Transfer Learning in Natural Language Processing

June 2, 2019
NAACL-HLT 2019

Sebastian Ruder

AllenNLP Matthew Peters

Swabha Swayamdipta AI2

Thomas Wolf

Transfer Learning in NLP

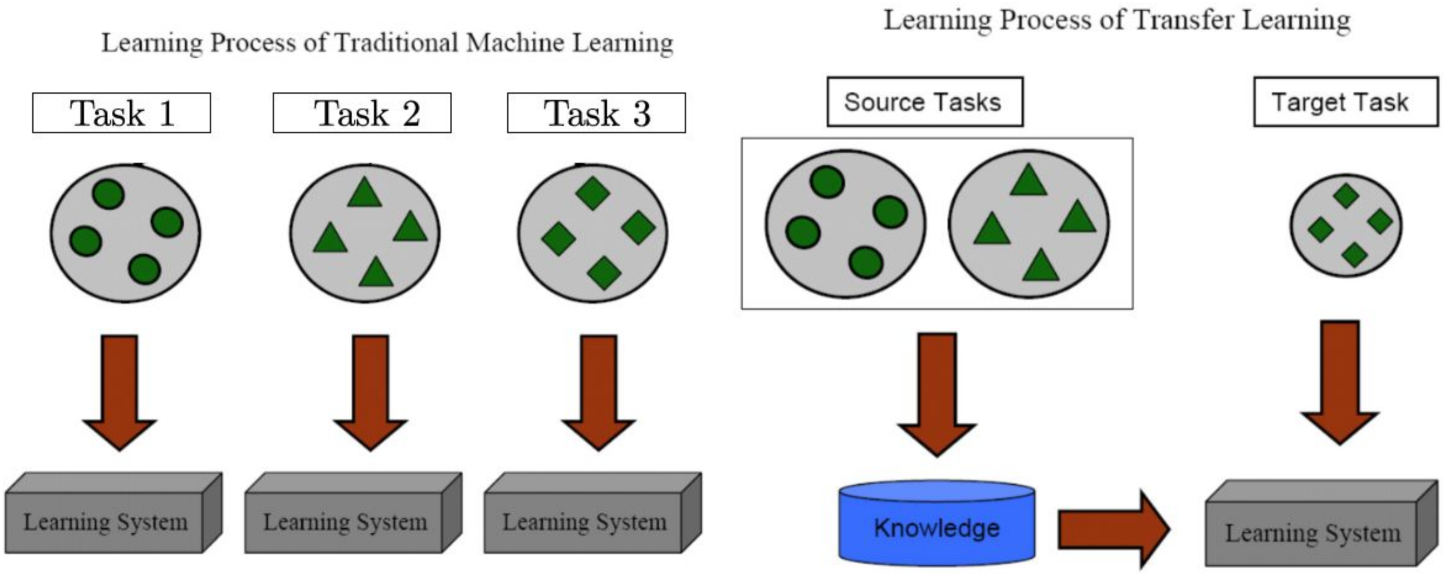
Follow along with the tutorial:

- ❑ Slides: http://tiny.cc/NAACLTransfer
- ❑ Colab: http://tiny.cc/NAACLTransferColab
- ❑ Code: http://tiny.cc/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

What is transfer learning?



(a) Traditional Machine Learning

(b) Transfer Learning

[Pan and Yang (2010)]

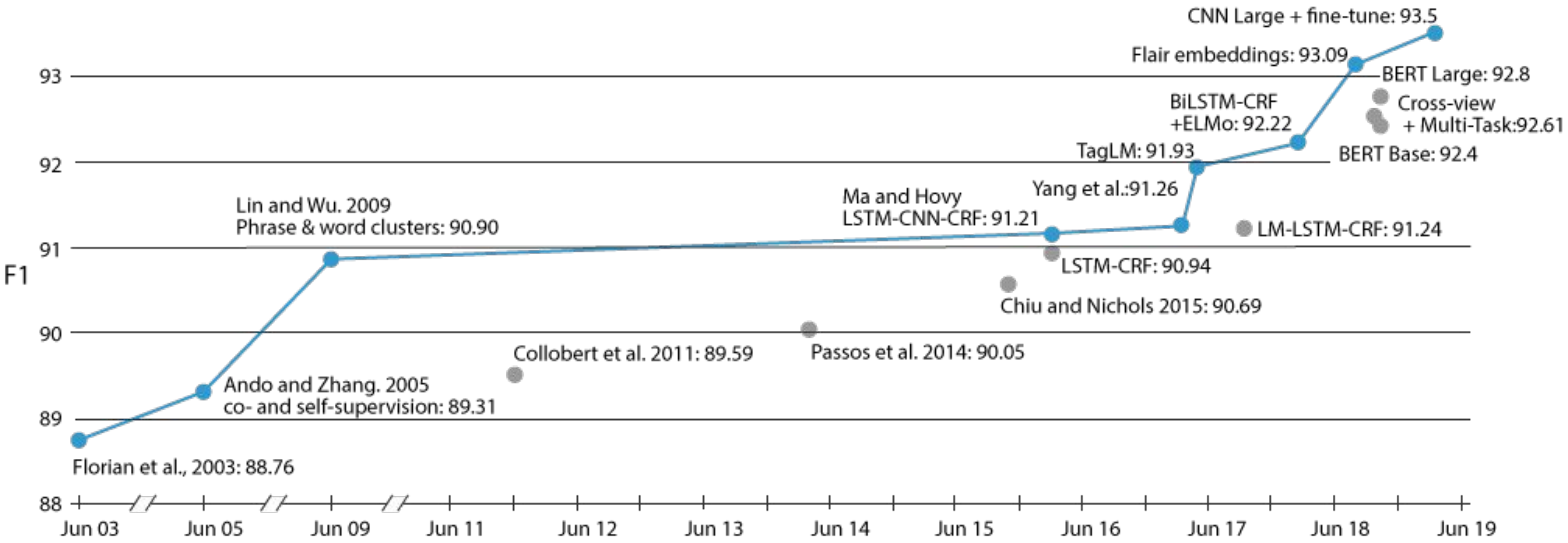
Why transfer learning in NLP?

- ❑ 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.

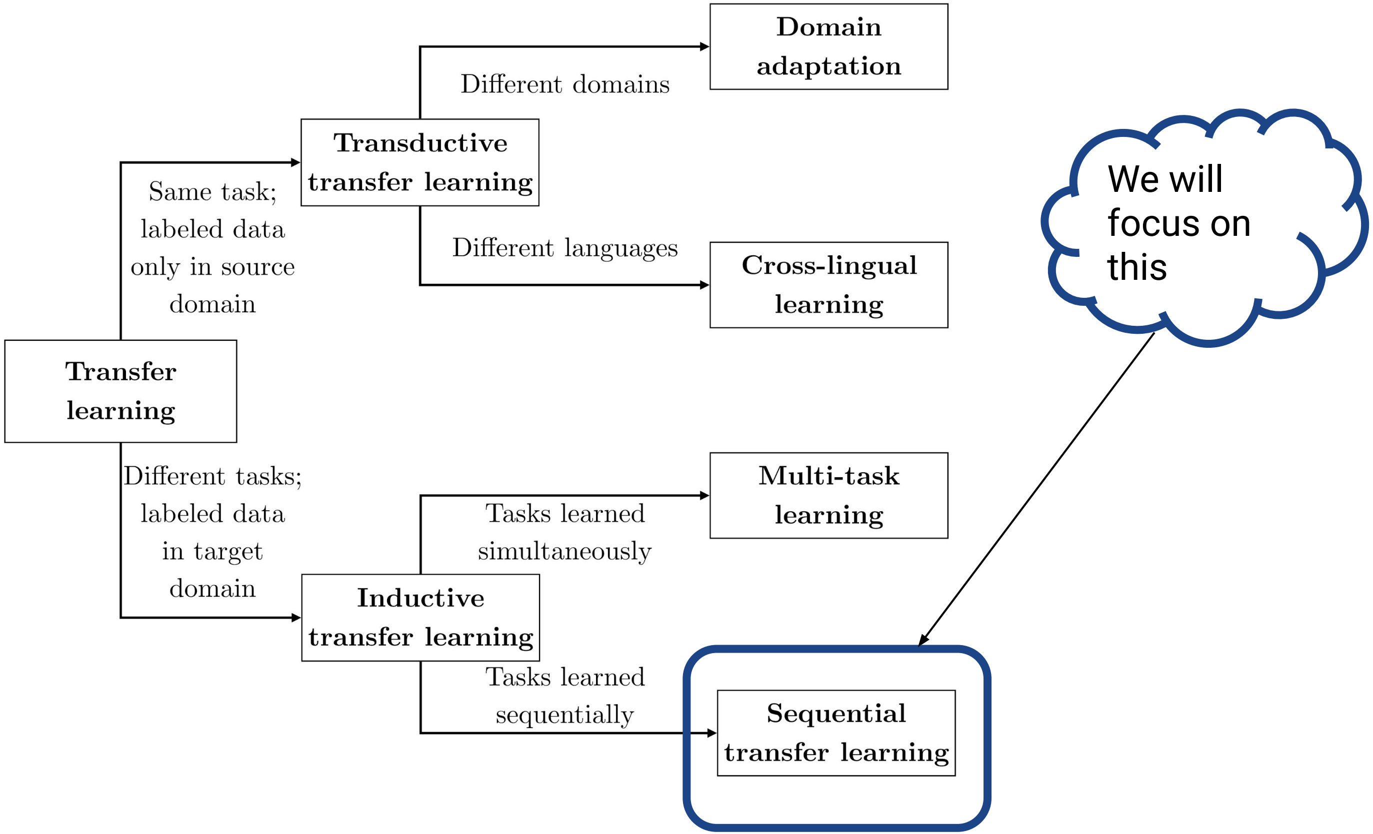
- ❑ Empirically, transfer learning has resulted in SOTA for many supervised NLP tasks (e.g. classification, information extraction, Q&A, etc).

Why transfer learning in NLP? (Empirically)

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



Types of transfer learning in NLP



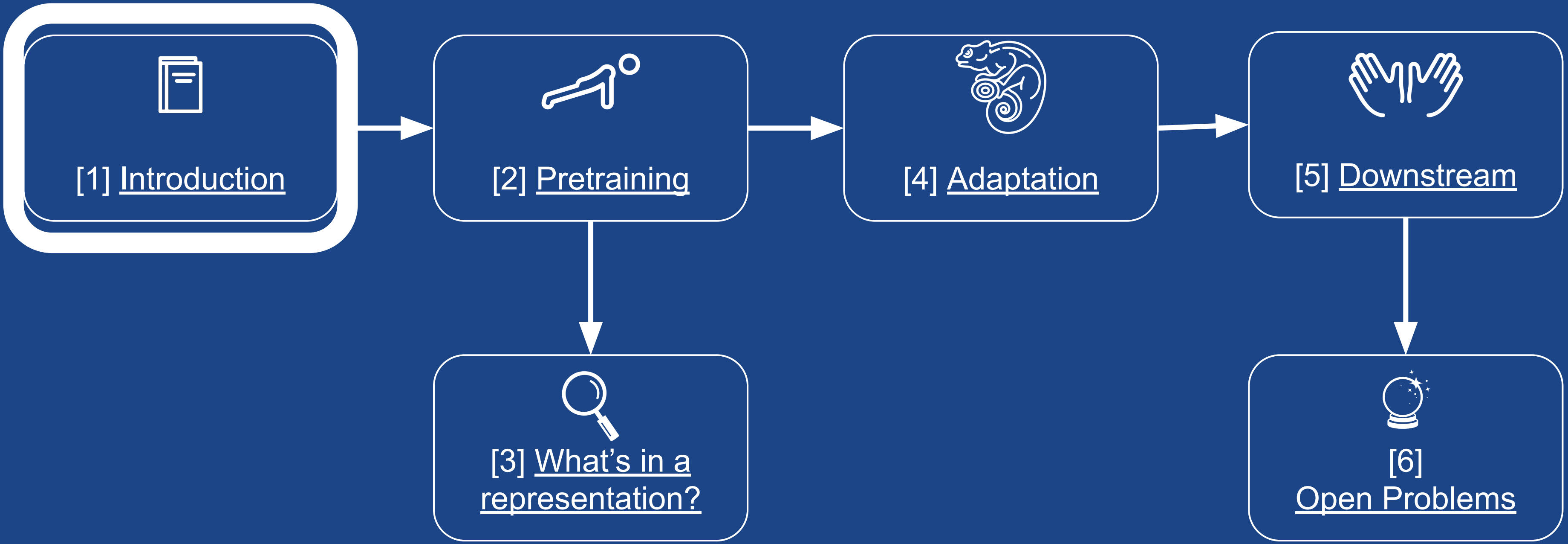
Ruder (2019)

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.)

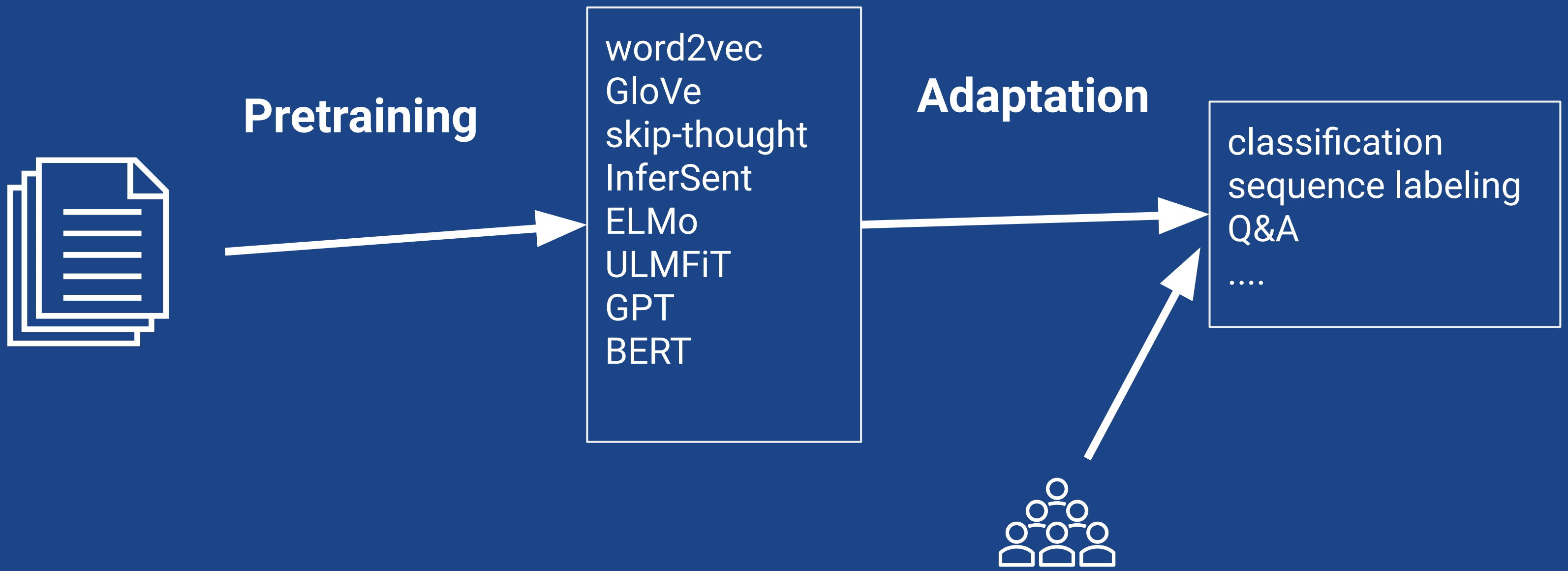
Agenda


[1] Introduction
[2] Pretraining
[3] What's in a representation?
[4] Adaptation
[5] Downstream
[6] Open Problems

1. Introduction

Sequential transfer learning

Learn on one task / dataset, then transfer to another task / dataset



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, …]

PRP	VBP	PRP	NN	CC	NN	.
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, …]

PRP	VBP	PRP	NN	CC	NN	.
I	love	my	cat	and	dog	.

I love my cat and dog . }-> “positive"

Major Themes

Major themes: From words to words-in-context

Word vectors

cats = [0.2, -0.3, ...]

dogs = [0.4, -0.5, ...]

Sentence / doc vectors

$\left.\begin{array}{l}\text{We have two}\\ \text{cats.}\end{array}\right\}$ [-1.2, 0.0, ...]

$\left.\begin{array}{l}\text{It's raining}\\ \text{cats and dogs.}\end{array}\right\}$ [0.8, 0.9, ...]

Word-in-context vectors

We have two $\overbrace{\text{cats}}^{[1.2,\ -0.3,\ \ldots]}$.

It's raining $\overbrace{\text{cats}}^{[-0.4,\ 0.9,\ \ldots]}$ and dogs.

