Transfer Learning in Natural Language Processing

June 2, 2019
NAACL-HLT 2019
Follow along with the tutorial:

- Slides: [http://tiny.cc/NAACLTransfer](http://tiny.cc/NAACLTransfer)
- Colab: [http://tiny.cc/NAACLTransferColab](http://tiny.cc/NAACLTransferColab)

Questions:

- Twitter: [#NAACLTransfer](#NAACLTransfer) during the tutorial
- Whova: “Questions for the tutorial on Transfer Learning in NLP” topic
- Ask us during the break or after the tutorial
What is transfer learning?

Pan and Yang (2010)
Why transfer learning in NLP?

- Many NLP tasks share common knowledge about language (e.g. linguistic representations, structural similarities)
- Tasks can inform each other—e.g. syntax and semantics
- Annotated data is rare, make use of as much supervision as available.

- Empirically, transfer learning has resulted in SOTA for many supervised NLP tasks (e.g. classification, information extraction, Q&A, etc).
Why transfer learning in NLP? (Empirically)

Performance on Named Entity Recognition (NER) on CoNLL-2003 (English) over time

- CNN Large + fine-tune: 93.5
- Flair embeddings: 93.09
- BERT Large: 92.8
- Cross-view + Multi-Task: 92.61
- BERT Base: 92.4
- TagLM: 91.93
- LSTM-CRF: 91.24
- LM-LSTM-CRF: 91.24
- Ma and Hovy LSTM-CNN-CRF: 91.21
- Yang et al.: 91.26

- Chiu and Nichols 2015: 90.69
- Passos et al. 2014: 90.05
- Collobert et al. 2011: 89.59
- Florian et al., 2003: 88.76
- Ando and Zhang, 2005 co- and self-supervision: 89.31
- Lin and Wu, 2009 Phrase & word clusters: 90.90

Types of transfer learning in NLP

Transfer learning

- Same task; labeled data only in source domain
- Different tasks; labeled data in target domain

Transductive transfer learning

- Different domains
- Different languages

- Cross-lingual learning

Inductive transfer learning

- Tasks learned simultaneously
- Tasks learned sequentially

Multi-task learning

- Sequential transfer learning

Domain adaptation

We will focus on this

Ruder (2019)
What this tutorial is about and what it’s not about

- Goal: provide broad overview of transfer methods in NLP, focusing on the most empirically successful methods as of today (mid 2019)
- Provide practical, hands on advice → by end of tutorial, everyone has ability to apply recent advances to text classification task

- What this is not: **Comprehensive** (it’s impossible to cover all related papers in one tutorial!)
- (Bender Rule: This tutorial is mostly for work done in English, extensibility to other languages depends on availability of data and resources.)
Agenda

[1] Introduction

[2] Pretraining


[4] Adaptation

[5] Downstream

1. Introduction
Sequential transfer learning
Learn on one task / dataset, then transfer to another task / dataset

Pretraining:
- word2vec
- GloVe
- skip-thought
- InferSent
- ELMo
- ULMFiT
- GPT
- BERT

Adaptation:
- classification
- sequence labeling
- Q&A
- ....
Pretraining tasks and datasets

- Unlabeled data and self-supervision
  - Easy to gather very large corpora: Wikipedia, news, web crawl, social media, etc.
  - Training takes advantage of distributional hypothesis: “You shall know a word by the company it keeps” (Firth, 1957), often formalized as training some variant of language model
  - Focus on efficient algorithms to make use of plentiful data

- Supervised pretraining
  - Very common in vision, less in NLP due to lack of large supervised datasets
  - Machine translation
  - NLI for sentence representations
  - Task-specific—transfer from one Q&A dataset to another
Target tasks and datasets

Target tasks are typically supervised and span a range of common NLP tasks:

- Sentence or document classification (e.g. sentiment)
- Sentence pair classification (e.g. NLI, paraphrase)
- Word level (e.g. sequence labeling, extractive Q&A)
- Structured prediction (e.g. parsing)
- Generation (e.g. dialogue, summarization)
Concrete example—word vectors

Word embedding methods (e.g. word2vec) learn one vector per word:

cat = [0.1, -0.2, 0.4, ...]
dog = [0.2, -0.1, 0.7, ...]
Concrete example—word vectors

Word embedding methods (e.g. word2vec) learn one vector per word:

cat = [0.1, -0.2, 0.4, ...]
dog = [0.2, -0.1, 0.7, ...]

I love my cat and dog.
Concrete example—word vectors

Word embedding methods (e.g. word2vec) learn one vector per word:

cat = [0.1, -0.2, 0.4, ...]
dog = [0.2, -0.1, 0.7, ...]
Major Themes
### Word vectors

<table>
<thead>
<tr>
<th>Word</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>cats</td>
<td>[0.2, -0.3,...]</td>
</tr>
<tr>
<td>dogs</td>
<td>[0.4, -0.5,...]</td>
</tr>
</tbody>
</table>

### Sentence / doc vectors

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>We have two cats.</td>
<td>[-1.2, 0.0,...]</td>
</tr>
<tr>
<td>It’s raining cats and dogs.</td>
<td>[0.8, 0.9,...]</td>
</tr>
</tbody>
</table>

### Word-in-context vectors

<table>
<thead>
<tr>
<th>Sentence</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>We have two cats.</td>
<td>[1.2, -0.3,...]</td>
</tr>
<tr>
<td>It’s raining cats and dogs.</td>
<td>[-0.4, 0.9,...]</td>
</tr>
</tbody>
</table>
Major themes: LM pretraining

- Many successful pretraining approaches are based on language modeling
- Informally, a LM learns $P_\theta(text)$ or $P_\theta(text \mid \text{some other text})$
- Doesn’t require human annotation
- Many languages have enough text to learn high capacity model
- Versatile—can learn both sentence and word representations with a variety of objective functions
Major themes: From shallow to deep

Bengio et al 2003: A Neural Probabilistic Language Model

Devlin et al 2019: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
Major themes: pretraining vs target task

Choice of pretraining and target tasks are coupled

- Sentence / document representations not useful for word level predictions
- Word vectors can be pooled across contexts, but often outperformed by other methods
- In contextual word vectors, bidirectional context important

In general:

- Similar pretraining and target tasks $\rightarrow$ best results
Agenda

[1] Introduction

[2] Pretraining

[3] What’s in a representation?

[4] Adaptation

[5] Downstream

2. Pretraining
Overview

- Language model pretraining
- Word vectors
- Sentence and document vectors
- Contextual word vectors
- Interesting properties of pretraining
- Cross-lingual pretraining
Word Type Representation

LM pretraining

word2vec, Mikolov et al (2013)

We [have a ??? and three] dogs

ELMo, Peters et al. 2018, ULMFiT (Howard & Ruder 2018), GPT (Radford et al. 2018)

We have a ???

We like pets. }

Skip-Thought (Kiros et al., 2015)

BERT, Devlin et al 2019

We have a MASK and three dogs
Word vectors
Why embed words?

- Embeddings are themselves parameters—can be learned
- Sharing representations across tasks
- Lower dimensional space
  - Better for computation—difficult to handle sparse vectors.
Unsupervised pretraining: Pre-Neural

Latent Semantic Analysis (LSA)—SVD of term-document matrix, (Deerwester et al., 1990)

Brown clusters, hard hierarchical clustering based on n-gram LMs, (Brown et al. 1992)

Latent Dirichlet Allocation (LDA)—Documents are mixtures of topics and topics are mixtures of words (Blei et al., 2003)
Word vector pretraining

n-gram neural language model
(Bengio et al. 2003)

Supervised multitask word embeddings
(Collobert and Weston, 2008)
word2vec

Efficient algorithm + large scale training → high quality word vectors

(Mikolov et al., 2013)

See also:
- Pennington et al. (2014): GloVe
- Bojanowski et al. (2017): fastText
Sentence and document vectors
Paragraph vector

Unsupervised paragraph embeddings (Le & Mikolov, 2014)

SOTA classification (IMDB, SST)

<table>
<thead>
<tr>
<th>Model</th>
<th>Error rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoW (bnc) (Maas et al., 2011)</td>
<td>12.20 %</td>
</tr>
<tr>
<td>BoW (bΔt‘c) (Maas et al., 2011)</td>
<td>11.77 %</td>
</tr>
<tr>
<td>LDA (Maas et al., 2011)</td>
<td>32.58 %</td>
</tr>
<tr>
<td>Full+BoW (Maas et al., 2011)</td>
<td>11.67 %</td>
</tr>
<tr>
<td>Full+Unlabeled+BoW (Maas et al., 2011)</td>
<td>11.11 %</td>
</tr>
<tr>
<td>WRRBM (Dahl et al., 2012)</td>
<td>12.58 %</td>
</tr>
<tr>
<td>WRRBM + BoW (bnc) (Dahl et al., 2012)</td>
<td>10.77 %</td>
</tr>
<tr>
<td>MNB-uni (Wang &amp; Manning, 2012)</td>
<td>16.45 %</td>
</tr>
<tr>
<td>MNB-bi (Wang &amp; Manning, 2012)</td>
<td>13.41 %</td>
</tr>
<tr>
<td>SVM-uni (Wang &amp; Manning, 2012)</td>
<td>13.05 %</td>
</tr>
<tr>
<td>SVM-bi (Wang &amp; Manning, 2012)</td>
<td>10.84 %</td>
</tr>
<tr>
<td>NBSVM-uni (Wang &amp; Manning, 2012)</td>
<td>11.71 %</td>
</tr>
<tr>
<td>NBSVM-bi (Wang &amp; Manning, 2012)</td>
<td>8.78 %</td>
</tr>
<tr>
<td>Paragraph Vector</td>
<td>7.42 %</td>
</tr>
</tbody>
</table>
Skip-Thought Vectors

Predict previous / next sentence with seq2seq model (Kiros et al., 2015)

Hidden state of encoder transfers to sentence tasks (classification, semantic similarity)

<table>
<thead>
<tr>
<th>Method</th>
<th>MR</th>
<th>CR</th>
<th>SUBJ</th>
<th>MPQA</th>
<th>TREC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NB-SVM [41]</td>
<td>79.4</td>
<td>81.8</td>
<td>93.2</td>
<td>86.3</td>
<td></td>
</tr>
<tr>
<td>MNB [41]</td>
<td>79.0</td>
<td>80.0</td>
<td>93.6</td>
<td>86.3</td>
<td></td>
</tr>
<tr>
<td>cBoW [6]</td>
<td>77.2</td>
<td>79.9</td>
<td>91.3</td>
<td>86.4</td>
<td>87.3</td>
</tr>
<tr>
<td>GrConv [6]</td>
<td>76.3</td>
<td>81.3</td>
<td>89.5</td>
<td>84.5</td>
<td>88.4</td>
</tr>
<tr>
<td>RNN [6]</td>
<td>77.2</td>
<td>82.3</td>
<td>93.7</td>
<td>90.1</td>
<td>90.2</td>
</tr>
<tr>
<td>BRNN [6]</td>
<td>82.3</td>
<td>82.6</td>
<td>94.2</td>
<td>90.3</td>
<td>91.0</td>
</tr>
<tr>
<td>CNN [4]</td>
<td>81.5</td>
<td>85.0</td>
<td>93.4</td>
<td>89.6</td>
<td><strong>93.6</strong></td>
</tr>
<tr>
<td>AdaSent [6]</td>
<td><strong>83.1</strong></td>
<td><strong>86.3</strong></td>
<td><strong>95.5</strong></td>
<td><strong>93.3</strong></td>
<td>92.4</td>
</tr>
<tr>
<td>Paragraph-vector [7]</td>
<td>74.8</td>
<td>78.1</td>
<td>90.5</td>
<td>74.2</td>
<td>91.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combination</th>
<th>MR</th>
<th>CR</th>
<th>SUBJ</th>
<th>MPQA</th>
<th>TREC</th>
</tr>
</thead>
<tbody>
<tr>
<td>uni-skip</td>
<td>75.5</td>
<td>79.3</td>
<td>92.1</td>
<td>86.9</td>
<td>91.4</td>
</tr>
<tr>
<td>bi-skip</td>
<td>73.9</td>
<td>77.9</td>
<td>92.5</td>
<td>83.3</td>
<td>89.4</td>
</tr>
<tr>
<td>combine-skip</td>
<td>76.5</td>
<td>80.1</td>
<td>93.6</td>
<td>87.1</td>
<td><strong>92.2</strong></td>
</tr>
<tr>
<td>combine-skip + NB</td>
<td>80.4</td>
<td>81.3</td>
<td>93.6</td>
<td>87.5</td>
<td></td>
</tr>
</tbody>
</table>
Autoencoder pretraining

Dai & Le (2015): Pretrain a sequence autoencoder (SA) and generative LM

SOTA classification (IMDB)

<table>
<thead>
<tr>
<th>Model</th>
<th>Test error rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM with tuning and dropout</td>
<td>13.50%</td>
</tr>
<tr>
<td>LSTM initialized with word2vec embeddings</td>
<td>10.00%</td>
</tr>
<tr>
<td>LM-LSTM (see Section 2)</td>
<td>7.64%</td>
</tr>
<tr>
<td>SA-LSTM (see Figure 1)</td>
<td>7.24%</td>
</tr>
<tr>
<td>SA-LSTM with linear gain (see Section 3)</td>
<td>9.17%</td>
</tr>
<tr>
<td>SA-LSTM with joint training (see Section 3)</td>
<td>14.70%</td>
</tr>
<tr>
<td>Full+Unlabeled+BoW [21]</td>
<td>11.11%</td>
</tr>
<tr>
<td>WRRBM + BoW (bnc) [21]</td>
<td>10.77%</td>
</tr>
<tr>
<td>NBSVM-bi (Naïve Bayes SVM with bigrams) [35]</td>
<td>8.78%</td>
</tr>
<tr>
<td>seq2-bown-CNN (ConvNet with dynamic pooling) [11]</td>
<td>7.67%</td>
</tr>
<tr>
<td>Paragraph Vectors [18]</td>
<td>7.42%</td>
</tr>
</tbody>
</table>

See also:
- Socher et. al (2011): Semi-supervised recursive auto encoder
- Bowman et al. (2016): Variational autoencoder (VAE)
- Hill et al. (2016): Denoising autoencoder
Supervised sentence embeddings

