self.bert.config.hidden_size, num_classes)

    self.learning_rate = learning_rate
    self.train_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
    self.val_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
    self.test_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
```

BERT classifier model

```
def forward(self, y:torch.Tensor, input_ids:torch.Tensor,
            attention_mask:torch.Tensor):
    # And then the forward function is pretty simple--
    # way simpler than with the LSTM
    bert_result = self.bert(input_ids=input_ids,
                            attention_mask=attention_mask)

    # Typically we just use the pooler output for classification
    # Which, again, is the hidden state output for the [CLS] token
    cls_output = bert_result['pooler_output']

    py_logits = self.output_layer(cls_output)
    py = torch.argmax(py_logits, dim=1)
    loss = torch.nn.functional.cross_entropy(py_logits, y,
                                             reduction='mean')
    return {'py':py,
            'loss':loss}
```

BERT classifier model

```python
classifier_model = BertClassifier(learning_rate=2e-5,
                                  num_classes=2)
print('Model:')
print(classifier_model)
```

```
BertClassifier(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (output_layer): Linear(in_features=768, out_features=2, bias=True)
  (train_accuracy): MulticlassAccuracy()
  (val_accuracy): MulticlassAccuracy()
  (test_accuracy): MulticlassAccuracy()
)
```

BERT classifier training

```python
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks.progress import TQDMProgressBar

# And then training is easy with our old friend PyTorch Lightning
classifier_trainer = Trainer(
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None,
    max_epochs=1,
    callbacks=[TQDMProgressBar(refresh_rate=20)],
    val_check_interval = 0.2,
    )

classifier_trainer.fit(model=classifier_model,
                       train_dataloaders=train_dataloader,
                       val_dataloaders=dev_dataloader)
```

```
Epoch 0 step 842 validation accuracy: tensor(0.9106, device='cuda:0')


Epoch 0 step 1684 validation accuracy: tensor(0.9232, device='cuda:0')


Epoch 0 step 2526 validation accuracy: tensor(0.9174, device='cuda:0')


Epoch 0 step 3368 validation accuracy: tensor(0.9106, device='cuda:0')


Epoch 0 step 4210 validation accuracy: tensor(0.9128, device='cuda:0')
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=1` reached.
 Epoch 0 training accuracy: tensor(0.9208, device='cuda:0')
```

Other stuff we can do with BERT

Virtually the only thing BERT can't do well is generate text autoregressively (decode it).

That means there's a lot it can do, including:

- Sequence tagging
- Infilling missing tokens

Transformer tokenizers

Sometimes we want to run a Transformer tokenizer over a list of tokens rather than a text

```
[ ]
# Another useful thing to know is that if your text is already tokenized, you
# can have the tokenizer further tokenize it
tokens = ['transformer','tokenizers','are','cool','.']
token_tokenized = tokenizer.encode_plus(text=tokens,
                                        is_split_into_words=True)
pprint(token_tokenized)
print(tokenizer.convert_ids_to_tokens(token_tokenized['input_ids']))
```

```
{'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 'input_ids': [101, 10938, 2121, 19204, 17629, 2015, 2024, 4658, 1012, 102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]}
['[CLS]', 'transform', '##er', 'token', '##izer', '##s', 'are', 'cool', '.', '[SEP]']
```

Transformer tokenizer

```python
# The tokenizer returns a BatchEncoding object which extends Python dict,
# but also has some functionality for aligning the wordpieces with the original
# text (or tokens), which (spoiler alert) we'll need to do later

print(type(token_tokenized))

# One example of this functionality is .word_ids(), which links each wordpiece
# back to the original token in the input
print(token_tokenized.word_ids())
```

```
<class 'transformers.tokenization_utils_base.BatchEncoding'>
[None, 0, 0, 1, 1, 1, 2, 3, 4, None]
```

CoNLL 2003

Classic named-entity recognition (NER) dataset

Consists of consists of a series of news articles, with each word in each article tagged for part-of-speech, constituency, and named-entity membership

We're only using the named-entity tags in this example

```
# Example:
'''
"Bill (B-PER) Gates (I-PER) founded (O) Liberty (B-ORG) Mutual (I-ORG) in (O)
"the (B-LOC) Grand (I-LOC) Duchy (I-LOC) of (I-LOC) Luxembourg (I-LOC)"
'''

conll_url = "https://data.deepai.org/conll2003.zip"
train_file = "train.txt"
dev_file = "valid.txt"