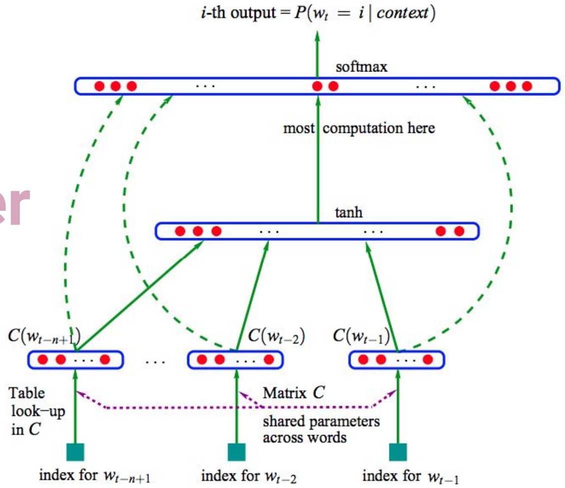
Major themes: LM pretraining

- Many successful pretraining approaches are based on language modeling
- Informally, a LM learns $P_\Theta(\textit{text})$ or $P_\Theta(\textit{text} \mid \textit{some other text})$

- 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

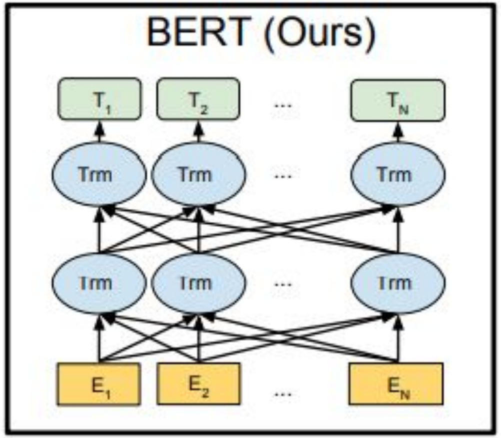
Major themes: From shallow to deep

1 layer



[Bengio et al 2003: A Neural Probabilistic Language Model]

24 layers



[Devlin et al 2019: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding]

Major themes: pretraining vs target task

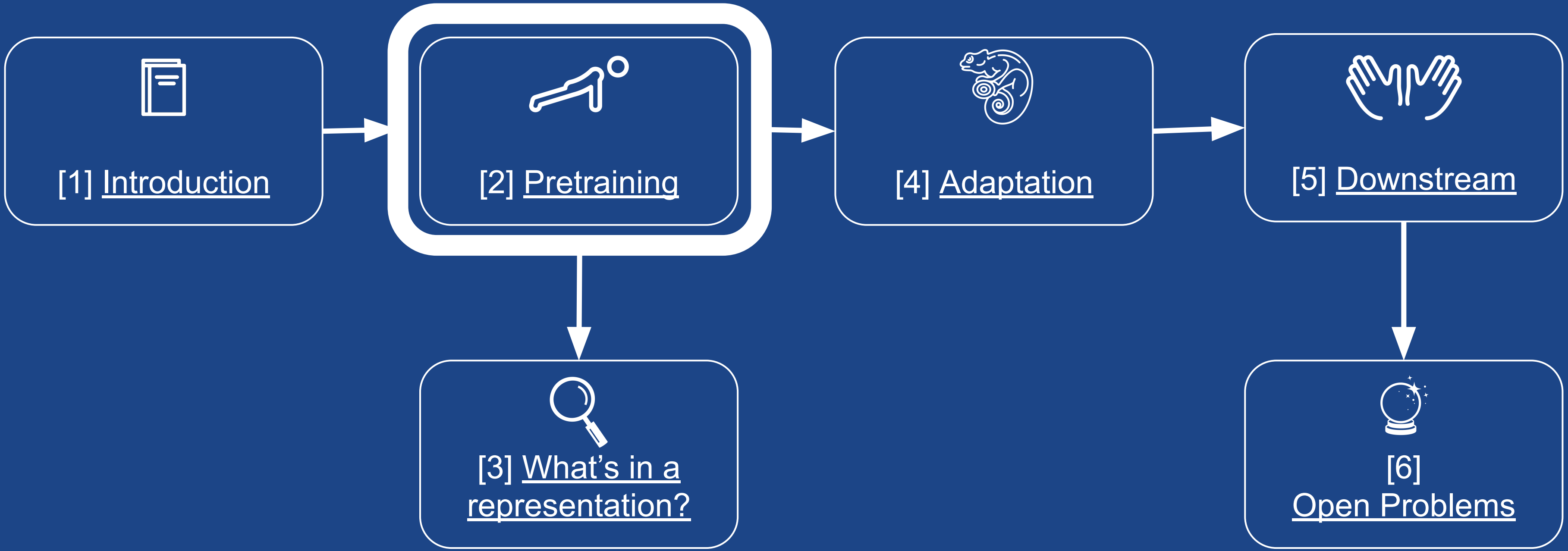
Choice of pretraining and target tasks are coupled

- ❏ 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 → best results

Agenda


[1] Introduction
[2] Pretraining
[3] What's in a representation?
[4] Adaptation
[5] Downstream
[6] Open Problems

2. Pretraining

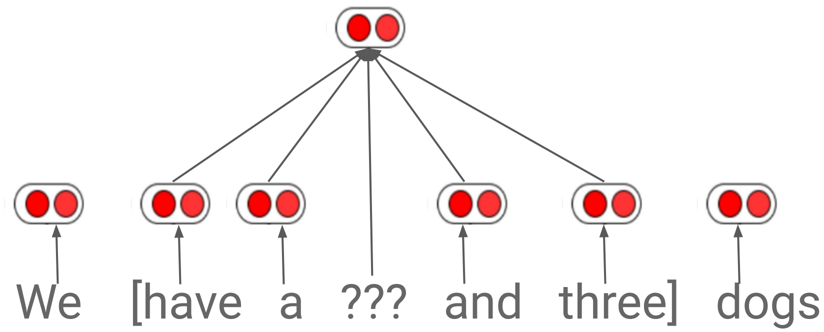
Image credit: Creative Stall

Overview

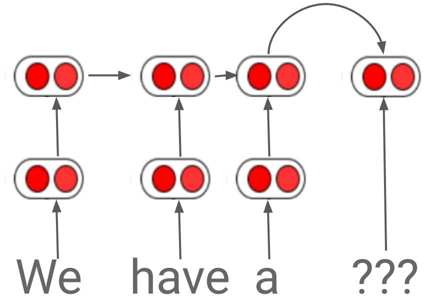
- ❑ Language model pretraining
- ❑ Word vectors
- ❑ Sentence and document vectors
- ❑ Contextual word vectors
- ❑ Interesting properties of pretraining
- ❑ Cross-lingual pretraining

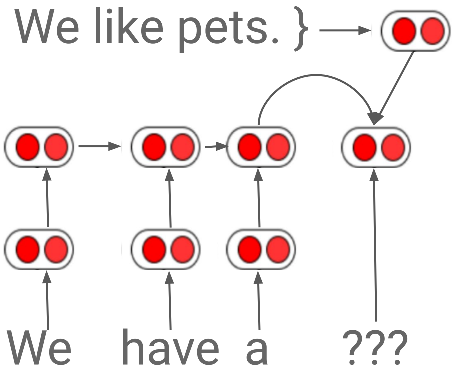
LM pretraining

word2vec, Mikolov et al (2013)



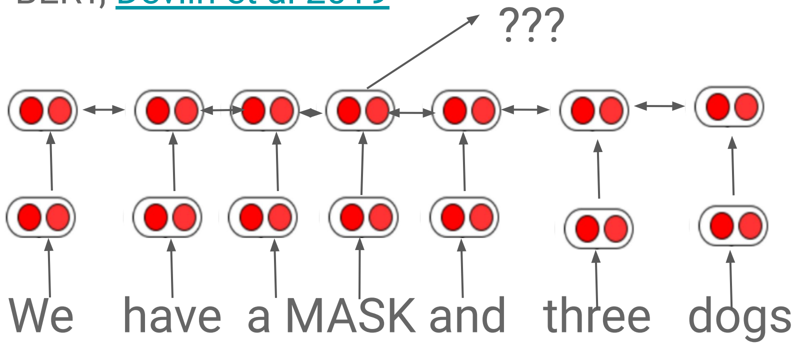
ELMo, Peters et al. 2018, ULMFiT (Howard & Ruder 2018), GPT (Radford et al. 2018)





Skip-Thought (Kiros et al., 2015)

BERT, Devlin et al 2019



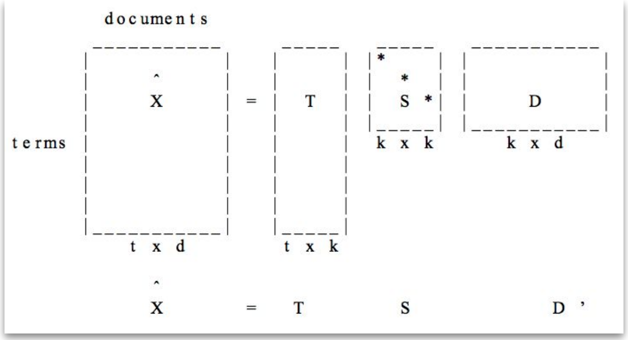
Word vectors

Why embed words?

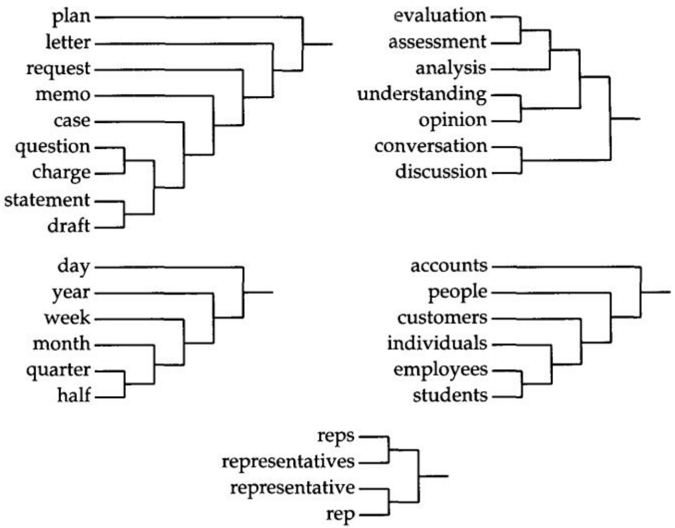
- Embeddings are themselves parameters—can be learned
- Sharing representations across tasks
- Lower dimensional space
  - Better for computation—difficult to handle sparse vectors.

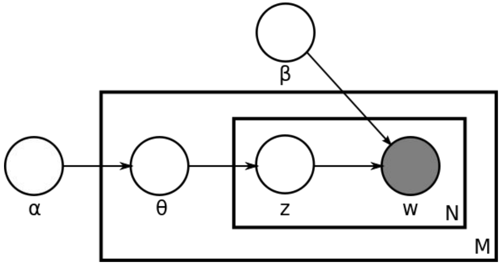
Unsupervised pretraining : Pre-Neural

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)

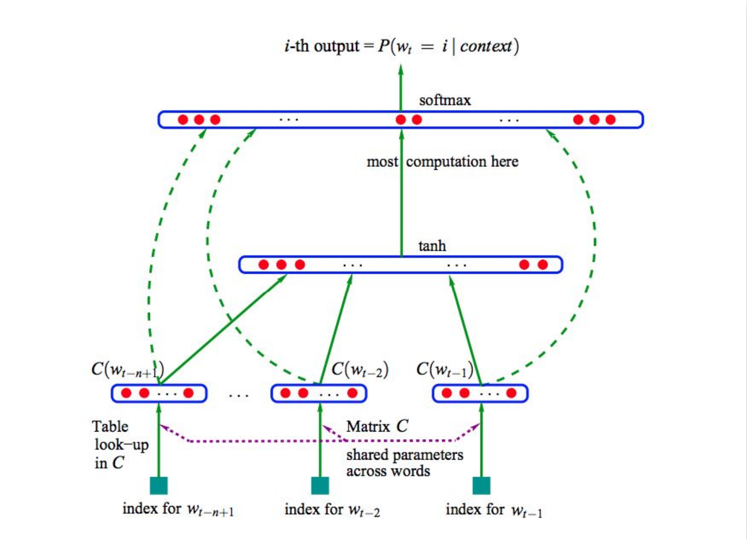




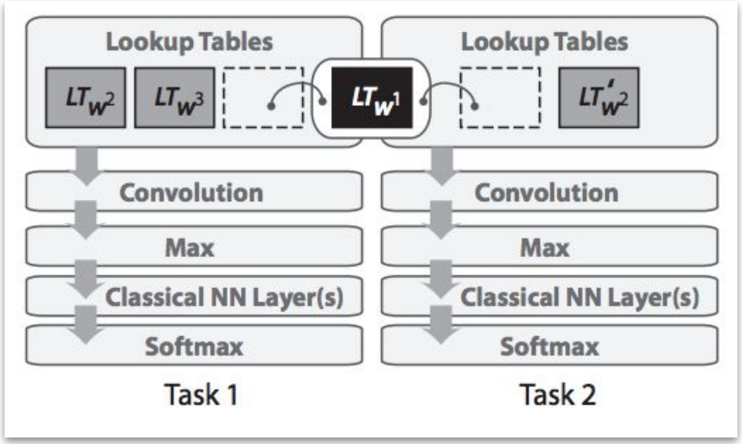
Latent Dirichlet Allocation (LDA)—Documents are mixtures of topics and topics are mixtures of words (Blei et al., 2003)

Word vector pretraining

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



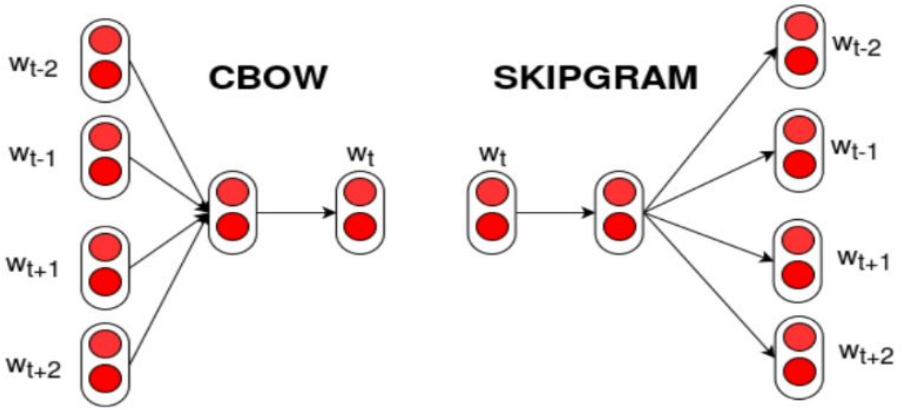
Supervised multitask word embeddings (Collobert and Weston, 2008)



word2vec

Efficient algorithm + large scale training → high quality word vectors

(Mikolov et al., 2013)



$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} logp(w_t|w_{t+j})$$

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} logp(w_{t+j}|w_t)$$

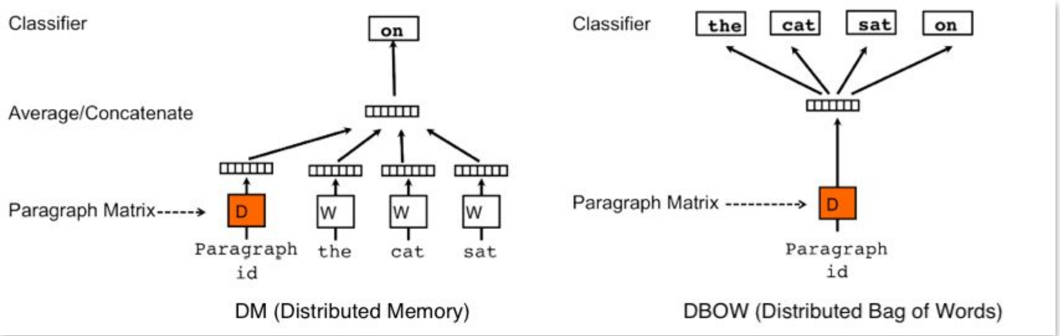
See also:

- ❑ Pennington et al. (2014): GloVe
- ❑ Bojanowski et al. (2017): fastText

Sentence and document vectors

Paragraph vector

Unsupervised paragraph embeddings (Le & Mikolov, 2014)

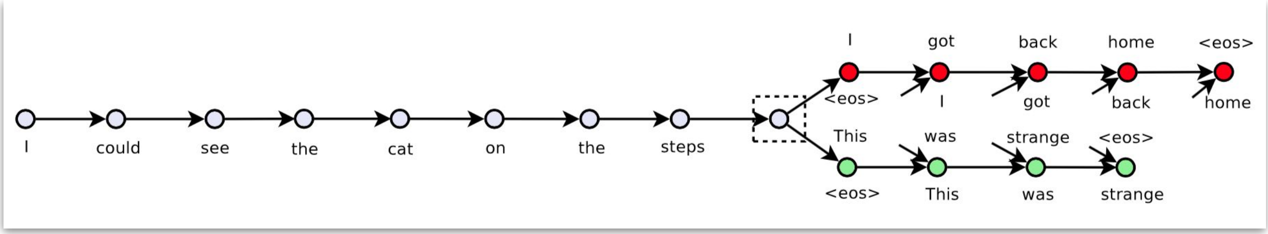


SOTA classification (IMDB, SST)

Model	Error rate
BoW (bnc) (Maas et al., 2011)	12.20 %
BoW (bΔt'c) (Maas et al., 2011)	11.77%
LDA (Maas et al., 2011)	32.58%
Full+BoW (Maas et al., 2011)	11.67%
Full+Unlabeled+BoW (Maas et al., 2011)	11.11%
WRRBM (Dahl et al., 2012)	12.58%
WRRBM + BoW (bnc) (Dahl et al., 2012)	10.77%
MNB-uni (Wang & Manning, 2012)	16.45%
MNB-bi (Wang & Manning, 2012)	13.41%
SVM-uni (Wang & Manning, 2012)	13.05%
SVM-bi (Wang & Manning, 2012)	10.84%
NBSVM-uni (Wang & Manning, 2012)	11.71%
NBSVM-bi (Wang & Manning, 2012)	8.78%
Paragraph Vector	**7.42%**

Skip-Thought Vectors

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

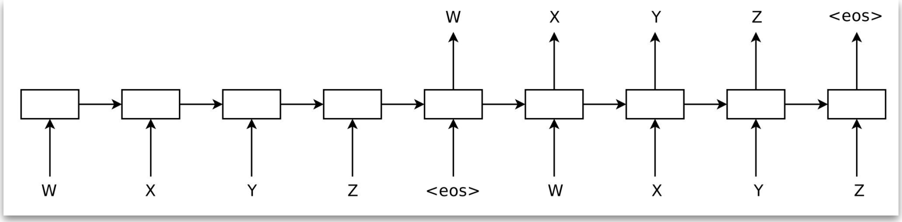


Method	MR	CR	SUBJ	MPQA	TREC
NB-SVM [41]	79.4	81.8	93.2	86.3	
MNB [41]	79.0	80.0	93.6	86.3	
cBoW [6]	77.2	79.9	91.3	86.4	87.3
GrConv [6]	76.3	81.3	89.5	84.5	88.4
RNN [6]	77.2	82.3	93.7	90.1	90.2
BRNN [6]	82.3	82.6	94.2	90.3	91.0
CNN [4]	81.5	85.0	93.4	89.6	**93.6**
AdaSent [6]	**83.1**	**86.3**	**95.5**	**93.3**	92.4
Paragraph-vector [7]	74.8	78.1	90.5	74.2	91.8
uni-skip	75.5	79.3	92.1	86.9	91.4
bi-skip	73.9	77.9	92.5	83.3	89.4
combine-skip	76.5	80.1	93.6	87.1	92.2
combine-skip + NB	80.4	81.3	93.6	87.5	

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

Autoencoder pretraining

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



SOTA classification (IMDB)

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

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

- Paragram-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	NOUN	ADJ	??
playing	game	multiplayer	plays
played	games		Play
	players		
	football		

Contextual word vectors - Key Idea

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

f(play | The kids play a game in the park.)

!=

f(play | The Broadway play premiered yesterday.)