Also possible to train sentence embeddings with supervised objective

- Paragraph-phrase: uses paraphrase database for supervision, best for paraphrase and semantic similarity (Wieting et al. 2016)
- InferSent: bi-LSTM trained on SNLI + MNLI (Conneau et al. 2017)
- GenSen: multitask training (skip-thought, machine translation, NLI, parsing) (Subramanian et al. 2018)
Contextual word vectors
Contextual word vectors - Motivation

Word vectors compress all contexts into a *single vector*

Nearest neighbor GloVe vectors to “**play**”

**VERB**  
playing  
played  

**NOUN**  
game  
games  
players  
football  

**ADJ**  
multiplayer  

??  
plays  
Play
Contextual word vectors - Key Idea

Instead of learning one vector per word, learn a vector that depends on context

\[ f(\text{play} \mid \text{The kids play a game in the park.}) \]

\[ \neq \]

\[ f(\text{play} \mid \text{The Broadway play premiered yesterday.}) \]

Many approaches based on language models
context2vec

Use bidirectional LSTM and cloze prediction objective (a 1 layer masked LM)

Learn representations for both words and contexts (minus word)

Sentence completion
Lexical substitution
WSD

<table>
<thead>
<tr>
<th></th>
<th>c2v iteration+</th>
<th>c2v</th>
<th>AWE</th>
<th>S-1</th>
<th>S-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCSS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test</td>
<td>64.0</td>
<td>62.7</td>
<td>48.4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>all</td>
<td>65.1</td>
<td>61.3</td>
<td>49.7</td>
<td>58.9</td>
<td>56.2</td>
</tr>
<tr>
<td>LST-07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test</td>
<td>56.1</td>
<td>54.8</td>
<td>41.9</td>
<td>55.2</td>
<td>-</td>
</tr>
<tr>
<td>all</td>
<td>56.0</td>
<td>54.6</td>
<td>42.5</td>
<td>55.1</td>
<td>53.6</td>
</tr>
<tr>
<td>LST-14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test</td>
<td>47.7</td>
<td>47.3</td>
<td>38.1</td>
<td>50.0</td>
<td>-</td>
</tr>
<tr>
<td>all</td>
<td>47.9</td>
<td>47.5</td>
<td>38.9</td>
<td>50.2</td>
<td>48.3</td>
</tr>
<tr>
<td>SE-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>test</td>
<td>72.8</td>
<td>71.2</td>
<td>61.4</td>
<td>74.1</td>
<td>73.6</td>
</tr>
</tbody>
</table>

(Melamud et al., CoNLL 2016)
TagLM

Pretrain two LMs (forward and backward) and add to sequence tagger. SOTA NER and chunking results

<table>
<thead>
<tr>
<th>Model</th>
<th>$F_1 \pm \text{std}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chiu and Nichols (2016)</td>
<td>90.91 ± 0.20</td>
</tr>
<tr>
<td>Lample et al. (2016)</td>
<td>90.94</td>
</tr>
<tr>
<td>Ma and Hovy (2016)</td>
<td>91.37</td>
</tr>
<tr>
<td>Our baseline without LM</td>
<td>90.87 ± 0.13</td>
</tr>
<tr>
<td>TagLM</td>
<td>91.93 ± 0.19</td>
</tr>
</tbody>
</table>

Table 1: Test set $F_1$ comparison on CoNLL 2003 NER task, using only CoNLL 2003 data and unlabeled text.

(Peters et al. ACL 2017)
Unsupervised Pretraining for Seq2Seq

Pretrain encoder and decoder with LMs (everything shaded is pretrained).

Large boost for MT.

(Ramachandran et al, EMNLP 2017)
Pretrain bidirectional encoder with MT supervision, extract LSTM states

Adding CoVe with GloVe gives improvements for classification, NLI, Q&A

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Random</th>
<th>GloVe</th>
<th>Char</th>
<th>CoVe-S</th>
<th>CoVe-M</th>
<th>CoVe-L</th>
<th>Char+CoVe-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>SST-2</td>
<td>84.2</td>
<td>88.4</td>
<td>90.1</td>
<td>89.0</td>
<td>90.9</td>
<td>91.1</td>
<td>91.2</td>
</tr>
<tr>
<td>SST-5</td>
<td>48.6</td>
<td>53.5</td>
<td>52.2</td>
<td>54.0</td>
<td>54.7</td>
<td>54.5</td>
<td>55.2</td>
</tr>
<tr>
<td>IMDb</td>
<td>88.4</td>
<td>91.1</td>
<td>91.3</td>
<td>90.6</td>
<td>91.6</td>
<td>91.7</td>
<td>92.1</td>
</tr>
<tr>
<td>TREC-6</td>
<td>88.9</td>
<td>94.9</td>
<td>94.7</td>
<td>94.7</td>
<td>95.1</td>
<td>95.8</td>
<td>95.8</td>
</tr>
<tr>
<td>TREC-50</td>
<td>81.9</td>
<td>89.2</td>
<td>89.8</td>
<td>89.6</td>
<td>89.6</td>
<td>90.5</td>
<td>91.2</td>
</tr>
<tr>
<td>SNLI</td>
<td>82.3</td>
<td>87.7</td>
<td>87.7</td>
<td>87.3</td>
<td>87.5</td>
<td>87.9</td>
<td>88.1</td>
</tr>
<tr>
<td>SQuAD</td>
<td>65.4</td>
<td>76.0</td>
<td>78.1</td>
<td>76.5</td>
<td>77.1</td>
<td>79.5</td>
<td>79.9</td>
</tr>
</tbody>
</table>

(McCann et al, NeurIPS 2017)
Pretrain deep bidirectional LM, extract contextual word vectors as learned linear combination of hidden states

SOTA for 6 diverse tasks

<table>
<thead>
<tr>
<th>TASK</th>
<th>PREVIOUS SOTA</th>
<th>OUR ELMo + BASELINE 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>
</tr>
<tr>
<td>SNLI</td>
<td>Chen et al. (2017)</td>
<td>88.6</td>
<td>88.0</td>
</tr>
<tr>
<td>SRL</td>
<td>He et al. (2017)</td>
<td>81.7</td>
<td>81.4</td>
</tr>
<tr>
<td>Coref</td>
<td>Lee et al. (2017)</td>
<td>67.2</td>
<td>67.2</td>
</tr>
<tr>
<td>NER</td>
<td>Peters et al. (2017)</td>
<td>91.93 ± 0.19</td>
<td>90.15</td>
</tr>
<tr>
<td>SST-5</td>
<td>McCann et al. (2017)</td>
<td>53.7</td>
<td>51.4</td>
</tr>
</tbody>
</table>

(Peters et al, NAACL 2018)
Pretrain AWD-LSTM LM, fine-tune LM in two stages with different adaptation techniques

SOTA for six classification datasets

(Howard and Ruder, ACL 2018)
Pretrain large 12-layer left-to-right Transformer, fine tune for sentence, sentence pair and multiple choice questions.

SOTA results for 9 tasks.

<table>
<thead>
<tr>
<th>Method</th>
<th>MNLI-m</th>
<th>MNLI-mm</th>
<th>SNLI</th>
<th>Sci'Tail</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>

(Radford et al., 2018)
BERT pretrains both sentence and contextual word representations, using masked LM and next sentence prediction. BERT-large has 340M parameters, 24 layers!

See also: Logeswaran and Lee, ICLR 2018

(Devlin et al. 2019)
BERT

SOTA GLUE benchmark results (sentence pair classification).

<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></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>
</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&lt;sub&gt;BASE&lt;/sub&gt;</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&lt;sub&gt;LARGE&lt;/sub&gt;</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. 2019)
# BERT

SOTA SQuAD v1.1 (and v2.0) Q&A

<table>
<thead>
<tr>
<th>System</th>
<th>Dev EM</th>
<th>Dev F1</th>
<th>Test EM</th>
<th>Test F1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Top Leaderboard Systems (Dec 10th, 2018)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Human</td>
<td>-</td>
<td>-</td>
<td>82.3</td>
<td>91.2</td>
</tr>
<tr>
<td>#1 Ensemble - nlnet</td>
<td>-</td>
<td>-</td>
<td>86.0</td>
<td>91.7</td>
</tr>
<tr>
<td>#2 Ensemble - QANet</td>
<td>-</td>
<td>-</td>
<td>84.5</td>
<td>90.5</td>
</tr>
<tr>
<td><strong>Published</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BiDAF+ELMo (Single)</td>
<td>-</td>
<td>85.6</td>
<td>-</td>
<td>85.8</td>
</tr>
<tr>
<td>R.M. Reader (Ensemble)</td>
<td>81.2</td>
<td>87.9</td>
<td>82.3</td>
<td>88.5</td>
</tr>
<tr>
<td><strong>Ours</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{BERT}_\text{BASE}$ (Single)</td>
<td>80.8</td>
<td>88.5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\text{BERT}_\text{LARGE}$ (Single)</td>
<td>84.1</td>
<td>90.9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\text{BERT}_\text{LARGE}$ (Ensemble)</td>
<td>85.8</td>
<td>91.8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\text{BERT}_\text{LARGE}$ (Sgl.+TriviaQA)</td>
<td>84.2</td>
<td>91.1</td>
<td>85.1</td>
<td>91.8</td>
</tr>
<tr>
<td>$\text{BERT}_\text{LARGE}$ (Ens.+TriviaQA)</td>
<td>86.2</td>
<td>92.2</td>
<td>87.4</td>
<td>93.2</td>
</tr>
</tbody>
</table>

(Devlin et al. 2019)
Other pretraining objectives

- Contextual string representations ([Akbik et al., COLING 2018](#))—SOTA NER results
- Cross-view training ([Clark et al. EMNLP 2018](#))—improve supervised tasks with unlabeled data
- Cloze-driven pretraining ([Baevski et al. (2019)](#))—SOTA NER and constituency parsing
Why does language modeling work so well?

- Language modeling is a very difficult task, even for humans.
- Language models are expected to compress any possible context into a vector that generalizes over possible completions.
  - “They walked down the street to ????”
- To have any chance at solving this task, a model is forced to learn syntax, semantics, encode facts about the world, etc.
- Given enough data, a huge model, and enough compute, can do a reasonable job!
- Empirically works better than translation, autoencoding: “Language Modeling Teaches You More Syntax than Translation Does” (Zhang et al. 2018)
Sample efficiency
Pretraining reduces need for annotated data

(Peters et al, NAACL 2018)
Pretraining reduces need for annotated data

(Howard and Ruder, ACL 2018)
Pretraining reduces need for annotated data

(Clark et al. EMNLP 2018)
Scaling up pretraining
Scaling up pretraining

More data $\rightarrow$ better word vectors

(Pennington et al 2014)
Scaling up pretraining

Figure 3: Average GLUE score with different amounts of Common Crawl data for pretraining.

Baevski et al. (2019)
Scaling up pretraining

Bigger model → better results

(Devlin et al 2019)

<table>
<thead>
<tr>
<th>Hyperparams</th>
<th>Dev Set Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>#L</td>
<td>#H</td>
</tr>
<tr>
<td>3</td>
<td>768</td>
</tr>
<tr>
<td>6</td>
<td>768</td>
</tr>
<tr>
<td>6</td>
<td>768</td>
</tr>
<tr>
<td>12</td>
<td>768</td>
</tr>
<tr>
<td>12</td>
<td>1024</td>
</tr>
<tr>
<td>24</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table 6: Ablation over BERT model size. #L = the number of layers; #H = hidden size; #A = number of attention heads. “LM (ppl)” is the masked LM perplexity of held-out training data.
Cross-lingual pretraining
Cross-lingual pretraining

- Much work on training cross-lingual word embeddings (Overview: Ruder et al. (2017))
- Idea: train each language separately, then align.
- Recent work aligning ELMo: Schuster et al., (NAACL 2019)
- ACL 2019 Tutorial on Unsupervised Cross-lingual Representation Learning
Cross-lingual Polyglot Pretraining

Key idea: **Share vocabulary** and representations across languages by training one model on many languages.

Advantages: Easy to implement, **enables** cross-lingual pretraining by itself

Disadvantages: Leads to **under-representation** of low-resource languages

- LASER: Use parallel data for sentence representations ([Artetxe & Schwenk, 2018](https://example.com))
- **Multilingual BERT**: BERT trained jointly on 100 languages
- Rosita: Polyglot contextual representations ([Mulcaire et al., NAACL 2019](https://example.com))
- XLM: Cross lingual LM ([Lample & Conneau, 2019](https://example.com))
Hands-on #1: Pretraining a Transformer Language Model

Image credit: Chanaky
Hands-on: Overview

Current developments in Transfer Learning combine new approaches for **training schemes** (sequential training) as well as **models** (transformers) can look intimidating and complex.

- **Goals:**
  - Let’s make these recent works “uncool again” i.e. as accessible as possible
  - Expose all the details in a simple, concise and self-contained code-base
  - Show that transfer learning can be simple (less hand-engineering) & fast (pretrained model)

- **Plan**
  - Build a GPT-2 / BERT model
  - Pretrain it on a rather large corpus with ~100M words
  - Adapt it for a target task to get SOTA performances

- **Material:**
  - Colab: [http://tiny.cc/NAACLTransferColab](http://tiny.cc/NAACLTransferColab) ➔ code of the following slides
  - Code: [http://tiny.cc/NAACLTransferCode](http://tiny.cc/NAACLTransferCode) ➔ same code organized in a repo
Hands-on pre-training

Colab: [http://tiny.cc/NAACLTransferColab](http://tiny.cc/NAACLTransferColab)

Our core model will be a Transformer. Large-scale transformer architectures (GPT-2, BERT, XLM...) are very similar to each other and consist of:

- summing words and position embeddings
- applying a succession of transformer blocks with:
  - layer normalisation
  - a self-attention module
  - dropout and a residual connection
- another layer normalisation
- a feed-forward module with one hidden layer and a non linearity: Linear $\xrightarrow{\text{ReLU/gelu}}$ Linear
- dropout and a residual connection

Main differences between GPT/GPT-2/BERT are the objective functions:

- causal language modeling for GPT
- masked language modeling for BERT (+ next sentence prediction)
Let’s code the backbone of our model!

PyTorch 1.1 now has a `nn.MultiHeadAttention` module: lets us encapsulate the self-attention logic while still controlling the internals of the Transformer.
Two attention masks?