# Associate each named entity class with an int
ner_tags = {'O': 0, 'B-PER': 1, 'I-PER': 2, 'B-ORG': 3, 'I-ORG': 4, 'B-LOC': 5, 'I-LOC': 6, 'B-MISC': 7, 'I-MISC': 8}
```

Getting CoNLL

```
[ ]  !wget $conll_url # this is a linux comand that will grab the file to the local directory
```

```
--2023-05-07 19:49:42--  https://data.deepai.org/conll2003.zip
Resolving data.deepai.org (data.deepai.org)... 169.150.236.100, 2400:52e0:1500::1021:1
Connecting to data.deepai.org (data.deepai.org)|169.150.236.100|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 982975 (960K) [application/zip]
Saving to: ‘conll2003.zip’

conll2003.zip       100%[===================>] 959.94K   834KB/s    in 1.2s

2023-05-07 19:50:42 (834 KB/s) - ‘conll2003.zip’ saved [982975/982975]
```

```
[ ]  # We can take a look at what we have
     !ls
```

```
conll2003.zip  lightning_logs  sample_data
```

```
[ ]  # We'll have to unzip it
     !unzip conll2003.zip
```

```
Archive:  conll2003.zip
  inflating: metadata
  inflating: test.txt
  inflating: train.txt
  inflating: valid.txt
```

Getting CoNLL

```
# You can use the `head` command to peak inside of a file on linux
# to see what the data looks like
!head -20 train.txt
```

```
-DOCSTART- -X- -X- O

EU NNP B-NP B-ORG
rejects VBZ B-VP O
German JJ B-NP B-MISC
call NN I-NP O
to TO B-VP O
boycott VB I-VP O
British JJ B-NP B-MISC
lamb NN I-NP O
. . O O

Peter NNP B-NP B-PER
Blackburn NNP I-NP I-PER

BRUSSELS NNP B-NP B-LOC
1996-08-22 CD I-NP O

The DT B-NP O
European NNP I-NP B-ORG
```

Preprocessing CoNLL

```
[ ]  # The first step here is to read these files into Pandas dataframes
     import pandas as pd
     train_token_df = pd.read_csv('train.txt', sep=' ', names=['token', 'pos_tag', 'chunk_tag', 'ner_tag'])
     val_token_df = pd.read_csv('valid.txt', sep=' ', names=['token', 'pos_tag', 'chunk_tag', 'ner_tag'])
```

```
[ ]  display(val_token_df)
```

	token	pos_tag	chunk_tag	ner_tag
0	-DOCSTART-	-X-	-X-	O
1	CRICKET	NNP	B-NP	O
2	-	:	O	O
3	LEICESTERSHIRE	NNP	B-NP	B-ORG
4	TAKE	NNP	I-NP	O
...	...	...	...	...
51573	.	.	O	O
51574	--	:	O	O
51575	Dhaka	NNP	B-NP	B-ORG
51576	Newsroom	NNP	I-NP	I-ORG
51577	880-2-506363	CD	I-NP	O

51578 rows × 4 columns

Preprocessing CoNLL2003