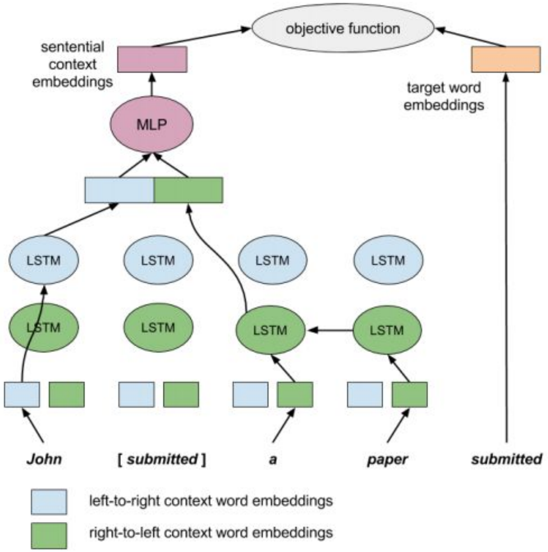
Many approaches based on language models

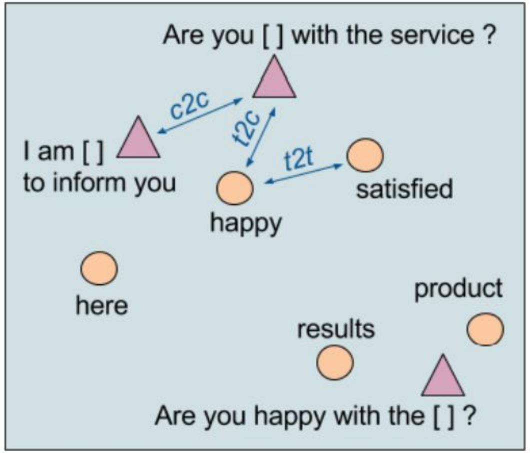
context2vec

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



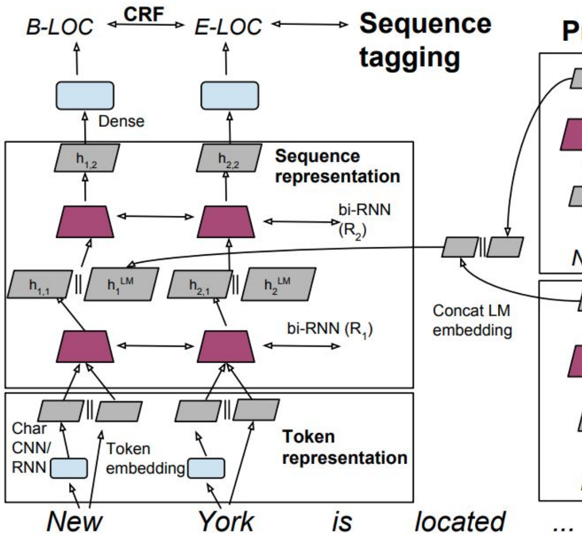


	c2v iters+	c2v	AWE	S-1	S-2
			MCSS		
test	**64.0**	62.7	48.4	-	-
all	**65.1**	61.3	49.7	58.9	56.2
			LST-07		
test	**56.1**	54.8	41.9	55.2	-
all	**56.0**	54.6	42.5	55.1	53.6
			LST-14		
test	47.7	47.3	38.1	**50.0**	-
all	47.9	47.5	38.9	**50.2**	48.3
			SE-3		
test	72.8	71.2	61.4	**74.1**	73.6

(Melamud et al., CoNLL 2016)

TagLM

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

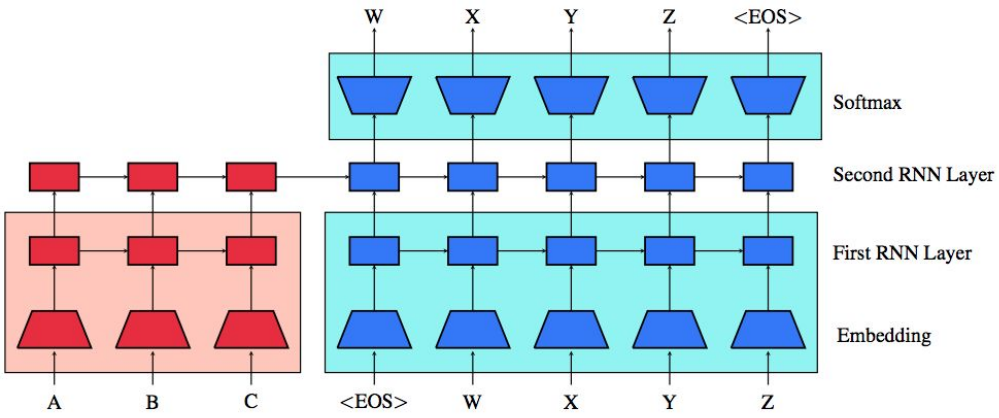


Model	$F_1 \pm$ std
Chiu and Nichols (2016)	90.91 ± 0.20
Lample et al. (2016)	90.94
Ma and Hovy (2016)	91.37
Our baseline without LM	90.87 ± 0.13
TagLM	$\mathbf{91.93 \pm 0.19}$

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



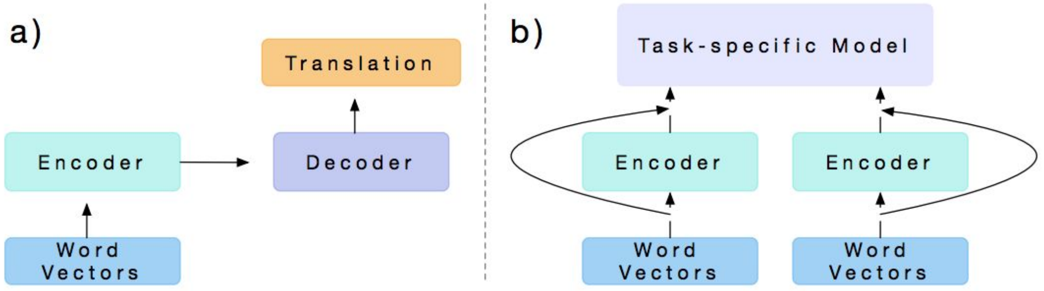
System	ensemble?	BLEU newstest2014	BLEU newstest2015
Phrase Based MT (Williams et al., 2016)	-	21.9	23.7
Supervised NMT (Jean et al., 2015)	single	-	22.4
Edit Distance Transducer NMT (Stahlberg et al., 2016)	single	21.7	24.1
Edit Distance Transducer NMT (Stahlberg et al., 2016)	ensemble 8	22.9	25.7
Backtranslation (Sennrich et al., 2015a)	single	22.7	25.7
Backtranslation (Sennrich et al., 2015a)	ensemble 4	23.8	26.5
Backtranslation (Sennrich et al., 2015a)	ensemble 12	**24.7**	27.6
No pretraining	single	21.3	24.3
Pretrained seq2seq	single	**24.0**	**27.0**
Pretrained seq2seq	ensemble 5	**24.7**	**28.1**

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

Large boost for MT.

[(Ramachandran et al, EMNLP 2017)]

CoVe



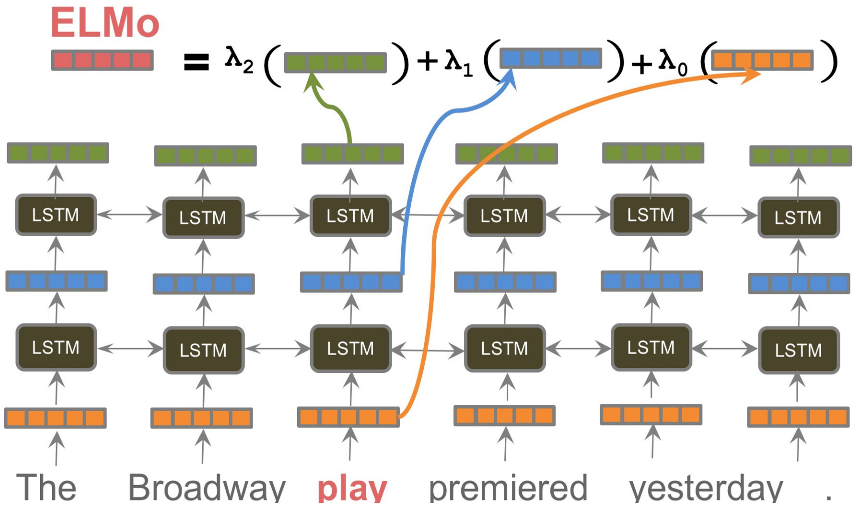
Dataset	Random	GloVe	GloVe+ Char	GloVe+ CoVe-S	GloVe+ CoVe-M	GloVe+ CoVe-L	GloVe+ Char+CoVe-L
SST-2	84.2	88.4	90.1	89.0	90.9	91.1	**91.2**
SST-5	48.6	53.5	52.2	54.0	54.7	54.5	**55.2**
IMDb	88.4	91.1	91.3	90.6	91.6	91.7	**92.1**
TREC-6	88.9	94.9	94.7	94.7	95.1	95.8	**95.8**
TREC-50	81.9	89.2	89.8	89.6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

Pretrain bidirectional encoder with MT supervision, extract LSTM states

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

(McCann et al, NeurIPS 2017)

ELMo



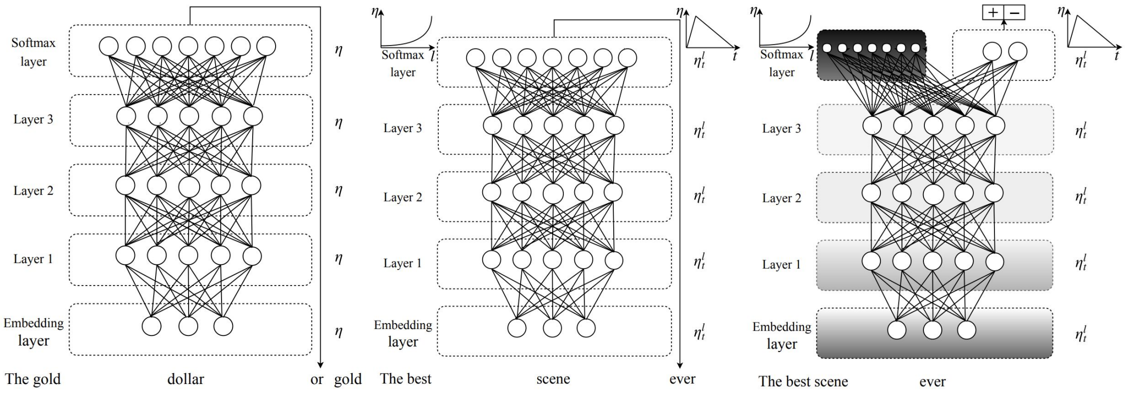
Pretrain deep bidirectional LM, extract contextual word vectors as learned linear combination of hidden states

SOTA for 6 diverse tasks

TASK	PREVIOUS SOTA		OUR BASELINE	ELMO + BASELINE	INCREASE (ABSOLUTE/ RELATIVE)
SQuAD	Liu et al. (2017)	84.4	81.1	85.8	4.7 / 24.9%
SNLI	Chen et al. (2017)	88.6	88.0	88.7 ± 0.17	0.7 / 5.8%
SRL	He et al. (2017)	81.7	81.4	84.6	3.2 / 17.2%
Coref	Lee et al. (2017)	67.2	67.2	70.4	3.2 / 9.8%
NER	Peters et al. (2017)	91.93 ± 0.19	90.15	92.22 ± 0.10	2.06 / 21%
SST-5	McCann et al. (2017)	53.7	51.4	54.7 ± 0.5	3.3 / 6.8%

(Peters et al, NAACL 2018)

ULMFiT



(a) LM pre-training (b) LM fine-tuning (c) Classifier fine-tuning

	Model	Test		Model	Test
IMDb	CoVe (McCann et al., 2017)	8.2	TREC-6	CoVe (McCann et al., 2017)	4.2
	oh-LSTM (Johnson and Zhang, 2016)	5.9		TBCNN (Mou et al., 2015)	4.0
	Virtual (Miyato et al., 2016)	5.9		LSTM-CNN (Zhou et al., 2016)	3.9
	ULMFiT (ours)	**4.6**		ULMFiT (ours)	**3.6**

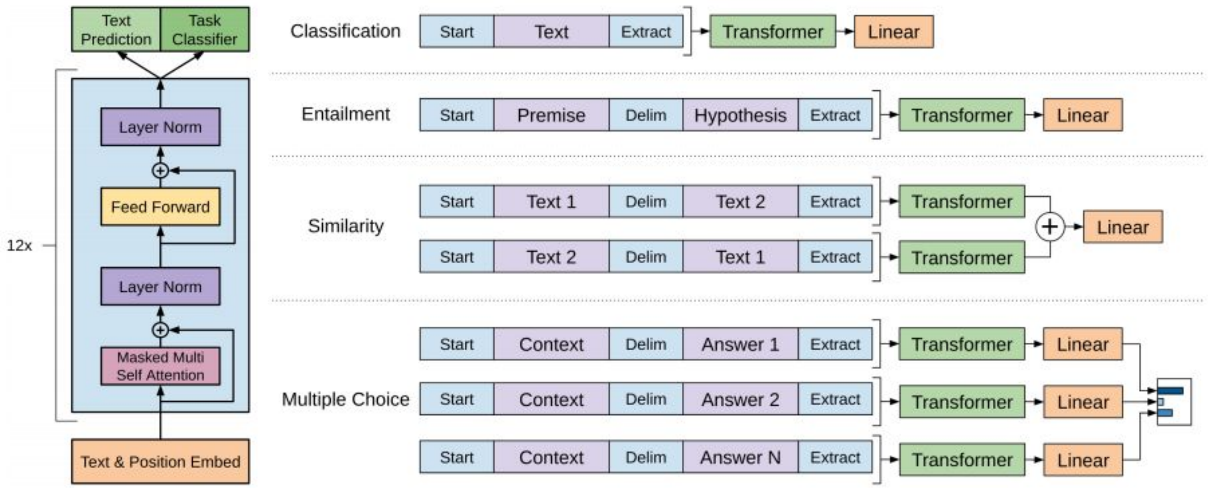
	AG	DBpedia	Yelp-bi	Yelp-full
Char-level CNN (Zhang et al., 2015)	9.51	1.55	4.88	37.95
CNN (Johnson and Zhang, 2016)	6.57	0.84	2.90	32.39
DPCNN (Johnson and Zhang, 2017)	6.87	0.88	2.64	30.58
ULMFiT (ours)	**5.01**	**0.80**	**2.16**	**29.98**

Pretrain AWD-LSTM LM,
fine-tune LM in two stages with different adaptation techniques

SOTA for six classification datasets

(Howard and Ruder, ACL 2018)

GPT



Method	MNLI-m	MNLI-mm	SNLI	SciTail	QNLI	RTE
ESIM + ELMo [44] (5x)	-	-	89.3	-	-	-
CAFE [58] (5x)	80.2	79.0	89.3	-	-	-
Stochastic Answer Network [35] (3x)	80.6	80.1	-	-	-	-
CAFE [58]	78.7	77.9	88.5	83.3		
GenSen [64]	71.4	71.3	-	-	82.3	59.2
Multi-task BiLSTM + Attn [64]	72.2	72.1	-	-	82.1	**61.7**
Finetuned Transformer LM (ours)	**82.1**	**81.4**	**89.9**	**88.3**	**88.1**	56.0

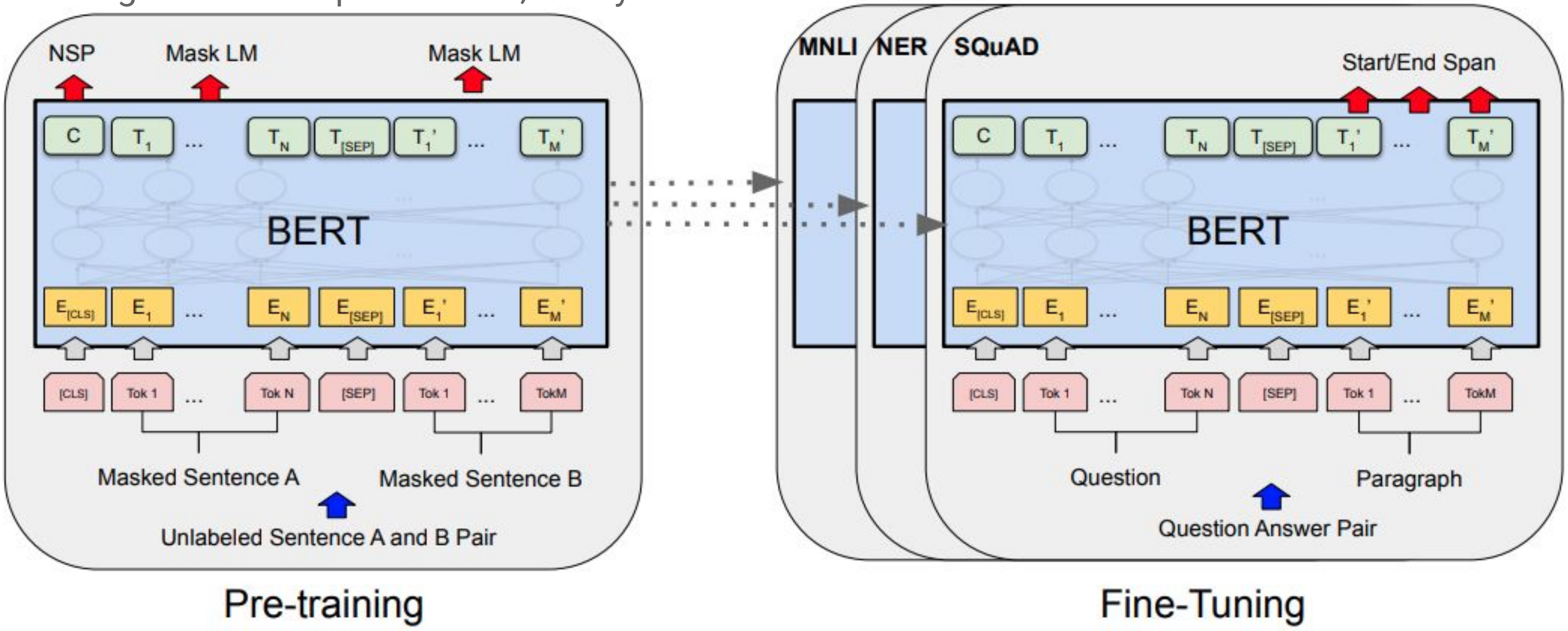
Pretrain large 12-layer left-to-right Transformer, fine tune for sentence, sentence pair and multiple choice questions.

SOTA results for 9 tasks.

(Radford et al., 2018)

BERT

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).

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

(Devlin et al. 2019)

BERT

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

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

(Devlin et al. 2019)

Other pretraining objectives

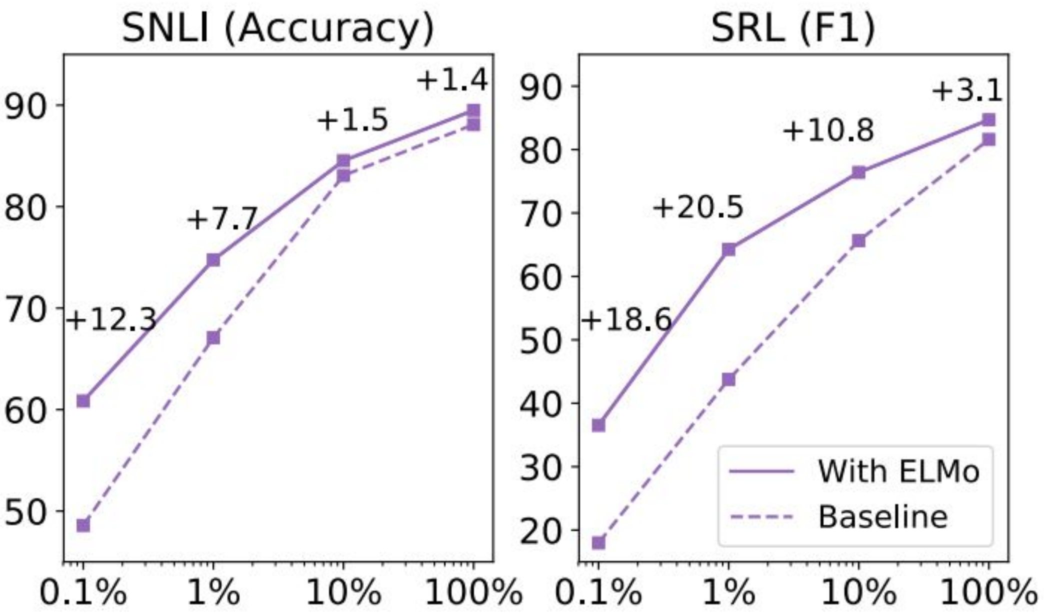
- Contextual string representations (Akbik et al., COLING 2018)—SOTA NER results
- Cross-view training (Clark et al. EMNLP 2018)—improve supervised tasks with unlabeled data
- Cloze-driven pretraining (Baevski et al. (2019)—SOTA NER and constituency parsing

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)