- **padding_mask** masks the padding tokens. It is specific to each sample in the batch:

<table>
<thead>
<tr>
<th>I love Mom</th>
<th>s cooking</th>
</tr>
</thead>
<tbody>
<tr>
<td>I love</td>
<td>you too</td>
</tr>
<tr>
<td>No</td>
<td></td>
</tr>
<tr>
<td>This</td>
<td>is the shit</td>
</tr>
<tr>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

- **attn_mask** is the same for all samples in the batch. It masks the previous tokens for causal transformers:

```python
import torch
import torch.nn as nn

class Transformer(nn.Module):
    def __init__(self, embed_dim, hidden_dim, num_embeddings, num_max_positions, num_heads, num_layers, dropout, causal):
        super().__init__()
        self.causal = causal
        self.token_embeddings = nn.Embedding(num_embeddings, embed_dim)
        self.position_embeddings = nn.Embedding(num_max_positions, embed_dim)
        self.dropout = nn.Dropout(dropout)

        self.attentions, self.feed_forwards = nn.ModuleList(), nn.ModuleList()
        self.layer_norms_1, self.layer_norms_2 = nn.ModuleList(), nn.ModuleList()
        for ln in range(num_layers):
            self.attentions.append(nn.MultiheadAttention(embed_dim, num_heads, dropout=dropout))
            self.feed_forwards.append(nn.Sequential(nn.Linear(embed_dim, hidden_dim),
                                                    nn.ReLU(),
                                                    nn.Linear(hidden_dim, embed_dim)))
            self.layer_norms_1.append(nn.LayerNorm(embed_dim, eps=1e-12))
            self.layer_norms_2.append(nn.LayerNorm(embed_dim, eps=1e-12))

    def forward(self, x, padding_mask=None):
        positions = torch.arange(len(x), device=x.device).unsqueeze(-1)
        h = self.token_embeddings(x)
        h = h + self.position_embeddings(positions).expand_as(h)
        h = self.dropout(h)

        attn_mask = None
        if self.causal:
            attn_mask = torch.full((len(x), len(x)), -float('Inf'), device=h.device, dtype=h.dtype)
            attn_mask = torch.triu(attn_mask, diagonal=1)

        for layer_norm_1, attention, layer_norm_2, feed_forward in zip(self.layer_norms_1, self.attentions, self.layer_norms_2, self.feed_forwards):
            h = layer_norm_1(h)
            x, _ = attention(h, h, h, attn_mask=attn_mask, need_weights=False, key_padding_mask=padding_mask)
            h = x + h
            h = layer_norm_2(h)
            x = feed_forward(h)
            h = x + h

        return h
```
1. **A pretraining head** on top of our core model: we choose a language modeling head with tied weights.

2. **Initialize** the weights

3. **Define a loss function**: we choose a cross-entropy loss on current (or next) token predictions.
We'll use a pre-defined open vocabulary tokenizer: BERT’s model cased tokenizer.

Hyper-parameters taken from [Dai et al., 2018](https://arxiv.org/abs/1811.00388) (Transformer-XL) \(\Rightarrow\) \(~50M\) parameters causal model.

Use a large dataset for pre-training: WikiText-103 with 103M tokens ([Merity et al., 2017](https://arxiv.org/abs/1705.02369)).

Instantiate our model and optimizer (Adam)
Hands-on pre-training

And we’re done: let’s train!

A simple update loop. We use gradient accumulation to have a large batch size even on 1 GPU (>64).

Learning rate schedule:
- linear warmup to start
- then cosine or inverse square root decrease

Go!

And we’re done: let’s train!

```python
import os from torch.utils.data import DataLoader from ignite.engine import Engine, Events from ignite.metrics import RunningAverage from ignite.handlers import ModelCheckpoint from ignite.contrib.handlers import CosineAnnealingScheduler, create_lr_scheduler_with_warmup, ProgressBar
data_loader = DataLoader(datasets['train'], batch_size=16, shuffle=True)

# Define training function
for batch in self.state.dataloader:
    self.state.batch = batch
    self.state.iteration += 1
    self._fire_event(Events.ITERATION_STARTED)
    self.state.output = self._process_function(self, batch)
    self._fire_event(Events.ITERATION_COMPLETED)

RunningAverage(output_transform=lambda x: x).attach(trainer, 'loss').attach(trainer, metric_names=['loss'])

# Learning rate scheduler: linearly warm-up to lr and then decrease the learning rate to zero with cosine
# scheduler = CosineAnnealingScheduler(optimizer, 'lr', args.lr, 0.0, len(data_loader) * args.n_epochs)
scheduler = create_lr_scheduler_with_warmup(cosine_scheduler, 0.0, args.lr, args.n_warmup)
trainer.add_event_handler(Events.ITERATION_STARTED, scheduler)

# Save checkpoints and training config
checkpoint_handler = ModelCheckpoint(args.log_dir, 'checkpoint', save_interval=1, n_saved=5)
trainer.add_event_handler(Events.EPOCH_COMPLETED, checkpoint_handler, {'mymodel': model})
torch.save(args, os.path.join(args.log_dir, 'training_args.bin'))

trainer.run(data_loader, max_epochs=args.n_epochs)
```

... Epoch [1/50] [365/28874] 1% , loss=2.30e+00 [03:43<4:52:22]
Hands-on pre-training — Concluding remarks

- **On pretraining**
  - **Intensive**: in our case 5h–20h on 8 V100 GPUs (few days w. 1 V100) to reach a good perplexity ⇨ share your pretrained models
  - **Robust to the choice of hyper-parameters** (apart from needing a warm-up for transformers)
  - Language modeling is a hard task, your model should **not have enough capacity to overfit** if your dataset is large enough ⇨ you can just start the training and let it run.
  - **Masked-language modeling**: typically 2-4 times slower to train than LM
    We only mask 15% of the tokens ⇨ smaller signal

- **For the rest of this tutorial**
  We don’t have enough time to do a full pretraining
  ⇨ we pretrained **two models** for you before the tutorial
First model:

- **exactly the one** we built together ➔ a 50M parameters causal Transformer
- Trained 15h on 8 V100
- Reached a **word-level perplexity of 29** on wikitext-103 validation set (quite competitive)

Second model:

- Same model but trained with a **masked-language modeling** objective (see the repo)
- Trained 30h on 8 V100
- Reached a “masked-word” perplexity of 8.3 on wikitext-103 validation set

---

**Wikitext-103 Validation/Test PPL**

<table>
<thead>
<tr>
<th>Model</th>
<th>#Params</th>
<th>Validation PPL</th>
<th>Test PPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grave et al. (2016b) – LSTM</td>
<td>-</td>
<td>-</td>
<td>48.7</td>
</tr>
<tr>
<td>Bai et al. (2018) – TCN</td>
<td>-</td>
<td>-</td>
<td>45.2</td>
</tr>
<tr>
<td>Dauphin et al. (2016) – GCNN-8</td>
<td>-</td>
<td>-</td>
<td>44.9</td>
</tr>
<tr>
<td>Grave et al. (2016b) – LSTM + Neural cache</td>
<td>-</td>
<td>-</td>
<td>40.8</td>
</tr>
<tr>
<td>Dauphin et al. (2016) – GCNN-14</td>
<td>-</td>
<td>-</td>
<td>37.2</td>
</tr>
<tr>
<td>Merity et al. (2015) – 4-layer QRNN</td>
<td>151M</td>
<td>32.0</td>
<td>33.0</td>
</tr>
<tr>
<td>Rae et al. (2018) – LSTM + Hebbian + Cache</td>
<td>-</td>
<td>29.7</td>
<td>29.9</td>
</tr>
<tr>
<td>Ours – Transformer-XL Standard</td>
<td>151M</td>
<td><strong>23.1</strong></td>
<td><strong>24.0</strong></td>
</tr>
<tr>
<td>Ours – Transformer-XL Large</td>
<td>257M</td>
<td><strong>17.7</strong></td>
<td><strong>18.3</strong></td>
</tr>
</tbody>
</table>

---

Dai et al., 2018
Agenda

[1] Introduction

[2] Pretraining


[4] Adaptation

[5] Downstream

3. What is in a Representation?
Why care about what is in a representation?

- Extrinsic evaluation with downstream tasks
  - Complex, diverse with task-specific quirks

- Language-aware representations
  - To generalize to other tasks, new inputs
  - As intermediates for possible improvements to pretraining

- Interpretability!
  - Are we getting our results because of the right reasons?
  - Uncovering biases...

Swayamdipta, 2019
What to analyze?

- Embeddings
  - Word
  - Contextualized

- Network Activations

- Variations
  - Architecture (RNN / Transformer)
  - Layers
  - Pretraining Objectives

Diagram:

- $T$
- $L_n$
- $L_1$
- $E$
Analysis Method 1: Visualization

Hold the embeddings / network activations static or **frozen**
Visualizing Embedding Geometries

- Plotting embeddings in a lower dimensional (2D/3D) space
  - t-SNE van der Maaten & Hinton, 2008
  - PCA projections

- Visualizing word analogies Mikolov et al., 2013
  - Spatial relations
    - $w_{\text{king}} - w_{\text{man}} + w_{\text{woman}} \sim w_{\text{queen}}$

- High-level view of lexical semantics
  - Only a limited number of examples
  - Connection to other tasks is unclear Goldberg, 2017
Neuron activation values correlate with features / labels

Indicates learning of recognizable features
- How to select which neuron? Hard to scale!
- Interpretable != Important (Morcos et al., 2018)

Cell that is sensitive to the depth of an expression:
```c
#define CONFIG_AUDITSYSCALL

static inline int audit_match_class_bits(int class, u32 *mask)
{
    int i;
    if (classes[class]) {
        for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
            if (mask[i] & classes[class][i])
                return 0;
    }
    return 1;
}
```

Radford et al., 2017

Morcos et al., 2018

Karpathy et al., 2016
Layer-wise analysis (static)

- How important is each layer for a **given performance** on a downstream task?
  - Weighted average of layers

- Task and architecture specific!

Also see Tenney et al., ACL 2019

Peters et al., EMNLP 2018
Popular in machine translation, or other seq2seq architectures:

- **Alignment** between words of source and target.
- Long-distance word-word **dependencies** (intra-sentence attention)

**Sheds light on architectures**

- Having sophisticated attention mechanisms can be a good thing!
- Layer-specific

**Interpretation can be tricky**

- Few examples only - cherry picking?
- Robust **corpus-wide** trends? Next!

---

Vaswani et al., 2017
Analysis Method 2: Behavioral Probes

- RNN-based language models
  - number agreement in subject-verb dependencies
  - natural and nonce or ungrammatical sentences
  - evaluate on output perplexity

- RNNs outperform other non-neural baselines.

- Performance improves when trained explicitly with syntax
  (Kuncoro et al. 2018)

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018

Kuncoro et al. 2018
Analysis Method 2: Behavioral Probes

- RNN-based language models (RNN-based)
  - **number agreement** in subject-verb dependencies
  - For natural and nonce/ungrammatical sentences
  - LM perplexity differences

- RNNs outperform other non-neural baselines.

- Performance improves when trained explicitly with syntax (Kuncoro et al. 2018)

- Probe: Might be vulnerable to co-occurrence biases
  - “dogs in the neighborhood bark(s)”
  - Nonce sentences might be too different from original...

Linzen et al., 2016; Gulordava et al. 2018; Marvin et al., 2018
Analysis Method 3: Classifier Probes

Hold the embeddings / network activations static and

train a **simple supervised** model on top

Probe classification task (Linear / MLP)
Probing Surface-level Features

- Given a sentence, predict properties such as:
  - Length
  - Is a word in the sentence?

- Given a word in a sentence predict properties such as:
  - Previously seen words, contrast with language model
  - Position of word in the sentence

- Checks ability to memorize
  - Well-trained, richer architectures tend to fare better
  - Training on linguistic data memorizes better

Zhang et al. 2018; Liu et al., 2018; Conneau et al., 2018
Morphology

Word-level syntax
- POS tags, CCG supertags
- Constituent parent, grandparent...

Partial syntax
- Dependency relations

Partial semantics
- Entity Relations
- Coreference
- Roles

Adi et al., 2017; Conneau et al., 2018; Belinkov et al., 2017; Zhang et al., 2018; Blevins et al., 2018; Tenney et al. 2019; Liu et al., 2019
# Probing classifier findings