```
[ ]  # Okay, now we are cooking with gas
     val_seq_df
```

sequence_id	token	pos_tag	chunk_tag	ner_tag	ner_id
0	[CRICKET, -, LEICESTERSHIRE, TAKE, OVER, AT, T...	[NNP, :, NNP, NNP, IN, NNP, NNP, NNP, NNP, NN,...	[B-NP, O, B-NP, I-NP, B-PP, B-NP, I-NP, I-NP, ...	[O, O, B-ORG, O, O, O, O, O, O, O, O, B-LOC, O...	[0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 7, 8, ...
1	[CRICKET, -, ENGLISH, COUNTY, CHAMPIONSHIP, SC...	[NNP, :, JJ, NNS, WDT, NNP, ., NNP, CD, NN, CC...	[B-NP, O, B-NP, I-NP, B-NP, I-NP, O, B-NP, I-N...	[O, O, B-MISC, I-MISC, I-MISC, O, O, B-LOC, O,...	[0, 0, 7, 8, 8, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, ...
2	[CRICKET, -, 1997, ASHES, INTINERARY, ., LONDO...	[NNP, :, CD, NNP, NNP, ., NNP, CD, NNP, MD, VB...	[B-NP, O, B-NP, I-NP, I-NP, O, B-NP, I-NP, B-N...	[O, O, O, B-MISC, O, O, B-LOC, O, B-LOC, O, O,...	[0, 0, 0, 7, 0, 0, 5, 0, 5, 0, 0, 0, 7, 0, 0, ...
3	[SOCCER, -, SHEARER, NAMED, AS, ENGLAND, CAPTA...	[NN, :, NN, VBD, NNP, NNP, NNP, ., NNP, CD, DT...	[B-NP, O, B-NP, B-VP, B-NP, I-NP, I-NP, O, B-N...	[O, O, B-PER, O, O, B-LOC, O, O, B-LOC, O, O, ...	[0, 0, 1, 0, 0, 5, 0, 0, 5, 0, 0, 0, 0, 0, 0, ...
4	[BASKETBALL, -, INTERNATIONAL, TOURNAMENT, RES...	[NNP, :, NNP, NNP, NNP, ., VB, CD, NN, IN, DT,...	[B-NP, O, B-NP, I-NP, I-NP, O, B-VP, B-NP, B-N...	[O, O, O, O, O, O, B-LOC, O, O, O, O, O, O, O,...	[0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, ...
...	...	...	...	...	...
211	[Nato, declines, comment, on, fighting, in, Ir...	[NNP, VBZ, NN, IN, VBG, IN, NNP, ., NNP, NNP, ...	[B-NP, B-VP, B-NP, B-PP, B-VP, B-PP, B-NP, O, ...	[O, O, O, O, O, O, B-LOC, O, B-LOC, O, O, B-OR...	[0, 0, 0, 0, 0, 0, 5, 0, 5, 0, 0, 3, 4, 4, 4, ...
212	[More, automatic, weapons, stolen, in, Belgium...	[RBR, JJ, NNS, VBN, IN, NNP, ., NNP, NNP, JJR,...	[B-ADJP, I-ADJP, B-NP, B-VP, B-PP, B-NP, O, B-...	[O, O, O, O, O, B-LOC, O, B-LOC, O, O, O, O, O...	[0, 0, 0, 0, 0, 5, 0, 5, 0, 0, 0, 0, 0, 0, 0, ...
213	[No, trace, of, two, missing, teenagers, in, B...	[DT, NN, IN, CD, JJ, NNS, IN, NNP, ., NNP, NNP...	[B-NP, I-NP, B-PP, B-NP, I-NP, I-NP, B-PP, B-N...	[O, O, O, O, O, O, O, B-LOC, O, B-LOC, O, B-MI...	[0, 0, 0, 0, 0, 0, 0, 5, 0, 5, 0, 7, 0, 0, 0, ...
214	[Controversial, IRA, film, screened, at, Venic...	[NNP, NNP, NN, VBD, IN, NNP, NN, ., NNP, NNP, ...	[B-NP, I-NP, I-NP, B-VP, B-PP, B-NP, I-NP, O, ...	[O, B-ORG, O, O, O, B-LOC, O, O, B-PER, I-PER,...	[0, 3, 0, 0, 0, 5, 0, 0, 1, 2, 0, 0, 5, 0, 0, ...
215	[Dhaka, stocks, seen, steady, in, absence, of,...	[NNP, NNS, VBN, JJ, IN, NN, IN, JJ, NNS, ., NN...	[B-NP, I-NP, B-VP, B-ADJP, B-PP, B-NP, B-PP, B...	[B-LOC, O, O, O, O, O, O, O, O, O, B-LOC, O, O...	[5, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, ...

CoNLL Dataset

The PyTorch Dataset for CoNLL is very simple and standard. Just return a set of token and a set of NER labels

```python
import torch
from torch.utils.data import Dataset


class SequenceTaggingDataset(Dataset):
  def __init__(self,
               tokens=None,
               labels=None):

    self.tokens = tokens
    self.labels = labels

  def __len__(self):
    return self.tokens.shape[0]

  def __getitem__(self, idx):
    rdict = {
      'labels': self.labels[idx],
      'tokens': self.tokens[idx]
    }
    return rdict
```

CoNLL DataLoader

The DataLoader is more complicated because we have to align the (double) tokenized tokens with the token tags.

We'll use that extra tokenizer information I mentioned earlier to do that.