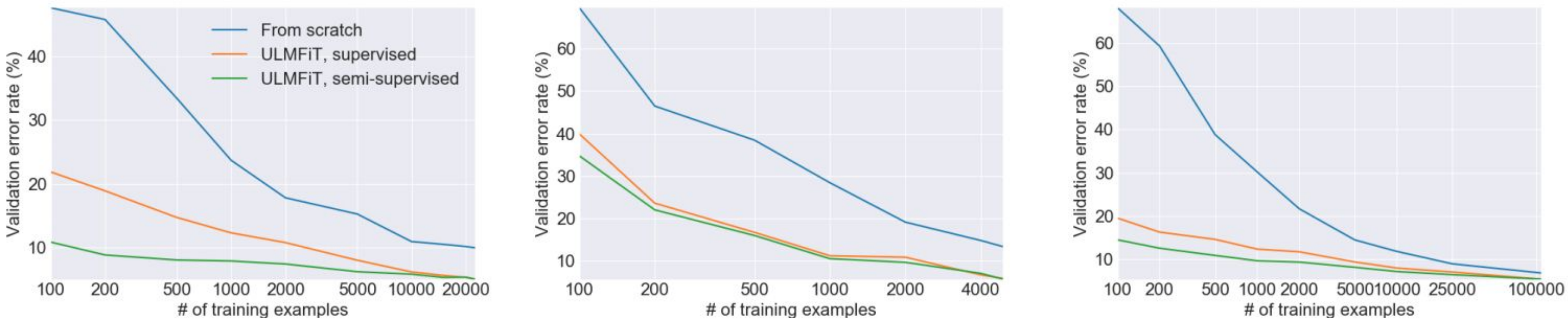
Sample efficiency

Pretraining reduces need for annotated data



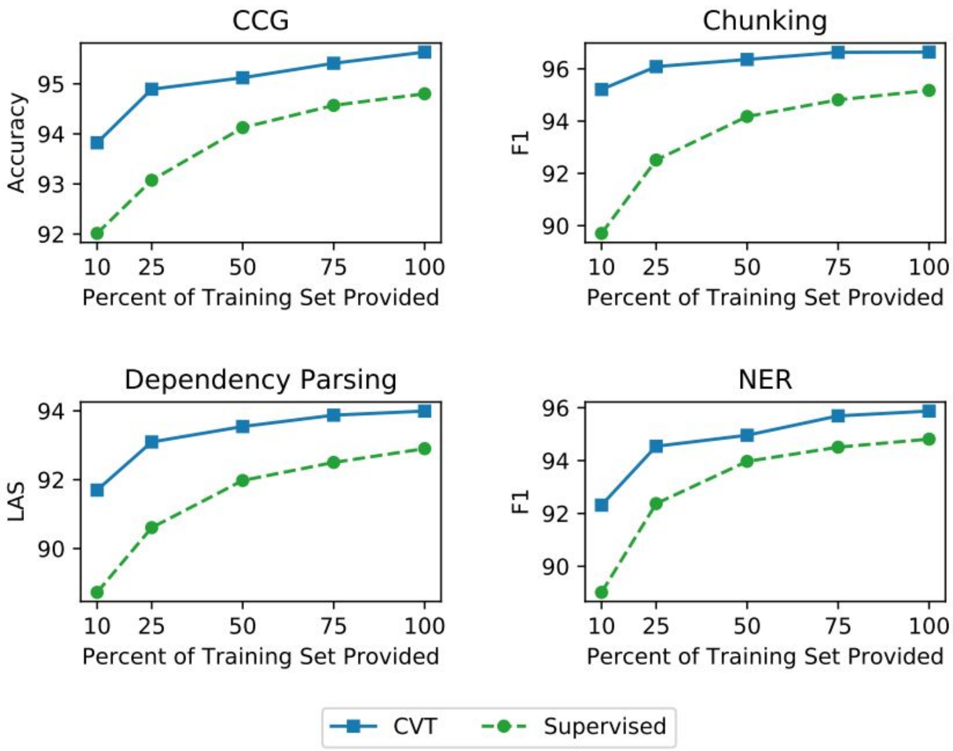
[(Peters et al, NAACL 2018)](#)

Pretraining reduces need for annotated data



[(Howard and Ruder, ACL 2018)](https://arxiv.org/abs/1801.06146)

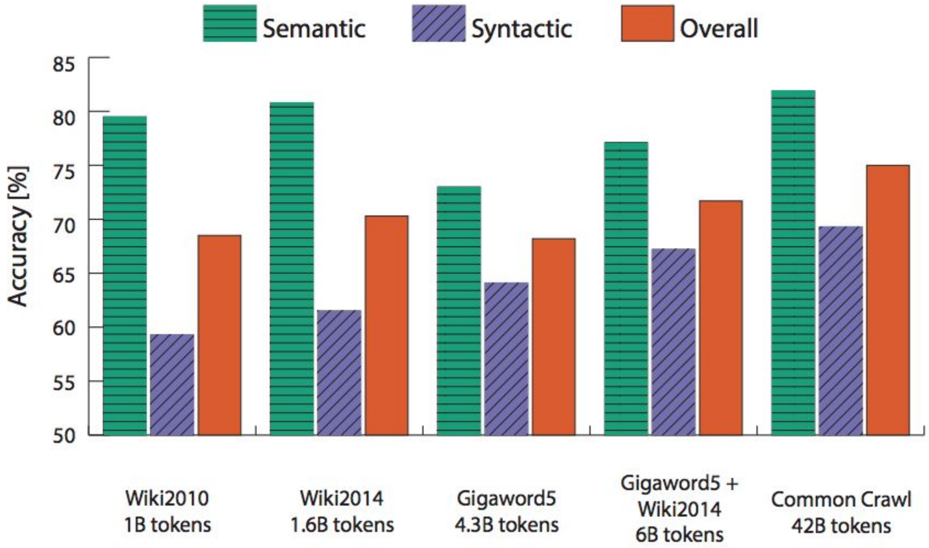
Pretraining reduces need for annotated data



(Clark et al. EMNLP 2018)

Scaling up pretraining

Scaling up pretraining



More data → better word vectors

([Pennington et al 2014](#))

Scaling up pretraining

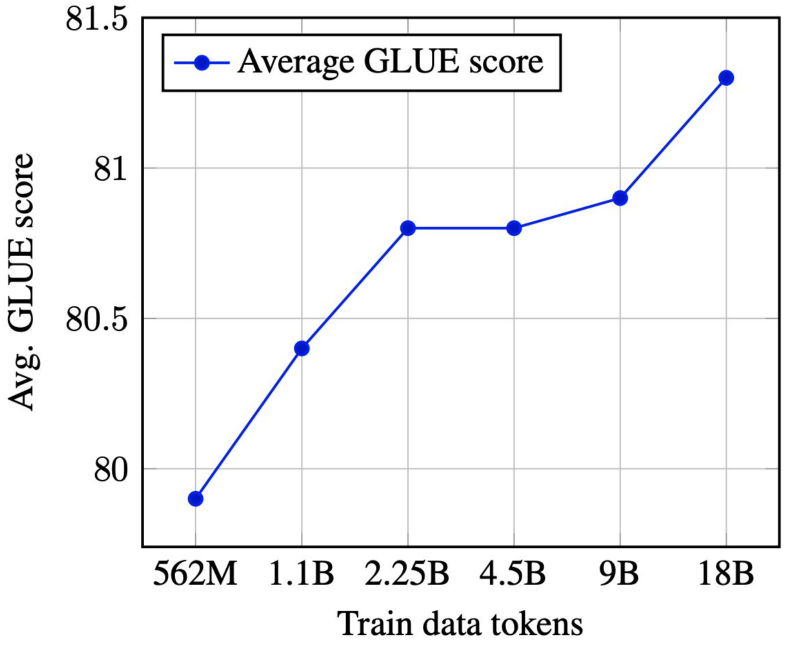


Figure 3: Average GLUE score with different amounts of Common Crawl data for pretraining.

Baevski et al. (2019)

Scaling up pretraining

Hyperparams				Dev Set Accuracy		
#L	#H	#A	LM (ppl)	MNLI-m	MRPC	SST-2
3	768	12	5.84	77.9	79.8	88.4
6	768	3	5.24	80.6	82.2	90.7
6	768	12	4.68	81.9	84.8	91.3
12	768	12	3.99	84.4	86.7	92.9
12	1024	16	3.54	85.7	86.9	93.3
24	1024	16	3.23	86.6	87.8	93.7

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.

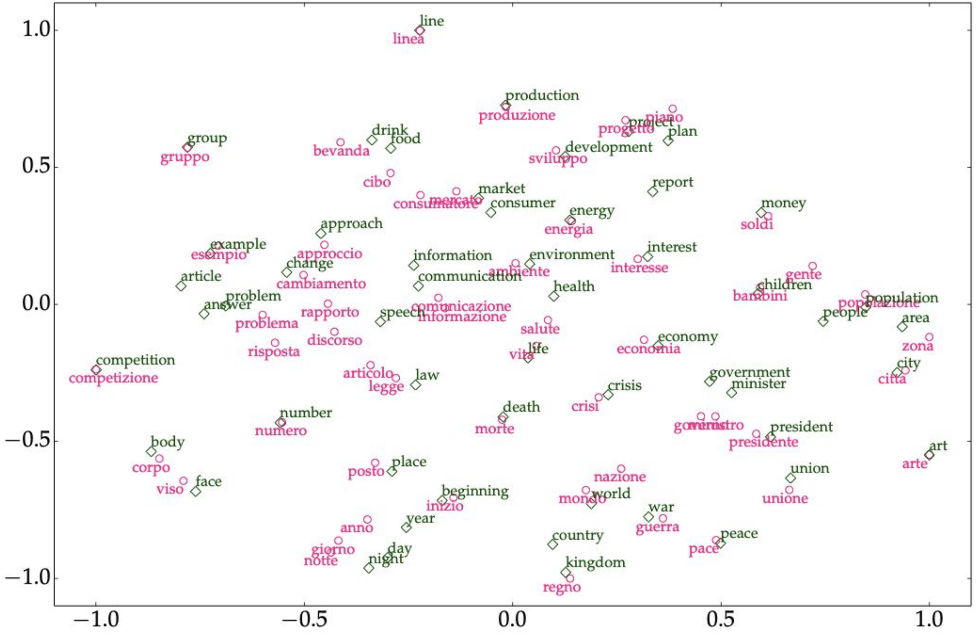
Bigger model → better results

(Devlin et al 2019)

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)
- ❏ Multilingual BERT: BERT trained jointly on 100 languages
- ❏ Rosita: Polyglot contextual representations (Mulcaire et al., NAACL 2019)
- ❏ XLM: Cross lingual LM (Lample & Conneau, 2019)

Hands-on #1:
Pretraining a Transformer Language Model

Image credit: Chanaky

Hands-on: Overview

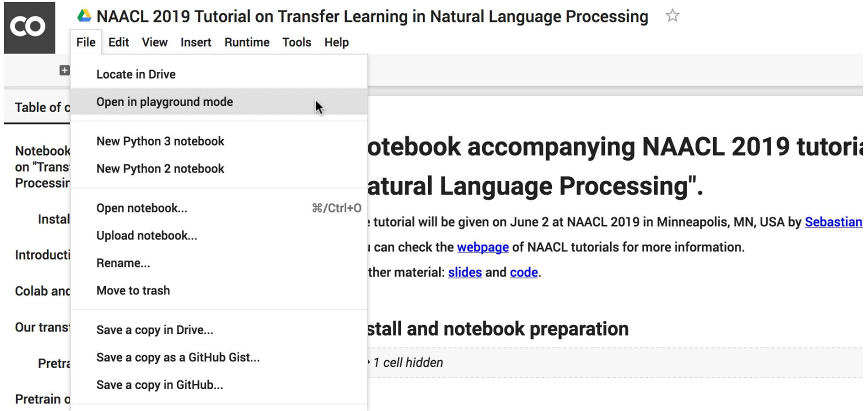
Current developments in Transfer Learning combine new approaches for training schemes (sequential training) as well as models (transformers) ➪ can look intimidating and complex

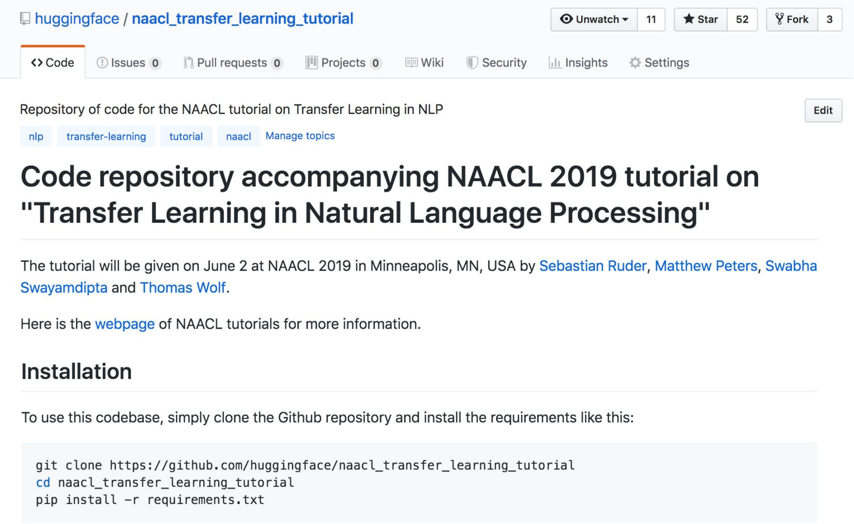
- Goals:
  - Let's make these recent works "uncool again" i.e. as accessible as possible
  - Expose all the details in a simple, concise and self-contained code-base
  - Show that transfer learning can be simple (less hand-engineering) & fast (pretrained model)
- Plan
  - Build a GPT-2 / BERT model
  - 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 ➪ code of the following slides
  - Code: http://tiny.cc/NAACLTransferCode ➪ same code organized in a repo

Hands-on pre-training

Colab: http://tiny.cc/NAACLTransferColab

Repo: http://tiny.cc/NAACLTransferCode

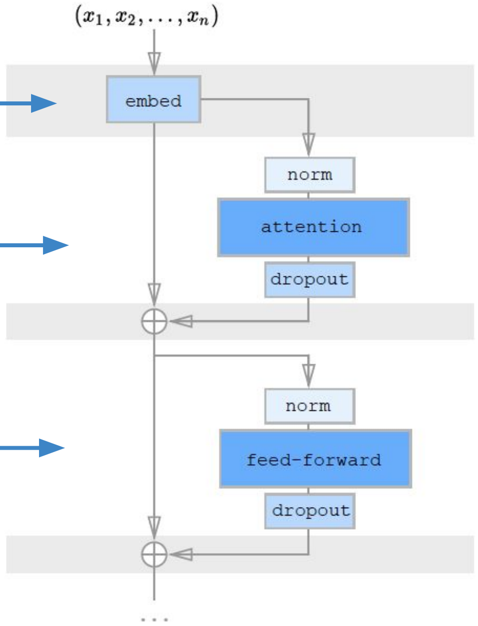




Hands-on pre-training

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 ➪ 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)

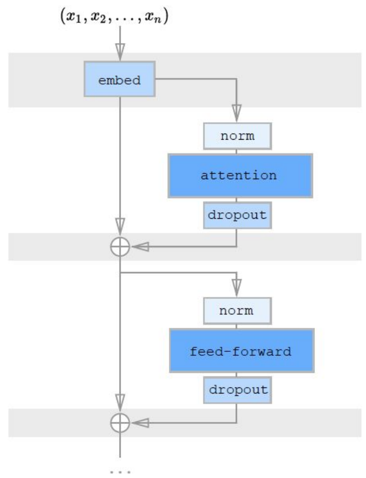
We'll play with both

(Child et al, 2019)

Hands-on pre-training

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.




```
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.tokens_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 _ 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.tokens_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)
            x = self.dropout(x)
            h = x + h

            h = layer_norm_2(h)
            x = feed_forward(h)
            x = self.dropout(x)
            h = x + h

        return h
```

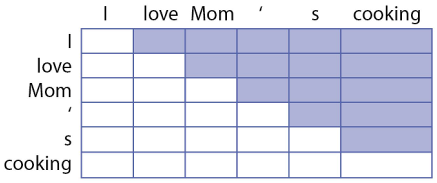
Hands-on pre-training

Two attention masks?

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

I	love	Mom	'	s	cooking
I	love	you	too	!	
No	way				
This	is	the	shit		
Yes					

- `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.tokens_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 _ 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.tokens_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)
            x = self.dropout(x)
            h = x + h

            h = layer_norm_2(h)
            x = feed_forward(h)
            x = self.dropout(x)
            h = x + h

        return h
```

Hands-on pre-training

To pretrain our model, we need to add a few elements: a head, a loss and initialize weights.

We add these elements with a pretraining model encapsulating our model.

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

```python
class TransformerWithLMHead(nn.Module):
    def __init__(self, config):
        """ Transformer with a language modeling head on top (tied weights) """
        super().__init__()
        self.config = config
        self.transformer = Transformer(config.embed_dim, config.hidden_dim, config.num_embeddings,
                                       config.num_max_positions, config.num_heads, config.num_layers,
                                       config.dropout, causal=not config.mlm)

        self.lm_head = nn.Linear(config.embed_dim, config.num_embeddings, bias=False)
        self.apply(self.init_weights)
        self.tie_weights()

    def tie_weights(self):
        self.lm_head.weight = self.transformer.tokens_embeddings.weight

    def init_weights(self, module):
        """ initialize weights - nn.MultiheadAttention is already initalized by PyTorch (xavier) """
        if isinstance(module, (nn.Linear, nn.Embedding, nn.LayerNorm)):
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if isinstance(module, (nn.Linear, nn.LayerNorm)) and module.bias is not None:
            module.bias.data.zero_()

    def forward(self, x, labels=None, padding_mask=None):
        """ x has shape [seq length, batch], padding_mask has shape [batch, seq length] """
        hidden_states = self.transformer(x, padding_mask)
        logits = self.lm_head(hidden_states)

        if labels is not None:
            shift_logits = logits[:-1] if self.transformer.causal else logits
            shift_labels = labels[1:] if self.transformer.causal else labels
            loss_fct = nn.CrossEntropyLoss(ignore_index=-1)
            loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1))
            return logits, loss

        return logits
```

Hands-on pre-training

Now let's take care of our data and configuration

We'll use a pre-defined open vocabulary tokenizer: BERT's model cased tokenizer.

```
from pytorch_pretrained_bert import BertTokenizer, cached_path

tokenizer = BertTokenizer.from_pretrained('bert-base-cased', do_lower_case=False)
```

Hyper-parameters taken from [Dai et al., 2018](#) (Transformer-XL) ➪ ~50M parameters causal model.

```
from collections import namedtuple

Config = namedtuple('Config',
  field_names="embed_dim, hidden_dim, num_max_positions, num_embeddings      , num_heads, num_layers,"
              "dropout, initializer_range, batch_size, lr, max_norm, n_epochs, n_warmup,"
              "mlm, gradient_accumulation_steps, device, log_dir, dataset_cache")
args = Config( 410      , 2100      , 256               , len(tokenizer.vocab), 10       , 16        ,
               0.1     , 0.02            , 16          , 2.5e-4, 1.0 , 50      , 1000     ,
               False, 4, "cuda" if torch.cuda.is_available() else "cpu", "./"    , "./dataset_cache.bin")
```

Use a large dataset for pre-trainining: WikiText-103 with 103M tokens ([Merity et al., 2017](#)).

```
dataset_file = cached_path("https://s3.amazonaws.com/datasets.huggingface.co/wikitext-103/"
                           "wikitext-103-train-tokenized-bert.bin")
datasets = torch.load(dataset_file)

# Convert our encoded dataset to torch.tensors and reshape in blocks of the transformer's input length
for split_name in ['train', 'valid']:
    tensor = torch.tensor(datasets[split_name], dtype=torch.long)
    num_sequences = (tensor.size(0) // args.num_max_positions) * args.num_max_positions
    datasets[split_name] = tensor.narrow(0, 0, num_sequences).view(-1, args.num_max_positions)
```

Instantiate our model and optimizer (Adam)

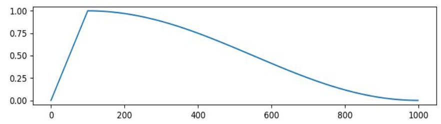
```
model = TransformerWithLMHead(args).to(args.device)
optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
```

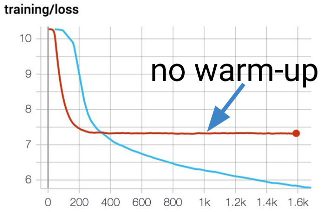
Hands-on pre-training

And we're done: let's train!

A simple update loop.
We use gradient accumulation to have a large batc
GPU (>64)

Learning r
- linear wa
- then cos
square root decrease






```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

dataloader = DataLoader(datasets['train'], batch_size=args.batch_size, shuffle=True)

# Define training function
```

```python
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)
```

```python
[seq length, batch]
```

```python
RunningAverage(output_transform=lambda x: x).attach(trainer, "loss")
ProgressBar(persist=True).attach(trainer, metric_names=['loss'])

# Learning rate schedule: linearly warm-up to lr and then decrease the learning rate to zero with cosine
cos_scheduler = CosineAnnealingScheduler(optimizer, 'lr', args.lr, 0.0, len(dataloader) * args.n_epochs)
scheduler = create_lr_scheduler_with_warmup(cos_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'))
```

Go!