<table>
<thead>
<tr>
<th></th>
<th>Lex.</th>
<th>CoVe</th>
<th>GPT</th>
<th>ELMo</th>
<th>Supersense ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full</td>
<td>cat</td>
<td>mix</td>
<td>Full</td>
<td>cat</td>
</tr>
<tr>
<td>Part-of-Speech</td>
<td>85.7</td>
<td>94.0</td>
<td>8.4</td>
<td>90.4</td>
<td>96.7</td>
</tr>
<tr>
<td>Constituents</td>
<td>56.1</td>
<td>81.6</td>
<td>25.4</td>
<td>69.1</td>
<td>84.6</td>
</tr>
<tr>
<td>Dependencies</td>
<td>75.6</td>
<td>83.6</td>
<td>8.6</td>
<td>80.4</td>
<td>93.9</td>
</tr>
<tr>
<td>Entities</td>
<td>88.4</td>
<td>90.3</td>
<td>1.9</td>
<td>92.0</td>
<td>95.6</td>
</tr>
<tr>
<td>SRL (all)</td>
<td>59.7</td>
<td>80.4</td>
<td>20.7</td>
<td>74.1</td>
<td>90.1</td>
</tr>
<tr>
<td>Core roles</td>
<td>56.2</td>
<td>81.0</td>
<td>24.7</td>
<td>73.6</td>
<td>92.6</td>
</tr>
<tr>
<td>Non-core roles</td>
<td>67.7</td>
<td>78.8</td>
<td>11.1</td>
<td>75.4</td>
<td>84.1</td>
</tr>
<tr>
<td>OntoNotes coref.</td>
<td>72.9</td>
<td>79.2</td>
<td>6.3</td>
<td>75.3</td>
<td>84.0</td>
</tr>
<tr>
<td>SPR1</td>
<td>73.7</td>
<td>77.1</td>
<td>3.4</td>
<td>80.1</td>
<td>84.8</td>
</tr>
<tr>
<td>SPR2</td>
<td>52.1</td>
<td>54.3</td>
<td>2.2</td>
<td>54.3</td>
<td>53.5</td>
</tr>
<tr>
<td>Winograd coref.</td>
<td>69.1</td>
<td>78.1</td>
<td>9.0</td>
<td>75.4</td>
<td>84.4</td>
</tr>
<tr>
<td>Rel. (SemEval)</td>
<td>69.1</td>
<td>78.1</td>
<td>9.0</td>
<td>75.4</td>
<td>84.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>BERT-base</th>
<th>Lex.</th>
<th>F1 Score</th>
<th>Abs. Δ</th>
<th>ELMo</th>
<th>BERT (without pretraining)</th>
<th>Lex.</th>
<th>F1 Score</th>
<th>Abs. Δ</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Part-of-Speech</td>
<td>88.4</td>
<td>97.0</td>
<td>96.7</td>
<td>0.0</td>
<td>96.5</td>
<td>96.9</td>
<td>0.2</td>
<td>0.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constituents</td>
<td>80.1</td>
<td>93.0</td>
<td>95.1</td>
<td>1.1</td>
<td>91.5</td>
<td>95.4</td>
<td>0.3</td>
<td>0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td>90.9</td>
<td>96.1</td>
<td>96.2</td>
<td>0.6</td>
<td>91.8</td>
<td>96.2</td>
<td>0.3</td>
<td>0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entities</td>
<td>75.4</td>
<td>89.4</td>
<td>91.3</td>
<td>1.2</td>
<td>87.5</td>
<td>92.3</td>
<td>1.0</td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRL (all)</td>
<td>74.9</td>
<td>91.4</td>
<td>93.6</td>
<td>1.0</td>
<td>76.3</td>
<td>89.9</td>
<td>1.0</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core roles</td>
<td>76.4</td>
<td>84.7</td>
<td>85.9</td>
<td>1.8</td>
<td>76.9</td>
<td>84.1</td>
<td>1.0</td>
<td>2.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-core roles</td>
<td>74.9</td>
<td>88.7</td>
<td>90.2</td>
<td>6.3</td>
<td>75.7</td>
<td>91.4</td>
<td>1.2</td>
<td>7.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OntoNotes coref.</td>
<td>79.2</td>
<td>84.7</td>
<td>86.1</td>
<td>1.3</td>
<td>79.6</td>
<td>85.1</td>
<td>1.0</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPR1</td>
<td>81.7</td>
<td>83.0</td>
<td>83.8</td>
<td>0.7</td>
<td>81.6</td>
<td>83.2</td>
<td>1.0</td>
<td>3.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPR2</td>
<td>65.4</td>
<td>53.6</td>
<td>54.9</td>
<td>1.4</td>
<td>53.0</td>
<td>53.8</td>
<td>1.0</td>
<td>7.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Winograd coref.</td>
<td>57.4</td>
<td>78.3</td>
<td>82.0</td>
<td>4.2</td>
<td>56.2</td>
<td>77.6</td>
<td>0.5</td>
<td>4.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rel. (SemEval)</td>
<td>75.1</td>
<td>84.8</td>
<td>86.3</td>
<td>1.9</td>
<td>75.2</td>
<td>84.2</td>
<td>1.0</td>
<td>2.9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Liu et al. NAACL 2019</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Tenney et al., ACL 2019</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Hewitt et al., 2019</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Method</th>
<th>Distance</th>
<th>Depth</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>48.9</td>
<td>0.58</td>
<td>2.9</td>
<td>0.27</td>
</tr>
<tr>
<td>ELMo</td>
<td>26.8</td>
<td>0.44</td>
<td>54.3</td>
<td>0.56</td>
</tr>
<tr>
<td>DECAY0</td>
<td>51.7</td>
<td>0.61</td>
<td>54.3</td>
<td>0.56</td>
</tr>
<tr>
<td>PRO1</td>
<td>59.8</td>
<td>0.73</td>
<td>64.4</td>
<td>0.75</td>
</tr>
<tr>
<td>ELMo1</td>
<td>77.0</td>
<td>0.83</td>
<td>86.5</td>
<td>0.87</td>
</tr>
<tr>
<td>BERTBase7</td>
<td>79.8</td>
<td>0.85</td>
<td>88.0</td>
<td>0.87</td>
</tr>
<tr>
<td>BERTLARGE15</td>
<td>82.5</td>
<td>0.86</td>
<td>89.4</td>
<td>0.88</td>
</tr>
<tr>
<td>BERTLARGE16</td>
<td>81.7</td>
<td>0.87</td>
<td>90.1</td>
<td>0.89</td>
</tr>
</tbody>
</table>
Probing classifier findings

- Contextualized > non-contextualized
  - Especially on syntactic tasks
  - Closer performance on semantic tasks
  - Bidirectional context is important

- BERT (large) almost always gets the highest performance
  - Grain of salt: Different contextualized representations were trained on different data, using different architectures...

---

<table>
<thead>
<tr>
<th></th>
<th>Lex.</th>
<th>CoVe Full</th>
<th>Abs. Δ</th>
<th>Lex.</th>
<th>ELMo Full</th>
<th>Abs. Δ</th>
<th>Lex.</th>
<th>GPT cat</th>
<th>mix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part-of-Speech</td>
<td>85.7</td>
<td>94.0</td>
<td>8.4</td>
<td>90.4</td>
<td>96.7</td>
<td>6.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constituents</td>
<td>56.1</td>
<td>81.6</td>
<td>25.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td>75.0</td>
<td>83.6</td>
<td>8.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entities</td>
<td>88.4</td>
<td>90.3</td>
<td>1.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRL (all)</td>
<td>59.7</td>
<td>80.4</td>
<td>20.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core roles</td>
<td>56.2</td>
<td>81.0</td>
<td>24.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-core roles</td>
<td>67.7</td>
<td>78.8</td>
<td>11.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OntoNotes coref.</td>
<td>72.9</td>
<td>79.2</td>
<td>6.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPR1</td>
<td>73.7</td>
<td>77.1</td>
<td>3.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPR2</td>
<td>76.6</td>
<td>80.2</td>
<td>3.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Winograd coref.</td>
<td>52.1</td>
<td>54.3</td>
<td>2.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rel. (SemEval)</td>
<td>51.0</td>
<td>60.6</td>
<td>9.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Macro Average</td>
<td>69.1</td>
<td>78.1</td>
<td>9.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Lex.</th>
<th>F1 Score</th>
<th>cat</th>
<th>mix</th>
<th>Exp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part-of-Speech</td>
<td>88.4</td>
<td>97.0</td>
<td>96.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constituents</td>
<td>68.4</td>
<td>83.7</td>
<td>86.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td>80.1</td>
<td>93.0</td>
<td>95.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entities</td>
<td>90.9</td>
<td>96.1</td>
<td>96.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRL (all)</td>
<td>75.4</td>
<td>89.4</td>
<td>91.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core roles</td>
<td>74.9</td>
<td>91.4</td>
<td>93.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-core roles</td>
<td>76.4</td>
<td>84.7</td>
<td>85.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OntoNotes coref.</td>
<td>74.9</td>
<td>88.7</td>
<td>90.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPR1</td>
<td>79.2</td>
<td>84.7</td>
<td>86.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPR2</td>
<td>81.7</td>
<td>83.0</td>
<td>83.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Winograd coref.</td>
<td>54.3</td>
<td>53.6</td>
<td>54.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rel. (SemEval)</td>
<td>57.4</td>
<td>78.3</td>
<td>82.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Macro Average</td>
<td>75.1</td>
<td>84.8</td>
<td>86.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Supersense ID**

<table>
<thead>
<tr>
<th></th>
<th>BERT</th>
<th>GED</th>
<th>PS-Role</th>
<th>PS-Fxn</th>
<th>EF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELM00</td>
<td>26.8</td>
<td>0.44</td>
<td>54.3</td>
<td>0.56</td>
<td></td>
</tr>
<tr>
<td>DECAy0</td>
<td>51.7</td>
<td>0.61</td>
<td>54.3</td>
<td>0.56</td>
<td></td>
</tr>
<tr>
<td>PRO1</td>
<td>59.8</td>
<td>0.73</td>
<td>64.4</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>ELM01</td>
<td>77.0</td>
<td>0.83</td>
<td>86.5</td>
<td>0.87</td>
<td></td>
</tr>
<tr>
<td>BERTBASE7</td>
<td>79.8</td>
<td>0.85</td>
<td>88.0</td>
<td>0.87</td>
<td></td>
</tr>
<tr>
<td>BERTLARGE15</td>
<td>82.5</td>
<td>0.86</td>
<td>89.4</td>
<td>0.88</td>
<td></td>
</tr>
<tr>
<td>BERTLARGE16</td>
<td>81.7</td>
<td>0.87</td>
<td>90.1</td>
<td>0.89</td>
<td></td>
</tr>
</tbody>
</table>

---

Tenney et al., ACL 2019

Hewitt et al., 2019
RNN layers: General linguistic properties
- Lowest layers: morphology
- Middle layers: syntax
- Highest layers: Task-specific semantics

Transformer layers:
- Different trends for different tasks; middle-heavy
- Also see Tenney et. al., 2019

Fig. from Liu et al. (NAACL 2019)
Language modeling **outperforms** other unsupervised and supervised objectives.
- Machine Translation
- Dependency Parsing
- Skip-thought

**Low-resource** settings (size of training data) might result in opposite trends.

Zhang et al., 2018; Blevins et al., 2018; Liu et al., 2019;
What have we learnt so far?

- Representations are **predictive** of certain linguistic phenomena:
  - Alignments in translation, Syntactic hierarchies

- Pretraining with and without syntax:
  - Better performance with syntax
  - But without, some notion of syntax at least ([Williams et al. 2018](#))

- Network architectures determine what is in a representation
  - Syntax and BERT Transformer ([Tenney et al., 2019; Goldberg, 2019](#))
  - Different layer-wise trends across architectures
Open questions about probes

- What information should a good probe look for?
  - Probing a probe!

- What does probing performance tell us?
  - Hard to synthesize results across a variety of baselines...

- Can introduce some complexity in itself
  - Linear or non-linear classification.
  - Behavioral: design of input sentences

- Should we be using probes as evaluation metrics?
  - Might defeat the purpose...
Progressively erase or mask network components
- Word embedding dimensions
- Hidden units
- Input - words / phrases

Figure 5: Heatmap of word importance (computed using Eq. 1) in sentiment analysis.

Li et al., 2016
So, what is in a representation?

- Depends on how you look at it!
  - **Visualization:**
    - *bird’s eye view*
    - *few* samples -- might call to mind cherry-picking
  - **Probes:**
    - discover corpus-wide **specific** properties
    - may introduce own biases...
  - **Network ablations:**
    - great for **improving modeling**,
    - could be task specific

- Analysis methods as tools to aid model development!
Very current and ongoing!

Citation counts by year in "Part 3. What do representations learn"?

- 2015
- 2016
- 2017
- 2018
- 2019

First column for citations in and before 2015
What’s next?

- Linguistic Awareness
- Interpretability

Interpretability + transferability to downstream tasks is key

➔ Up next!

Correlation of probes to downstream tasks

Conneau et al., 2018
Some Pointers

- Suite of word-based and word-pair-based tasks: Liu et al. 2019 (3B Semantics)
  
  https://github.com/nelson-liu/contextual-repr-analysis

- Structural Probes: Hewitt & Manning 2019 (9E Machine Learning)

- Overview of probes: Belinkov & Glass, 2019 (7F Poster #18)
Transfer Learning in NLP

Follow along with the tutorial:

- Slides: https://tinyurl.com/NAACLTransfer
- Colab: https://tinyurl.com/NAACLTransferColab
- Code: https://tinyurl.com/NAACLTransferCode

Questions:

- Twitter: #NAACLTransfer during the tutorial
- Whova: “Questions for the tutorial on Transfer Learning in NLP” topic
- Ask us during the break or after the tutorial
Agenda

[1] Introduction

[2] Pretraining

[3] What’s in a representation?

[4] Adaptation

[5] Downstream

4. Adaptation
4 – How to adapt the pretrained model

Several orthogonal directions we can make decisions on:

1. **Architectural** modifications?
   *How much to change the pretrained model architecture for adaptation*

2. **Optimization** schemes?
   *Which weights to train during adaptation and following what schedule*

3. **More signal**: Weak supervision, Multi-tasking & Ensembling
   *How to get more supervision signal for the target task*
4.1 – Architecture

Two general options:

A. **Keep** pretrained model **internals unchanged**:
   
   *Add classifiers on top, embeddings at the bottom, use outputs as features*

B. **Modify** pretrained model internal architecture:
   
   *Initialize encoder-decoders, task-specific modifications, adapters*
4.1.A – Architecture: Keep model unchanged

General workflow:

1. **Remove pretraining task head** if not useful for target task
   a. **Example**: remove softmax classifier from pretrained LM
   b. **Not always needed**: some adaptation schemes re-use the pretraining objective/task, e.g. for multi-task learning
4.1.A – Architecture: Keep model unchanged

General workflow:

2. Add target task-specific layers on top/bottom of pretrained model
   a. **Simple**: adding linear layer(s) on top of the pretrained model
4.1.A – Architecture: Keep model unchanged

General workflow:

2. Add target task-specific layers on top/bottom of pretrained model
   a. Simple: adding linear layer(s) on top of the pretrained model
   b. More complex: model output as input for a separate model
   c. Often beneficial when target task requires interactions that are not available in pretrained embedding
4.1.B – Architecture: Modifying model internals

Various reasons:

1. Adapting to a **structurally different** target task
   a. Ex: Pretraining with a **single** input sequence (ex: language modeling) but adapting to a task with **several** input sequences (ex: translation, conditional generation...)
   b. Use the pretrained model weights to initialize as much as possible of a structurally different target task model
   c. Ex: Use monolingual LMs to initialize encoder and decoder parameters for MT ([Ramachandran et al., EMNLP 2017; Lample & Conneau, 2019](#))
4.1.B – Architecture: Modifying model internals

Various reasons:

2. Task-specific **modifications**
   a. Provide pretrained model with capabilities that are useful for the target task
   b. Ex: Adding skip/residual connections, attention ([Ramachandran et al., EMNLP 2017](#))
3. Using **less parameters** for adaptation:
   a. Less parameters to fine-tune
   b. Can be very useful given the increasing size of model parameters
   c. Ex: add bottleneck modules (“adapters”) between the layers of the pretrained model ([Rebuffi et al., NIPS 2017; CVPR 2018](#))
4.1.B – Architecture: Modifying model internals