```
from typing import List, Dict, Union
from torch.utils.data import DataLoader

def sequence_tagging_collate(batch:List[Dict[str, Union[torch.Tensor,str]]]):
  # First we have the tokenizer tokenize the batch, with padding just like earlier
  # The documents in this dataset are long, so we have to do truncation
  tokenized_batch = tokenizer.batch_encode_plus([example['tokens'] for example in batch],
                                                 is_split_into_words=True,
                                                 max_length=512,
                                                 truncation=True,
                                                 return_tensors='pt',
                                                 padding=True)

  # We'll do the alignment by making a tensor the size of the input_ids tensor,
  # then filling its values in using the tokenizer.word_ids() alignment information
  labels = torch.zeros_like(tokenized_batch['input_ids'])
  for row_num in range(labels.shape[0]):
    row_words = tokenized_batch.word_ids(row_num)
    for col_num in range(labels.shape[1]):
      if row_words[col_num] is not None:
        labels[row_num,col_num] = batch[row_num]['labels'][row_words[col_num]]

  return {
      'input_ids':tokenized_batch['input_ids'],
      'attention_mask':tokenized_batch['attention_mask'],
      'labels': labels
  }
```

CoNLL sequence tagging model

```
class SequenceTaggingModel(pl.LightningModule):
  def __init__(self,
               learning_rate:float,
               num_classes:int,
               **kwargs):
    super().__init__(**kwargs)
    # The elements of a tagger model are identical to those of a classifier
    self.bert = BertModel.from_pretrained('bert-base-uncased')
    self.output_layer = torch.nn.Linear(self.bert.config.hidden_size, num_classes)
    self.learning_rate = learning_rate
    self.train_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
    self.val_accuracy = Accuracy(task='multiclass', num_classes=num_classes)

    # Only difference is that we'll also take a look at the F1 for each class, to
    # look at some problems with accuracy as a metric for sequence tagging
    self.train_f1 = F1Score(task='multiclass', num_classes = num_classes, average='none')
    self.val_f1 = F1Score(task='multiclass', num_classes = num_classes, average='none')
```

CoNLL sequence tagging model

```python
def forward(self,
            input_ids:torch.Tensor,
            attention_mask:torch.Tensor,
            labels:torch.Tensor):

  bert_result = self.bert(input_ids=input_ids,
                          attention_mask=attention_mask)

  # This only difference between this and a classifier is that we run the
  # output layer on the 'last_hidden_state' rather than 'pooler_output', so that
  # we end up with one prediction per wordpiece, rather than one per sequence
  last_hidden = bert_result['last_hidden_state']
  predicted_logits = self.output_layer(last_hidden)
  predicted_labels = torch.argmax(predicted_logits, dim=2)

  loss = torch.nn.functional.cross_entropy(predicted_logits.transpose(1,2),
                                           labels,
                                           reduction='mean')
  return {'predicted_labels':predicted_labels,
          'loss':loss}
```

CoNLL sequence tagging model

```
# Then do all the usual PyTorch Lightning functions
def configure_optimizers(self):
  return [torch.optim.Adam(self.parameters(), lr=self.learning_rate)]

def training_step(self, batch, batch_idx):
  result = self.forward(**batch)
  loss = result['loss']
  self.log('train_loss', result['loss'])
  self.train_accuracy.update(result['predicted_labels'], batch['labels'])
  self.train_f1.update(result['predicted_labels'], batch['labels'])
  return loss

def training_epoch_end(self, outs):
  # print(f'Epoch {self.current_epoch} training accuracy:', self.train_accuracy.compute())
  self.train_accuracy.reset()
  # print(f'Epoch {self.current_epoch} training F1s:', self.train_f1.compute())
  # print(f'Epoch {self.current_epoch} training mean F1:', self.train_f1.compute().mean())
  self.train_f1.reset()
```

CoNLL sequence tagging model

```
def validation_step(self, batch, batch_idx):
    result = self.forward(**batch)
    self.val_accuracy.update(result['predicted_labels'], batch['labels'])
    self.val_f1.update(result['predicted_labels'], batch['labels'])
    return result['loss']

def validation_epoch_end(self, outs):
    print(f'Epoch {self.current_epoch} step {self.global_step} validation accuracy:', self.val_accuracy.compute())
    self.val_accuracy.reset()
    print(f'Epoch {self.current_epoch} step {self.global_step} validation F1s:', self.val_f1.compute())
    print(f'Epoch {self.current_epoch} step {self.global_step} validation mean F1:', self.val_f1.compute().mean())
    self.val_f1.reset()
```

CoNLL model training