```python
trainer.run(train_dataloader, max_epochs=args.n_epochs)
```

...

Epoch [1/50] [365/28874] 1%| , loss=2.30e+00 [03:43<4:52:22]

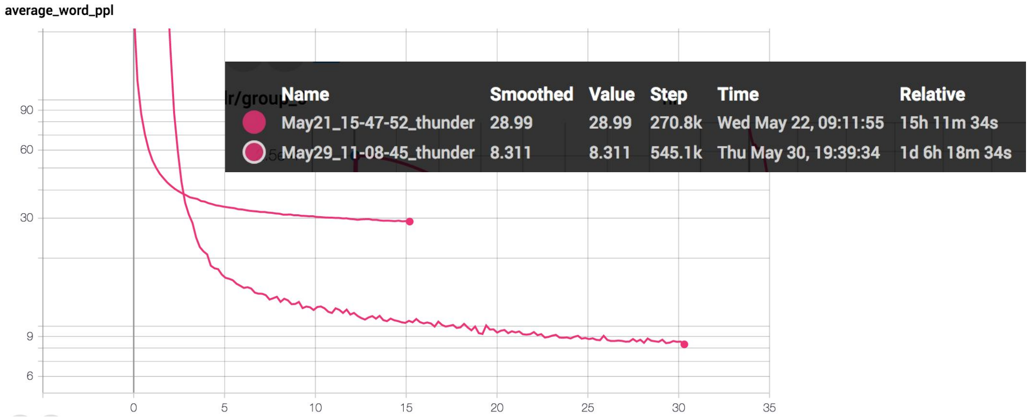
Hands-on pre-training — Concluding remarks

- 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

Hands-on pre-training — Concluding remarks

- **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



Model	#Params	Validation PPL	Test PPL
Grave et al. (2016b) – LSTM	-	-	48.7
Bai et al. (2018) – TCN	-	-	45.2
Dauphin et al. (2016) – GCNN-8	-	-	44.9
Grave et al. (2016b) – LSTM + Neural cache	-	-	40.8
Dauphin et al. (2016) – GCNN-14	-	-	37.2
Merity et al. (2018) – 4-layer QRNN	151M	32.0	33.0
Rae et al. (2018) – LSTM + Hebbian + Cache	-	29.7	29.9
Ours – Transformer-XL Standard	151M	**23.1**	**24.0**
Baevski & Auli (2018) – adaptive input◇	247M	19.8	20.5
Ours – Transformer-XL Large	257M	**17.7**	**18.3**

Wikitext-103 Validation/Test PPL

Dai et al., 2018

Agenda

[1] Introduction

[2] Pretraining

[3] What's in a representation?

[4] Adaptation

[5] Downstream

[6] Open Problems

3. What is in a Representation?

Image credit: Caique Lima

Why care about what is in a representation?

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

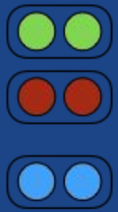
- ❑ Language-aware representations
  - ❑ To generalize to other tasks, new inputs
  - ❑ As intermediates for possible improvements to pretraining

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

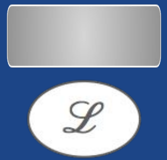
Swayamdipta, 2019

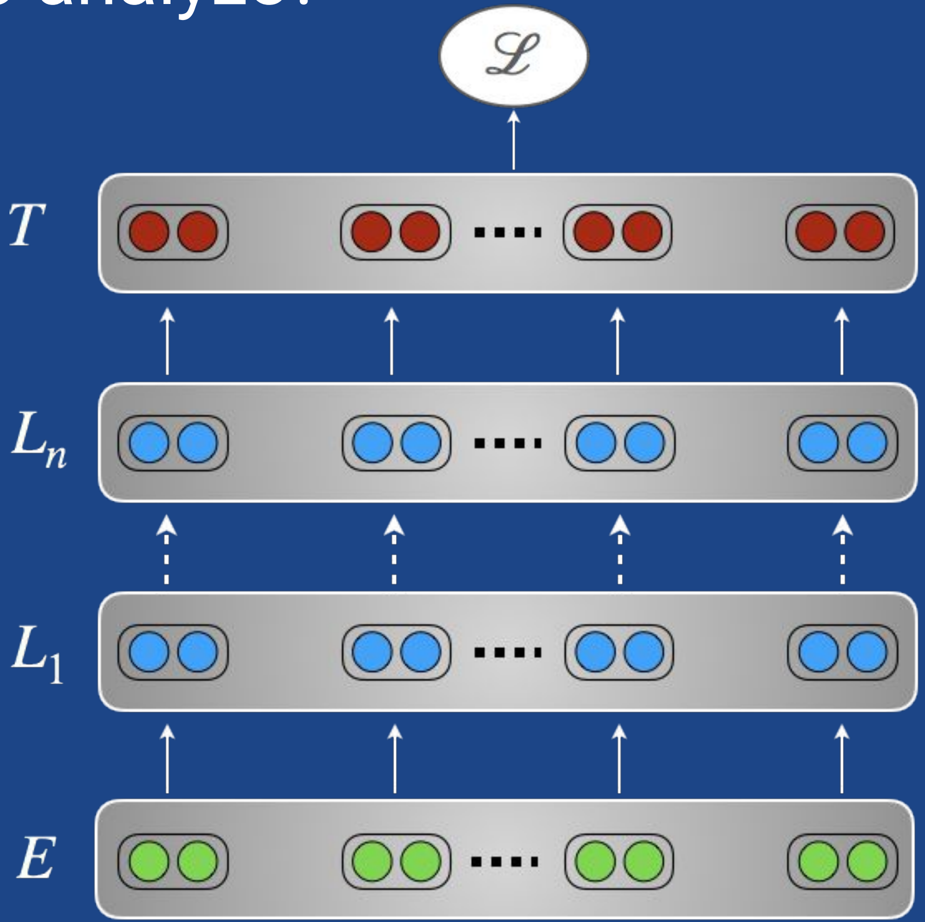
What to analyze?

- Embeddings
  - Word
  - Contextualized
- Network Activations



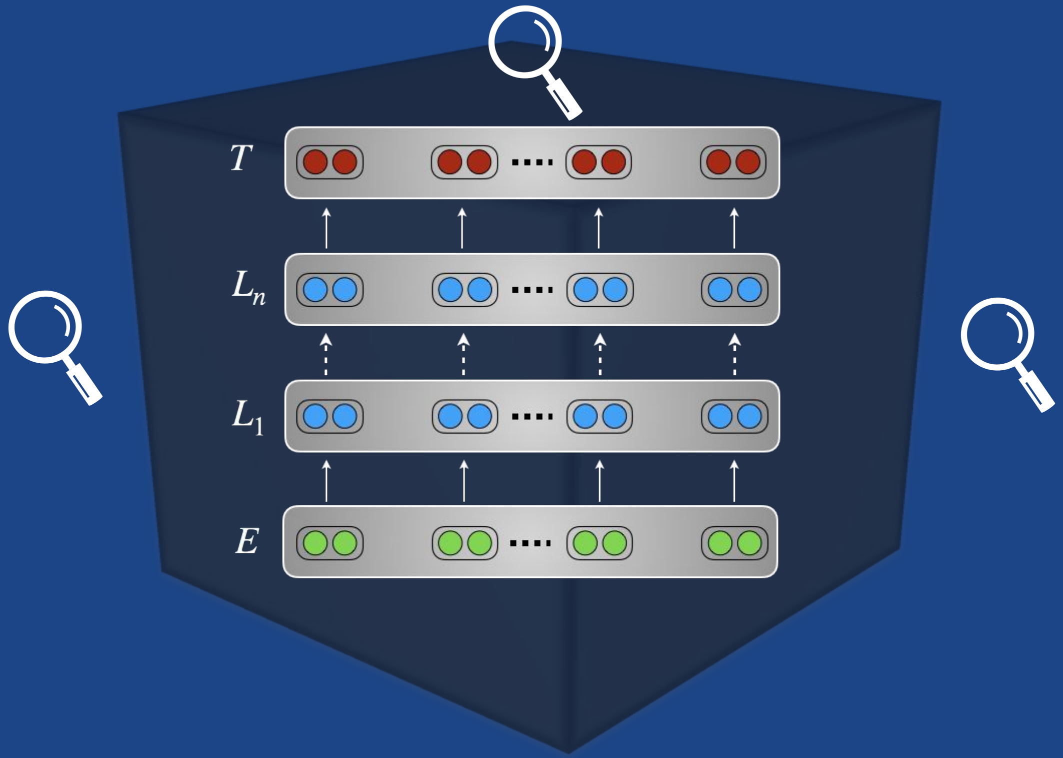
- Variations
  - Architecture (RNN / Transformer)
  - Layers
  - Pretraining Objectives





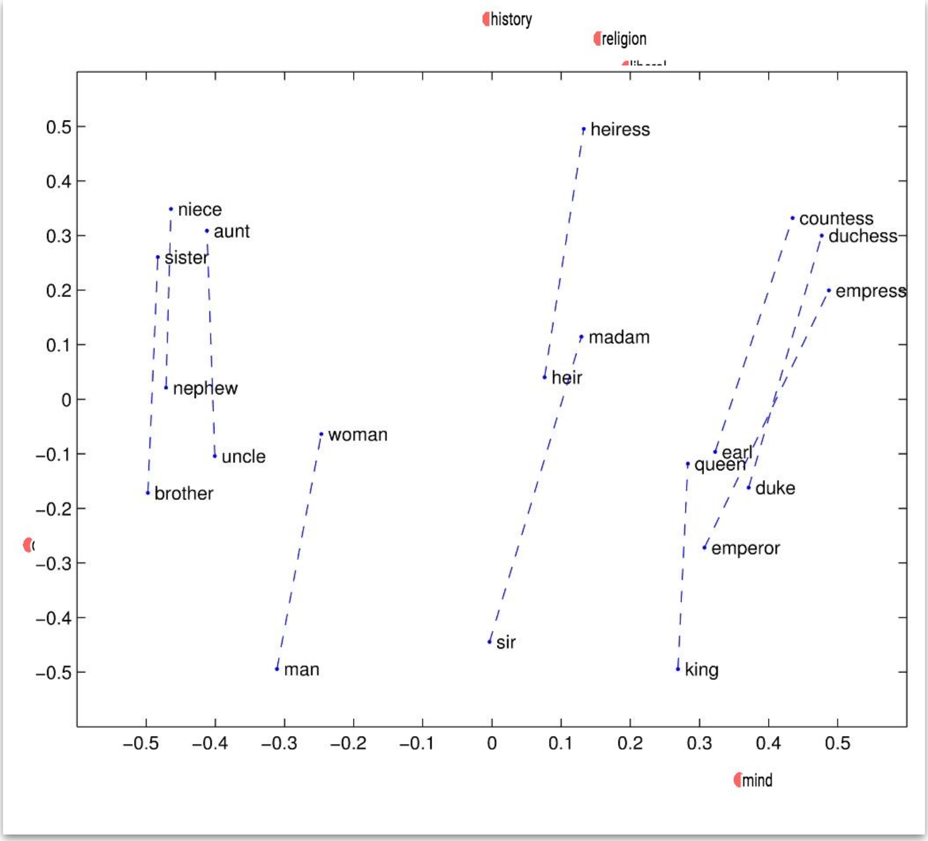
Analysis Method 1: Visualization

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
  - $w_{king} - w_{man} + w_{woman} \sim w_{queen}$
- High-level view of lexical semantics
  - Only a limited number of examples
  - Connection to other tasks is unclear Goldberg, 2017



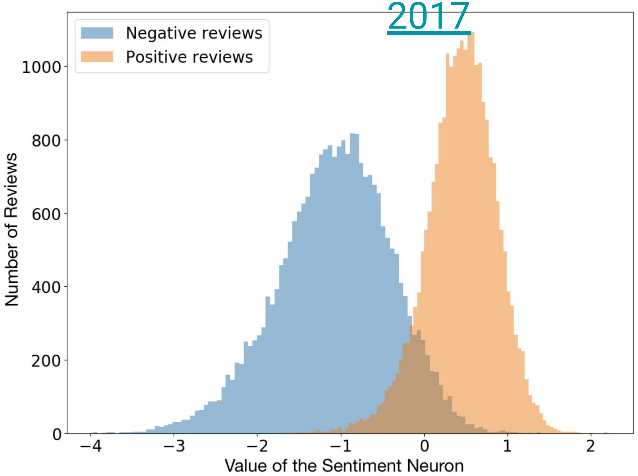
Pennington et al., 2014

Image: Tensorflow

Visualizing Neuron Activations

- 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](#))

[Radford et al., 2017](#)



Cell that is sensitive to the depth of an expression:

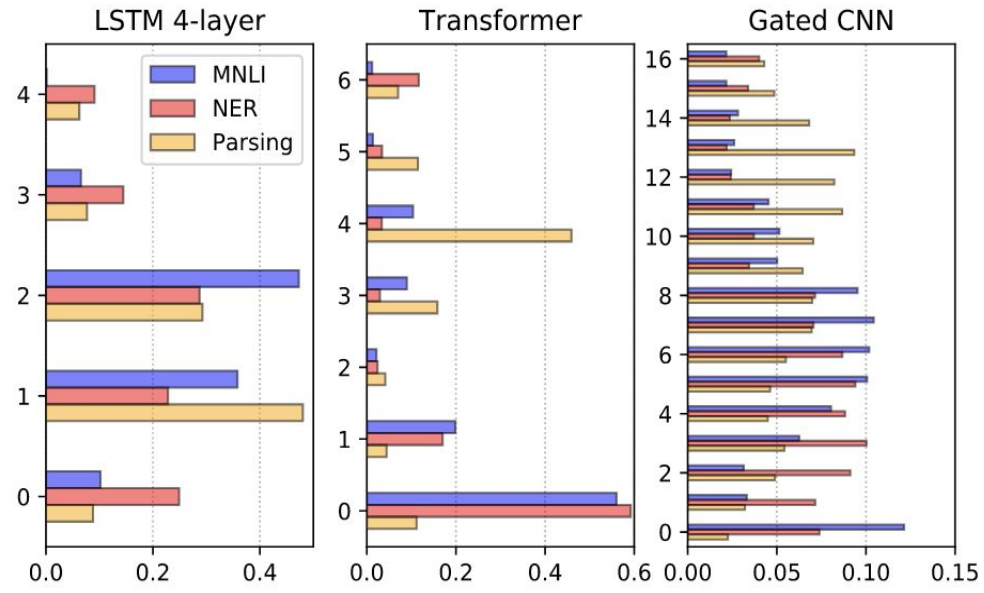
```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
 int i;
 if (classes[class]) {
  for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
   if (mask[i] & classes[class][i])
    return 0;
 }
 return 1;
}
```

[Karpathy et al., 2016](#)

Visualizing Layer-Importance Weights

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

- Task and architecture specific!

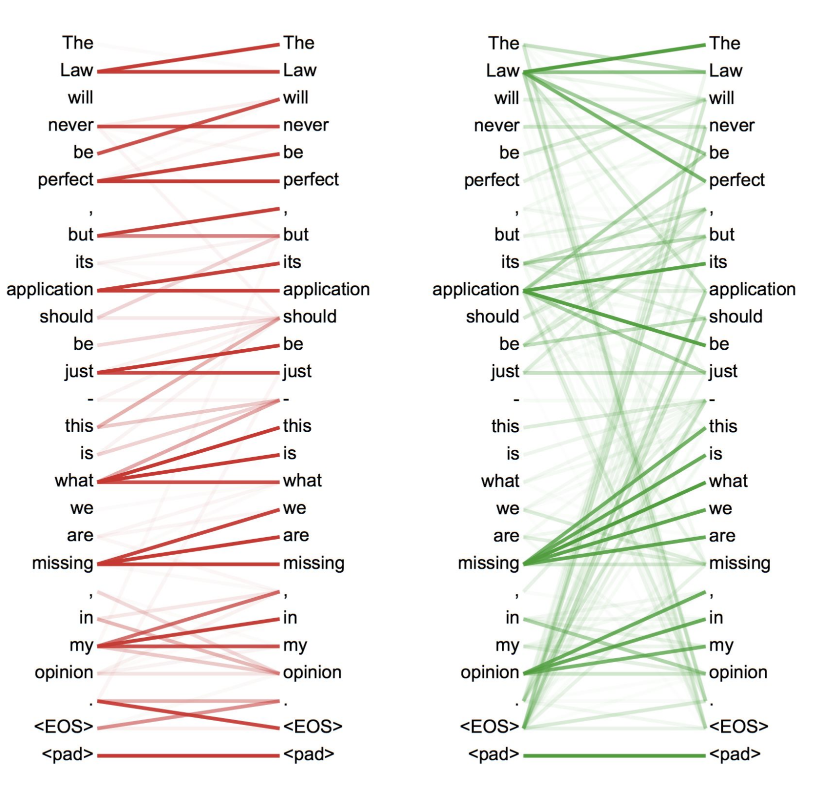


Also see Tenney et al., ACL 2019

Peters et al.. EMNLP 2018

Visualizing Attention Weights

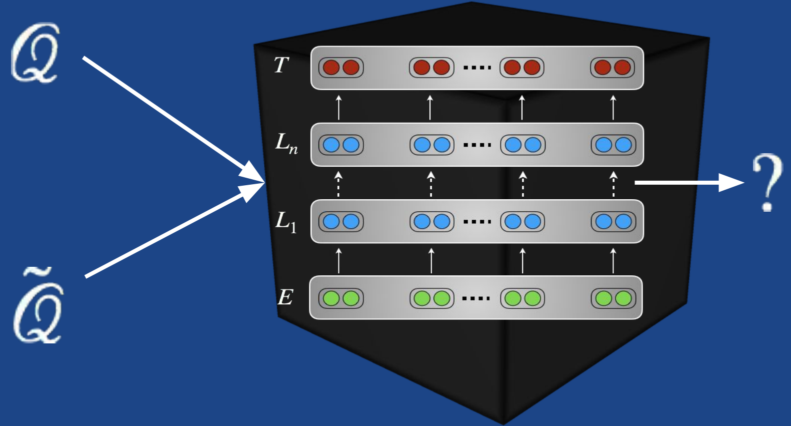
- Popular in machine translation, or other seq2seq architectures:
  - **Alignment** between words of source and target.
  - Long-distance word-word **dependencies** (intra-sentence attention)
- Sheds light on architectures
  - Having sophisticated attention mechanisms can be a good thing!
  - Layer-specific
- 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



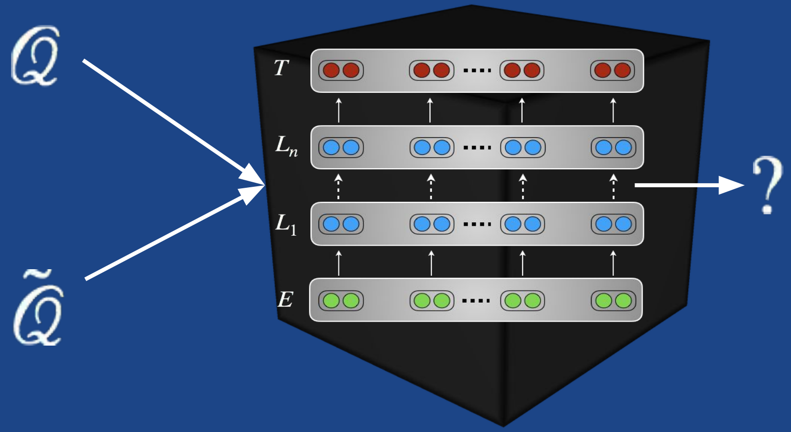
- RNNs outperform other non-neural baselines.

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

Kuncoro et al. 2018

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

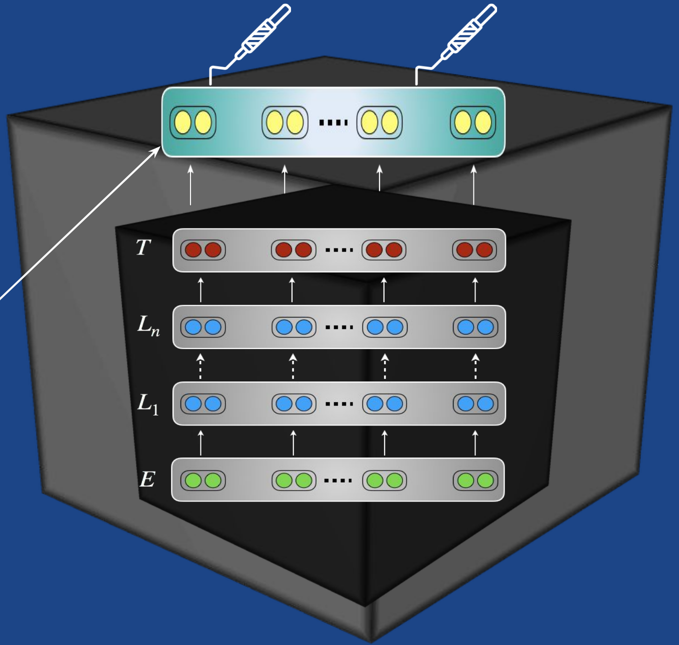
Kuncoro et al. 2018

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

Probing Morphology, Syntax, Semantics

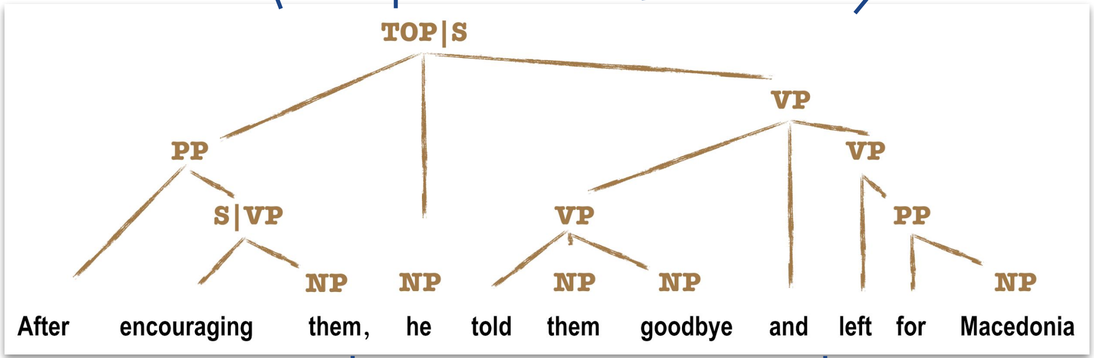
- ❑ Morphology
- ❑ Word-level syntax
  - ❑ POS tags, CCG supertags
  - ❑ Constituent parent, grandparent…
- ❑ Partial syntax
  - ❑ Dependency relations
- ❑ Partial semantics
  - ❑ Entity Relations
  - ❑ Coreference
  - ❑ Roles

Long-distance number agreement

Tree Depth

Subject-Verb Agreement

Top Constituents



Objects

Tense of main clause verb

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

	CoVe			ELMo			GPT		
	Lex.	Full	Abs. Δ	Lex.	Full	Abs. Δ	Lex.	cat	mix
Part-of-Speech	85.7	94.0	8.4	90.4	**96.7**	6.3			
Constituents	56.1	81.6	25.4	69.1	**84.6**	15.4			
Dependencies	75.0	83.6	8.6	80.4	**93.9**	13.6			
Entities	88.4	90.3	1.9	92.0	**95.6**	3.5			
SRL (all)	59.7	80.4	20.7	74.1	**90.1**	16.0			
Core roles	*56.2*	*81.0*	*24.7*	*73.6*	***92.6***	*19.0*			
Non-core roles	*67.7*	*78.8*	*11.1*	*75.4*	***84.1***	*8.8*			
OntoNotes coref.	72.9	79.2	6.3	75.3	84.0	8.7			
SPR1	73.7	77.1	3.4	80.1	**84.8**	4.7			