Adapters

- Commonly connected with a **residual connection** in parallel to an existing layer
- Most effective when placed at **every layer** (smaller effect at bottom layers)
- **Different operations** (convolutions, self-attention) possible
- Particularly suitable for modular architectures like Transformers

(Houlsby et al., ICML 2019; Stickland and Murray, ICML 2019)
4.1.B – Architecture: Modifying model internals

Adapters (Stickland & Murray, ICML 2019)

- Multi-head attention (MH; shared across layers) is used in parallel with self-attention (SA) layer of BERT
- Both are added together and fed into a layer-norm (LN)
Hands-on #2: Adapting our pretrained model
Hands-on: Model adaptation

Let’s see how a simple fine-tuning scheme can be implemented with our pretrained model:

- **Plan**
  - Start from our Transformer language model
  - Adapt the model to a target task:
    - keep the model **core unchanged**, load the pretrained weights
    - add a linear layer **on top**, newly initialized
    - use additional embeddings **at the bottom**, newly initialized

- **Reminder — material is here:**
  - Colab [http://tiny.cc/NAACLTTransferColab](http://tiny.cc/NAACLTTransferColab) ⇒ code of the following slides
Adaptation task

- We select a text classification task as the downstream task
- TREC-6: The Text REtrieval Conference (TREC) Question Classification ([Li et al., COLING 2002](https://www.aclweb.org/anthology/C02-1021))
- TREC consists of open-domain, fact-based questions divided into broad semantic categories contains 5500 labeled training questions & 500 testing questions with 6 labels:
  - `NUM`, `LOC`, `HUM`, `DESC`, `ENTY`, `ABBR`

Ex:

★ How did serfdom develop in and then leave Russia?  \(\rightarrow\)  DESC
★ What films featured the character Popeye Doyle?  \(\rightarrow\)  ENTY

<table>
<thead>
<tr>
<th>Model</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoVe (McCann et al., 2017)</td>
<td>4.2</td>
</tr>
<tr>
<td>TBCNN (Mou et al., 2015)</td>
<td>4.0</td>
</tr>
<tr>
<td>LSTM-CNN (Zhou et al., 2016)</td>
<td>3.9</td>
</tr>
<tr>
<td>ULMFiT (ours)</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Transfer learning models shine on this type of low-resource task

(Howard and Ruder, ACL 2018)
First adaptation scheme

- **Modifications:**
  - Keep model internals unchanged
  - Add a linear layer on top
  - Add an additional embedding (classification token) at the bottom

- **Computation flow:**
  - Model input: the tokenized question with a classification token at the end
  - Extract the last hidden-state associated to the classification token
  - Pass the hidden-state in a linear layer and softmax to obtain class probabilities

(Radford et al., 2018)
Hands-on: Model adaptation

Fine-tuning hyper-parameters:
- 6 classes in TREC-6
- Use fine tuning hyper parameters from Radford et al., 2018:
  - learning rate from 6.5e-5 to 0.0
  - fine-tune for 3 epochs

Let’s load and prepare our dataset:
- trim to the transformer input size & add a classification token at the end of each sample,
- pad to the left,
- convert to tensors,
- extract a validation set.

```
import random
from torch.utils.data import Import TensorDataset, random_split

dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/trec/" "+trec-tokenized-bert.bin")
datasets = torch.load(dataset_file)
for split_name in ['train', 'test']:
    # Trim the samples to the transformer's input length minus 1 & add a classification token
    datasets[split_name] = [x[:args.num_max_positions-1] + [tokenizer.vocab[ '[CLS]' ]] for x in datasets[split_name]]

    # Pad the dataset to max length
    padding_length = max(len(x) for x in datasets[split_name])
    datasets[split_name] = [x + [tokenizer.vocab[ '[PAD]' ]] * (padding_length - len(x)) for x in datasets[split_name]]

    # Convert to torch.Tensor and gather inputs and labels
    tensor = torch.tensor(datasets[split_name], dtype=torch.long)
    labels = torch.tensor(datasets[split_name + ' _labels'], dtype=torch.long)
    datasets[split_name + ' _labels'] = TensorDataset(tensor, labels)

    # Create a validation dataset from a fraction of the training dataset
    valid_size = int(adapt_args.valid_set_prop * len(datasets[ 'train' ]))
    train_size = len(datasets[ 'train' ]) - valid_size
    valid_dataset, train_dataset = random_split(datasets[ 'train' ], [valid_size, train_size])

    train_loader = DataLoader(train_dataset, batch_size=adapt_args.batch_size, shuffle=True)
    valid_loader = DataLoader(valid_dataset, batch_size=adapt_args.batch_size, shuffle=True)
    test_loader = DataLoader(datasets[ 'test' ], batch_size=adapt_args.batch_size, shuffle=False)
```
Adapt our model architecture

Keep our pretrained model unchanged as the backbone.

Replace the pre-training head (language modeling) with the classification head:

A linear layer, which takes as input the hidden-state of the [CLF] token (using a mask)

* Initialize all the weights of the model.
* Reload common weights from the pretrained model.

```python
class TransformerWithClfHead(nn.Module):
    def __init__(self, config, fine_tuning_config):
        super().__init__()
        self.config = fine_tuning_config
        self.transformer = Transformer(config.embed_dim, config.hidden_dim, config.num_embeddings,
                                        config.num_max_positions, config.num_heads, config.num_layers,
                                        fine_tuning_config.dropout, causal=not config.mlm)

        self.classification_head = nn.Linear(config.embed_dim, fine_tuning_config.num_classes)

    def apply(self, init_weights):
        self.apply(init_weights)

    def forward(self, x, clf_tokens_mask, clf_labels=None, padding_mask=None):
        hidden_states = self.transformer(x, padding_mask)
        clf_token_states = (hidden_states * clf_tokens_mask.unsqueeze(-1).float()).sum(dim=0)
        clf_logits = self.classification_head(clf_token_states)

        if clf_labels is not None:
            loss_fct = nn.CrossEntropyLoss(ignore_index=-1)
            loss = loss_fct(clf_logits.view(-1), clf_labels.view(-1))
            return clf_logits, loss

        return clf_logits
```

# If you have pretrained a model in the first section, you can use its weights
# state_dict = model.state_dict()

# Otherwise, just load pretrained model weights (and reload the training config as well)
state_dict = torch.load(torchcached_path(https://s3.amazonaws.com/models.huggingface.co/
"nacsl-2019-tutorial/model_checkpoint.pth", map_location='cpu')
args = torch.load(torchcached_path(https://s3.amazonaws.com/models.huggingface.co/
"nacsl-2019-tutorial/model_training_args.bin"))
adaptation_model = TransformerWithClfHead(config=args, fine_tuning_config=adapt_args, device=adapt_args.device)
incompatible_keys = adaptation_model.load_state_dict(state_dict, strict=False)
print(f'Parameters discarded from the pretrained model: {incompatible_keys.unexpected_keys}"
print(f'Parameters added in the adaptation model: {incompatible_keys.missing_keys}"
```

Parameters discarded from the pretrained model: ['lm_head.weight']
Parameters added in the adaptation model: ['classification_head.weight', 'classification_head.bias']
```
Hands-on: Model adaptation

Our fine-tuning code:

A simple training update function:
* prepare inputs: transpose and build padding & classification token masks
  * we have options to clip and accumulate gradients

We will evaluate on our validation and test sets:
* validation: after each epoch
  * test: at the end

Schedule:
* linearly increasing to lr
  * linearly decreasing to 0.0

```python
optimizer = torch.optim.Adam(adaptation_model.parameters(), lr=adapt_args.lr)

# Training function and trainer
def update(engine, batch):
    adaptation_model.train()
    batch, labels = (t.to(adapt_args.device) for t in batch)
    inputs = batch.transpose(0, 1).contiguous()  # to shape [seg length, batch]
    _, loss = adaptation_model(inputs, clf_tokens_mask=(inputs == tokenizer.vocab('[CLS]')), clf_labels=labels, padding_mask=(batch == tokenizer.vocab('[PAD]'))
    loss.backward()
    torch.nn.utils.clip_grad_norm_(adaptation_model.parameters(), adapt_args.max_norm)
    if engine.state.iteration & adapt_args.gradient_accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
    return loss.item()
trainer = Engine(update)

# Evaluation function and evaluator (evaluator output is the input of the metrics)
def inference(engine, batch):
    adaptation_model.eval()
    with torch.no_grad():
        batch, labels = (t.to(adapt_args.device) for t in batch)
        clf_logits = adaptation_model(inputs, clf_tokens_mask=(inputs == tokenizer.vocab('[CLS]')), padding_mask=(batch == tokenizer.vocab('[PAD]')))
    return clf_logits, labels
evaluator = Engine(inference)

# Attach metric to evaluator & evaluation to trainer: evaluate on valid set after each epoch
Accuracy().attach(evaluator, "accuracy")
@trainer.on(Events.EPOCH_COMPLETED)
def log_validation_results(engine):
    evaluator.run(valid_loader)
    print(f"Validation Epoch {engine.state.epoch} Error rate: {100*(1 - evaluator.state.metrics['accuracy'])}%")

# Learning rate schedule: linearly warm-up to lr and then to zero
scheduler = PiecewiseLinear(optimizer, 'lr', [(0, 0.0), (adapt_args.n_warmup, adapt_args.lr), (len(train_loader)*adapt_args.n_epochs, 0.0)])
trainer.add_event_handler(Events.ITERATION_STARTED, scheduler)

# Add progressbar with loss
RunningAverage(output_transform=lambda x: x).attach(trainer, 'loss').attach(trainer, metric_names=['loss'])

# Save checkpoints and finetuning config
checkpoint_handler = ModelCheckpoint(adapt_args.log_dir, 'finetuning_checkpoint', save_interval=1, require_empty=False)
trainer.add_event_handler(Events.EPOCH_COMPLETED, checkpoint_handler, {'symodel': adaptation_model})
torch.save(args, os.path.join(adapt_args.log_dir, 'fine_tuning_args.bin'))
```
We can now fine-tune our model on TREC:

```python
[50] trainer.run(train_loader, max_epochs=adapt_args.n_epochs)
```

- **Epoch [1/3]**
  - Validation Epoch: 1 Error rate: 9.174311926605505
  - Loss: 3.85e-01 [01:10:00:00]

- **Epoch [2/3]**
  - Validation Epoch: 2 Error rate: 5.871559633027523
  - Loss: 1.73e-01 [01:10:00:00]

- **Epoch [3/3]**
  - Validation Epoch: 3 Error rate: 5.688073394495408
  - Loss: 9.63e-02 [01:10:00:00]

Remarks:

- The error rate goes down quickly! After one epoch we already have >90% accuracy.
  - Fine-tuning is highly **data efficient** in Transfer Learning
- We took our pre-training & fine-tuning hyper-parameters straight from the literature on related models.
  - Fine-tuning is often **robust** to the exact choice of hyper-parameters

We are at the state-of-the-art (ULMFiT)
Let’s conclude this hands-on with a few additional words on robustness & variance.

- Large pretrained models (e.g. BERT large) are prone to degenerate performance when fine-tuned on tasks with small training sets.
- Observed behavior is often “on-off”: it either works very well or doesn’t work at all.
- Understanding the conditions and causes of this behavior (models, adaptation schemes) is an open research question.

Phang et al., 2018
4.2 – Optimization

Several directions when it comes to the optimization itself:

A. Choose **which weights** we should update
   *Feature extraction, fine-tuning, adapters*

B. Choose **how and when** to update the weights
   *From top to bottom, gradual unfreezing, discriminative fine-tuning*

C. Consider **practical trade-offs**
   *Space and time complexity, performance*
4.2.A – Optimization: Which weights?

The main question: **To tune or not to tune (the pretrained weights)?**

A. **Do not change** pretrained weights
   - Feature extraction, adapters

B. **Change** pretrained weights
   - Fine-tuning
Don’t touch the pretrained weights!

Feature extraction:
- Weights are **frozen**
4.2.A – Optimization: Which weights?

Don’t touch the pretrained weights!

Feature extraction:
- Weights are **frozen**
- A **linear classifier** is trained on top of the pretrained representations
4.2.A – Optimization: Which weights?

Don’t touch the pretrained weights!

Feature extraction:
- Weights are **frozen**
- A linear classifier is trained on top of the pretrained representations
- Don’t just use features of the top layer!
- Learn a linear combination of layers

(Peters et al., NAACL 2018, Ruder et al., AAAI 2019)
4.2.A – Optimization: Which weights?

Don’t touch the pretrained weights!

Feature extraction:
- Alternatively, pretrained representations are **used as features** in downstream model
4.2.A – Optimization: Which weights?

Don’t touch the pretrained weights!

Adapters

- Task-specific modules that are added **in between** existing layers
4.2.A – Optimization: Which weights?

Don’t touch the pretrained weights!

Adapters

- Task-specific modules that are added **in between** existing layers
- Only adapters are trained
4.2.A – Optimization: Which weights?

Yes, change the pretrained weights!

Fine-tuning:

- Pretrained weights are used as *initialization* for parameters of the downstream model
- The **whole pretrained architecture** is trained during the adaptation phase
Hands-on #3: Using Adapters and freezing
Second adaptation scheme: Using Adapters

- Modifications:
  - add Adapters inside the backbone model: Linear $\xrightarrow{\text{ReLU}}$ Linear with a skip-connection

- As previously:
  - add a linear layer on top
  - use an additional embedding (classification token) at the bottom

We will only train the adapters, the added linear layer and the embeddings. The other parameters of the model will be frozen.

Houlsby et al., ICML 2019
Let’s adapt our model architecture

Inherit from our pretrained model to have all the modules.