```python
# Training is identical to the classifier. Hooray for Pytorch Lightning.
seq_trainer = Trainer(
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None,
    max_epochs=3,
    callbacks=[TQDMProgressBar(refresh_rate=20)],
    val_check_interval = 0.2,
    )

seq_trainer.fit(model=seq_tag_model,
                train_dataloaders=train_seq_dataloader,
                val_dataloaders=val_seq_dataloader)
```

```
Epoch 0 step 19 validation accuracy: tensor(0.8916, device='cuda:0')
Epoch 0 step 19 validation F1s: tensor([0.9427, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
       device='cuda:0')
Epoch 0 step 19 validation mean F1: tensor(0.1047, device='cuda:0')

...

Epoch 2 step 285 validation accuracy: tensor(0.9880, device='cuda:0')
Epoch 2 step 285 validation F1s: tensor([0.9968, 0.9749, 0.9797, 0.9135, 0.8598, 0.9413, 0.8098, 0.7978, 0.6095],
       device='cuda:0')
Epoch 2 step 285 validation mean F1: tensor(0.8759, device='cuda:0')
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=3` reached.
```

Masked language modeling

Even though BERT isn't trained autoregressively, it is trained to in-fill missing words based on the surrounding context.

…and we can try that aspect of the model out!

BERT masked language modeling

```
from transformers import BertForMaskedLM
# We instantiate the model the same way, just using the new class
lm_bert = BertForMaskedLM.from_pretrained('bert-base-uncased')
```

BERT masked language modeling

```python
# We can reuse the same tokenizer
masked_sentence = 'What will the missing [MASK] in this sentence be?'

tokenized_masked = tokenizer.batch_encode_plus([masked_sentence], return_tensors='pt')
print(tokenized_masked)
```

```
{'input_ids': tensor([[ 101, 2054, 2097, 1996, 4394,  103, 1999, 2023, 6251, 2022, 1029,  102]]),
'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1]])}
```

```python
# '[MASK]' is another special token in the BERT vocabulary
print(tokenizer.convert_ids_to_tokens(tokenized_masked['input_ids'][0]))
```

```
['[CLS]', 'what', 'will', 'the', 'missing', '[MASK]', 'in', 'this', 'sentence', 'be', '?', '[SEP]']
```

BERT masked language modeling

```python
# We can use the model the same way
with torch.no_grad():
    masked_output = lm_bert(input_ids=tokenized_masked['input_ids'],
                            attention_mask=tokenized_masked['attention_mask'])

# Note how instead of a 'last_hidden_state' or 'pooler_output', we now have 'logits'
print(masked_output)

# The dimensionality of the logits is batch size x sequence length x vocab size
print(masked_output['logits'].shape)
```

```
MaskedLMOutput(loss=None, logits=tensor([[[ -6.7112,  -6.6641,  -6.6809,  ...,  -5.9996,  -5.8256,  -4.1387],
         [-13.5474, -13.9943, -13.7179,  ..., -13.4256, -12.6709, -10.0079],
         [-10.2931, -10.2157,  -9.7521,  ...,  -9.0000,  -6.3156,  -6.8798],
         ...,
         [-12.1131, -11.9527, -12.4436,  ..., -10.3881,  -9.2020,  -9.7878],
         [ -9.4531,  -9.0963,  -9.3015,  ...,  -8.0880,  -8.5506,  -3.7322],
         [-12.5165, -12.1793, -12.2511,  ...,  -9.8303, -10.1499, -10.0455]]]), hidden_states=None,
attentions=None)
torch.Size([1, 12, 30522])
```

BERT masked language modeling

```
# We can use the model the same way
with torch.no_grad():
    masked_output = lm_bert(input_ids=tokenized_masked['input_ids'],
                            attention_mask=tokenized_masked['attention_mask'])

# Note how instead of a 'last_hidden_state' or 'pooler_output', we now have 'logits'
print(masked_output)

# The dimensionality of the logits is batch size x sequence length x vocab size
print(masked_output['logits'].shape)
```

```
MaskedLMOutput(loss=None, logits=tensor([[[ -6.7112,  -6.6641,  -6.6809,  ...,  -5.9996,  -5.8256,  -4.1387],
         [-13.5474, -13.9943, -13.7179,  ..., -13.4256, -12.6709, -10.0079],
         [-10.2931, -10.2157,  -9.7521,  ...,  -9.0000,  -6.3156,  -6.8798],
         ...,
         [-12.1131, -11.9527, -12.4436,  ..., -10.3881,  -9.2020,  -9.7878],
         [ -9.4531,  -9.0963,  -9.3015,  ...,  -8.0880,  -8.5506,  -3.7322],
         [-12.5165, -12.1793, -12.2511,  ...,  -9.8303, -10.1499, -10.0455]]]), hidden_states=None,
attentions=None)
torch.Size([1, 12, 30522])
```

BERT masked language modeling