SPR2	76.6	80.2	3.6	82.1	83.1	1.0			
Winograd coref.	52.1	**54.3**	2.2	**54.3**	53.5	-0.8			
Rel. (SemEval)	51.0	60.6	9.6	55.7	77.8	22.1			
Macro Average	69.1	78.1	9.0	75.4	**84.4**	9.1			

	BERT-base				BERT				
	F1 Score			Abs. Δ	F1 Score				
	Lex.	cat	mix	ELMo	Lex.	cat	mi:		
Part-of-Speech	88.4	**97.0**	96.7	0.0	88.1	96.5	**96.9**	0.2	0.2
Constituents	68.4	83.7	86.7	2.1	69.0	80.1	**87.0**	0.4	2.5
Dependencies	80.1	93.0	95.1	1.1	80.2	91.5	**95.4**	0.3	1.4
Entities	90.9	96.1	96.2	0.6	91.8	96.2	**96.5**	0.3	0.9
SRL (all)	75.4	89.4	91.3	1.2	76.5	88.2	**92.3**	1.0	2.2
Core roles	*74.9*	*91.4*	*93.6*	*1.0*	*76.3*	*89.9*	***94.6***	*1.0*	*2.0*
Non-core roles	*76.4*	*84.7*	*85.9*	*1.8*	*76.9*	*84.1*	***86.9***	*1.0*	*2.8*
OntoNotes coref.	74.9	88.7	90.2	6.3	75.7	89.6	**91.4**	1.2	7.4
SPR1	79.2	84.7	**86.1**	1.3	79.6	85.1	**85.8**	-0.3	1.0
SPR2	81.7	83.0	**83.8**	0.7	81.6	83.2	**84.1**	0.3	1.0
Winograd coref.	54.3	53.6	54.9	1.4	53.0	53.8	**61.4**	6.5	7.8
Rel. (SemEval)	57.4	78.3	82.0	4.2	56.2	77.6	**82.4**	0.5	4.6
Macro Average	75.1	84.8	86.3	1.9	75.2	84.2	**87.3**	1.0	2.9

[Tenney et al., ACL 2019]

Pretrained Representation	Avg.	CCG	POS PTB	POS EWT	Chunk	NER	ST	GED	Supersense ID PS-Role	Supersense ID PS-Fxn	EF
ELMo (original) best layer	81.58	93.31	97.26	95.61	90.04	82.85	93.82	29.37	75.44	84.87	73.20
ELMo (4-layer) best layer	81.58	93.81	**97.31**	95.60	89.78	82.06	**94.18**	29.24	74.78	85.96	73.03
ELMo (transformer) best layer	80.97	92.68	97.09	95.13	93.06	81.21	93.78	30.80	72.81	82.24	70.88
OpenAI transformer best layer	75.01	82.69	93.82	91.28	86.06	58.14	87.81	33.10	66.23	76.97	74.03
BERT (base, cased) best layer	84.09	93.67	96.95	95.21	92.64	82.71	93.72	43.30	**79.61**	87.94	75.11
BERT (large, cased) best layer	**85.07**	**94.28**	96.73	**95.80**	**93.64**	**84.44**	93.83	**46.46**	79.17	**90.13**	**76.25**
GloVe (840B.300d)	59.94	71.58	90.49	83.93	62.28	53.22	80.92	14.94	40.79	51.54	49.70
Previous state of the art (without pretraining)	83.44	94.7	97.96	95.82	95.77	91.38	95.15	39.83	66.89	78.29	77.10

[Liu et al. NAACL 2019]

Method	Distance UUAS	Distance DSpr.	Depth Root%	Depth NSpr.
Linear	48.9	0.58	2.9	0.27
ELMo0	26.8	0.44	54.3	0.56
Decay0	51.7	0.61	54.3	0.56
Proj0	59.8	0.73	64.4	0.75
ELMo1	77.0	0.83	86.5	0.87
BERTbase7	79.8	0.85	88.0	0.87
BERTlarge15	**82.5**	0.86	89.4	0.88
BERTlarge16	81.7	**0.87**	**90.1**	**0.89**

[Hewitt et al., 2019]

Probing classifier findings

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

	BERT-base								
	F1 Score			A					
	Lex.	cat	mix	F					
Part-of-Speech	88.4	**97.0**	96.7						
Constituents	68.4	83.7	86.7						
Dependencies	80.1	93.0	95.1						
Entities	90.9	96.1	96.2						
SRL (all)	75.4	89.4	91.3						
Core roles	*74.9*	*91.4*	*93.6*						
Non-core roles	*76.4*	*84.7*	*85.9*						
OntoNotes coref.	74.9	88.7	90.2	6.3	75.7	89.6	**91.4**	1.2	7.4
SPR1	79.2	84.7	**86.1**	1.3	79.6	85.1	**85.8**	-0.3	1.0
SPR2	81.7	83.0	**83.8**	0.7	81.6	83.2	**84.1**	0.3	1.0
Winograd coref.	54.3	53.6	54.9	1.4	53.0	53.8	**61.4**	6.5	7.8
Rel. (SemEval)	57.4	78.3	82.0	4.2	56.2	77.6	**82.4**	0.5	4.6
Macro Average	75.1	84.8	86.3	1.9	75.2	84.2	**87.3**	1.0	2.9

[Tenney et al., ACL 2019]

- 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...

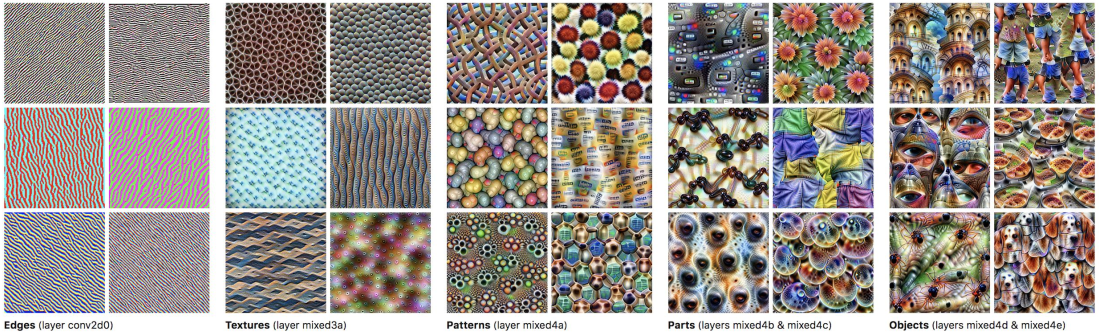
		Supersense ID		
T	GED	PS-Role	PS-Fxn	EF
.82	29.37	75.44	84.87	73.20
.18	29.24	74.78	85.96	73.03
.78	30.80	72.81	82.24	70.88
.81	33.10	66.23	76.97	74.03
.72	43.30	**79.61**	87.94	75.11
.83	**46.46**	79.17	**90.13**	**76.25**
.92	14.94	40.79	51.54	49.70
.15	39.83	66.89	78.29	77.10

[2019)]

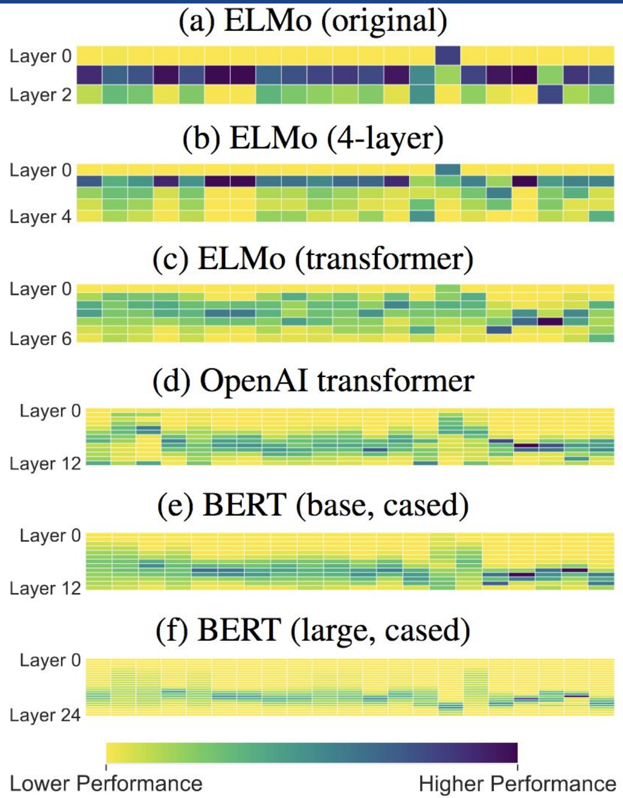
				Spr.
				.27
ELMo0	26.8	0.44	54.3	0.56
Decay0	51.7	0.61	54.3	0.56
Proj0	59.8	0.73	64.4	0.75
ELMo1	77.0	0.83	86.5	0.87
BERTbase7	79.8	0.85	88.0	0.87
BERTlarge15	**82.5**	0.86	89.4	0.88
BERTlarge16	81.7	**0.87**	**90.1**	**0.89**

[Hewitt et. al., 2019]

Probing: Layers of the network



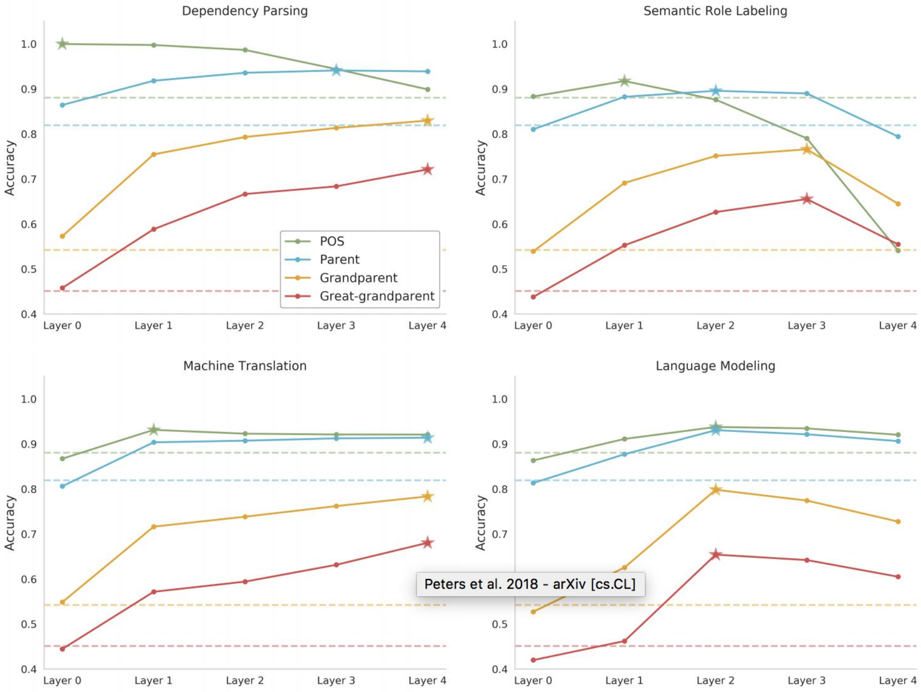
- ❑ **RNN** layers: General linguistic properties
  - ❑ Lowest layers: **morphology**
  - ❑ Middle layers: **syntax**
  - ❑ Highest layers: Task-specific **semantics**
- ❑ **Transformer** layers:
  - ❑ Different trends for different tasks; **middle-heavy**
  - ❑ Also see [Tenney et. al., 2019]



[Fig. from Liu et al. (NAACL 2019)]

Probing: Pretraining Objectives

- 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?

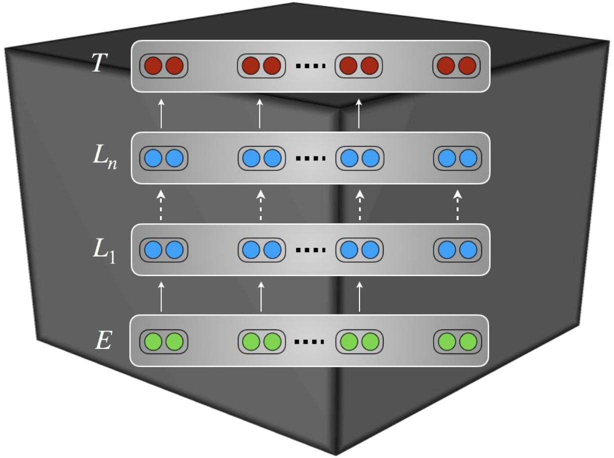
- Representations are **predictive** of certain linguistic phenomena:
  - **Alignments** in translation, Syntactic **hierarchies**
- Pretraining with and without syntax:
  - Better performance with syntax
  - But without, some notion of syntax at least (Williams et al. 2018)