Add the adapter modules: **Bottleneck layers with 2 linear layers and a non-linear activation function (ReLU)**

Hidden dimension is small: e.g. 32, 64, 256

The Adapters are inserted inside skip-connections after:
- the attention module
- the feed-forward module
Hands-on: Model adaptation

Now we need to freeze the portions of our model we don’t want to train. We just indicate that no gradient is needed for the frozen parameters by setting `param.requires_grad` to `False` for the frozen parameters:

```python
for name, param in adaptation_model.named_parameters():
    if 'embeddings' not in name and 'classification' not in name and 'adapters_1' not in name and 'adapters_2' not in name:
        param.detach()
        param.requires_grad = False
    else:
        param.requires_grad = True

full_parameters = sum(p.numel() for p in adaptation_model.parameters())
trained_parameters = sum(p.numel() for p in adaptation_model.parameters() if p.requires_grad)

print(f"We will train {trained_parameters:.3e} parameters out of {full_parameters:.3e}, i.e. \{100 * trained_parameters/full_parameters:.2f\}%")
```

In our case we will train 25% of the parameters. The model is small & deep (many adapters) and we need to train the embeddings so the ratio stay quite high. For a larger model this ratio would be a lot lower.
Hands-on: Model adaptation

We use a hidden dimension of 32 for the adapters and a learning rate ten times higher for the fine-tuning (we have added quite a lot of newly initialized parameters to train from scratch).

```python
[185] trainer.run(train_loader, max_epochs=adapt_args.n_epochs)
```

- **Epoch [1/3]**: [307/307] 100% [loss=2.04e-01 [01:00<00:00]]
- **Validation Epoch: 1 Error rate: 9.174311926605505**
- **Epoch [2/3]**: [307/307] 100% [loss=8.40e-02 [00:57<00:00]]
- **Validation Epoch: 2 Error rate: 7.522935779816509**
- **Epoch [3/3]**: [307/307] 100% [loss=4.83e-02 [01:00<00:00]]
- **Validation Epoch: 3 Error rate: 7.522935779816509**

```python
<ignite.engine.engine.State at 0x7ff4c60fd710>
```

```python
evaluator.run(test_loader)
print(f"Test Results - Error rate: {100*(1.00 - evaluator.state.metrics['accuracy']):.3f}"")
```

- **Test Results - Error rate: 4.000**

Results similar to full-fine-tuning case with advantage of training only 25% of the full model parameters. For a small 50M parameters model this method is overkill ⇒ for 300M–1.5B parameters models.
We have decided which weights to update, but in which order and how should be update them?

Motivation: We want to avoid overwriting useful pretrained information and maximize positive transfer.

Related concept: Catastrophic forgetting (McCloskey & Cohen, 1989; French, 1999)
When a model forgets the task it was originally trained on.
4.2.B – Optimization: What schedule?

A guiding principle:

**Update from top-to-bottom**

- Progressively in **time**: freezing
- Progressively in **intensity**: Varying the learning rates
- Progressively vs. the **pretrained model**: Regularization
Main intuition: Training all layers at the same time on data of a different distribution and task may lead to instability and poor solutions.

Solution: Train layers individually to give them time to adapt to new task and data.

Goes back to layer-wise training of early deep neural networks (Hinton et al., 2006; Bengio et al., 2007).
4.2.B – Optimization: Freezing

- Freezing all but the top layer ([Long et al., ICML 2015](#))
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
  1. Train new layer
4.2.B – Optimization: Freezing

- Freezing all but the top layer ([Long et al., ICML 2015](#))
- Chain-thaw ([Felbo et al., EMNLP 2017](#)): training one layer at a time
  1. Train new layer
  2. Train one layer at a time
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
  1. Train new layer
  2. Train one layer at a time
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
  1. Train new layer
  2. Train one layer at a time
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
  1. Train new layer
  2. Train one layer at a time
  3. Train all layers
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
Freezing all but the top layer (Long et al., ICML 2015)

Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time

Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
4.2.B – Optimization: Freezing

- Freezing all but the top layer ([Long et al., ICML 2015](#))
- Chain-thaw ([Felbo et al., EMNLP 2017](#)): training one layer at a time
- Gradually unfreezing ([Howard & Ruder, ACL 2018](#)): unfreeze one layer after another
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
- Sequential unfreezing (Chronopoulou et al., NAACL 2019): hyper-parameters that determine length of fine-tuning
  1. Fine-tune additional parameters for $n$ epochs
4.2.B – Optimization: Freezing

- Freezing all but the top layer ([Long et al., ICML 2015](https://icml.cc/2015/paper/1351.pdf))
- Chain-thaw ([Felbo et al., EMNLP 2017](https://aclanthology.org/D17-1029)): training one layer at a time
- Gradually unfreezing ([Howard & Ruder, ACL 2018](https://aclanthology.org/P18-1018)): unfreeze one layer after another
- Sequential unfreezing ([Chronopoulou et al., NAACL 2019](https://www.aclweb.org/anthology/N19-1065)): hyper-parameters that determine length of fine-tuning
  1. Fine-tune additional parameters for \( n \) epochs
  2. Fine-tune pretrained parameters without embedding layer for \( k \) epochs
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
- Sequential unfreezing (Chronopoulou et al., NAACL 2019): hyper-parameters that determine length of fine-tuning
  1. Fine-tune additional parameters for $n$ epochs
  2. Fine-tune pretrained parameters without embedding layer for $k$ epochs
  3. Train all layers until convergence
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
- Sequential unfreezing (Chronopoulou et al., NAACL 2019): hyper-parameters that determine length of fine-tuning

Commonality: **Train all parameters jointly** in the end
Hands-on #4:
Using gradual unfreezing
Gradual unfreezing is similar to our previous freezing process. We start by freezing all the model except the newly added parameters:

```python
for name, param in adaptation_model.named_parameters():
    if 'embeddings' not in name and 'classification' not in name:
        param.detach()
        param.requires_grad = False
    else:
        param.requires_grad = True
full_parameters = sum(p.numel() for p in adaptation_model.parameters())
trained_parameters = sum(p.numel() for p in adaptation_model.parameters() if p.requires_grad)
print(f"We will start by training (trained_parameters:3e) parameters out of (full_parameters:3e)," +
      f" i.e. (100 * trained_parameters/full_parameters:.2f)\%\")
```

We then gradually unfreeze an additional block along the training so that we train the full model at the end:

Unfreezing interval
Find index of layer to unfreeze
Name pattern matching
Gradual unfreezing has not been investigated in details for Transformer models
- no specific hyper-parameters advocated in the literature
- Residual connections may have an impact on the method
- should probably adapt LSTM hyper-parameters

We show simple experiments in the Colab. Better hyper-parameters settings can probably be found.
4.2.B – Optimization: Learning rates

*Main idea:* Use **lower learning rates** to avoid **overwriting** useful information.

Where and when?

- **Lower layers** (capture general information)
- **Early** in training (model still needs to adapt to target distribution)
- **Late** in training (model is close to convergence)
4.2.B – Optimization: Learning rates

- Discriminative fine-tuning ([Howard & Ruder, ACL 2018](https://doi.org/10.18653/v1/P18-1147))
  - Lower layers capture general information
  - Use lower learning rates for lower layers
  \[ \eta^{(i)} = \eta \times d_f^{\leftarrow i} \]
4.2.B – Optimization: Learning rates

- Discriminative fine-tuning
- Triangular learning rates (*Howard & Ruder, ACL 2018*)
  - Quickly move to a suitable region, then slowly converge over time
4.2.B – Optimization: Learning rates

- Discriminative fine-tuning
- Triangular learning rates ([Howard & Ruder, ACL 2018](https://doi.org/10.18653/v1/p18-1063))
  - Quickly move to a suitable region, then slowly converge over time
  - Also known as “learning rate warm-up”
  - Used e.g. in Transformer ([Vaswani et al., NIPS 2017](https://papers.neurips.cc/paper/8337-attention-for-all.pdf)) and Transformer-based methods (BERT, GPT)
  - Facilitates optimization; easier to escape suboptimal local minima
4.2.B – Optimization: Regularization

**Main idea:** minimize catastrophic forgetting by encouraging target model parameters to *stay* close to pretrained model parameters using a regularization term $\Omega$. 
4.2.B – Optimization: Regularization

- **Simple method:**
  Regularize new parameters not to deviate too much from pretrained ones (Wiese et al., CoNLL 2017):

\[ \Omega = \sum_1 \| L_i - L'_i \|_2 \]
4.2.B – Optimization: Regularization

- More advanced (elastic weight consolidation; EWC): Focus on parameters $\theta$ that are important for the pretrained task based on the Fisher information matrix $F$ (Kirkpatrick et al., PNAS 2017):

$$\Omega = \sum_i \frac{1}{2} F_i (\theta'_i - \theta_i)^2$$
EWC has downsides in continual learning:

- May over-constrain parameters
- Computational cost is linear in the number of tasks

(Schwarz et al., ICML 2018)
4.2.B – Optimization: Regularization

- If tasks are similar, we may also encourage source and target predictions to be close based on cross-entropy, similar to distillation:

\[ \Omega = \mathcal{H}(\hat{y}, \hat{y}') \]
Hands-on #5: Using discriminative learning
Discriminative learning rate can be implemented using two steps in our example:

First we organize the parameters of the various layers in labelled parameters groups in the optimizer:

```python
import re

# Build parameters groups by layer, numbered from the top ['1', '2', ..., '15']
parameter_groups = []
for i in range(args.num_layers):
    name_pattern = r'Transformer\.[^\s.]+\.' + str(i) + r'\.'
    group = {'name': str(args.num_layers - i),
             'params': [p for n, p in adaptation_model.named_parameters() if re.match(name_pattern, n)]}
    parameter_groups.append(group)

# Add the rest of the parameters (embeddings and classification layer) in a group labeled '0'
name_pattern = r'Transformer\.[^\s.]+\.*\.'
for n, p in adaptation_model.named_parameters() if not re.match(name_pattern, n)]
    parameter_groups.append(group)

# Sanity check that we still have the same number of parameters
assert sum(p.numel() for g in parameter_groups for p in g['params'])
assert sum(p.numel() for p in adaptation_model.parameters())

optimizer = torch.optim.Adam(parameter_groups, lr=adapt_args.lr)
```

We can then compute the learning rate of each group depending on its label (at each training iteration):

\[ \eta^i = \eta \times d_f^{-i} \]

Hyper-parameter
4.2.C – Optimization: Trade-offs

Several trade-offs when choosing which weights to update:

A. **Space** complexity
   
   *Task-specific modifications, additional parameters, parameter reuse*

B. **Time** complexity
   
   *Training time*

C. **Performance**
4.2.C – Optimization trade-offs: Space

Task-specific modifications

- Feature extraction: Many
- Adapters: Few
- Fine-tuning: None

Additional parameters

- Feature extraction: Many
- Adapters: Few
- Fine-tuning: None

Parameter reuse

- Feature extraction: All
- Adapters: None
- Fine-tuning: None
4.2.C – Optimization trade-offs: Time

Training time

Feature extraction

Adapters

Fine-tuning

Slow

Fast
Rule of thumb: If task source and target tasks are disimilar*, use feature extraction (Peters et al., 2019)

Otherwise, feature extraction and fine-tuning often perform similar

Fine-tuning BERT on textual similarity tasks works significantly better

Adapters achieve performance competitive with fine-tuning

Anecdotally, Transformers are easier to fine-tune (less sensitive to hyper-parameters) than LSTMs

*dissimilar: certain capabilities (e.g. modelling inter-sentence relations) are beneficial for target task, but pretrained model lacks them (see more later)
4.3 – Getting more signal

The target task is often a **low-resource** task. We can often improve the performance of transfer learning by combining a diverse set of signals:

A. From **fine-tuning** a single model on a single adaptation task....
   *The Basic: fine-tuning the model with a simple classification objective*

B. ... to **gathering signal** from other datasets and related tasks ...
   *Fine-tuning with Weak Supervision, Multi-tasking and Sequential Adaptation*

C. ... to **ensembling** models
   *Combining the predictions of several fine-tuned models*
4.3.A – Getting more signal: Basic fine-tuning

Simple example of fine-tuning on a text classification task:

A. Extract a single fixed-length vector from the model:
   hidden state of first/last token or mean/max of hidden-states

B. Project to the classification space with an additional classifier

C. Train with a classification objective

\[ \mathcal{L} = \mathcal{H}(y, \hat{y}) \]
4.3.B – Getting more signal: Related datasets/tasks

A. Sequential adaptation
   *Intermediate fine-tuning on related datasets and tasks*

B. Multi-task fine-tuning with related tasks
   *Such as NLI tasks in GLUE*

C. Dataset Slicing
   *When the model consistently underperforms on particular slices of the data*

D. Semi-supervised learning
   *Use unlabelled data to improve model consistency*
4.3.B – Getting more signal: Sequential adaptation

Fine-tuning on related high-resource dataset

1. Fine-tune model on related task with more data
4.3.B – Getting more signal: Sequential adaptation

Fine-tuning on related high-resource dataset

1. Fine-tune model on related task with more data
2. Fine-tune model on target task

- Helps particularly for tasks with limited data and similar tasks (Phang et al., 2018)
- Improves sample complexity on target task (Yogatama et al., 2019)
4.3.B – Getting more signal: Multi-task fine-tuning

Fine-tune the model jointly on related tasks

- For each optimization step, sample a task and a batch for training.
- Train via multi-task learning for a couple of epochs.

\[ \mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3 \]

1) \( T_{1-3} \), \( L_n \), \( L_1 \), \( E \)
4.3.B – Getting more signal: Multi-task fine-tuning

Fine-tune the model jointly on related tasks

- For each optimization step, sample a task and a batch for training.
- Train via multi-task learning for a couple of epochs.
- Fine-tune on the target task only for a few epochs at the end.
4.3.B – Getting more signal: Multi-task fine-tuning

Fine-tune the model with an unsupervised auxiliary task

- Language modelling is a related task!
- Fine-tuning the LM helps adapting the pretrained parameters to the target dataset.
- Helps even without pretraining (Rei et al., ACL 2017)
- Can optionally anneal ratio $\lambda$ (Chronopoulou et al., NAACL 2019)
- Used as a separate step in ULMFiT

$$\mathcal{L} = \mathcal{L}_1 + \lambda \mathcal{L}_2$$
4.3.B – Getting more signal: Dataset slicing

Use auxiliary heads that are trained **only on particular subsets** of the data

- Analyze errors of the model
- Use heuristics to automatically identify challenging subsets of the training data
- Train auxiliary heads jointly with main head

See also [Massive Multi-task Learning with Snorkel MeTaL](#)
4.3.B – Getting more signal: Semi-supervised learning

Can be used to make model predictions more consistent using unlabelled data

- Main idea: Minimize distance between predictions on original input $x$ and perturbed input $x'$
4.3.B – Getting more signal: Semi-supervised learning

Can be used to make model predictions more consistent using unlabelled data

- Perturbation can be noise, masking (Clark et al., EMNLP 2018), data augmentation, e.g. back-translation (Xie et al., 2019)
4.3.C – Getting more signal: Ensembling

Reaching the state-of-the-art by ensembling independently fine-tuned models

- **Ensembling** models
  
  *Combining the predictions of models fine-tuned with various hyper-parameters*

- **Knowledge distillation**
  
  *Distill an ensemble of fine-tuned models in a single smaller model*
4.3.C – Getting more signal: Ensembling

Combining the predictions of models fine-tuned with various hyper-parameters.