```python
# The logits represent a prediction of the most likely word in each position.
# We can decode these predictions by mapping the logits back to the vocabulary

# An optional first step is to convert the logits to probabilities using a softmax
masked_probs = torch.softmax(masked_output['logits'], dim=2)

# Then we can find the index of the top probability for each word
max_idxs = torch.argmax(masked_probs, dim=2)

# And then those indexes will just be word IDs, so we can map them back to words
# using the tokenizer
sequences = [' '.join([str(token) for token in tokenizer.convert_ids_to_tokens(row,
skip_special_tokens=True)]) for row in max_idxs]

# And hey, look! It correctly predicted the word 'word'!
# It also predicted a '.' for the [CLS] and [SEP], which is interesting
# We can just ignore those outputs.
print(sequences)
```

```
['. what will the missing word in this sentence be ? .']
```

BERT masked language modeling

```python
# If we do an argsort on the logits for that one slot, we can see what our other options
# were, and how likely the model thought they were

# We'll grab the top 15

sorted_idxs = torch.argsort(masked_probs[0,5,:], descending=True)[0:15]

# Looks right to me!
for idx in sorted_idxs:
  prob  =masked_probs[0,5,idx]
  print(f'Word: "{tokenizer.convert_ids_to_tokens([idx])[0]}"; Prob: {prob:.4f}')
```

```
Word: "word"; Prob: 0.5044
Word: "words"; Prob: 0.1150
Word: "sentence"; Prob: 0.1145
Word: "consonant"; Prob: 0.0183
Word: "item"; Prob: 0.0135
Word: "information"; Prob: 0.0129
Word: "line"; Prob: 0.0128
Word: "part"; Prob: 0.0102
Word: "thing"; Prob: 0.0093
Word: "link"; Prob: 0.0093
Word: "piece"; Prob: 0.0091
Word: "ingredient"; Prob: 0.0083
Word: "vowel"; Prob: 0.0063
Word: "clue"; Prob: 0.0060
Word: "phrase"; Prob: 0.0055
```

BERT masked language modeling

```python
# We can use the torch.multinomial function to sample from this probability distribution

for i in range(15):
    sampled_idx = torch.multinomial(masked_probs[0,5,:],1)
    sampled_word = tokenizer.convert_ids_to_tokens(sampled_idx)[0]
    print(f'Sampled word: "{sampled_word}"')
```

```
Sampled word: "word"
Sampled word: "word"
Sampled word: "word"
Sampled word: "faces"
Sampled word: "word"
Sampled word: "item"
Sampled word: "bit"
Sampled word: "word"
Sampled word: "word"
Sampled word: "word"
Sampled word: "word"
Sampled word: "word"
Sampled word: "sentence"
Sampled word: "word"
Sampled word: "word"
```

BERT masked language modeling

```python
# Adding a "temperature" causes the distribution to be flattened out, leading to more
# randomness

temperature = 1.5
temperature_probs = torch.softmax(masked_output['logits'][0,5,:] / temperature, dim=0)

for i in range(15):
  sampled_idx = torch.multinomial(temperature_probs,1)
  sampled_word = tokenizer.convert_ids_to_tokens(sampled_idx)[0]
  print(f'Sampled word: "{sampled_word}"')
```

```
Sampled word: "paragraph"
Sampled word: "word"
Sampled word: "words"
Sampled word: "moisture"
Sampled word: "words"
Sampled word: "entry"
Sampled word: "relative"
Sampled word: "silence"
Sampled word: "creepy"
Sampled word: "hydrogen"
Sampled word: "word"
Sampled word: "word"
Sampled word: "additional"
Sampled word: "word"
Sampled word: "horseshoe"
```

Concluding thoughts

Pretrained transformer models

- BERT, RoBERTa, XLNet, RoBERTa, DistilBERT, T5, GPT-X

Encoder-decoder, encoder-only, decoder-only

How to choose?

Looking forward:

- Zero- and few-shot learning
- Prompt engineering