- Network architectures determine 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...

Analysis Method 4: Model Alterations

- Progressively erase or mask network components
  - Word embedding dimensions
  - Hidden units
  - Input - words / phrases



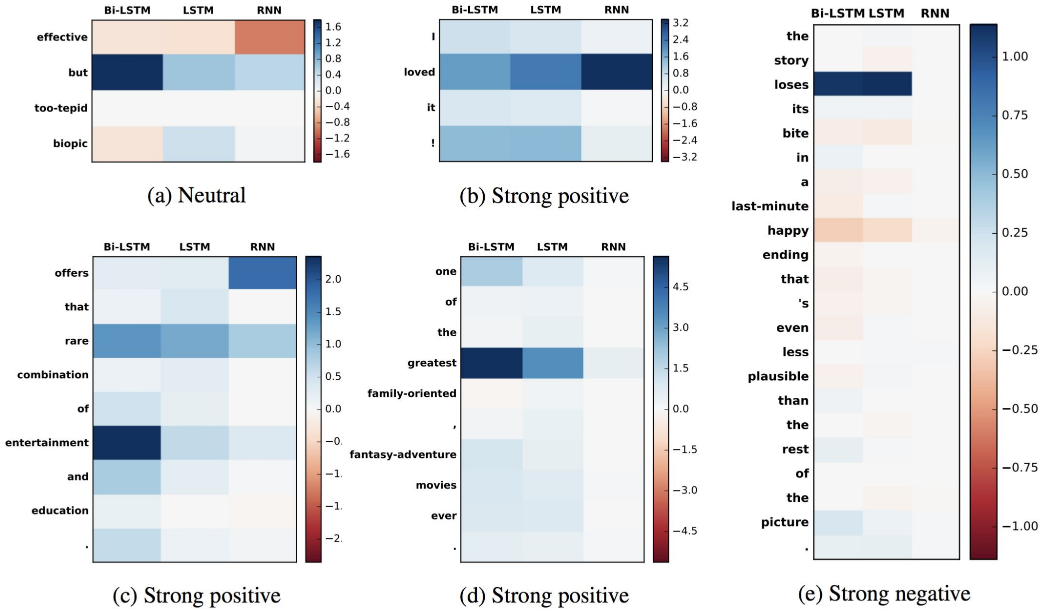
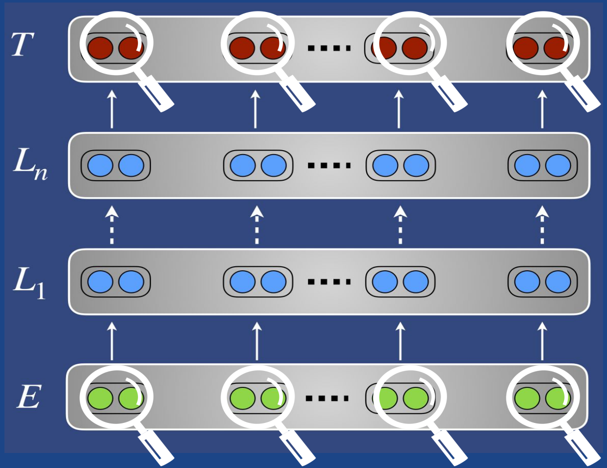


Figure 5: Heatmap of word importance (computed using Eq. 1) in sentiment analysis.

Li et al., 2016

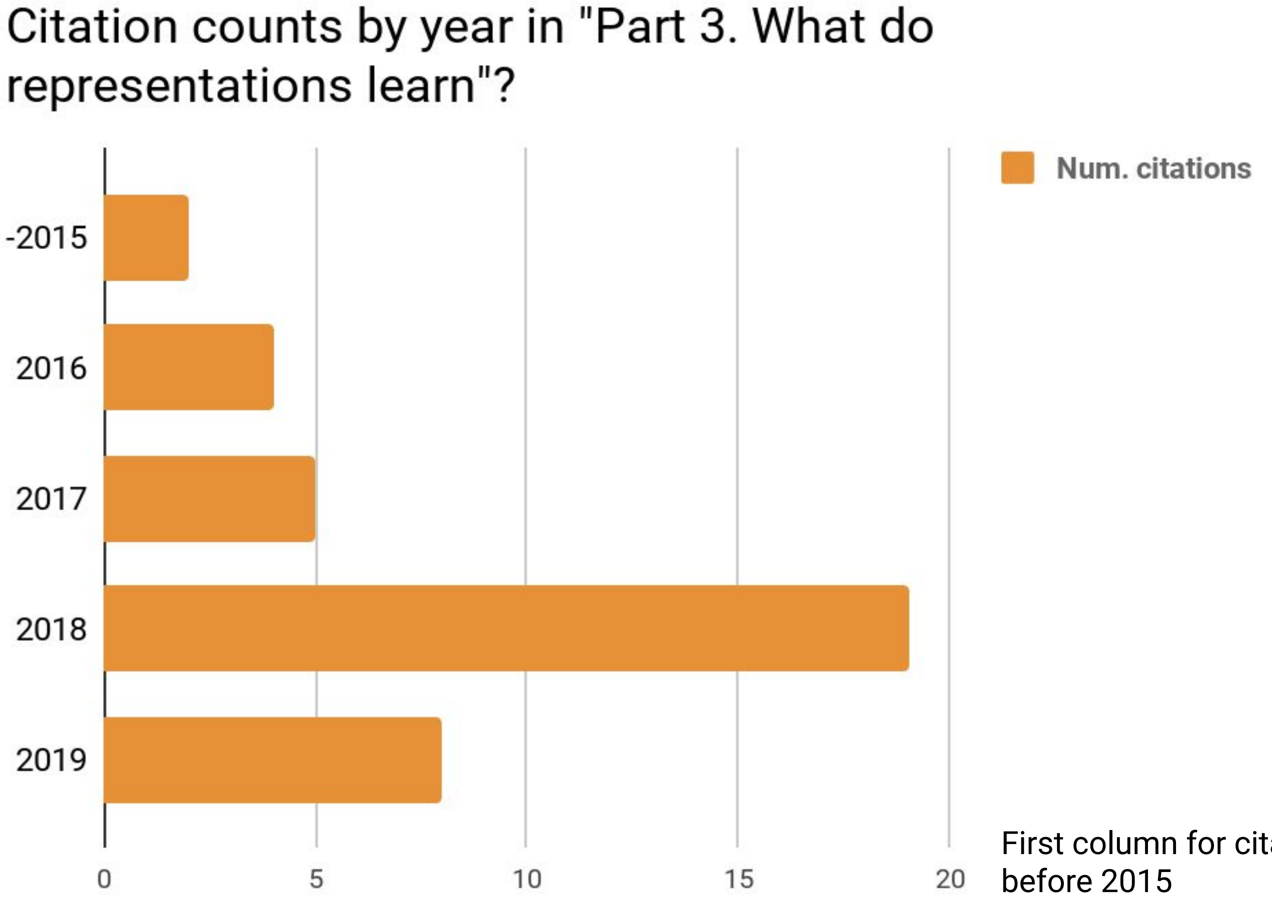
So, what is in a representation?

- 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!



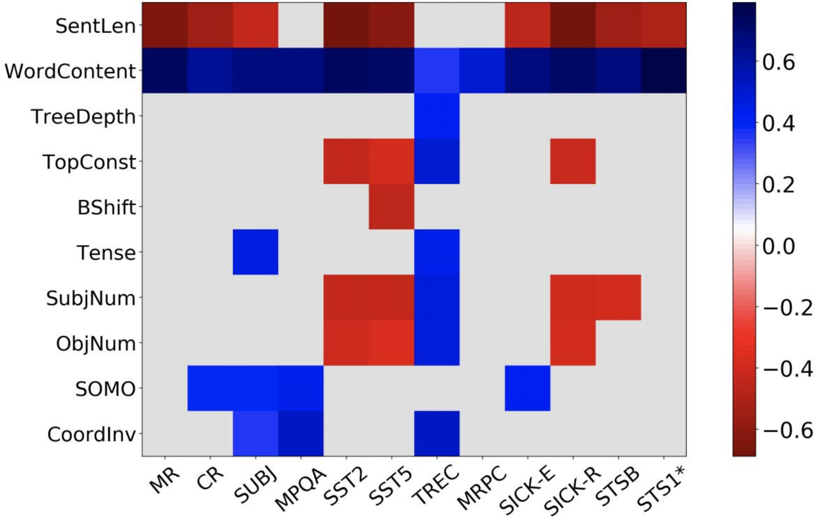
First column for citations in and before 2015

What's next?

- ❑ Linguistic Awareness 
- ❑ Interpretability 

Interpretability + transferability to downstream tasks is key

→ Up next!



[Conneau et al., 2018]

Correlation of probes to downstream tasks

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**)

Break

Image credit: Andrejs Kirma

Transfer Learning in NLP

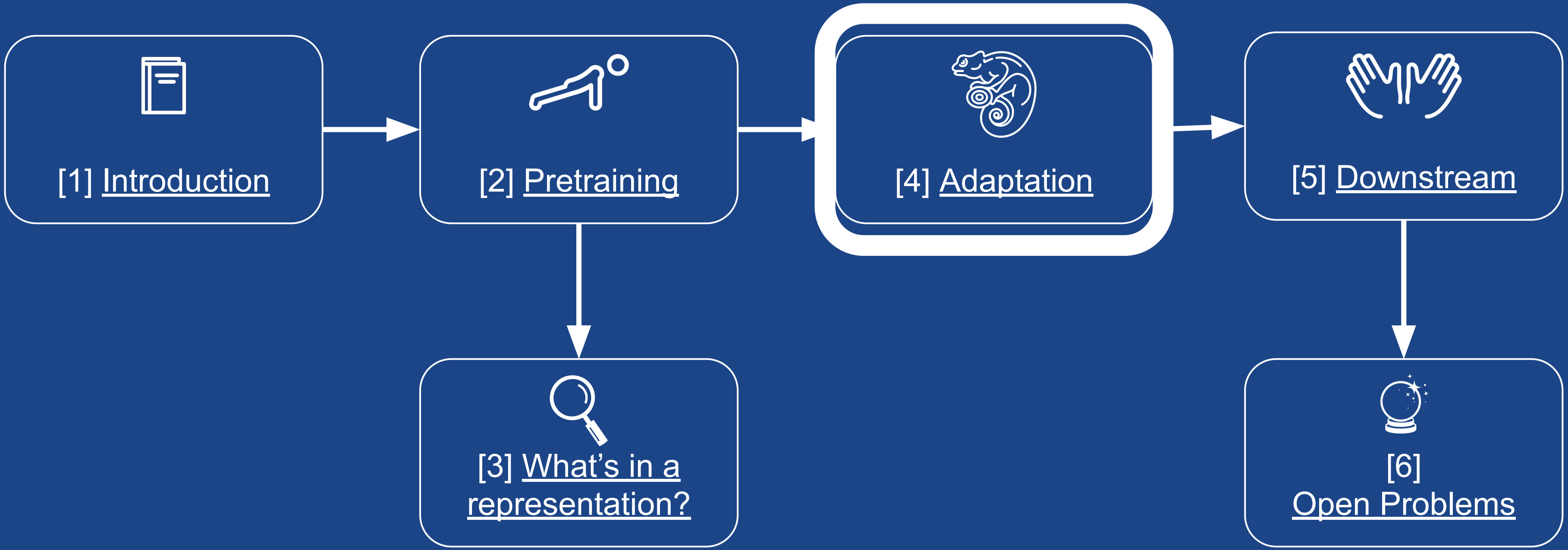
Follow along with the tutorial:

- ❑ 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
[6] Open Problems

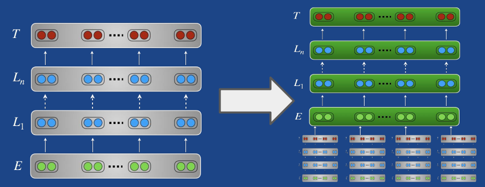
4. Adaptation

Image credit: Ben Didier

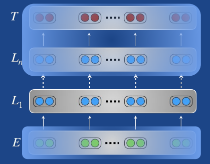
4 – How to adapt the pretrained model

Several orthogonal directions we can make decisions on:

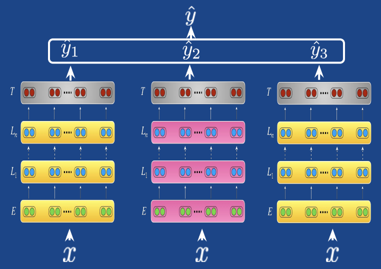
1. **Architectural** modifications?
   How much to change the pretrained model architecture for adaptation



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

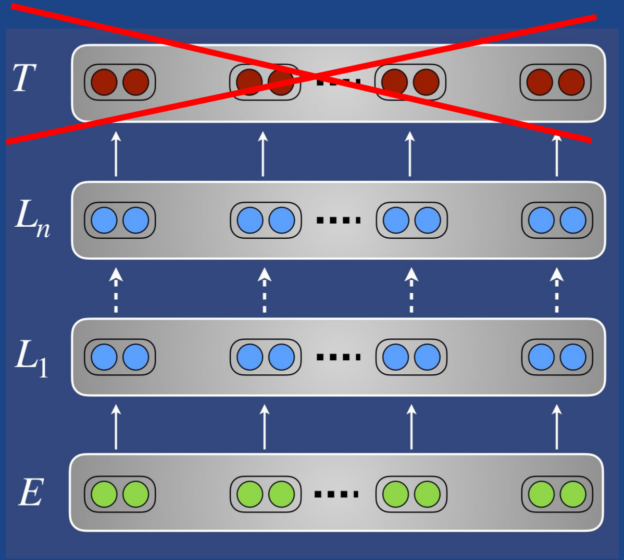
B. **Modify** pretrained model internal architecture:
Initialize encoder-decoders, task-specific modifications, adapters

Image credit: Darmawansyah

4.1.A – Architecture: Keep model **unchanged**

General workflow:

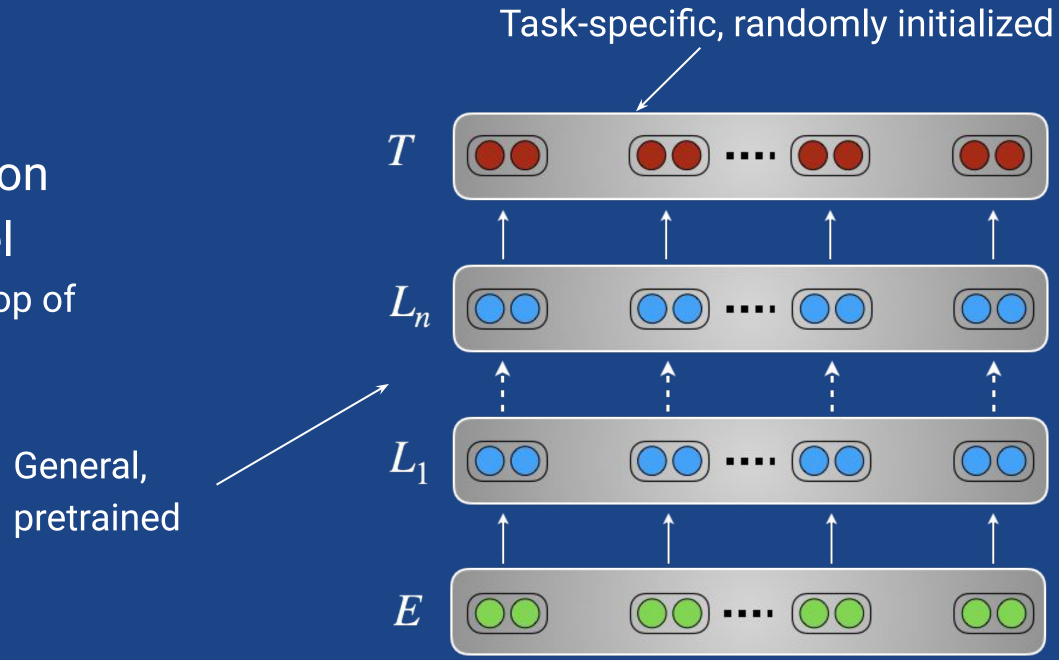
1. **Remove pretraining task head** if not useful for target task
   a. **Example**: remove softmax classifier from pretrained LM
   b. **Not always needed**: some adaptation schemes re-use the pretraining objective/task, e.g. for **multi-task learning**



4.1.A – Architecture: Keep model **unchanged**

General workflow:

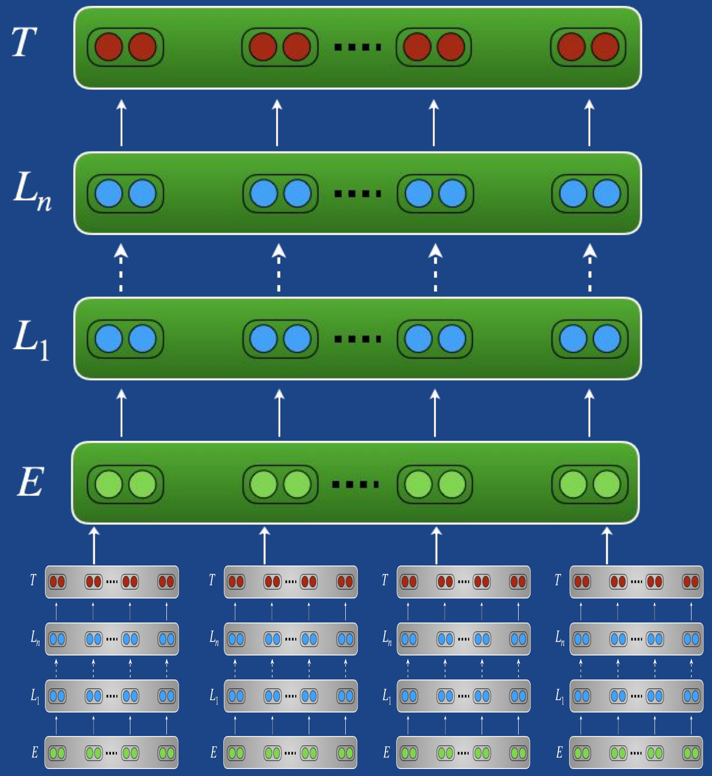
2. Add target task-specific layers on top/bottom of pretrained model
    a. **Simple**: adding linear layer(s) on top of the pretrained model



4.1.A – Architecture: Keep model **unchanged**

General workflow:

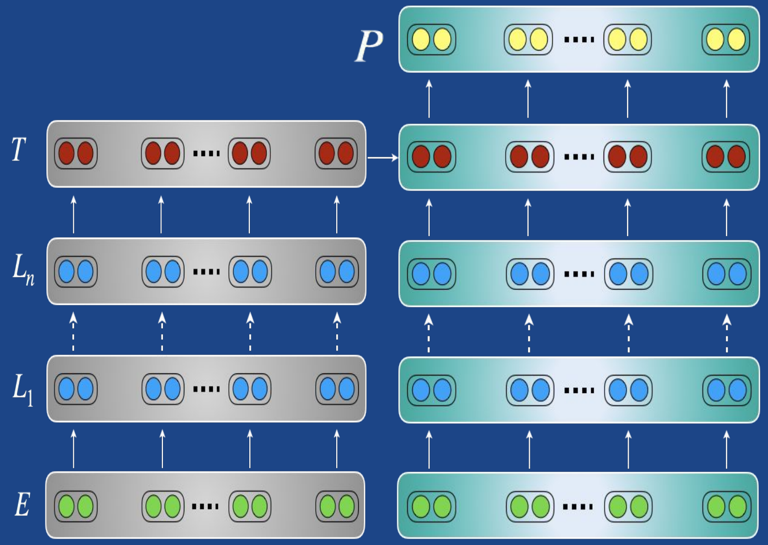
2. Add target task-specific layers on top/bottom of pretrained model
    a. **Simple**: adding linear layer(s) on top of the pretrained model
    b. **More complex**: model output as input for a separate model
    c. Often beneficial when target task requires **interactions** that are not available in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



4.1.B – Architecture: Modifying model internals

Various reasons:

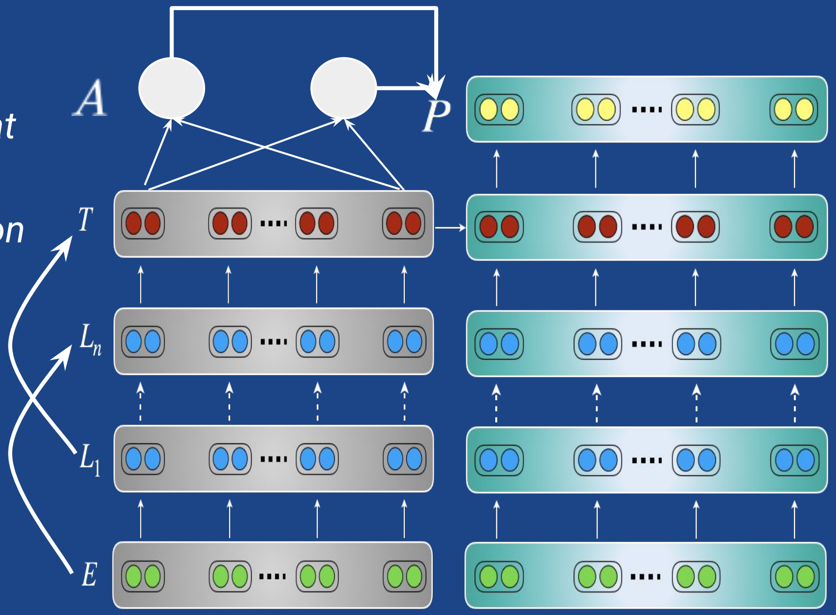
1. Adapting to a **structurally different** target task
    a. *Ex: Pretraining with a <u>single</u> input sequence (ex: language modeling) but adapting to a task with <u>several</u> input sequences (ex: translation, conditional generation...)*
    b. *Use the pretrained model weights to initialize as 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:

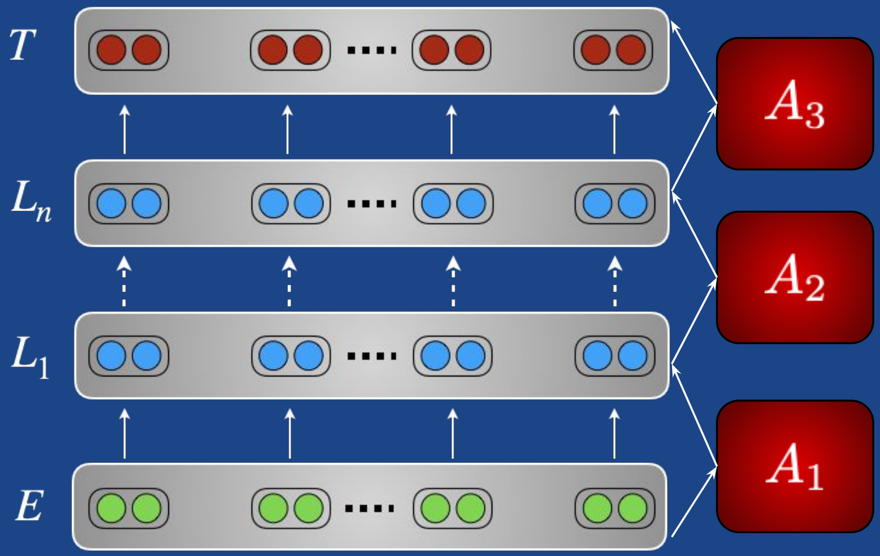
2. Task-specific **modifications**
   a. *Provide pretrained model with capabilities that are useful for the target task*
   b. *Ex: Adding skip/residual connections, attention ([Ramachandran et al., EMNLP 2017](#))*



4.1.B – Architecture: Modifying model internals

Various reasons:

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)

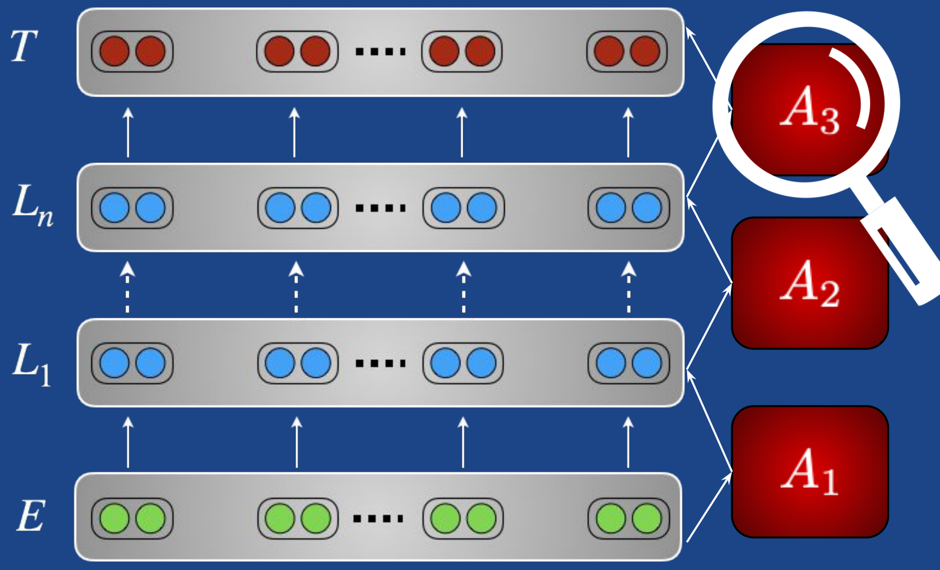
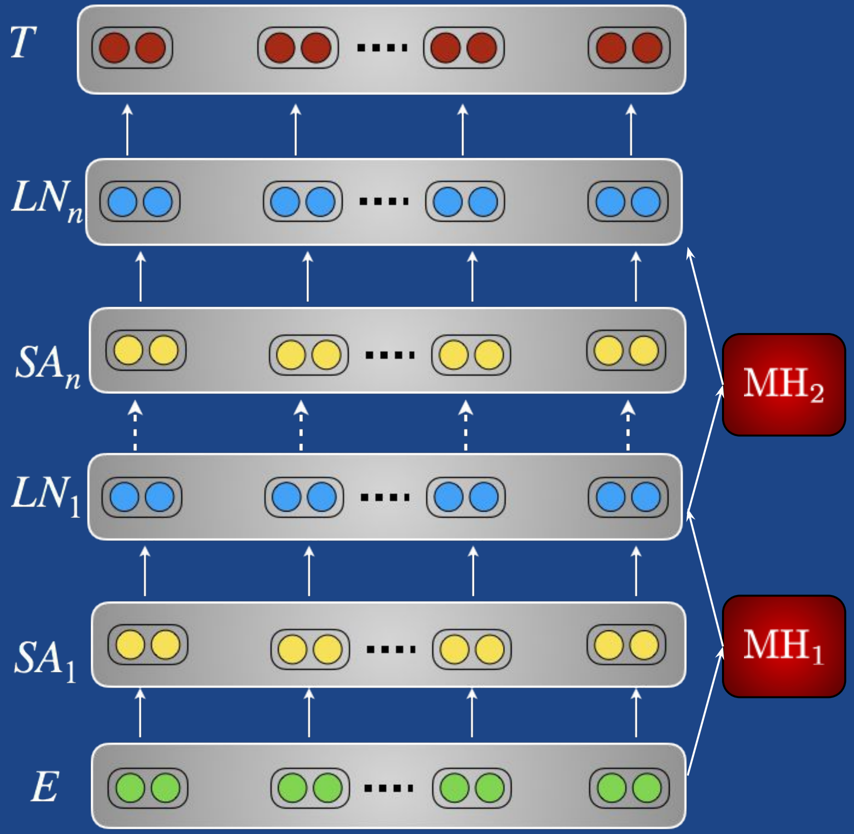


Image credit: Caique Lima

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

Image credit: Chanaky

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/NAACLTransferColab ⇨ code of the following slides
  - Code http://tiny.cc/NAACLTransferCode ⇨ same code in a repo

Hands-on: Model adaptation

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)
- 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 ? —> *DESC*
  - ★ What films featured the character Popeye Doyle ? —> *ENTY*

	Model	Test
TREC-6	CoVe (McCann et al., 2017)	4.2
	TBCNN (Mou et al., 2015)	4.0
	LSTM-CNN (Zhou et al., 2016)	3.9
	ULMFiT (ours)	**3.6**

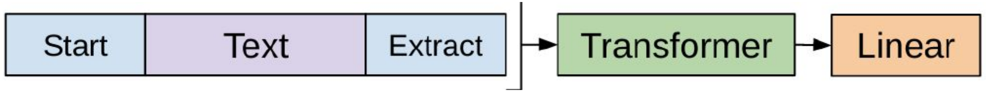
Transfer learning models shine on this type of low-resource task

(Howard and Ruder, ACL 2018)

Hands-on: Model adaptation

First adaptation scheme

Classification



- 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