- Model fine-tuned...
  - on different tasks
  - on different dataset-splits
  - with different parameters (dropout, initializations...)
  - from variant of pre-trained models (e.g. cased/uncased)

\[ Q(c \mid x) = \text{avg}([Q^1, Q^2, Q^3]) \]
4.3.C – Getting more signal: Distilling

Distilling ensembles of large models back in a single model

- Knowledge distillation: train a student model on soft targets produced by the teacher (the ensemble)

\[- \sum_{c} Q(c \mid X) \log(P_{r}(c \mid X))\]

- Relative probabilities of the teacher labels contain information about how the teacher generalizes

\[Q(c \mid x) = \text{avg}([Q^1, Q^2, Q^3])\]
Hands-on #6:
Using multi-task learning
Multitasking with a classification loss + language modeling loss.

Create **two heads**:
- language modeling head
- classification head

Total loss is a **weighted sum** of
- language modeling loss and
- classification loss
Hands-on: Multi-task learning

We use a coefficient of 1.0 for the classification loss and 0.5 for the language modeling loss and fine-tune a little longer (6 epochs instead of 3 epochs, the validation loss was still decreasing).

Multi-tasking helped us improve over single-task full-model fine-tuning!
Agenda

[1] Introduction

[2] Pretraining


[4] Adaptation

[5] Downstream

5. Downstream applications
Hands-on examples
5. Downstream applications - Hands-on examples

In this section we will explore downstream applications and practical considerations along two orthogonal directions:

A. What are the various applications of transfer learning in NLP
   Document/sequence classification, Token-level classification, Structured prediction and Language generation

B. How to leverage several frameworks & libraries for practical applications
   Tensorflow, PyTorch, Keras and third-party libraries like fast.ai, HuggingFace...
Frameworks & libraries: practical considerations

- Pretraining large-scale models is costly
  
  *Use open-source models*

  *Share your pretrained models*

- Sharing/accessing pretrained models
  
  - **Hubs**: Tensorflow Hub, PyTorch Hub
  
  - **Author released** checkpoints: ex BERT, GPT...
  
  - **Third-party** libraries: AllenNLP, fast.ai, HuggingFace

- Design considerations
  
  - **Hubs/libraries**:
    
    - Simple to use but can be difficult to modify model internal architecture
  
  - **Author released checkpoints**:
    
    - More difficult to use but you have full control over the model internals

---

<table>
<thead>
<tr>
<th>Consumption</th>
<th>CO$_2$e (lbs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Air travel, 1 passenger, NY↔SF</td>
<td>1984</td>
</tr>
<tr>
<td>Human life, avg, 1 year</td>
<td>11,023</td>
</tr>
<tr>
<td>American life, avg, 1 year</td>
<td>36,156</td>
</tr>
<tr>
<td>Car, avg incl. fuel, 1 lifetime</td>
<td>126,000</td>
</tr>
</tbody>
</table>

**Training one model**

- SOTA NLP model (tagging)                          | 13            |
  
  - w/ tuning & experimentation                      | 33,486        |

- Transformer (large)                               | 121           |
  
  - w/ neural architecture search                   | 394,863       |
5. Downstream applications - Hands-on examples

A. Sequence and document level classification
   *Hands-on: Document level classification (fast.ai)*

B. Token level classification
   *Hands-on: Question answering (Google BERT & Tensorflow/TF Hub)*

C. Language generation
   *Hands-on: Dialog Generation (OpenAI GPT & HuggingFace/PyTorch Hub)*
5.A – Sequence & document level classification

Transfer learning for document classification using the fast.ai library.

- **Target task:**
  
  *IMDB: a binary sentiment classification dataset containing 25k highly polar movie reviews for training, 25k for testing and additional unlabeled data.*
  

- **Fast.ai** has in particular:
  
  - a pre-trained English model available for download
  - a standardized data block API
  - easy access to standard datasets like IMDB

- **Fast.ai** is based on PyTorch
fast.ai gives access to many high-level API out-of-the-box for vision, text, tabular data and collaborative filtering.

The library is designed for speed of experimentation, e.g. by importing all necessary modules at once in interactive computing environments, like:

```python
from fastai.text import *  # Quick access to NLP functionality
```

Fast.ai then comprises all the high level modules needed to quickly setup a transfer learning experiment.

Load IMDB dataset & inspect it.

DataBunch for the language model and the classifier

Load an AWD-LSTM (Merity et al., 2017) pretrained on WikiText-103 & fine-tune it on IMDB using the language modeling loss.
5.A – Document level classification using fast.ai

Once we have a fine-tune language model (AWD-LSTM), we can create a text classifier by adding a classification head with:

- A layer to concatenate the final outputs of the RNN with the maximum and average of all the intermediate outputs (along the sequence length)
- Two blocks of $\text{nn.BatchNorm1d} \circ \text{nn.Dropout} \circ \text{nn.Linear} \circ \text{nn.ReLU}$ with a hidden dimension of 50.

Now we fine-tune in two steps:

1. train the classification head only while keeping the language model frozen, and

2. fine-tune the whole architecture.

Colab: [http://tiny.cc/NAACLTtransferFastAicolab](http://tiny.cc/NAACLTtransferFastAicolab)
5.B – Token level classification: BERT & Tensorflow

Transfer learning for token level classification: Google’s BERT in TensorFlow.

- Target task:
  SQuAD: a question answering dataset.
  [https://rajpurkar.github.io/SQuAD-explorer/](https://rajpurkar.github.io/SQuAD-explorer/)

- In this example we will directly use a Tensorflow checkpoint
  - Example: [https://github.com/google-research/bert](https://github.com/google-research/bert)
  - We use the usual Tensorflow workflow: create model graph comprising the core model and the added/modified elements
  - Take care of variable assignments when loading the checkpoint
Let’s adapt BERT to the target task. Keep our core model unchanged.

Replace the pre-training head (language modeling) with a classification head: a linear projection layer to estimate 2 probabilities for each token:

- being the start of an answer
- being the end of an answer.

```python
def create_model(bert_config, is_training, input_ids, input_mask, segment_ids, use one hot embeddings):
    """Creates a classification model."""
    model = modeling.BertModel(
        config=bert_config,
        is_training=is_training,
        input_ids=input_ids,
        input_mask=input_mask,
        token_type_ids=segment_ids,
        use one hot embeddings=use one hot_embeddings)

    final_hidden = model.get_sequence_output()

    final_hidden_shape = modeling.get_shape_list(final_hidden, expected_rank=3)
    batch_size = final_hidden_shape[0]
    seq_length = final_hidden_shape[1]
    hidden_size = final_hidden_shape[2]

    output_weights = tf.get_variable("cls/squad/output_weights", [2, hidden_size],
                                     initializer=tf.truncated_normal_initializer(stddev=0.02))

    output_bias = tf.get_variable("cls/squad/output_bias", [2],
                                   initializer=tf.zeros_initializer())

    final_hidden_matrix = tf.reshape(final_hidden,
                                      [batch_size * seq_length, hidden_size])
    logits = tf.matmul(final_hidden_matrix, output_weights, transpose_b=True)
    logits = tf.nn.bias_add(logits, output_bias)

    logits = tf.reshape(logits, [batch_size, seq_length, 2])
    logits = tf.transpose(logits, [2, 0, 1])

    unstacked_logits = tf.unstack(logits, axis=0)

    (start_logits, end_logits) = (unstacked_logits[0], unstacked_logits[1])

    return (start_logits, end_logits)
```
5.B – SQuAD with BERT & Tensorflow

Load our pretrained checkpoint

To load our checkpoint, we just need to setup an assignment_map from the variables of the checkpoint to the model variable, keeping only the variables in the model.

And we can use `tf.train.init_from_checkpoint`

```python
def get_assignment_map_from_checkpoint(tvars, init_checkpoint):
    """Compute the union of the current variables and checkpoint variables."""
    assignment_map = {}
    initialized_variable_names = {}

    name_to_variable = collections.OrderedDict()
    for var in tvars:
        name = var.name
        m = re.match("^\(.*;\d+\)$", name)
        if m is not None:
            name = m.group(1)
        name_to_variable[name] = var

    init_vars = tf.train.list_variables(init_checkpoint)

    assignment_map = collections.OrderedDict()
    for x in init_vars:
        [name, var] = x
        if name not in name_to_variable:
            continue
        assignment_map[name] = name
        initialized_variable_names[name] = 1
        initialized_variable_names[name + ":0"] = 1

    return (assignment_map, initialized_variable_names)

(start_logits, end_logits) = create_model(
    bert_config=bert_config,
    is_training=is_training,
    input_ids=input_ids,
    input_mask=input_mask,
    segment_ids=segment_ids,
    use_one_hot_embeddings=use_one_hot_embeddings)

tvars = tf.trainable_variables()

{assignment_map,
 initialized_variable_names} = get_assignment_map_from_checkpoint(tvars, init_checkpoint)

tf.train.init_from_checkpoint(init_checkpoint, assignment_map)
```
TensorFlow-Hub

Working directly with TensorFlow requires to have access to—and include in your code—the full code of the pretrained model. TensorFlow Hub is a library for sharing machine learning models as self-contained pieces of TensorFlow graph with their weights and assets. Modules are automatically downloaded and cached when instantiated.

Each time a module $m$ is called e.g. $y = m(x)$, it adds operations to the current TensorFlow graph to compute $y$ from $x$. 

```python
def create_model(bert_config, is_training, input_ids, input_mask, segment_ids, 
                 use_one_hot_embeddings):
    """Creates a classification model."""

    model = modeling.BertModel(
        config=bert_config, 
        is_training=is_training, 
        input_ids=input_ids, 
        input_mask=input_mask, 
        token_type_ids=segment_ids, 
        use_one_hot_embeddings=use_one_hot_embeddings)

    final_hidden = model.get_sequence_output()

model = hub.Module(
    BERT_MODEL_HUB, trainable=True)

bert_inputs = dict(
    input_ids=input_ids, input_mask=input_mask, segment_ids=segment_ids)

bert_outputs = model(inputs=bert_inputs, signature="tokens", as_dict=True)

# Use "pooled_output" for classification tasks on an entire sentence. 
# Use "sequence_outputs" for token-level output.
final_hidden = bert_outputs["sequence_outputs"]
```
Tensorflow Hub host a nice selection of pretrained models for NLP

Tensorflow Hub can also used with Keras exactly how we saw in the BERT example

The main limitations of Hubs are:

- No access to the source code of the model (*black-box*)
- Not possible to modify the internals of the model (e.g. *to add Adapters*)
Transfer learning for language generation: OpenAI GPT and HuggingFace library.

- **Target task:**
  
  *ConvAI2 – The 2nd Conversational Intelligence Challenge for training and evaluating models for non-goal-oriented dialogue systems, i.e. chit-chat*
  
  [http://convai.io](http://convai.io)

- **HuggingFace library of pretrained models**
  
  - A repository of large scale pre-trained models with BERT, GPT, GPT-2, Transformer-XL
  
  - Provide an easy way to download, instantiate and train pre-trained models in PyTorch

- **HuggingFace’s models are now also accessible using PyTorch Hub**
A dialog generation task:

Language generation tasks are close to the language modeling pre-training objective, but:

- Language modeling pre-training involves a single input: a sequence of words.
- In a dialog setting: several types of contexts are provided to generate an output sequence:
  - knowledge base: persona sentences,
  - history of the dialog: at least the last utterance from the user,
  - tokens of the output sequence that have already been generated.

How should we adapt the model?
Several options:
- Duplicate the model to initialize an encoder-decoder structure
e.g. Lample & Conneau, 2019
- Use a single model with concatenated inputs
see e.g. Wolf et al., 2019, Khandelwal et al. 2019

Concatenate the various context separated by delimiters and add position and segment embeddings
Let’s import pretrained versions of OpenAI GPT tokenizer and model.

And add a few new tokens to the vocabulary.

Now most of the work is about preparing the inputs for the model.

We organize the contexts in segments.

Add delimiter at the extremities of the segments.

And build our word, position and segment inputs for the model.

Then train our model using the pretraining language modeling objective.
PyTorch Hub

Last Friday, the PyTorch team soft-launched a beta version of *PyTorch Hub*. Let’s have a quick look.

- PyTorch Hub is based on [GitHub repositories](https://github.com).
- A model is shared by adding a *hubconf.py* script to the root of a GitHub repository.
- Both **model definitions** and **pre-trained weights** can be shared.

In our case, to use `torch.hub` instead of `pytorch-pretrained-bert`, we can simply call `torch.hub.load` with the path to `pytorch-pretrained-bert` GitHub repository:

```python
import torch

tokenizer = torch.hub.load('huggingface/pytorch-pretrained-BERT', 'openAIGPTTokenizer', 'openai-gpt')
model = torch.hub.load('huggingface/pytorch-pretrained-BERT', 'openAIGPTLMHeadModel', 'openai-gpt')
```

PyTorch Hub will fetch the model from the **master branch** on GitHub. This means that you don’t need to package your model (pip) & users will always access the most recent version (master).
Agenda

[1] Introduction

[2] Pretraining


[4] Adaptation

[5] Downstream

6. Open problems and future directions
6. Open problems and future directions

A. Shortcomings of pretrained language models
B. Pretraining tasks
C. Tasks and task similarity
D. Continual learning and meta-learning
E. Bias
Shortcomings of pretrained language models

- Recap: LM can be seen as a general pretraining task; with enough data, compute, and capacity a LM can learn a lot.
- In practice, many things that are less represented in text are harder to learn
- Pretrained language models are bad at
  - fine-grained linguistic tasks ([Liu et al., NAACL 2019](#))
  - common sense (when you actually make it difficult; [Zellers et al., ACL 2019](#)); natural language generation (maintaining long-term dependencies, relations, coherence, etc.)
  - tend to overfit to surface form information when fine-tuned; ‘rapid surface learners’
  - ...
Shortcomings of pretrained language models

Large, pretrained language models can be difficult to optimize.

- Fine-tuning is often **unstable** and has a **high variance**, particularly if the target datasets are very small.
- Devlin et al. (NAACL 2019) note that large (24-layer) version of BERT is particularly prone to degenerate performance; multiple random restarts are sometimes necessary as also investigated in detail in (Phang et al., 2018)
Shortcomings of pretrained language models

Current pretrained language models are very large.

- Do we really need all these parameters?
- Recent work shows that only a few of the attention heads in BERT are required (Voita et al., ACL 2019).
- More work needed to understand model parameters.
- Pruning and distillation are two ways to deal with this.
- See also: the lottery ticket hypothesis (Frankle et al., ICLR 2019).
Pretraining tasks

Shortcomings of the language modeling objective:

- Not appropriate for all models
  - If we condition on more inputs, need to pretrain those parts
  - E.g. the decoder in sequence-to-sequence learning (Song et al., ICML 2019)
- Left-to-right bias not always be best
  - Objectives that take into account more context (such as masking) seem useful (less sample-efficient)
  - Possible to combine different LM variants (Dong et al., 2019)
- Weak signal for semantics and long-term context vs. strong signal for syntax and short-term word co-occurrences
  - Need incentives that promote encoding what we care about, e.g. semantics
More diverse self-supervised objectives

- Taking inspiration from computer vision
- Self-supervision in language mostly based on word co-occurrence \cite{Ando:2005}
- Supervision on different levels of meaning
  - Discourse, document, sentence, etc.
  - Using other signals, e.g. meta-data
- Emphasizing different qualities of language

Example:

Sampling a patch and a neighbour and predicting their spatial configuration \cite{Doersch:2015}

Image colorization \cite{Zhang:2016}
Pretraining tasks

Specialized pretraining tasks that teach what our model is missing

- Develop **specialized pretraining tasks** that explicitly learn such relationships
  - Word-pair relations that capture background knowledge ([Joshi et al., NAACL 2019](#))
  - Span-level representations ([Swayamdipta et al., EMNLP 2018](#))
  - Different pretrained word embeddings are helpful ([Kiela et al., EMNLP 2018](#))

- Other pretraining tasks could explicitly learn **reasoning** or **understanding**
  - Arithmetic, temporal, causal, etc.; discourse, narrative, conversation, etc.

- Pretrained representations could be **connected in a sparse and modular way**
  - Based on linguistic substructures ([Andreas et al., NAACL 2016](#)) or experts ([Shazeer et al., ICLR 2017](#))
Pretraining tasks

Need for grounded representations

- Limits of distributional hypothesis—difficult to learn certain types of information from raw text
  - Human reporting bias: not stating the obvious (Gordon and Van Durme, AKBC 2013)
  - Common sense isn't written down
  - Facts about named entities
  - No grounding to other modalities

- Possible solutions:
  - Incorporate other structured knowledge (e.g. knowledge bases like ERNIE, Zhang et al 2019)
  - Multimodal learning (e.g. with visual representations like VideoBERT, Sun et al. 2019)
  - Interactive/human-in-the-loop approaches (e.g. dialog, Hancock et al. 2018)
Tasks and task similarity

Many tasks can be expressed as **variants of language modeling**

- Language itself can directly be used to specify tasks, inputs, and outputs, e.g. by **framing as QA** ([McCann et al., 2018](#))
- **Dialog-based learning** without supervision by forward prediction ([Weston, NIPS 2016](#))
- NLP tasks formulated as **cloze prediction objective** (Children Book Test, LAMBADA, Winograd, ...)
- **Triggering task behaviors via prompts** e.g. *TL; DR; translation prompt* ([Radford, Wu et al. 2019](#)); enables zero-shot adaptation
- Questioning the notion of a “task” in NLP
Tasks and task similarity

- Intuitive similarity of pretraining and target tasks (NLI, classification) correlates with better downstream performance.
- Do not have a clear understanding of when and how two tasks are similar and relate to each other.
- One way to gain more understanding: Large-scale empirical studies of transfer such as Taskonomy (Zamir et al., CVPR 2018).
- Should be helpful for designing better and specialized pretraining tasks.
Continual and meta-learning

- Current transfer learning **performs adaptation once**.
-Ultimately, we’d like to have models that continue to **retain and accumulate knowledge** across many tasks ([Yogatama et al., 2019](#)).
- No distinction between pretraining and adaptation; just **one stream of tasks**.
- Main challenge towards this: **Catastrophic forgetting**.
- Different approaches from the literature:
  - Memory, regularization, task-specific weights, etc.
Continual and meta-learning

- Objective of transfer learning: Learn a representation that is **general** and **useful** for many tasks.

- Objective **does not incentivize ease of adaptation** (often unstable); **does not learn how to adapt it**.

- Meta-learning combined with transfer learning could make this more feasible.

- However, most existing approaches are **restricted to the few-shot setting** and only **learn a few steps of adaptation**.
Bias

- Bias has been shown to be pervasive in word embeddings and neural models in general.
- Large pretrained models necessarily have their own sets of biases.
- There is a blurry boundary between common-sense and bias.
- We need ways to remove such biases during adaptation.
- A small fine-tuned model should be harder to misuse.
Conclusion

- Themes: words-in-context, LM pretraining, deep models
- Pretraining gives better sample-efficiency, can be scaled up
- Predictive of certain features—depends how you look at it
- Performance trade-offs, from top-to-bottom
- Transfer learning is simple to implement, practically useful
- Still many shortcomings and open problems
Questions?

- Twitter: #NAACLTransfer
- Whova: “Questions for the tutorial on Transfer Learning in NLP” topic

- Slides: http://tiny.cc/NAACLTransfer
- Colab: http://tiny.cc/NAACLTransferColab
- Code: http://tiny.cc/NAACLTransferCode
Extra slides
Why transfer learning in NLP? (Empirically)

Question Answering on SQuAD2.0

https://paperswithcode.com/sota/question-answering-on-squad20
*General Language Understanding Evaluation (GLUE; Wang et al., 2019): includes 11 diverse NLP tasks
**GLoVe**: very large scale (840B tokens), co-occurrence based. Learns linear relationships (SOTA word analogy) ([Pennington et al., 2014](#)).

**fastText**: incorporates subword information ([Bojanowski et al., 2017](#)).
### Semi-supervised Sequence Modeling with Cross-View Training

**Learning on a Labeled Example**

```
"She lives in Washington."
```

**Location**

```
BiLSTM Encoder -> Primary Prediction Module -> p_θ
```

---

**Learning on an Unlabeled Example**

```
"They traveled to Washington by plane"
```

**Auxiliary Prediction Modules**

- Auxiliary 1: They traveled to __________
- Auxiliary 2: They traveled to Washington __________
- Auxiliary 3: __________ Washington by plane
- Auxiliary 4: __________ by plane

---

### SOTA sequence modeling results

<table>
<thead>
<tr>
<th>Method</th>
<th>CCG Acc</th>
<th>Chunk F1</th>
<th>NER F1</th>
<th>FGN F1</th>
<th>POS Acc</th>
<th>Dep. UAS</th>
<th>Parse LAS</th>
<th>Translate BLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortcut LSTM (Wu et al., 2017)</td>
<td>95.1</td>
<td>90.7</td>
<td>86.8</td>
<td>97.53</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID-CNN-CRF (Strubell et al., 2017)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMT† (Hashimoto et al., 2017)</td>
<td>95.8</td>
<td></td>
<td>97.55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TagLM* (Peters et al., 2017)</td>
<td>96.4</td>
<td>91.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELMo* (Peters et al., 2018)</td>
<td>92.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Biaffine (Dozat and Manning, 2017)</td>
<td>95.7</td>
<td>94.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack Pointer (Ma et al., 2018)</td>
<td>95.9</td>
<td>94.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stanford (Luong and Manning, 2015)</td>
<td>94.9</td>
<td>95.1</td>
<td>91.2</td>
<td>87.5</td>
<td>97.60</td>
<td>95.1</td>
<td>93.3</td>
<td>28.9</td>
</tr>
<tr>
<td>Google (Luong et al., 2017)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supervised</td>
<td>95.1</td>
<td>95.1</td>
<td>91.8</td>
<td>87.9</td>
<td>97.64</td>
<td>95.4</td>
<td>93.7</td>
<td></td>
</tr>
<tr>
<td>Virtual Adversarial Training*</td>
<td>95.2</td>
<td>95.8</td>
<td>92.1</td>
<td>88.1</td>
<td>97.66</td>
<td>95.6</td>
<td>93.8</td>
<td>29.3</td>
</tr>
<tr>
<td>Word Dropout*</td>
<td>95.8</td>
<td>96.5</td>
<td>92.2</td>
<td>88.5</td>
<td>97.72</td>
<td>96.2</td>
<td>94.4</td>
<td></td>
</tr>
<tr>
<td>ELMo (our implementation)*</td>
<td>95.9</td>
<td>96.8</td>
<td>92.3</td>
<td>88.4</td>
<td>97.79</td>
<td>96.4</td>
<td>94.4</td>
<td></td>
</tr>
<tr>
<td>ELMo + Multi-task*†</td>
<td>95.7</td>
<td>96.6</td>
<td>92.3</td>
<td>88.7</td>
<td>97.70</td>
<td>95.9</td>
<td>94.1</td>
<td>29.6</td>
</tr>
<tr>
<td>CVT*</td>
<td>96.0</td>
<td>96.9</td>
<td>92.4</td>
<td>88.4</td>
<td>97.76</td>
<td>96.4</td>
<td>94.1</td>
<td></td>
</tr>
<tr>
<td>CVT + Multi-task†</td>
<td>96.1</td>
<td>97.0</td>
<td>92.6</td>
<td>88.8</td>
<td>97.74</td>
<td>96.6</td>
<td>95.0</td>
<td></td>
</tr>
<tr>
<td>CVT + Multi-task + Large†</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Clark et al. EMNLP 2018)
Pretrain bidirectional character level model, extract embeddings from first/last character

SOTA CoNLL 2003 NER results

<table>
<thead>
<tr>
<th>Task</th>
<th>PROPOSED</th>
<th>Previous best</th>
</tr>
</thead>
<tbody>
<tr>
<td>NER English</td>
<td>93.09±0.12</td>
<td>92.22±0.1</td>
</tr>
<tr>
<td>(Peters et al., 2018)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NER German</td>
<td>88.32±0.2</td>
<td>78.76</td>
</tr>
<tr>
<td>(Lample et al., 2016)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chunking</td>
<td>96.72±0.05</td>
<td>96.37±0.05</td>
</tr>
<tr>
<td>(Peters et al., 2017)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PoS tagging</td>
<td>97.85±0.01</td>
<td>97.64</td>
</tr>
<tr>
<td>(Choi, 2016)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Akbik et al., COLING 2018) (see also Akbik et al., NAACL 2019)
Cloze-driven Pretraining of Self-attention Networks

Pretraining

Fine-tuning

SOTA NER and PTB constituency parsing, ~3.3% less than BERT-large for GLUE

Baevski et al. (2019)
UniLM - Dong et al., 2019

Model is jointly pretrained on three variants of LM (bidirectional, left-to-right, seq-to-seq)

SOTA on three natural language generation tasks
Masked Sequence to Sequence Pretraining (MASS)

Pretrain encoder-decoder

(Song et al., ICML 2019)
Probing tasks for sentential features:

- Bag-of-Vectors is surprisingly good at capturing sentence-level properties, thanks to redundancies in natural linguistic input.
- BiLSTM-based models are better than CNN-based models at capturing interesting linguistic knowledge, with same objective.
- Objective matters - training on NLI is bad. Most tasks are structured so a seq 2 tree objective works best.
- Supervised objectives for sentence embeddings do better than unsupervised, like SkipThought (Kiros et al.)
From lower to higher layers, information goes from general to task-specific.
Other methods for analysis

- Textual omission and multi-modal: Kadar et al., 2016

- Adversarial Approaches
  - Adversary: input which differs from original just enough to change the desired prediction
    - SQuAD: Jia & Liang, 2017
    - NLI: Glockner et al., 2018; Minervini & Riedel, 2018
    - Machine Translation: Belinkov & Bisk, 2018
  - Requires identification (manual or automatic) of inputs to modify.
Analysis: Inputs and Outputs

What to analyze?

- Embeddings
  - Word types and tokens
  - Sentence
  - Document
- Network Activations
  - RNNs
  - CNNs
  - Feed-forward nets
- Layers
- Pretraining Objectives

What to look for?

- Surface-level features
- Lexical features
  - E.g. POS tags
- Morphology
- Syntactic Structure
  - Word-level
  - Sentence-level
- Semantic Structure
  - E.g. Roles, Coreference

Belinkov et al. (2019) — More details in Table 1
Analysis: Methods

- **Visualization:**
  - 2-D plots
  - Attention mechanisms
  - Network activations

- **Model Probes:**
  - Surface-level features
  - Syntactic features
  - Semantic features

- **Model Alterations:**
  - Network Erasure
  - Perturbations

* Not hard and fast categories
On 31 December 1687 the first organized group of Huguenots set sail from the Netherlands to the Dutch East India Company post at the Cape of Good Hope. The largest portion of the Huguenots to settle in the Cape arrived between 1688 and 1689 in seven ships as part of the organised migration, but quite a few arrived as late as 1700; thereafter the numbers declined and only small groups arrived at a time.

The number of old Acadian colonists declined after the year 1675.

The number of new Huguenot colonists declined after what year?

How does this say what’s in a representation?

Roundabout: what’s wrong with a representation...
Probes are simple linear / neural layers

Liu et al., NAACL 2019
Interpretability is difficult \cite{lipton2016}. Many variables make synthesis challenging. Choice of model architecture, pretraining objective determines informativeness of representations.

Interpretability is important, but not enough on its own. Interpretability + transferability to downstream tasks is key - that’s next!

Transferability to downstream tasks