```
AdaptationConfig = namedtuple('AdaptationConfig',
  field_names="num_classes, dropout, initializer_range, batch_size, lr, max_norm, n_epochs,"
              "n_warmup, valid_set_prop, gradient_accumulation_steps, device,"
              "log_dir, dataset_cache")
adapt_args = AdaptationConfig(
               6          , 0.1     , 0.02              , 16         , 6.5e-5, 1.0    , 3,
               10       , 0.1          , 1, "cuda" if torch.cuda.is_available() else "cpu",
               "./"    , "./dataset_cache.bin")
```

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.

I	love	Mom	'	s	cooking	[CLS]
I	love	you	too	!	[CLS]	
No	way	[CLS]				
This	is	the	one	[CLS]		
Yes	[CLS]					

```
import random
from torch.utils.data 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] = 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=False)
test_loader = DataLoader(datasets['test'], batch_size=adapt_args.batch_size, shuffle=False)
```

Hands-on: Model adaptation

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)

```
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)

        self.apply(self.init_weights)

    def init_weights(self, module):
        if isinstance(module, (nn.Linear, nn.Embedding, nn.LayerNorm)):
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if isinstance(module, (nn.Linear, nn.LayerNorm)) and module.bias is not None:
            module.bias.data.zero_()

    def forward(self, x, clf_tokens_mask, clf_labels=None, padding_mask=None):
        hidden_states = self.transformer(x, padding_mask)

        clf_tokens_states = (hidden_states * clf_tokens_mask.unsqueeze(-1).float()).sum(dim=0)
        clf_logits = self.classification_head(clf_tokens_states)

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

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

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

# Otherwise, just load pretrained model weigths (and reload the training config as well)
state_dict = torch.load(cached_path("https://s3.amazonaws.com/models.huggingface.co/"
                                    "naacl-2019-tutorial/model_checkpoint.pth"), map_location='cpu')
args = torch.load(cached_path("https://s3.amazonaws.com/models.huggingface.co/"
                              "naacl-2019-tutorial/model_training_args.bin"))

adaptation_model = TransformerWithClfHead(config=args, fine_tuning_config=adapt_args).to(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']
```

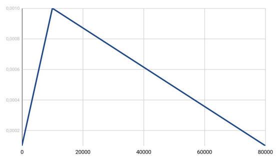
Hands-on: Model adaptation

Our fine-tuning code:

A simple training update function:
** prepare inputs: transpose and <u>build padding & classification token masks</u>*
** 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 [seq length, batch]
    _, loss = adaptation_model(inputs, clf_tokens_mask=(inputs == tokenizer.vocab['[CLS]']), clf_labels=labels,
                               padding_mask=(batch == tokenizer.vocab['[PAD]']))
    loss = loss / adapt_args.gradient_accumulation_steps
    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)
        inputs = batch.transpose(0, 1).contiguous()  # to shape [seq length, 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)

# Attache 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")
ProgressBar(persist=True).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, {'mymodel': adaptation_model})
torch.save(args, os.path.join(adapt_args.log_dir, 'fine_tuning_args.bin'))
```

Hands-on: Model adaptation – Results

We can now fine-tune our model on TREC:

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

Epoch [1/3]      [307/307] 100%| , loss=3.85e-01 [01:10<00:00]
Validation Epoch: 1 Error rate: 9.174311926605505
Epoch [2/3]      [307/307] 100%| , loss=1.73e-01 [01:10<00:00]
Validation Epoch: 2 Error rate: 5.871559633027523
Epoch [3/3]      [307/307] 100%| , loss=9.63e-02 [01:10<00:00]
Validation Epoch: 3 Error rate: 5.688073394495408
<ignite.engine.engine.State at 0x7ff4c8b385f8>
```

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

Test Results - Error rate: 3.600
```

	Model	Test
TREC-6	CoVe (McCann et al., 2017)	4.2
	TBCNN (Mou et al., 2015)	4.0
	LSTM-CNN (Zhou et al., 2016)	3.9
	ULMFiT (ours)	**3.6**

We are at the state-of-the-art (ULMFiT)

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

Hands-on: Model adaptation – Results

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.

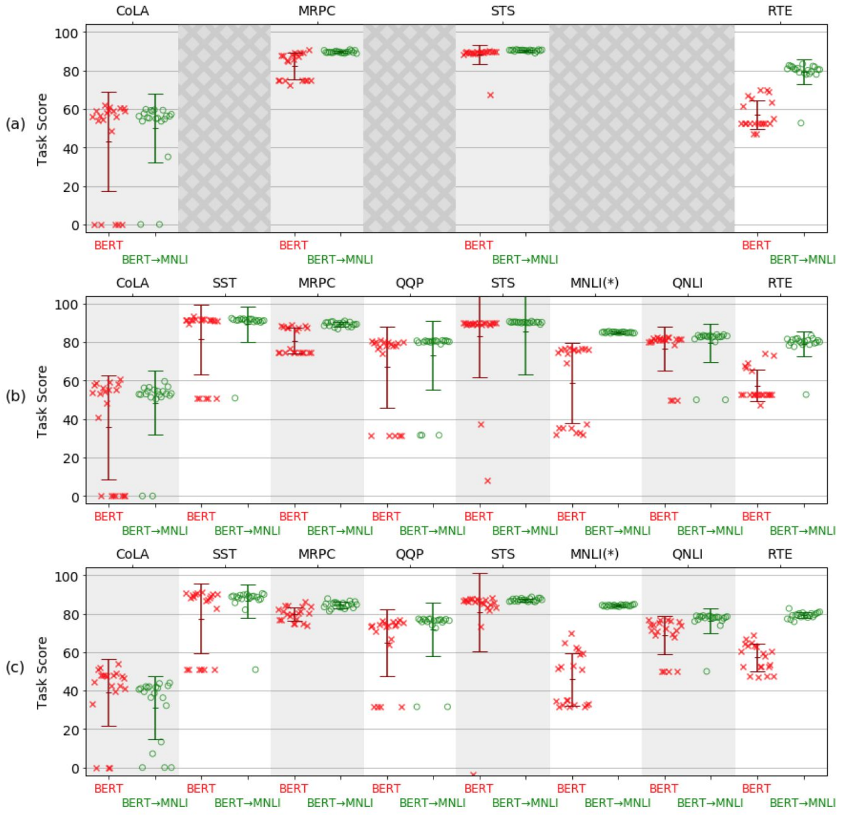


Figure 1: Distribution of task scores across 20 random restarts for BERT, and BERT with intermediary fine-tuning on MNLI. Each cross represents a single run. Error lines show mean±1std. (a) Fine-tuned on all data, for tasks with <10k training examples. (b) Fine-tuned on no more than 5k examples for each task. (c) Fine-tuned on no more than 1k examples for each task. (*) indicates that the intermediate task is the same as the target task.

[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

Image credit: ProSymbols, purplestudio, Markus, Alfredo

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

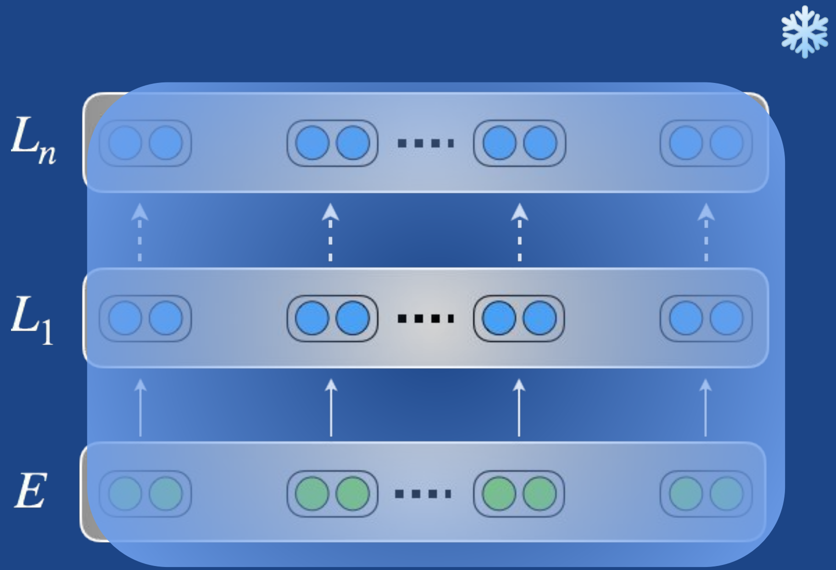
Image credit: purplestudio

4.2.A – Optimization: Which weights?

Don't touch the pretrained weights!

Feature extraction:

- ❑ Weights are **frozen**

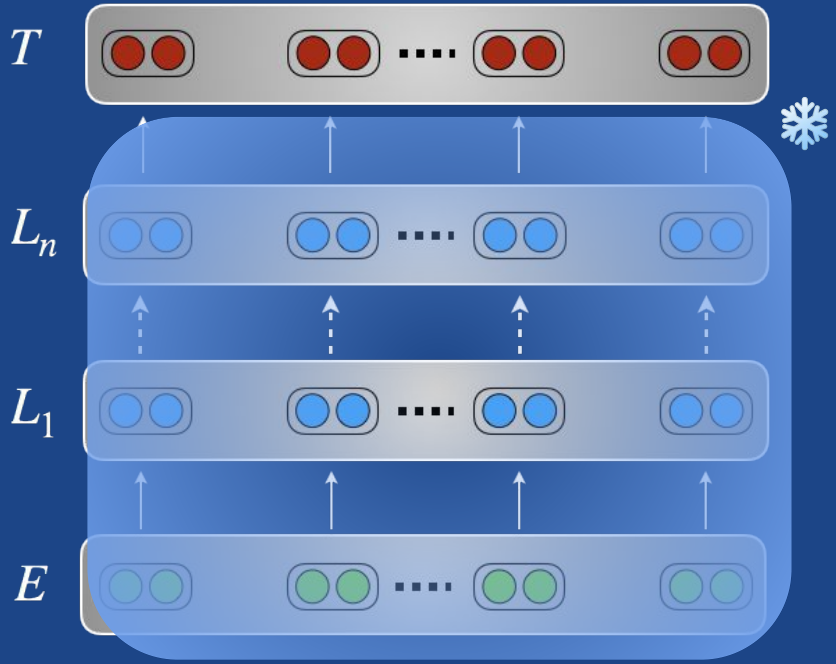


4.2.A – Optimization: Which weights?

Don't touch the pretrained weights!

Feature extraction:

- ❑ Weights are **frozen**
- ❑ A **linear classifier** is trained on top of the pretrained representations

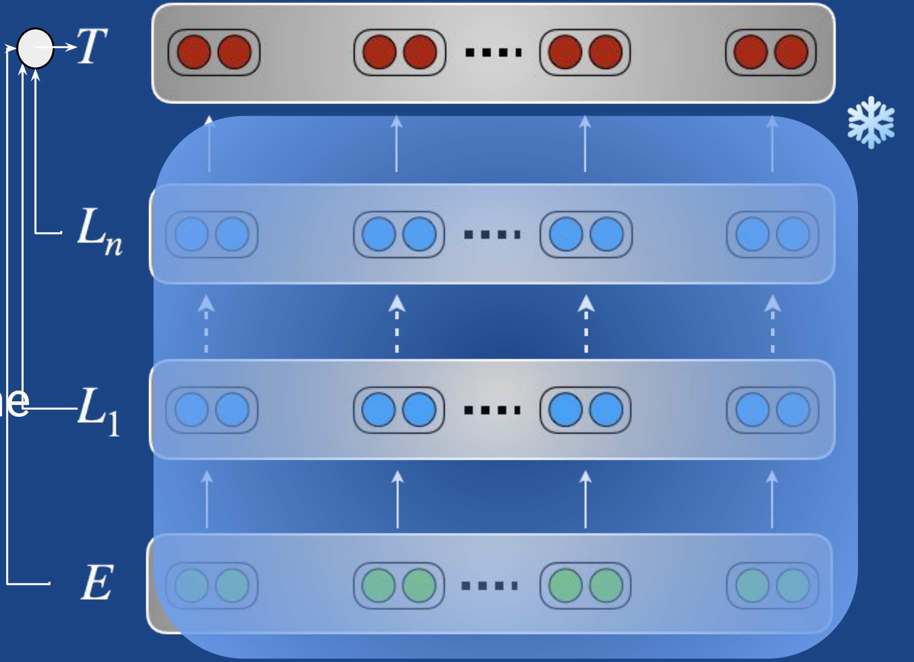


4.2.A – Optimization: Which weights?

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)

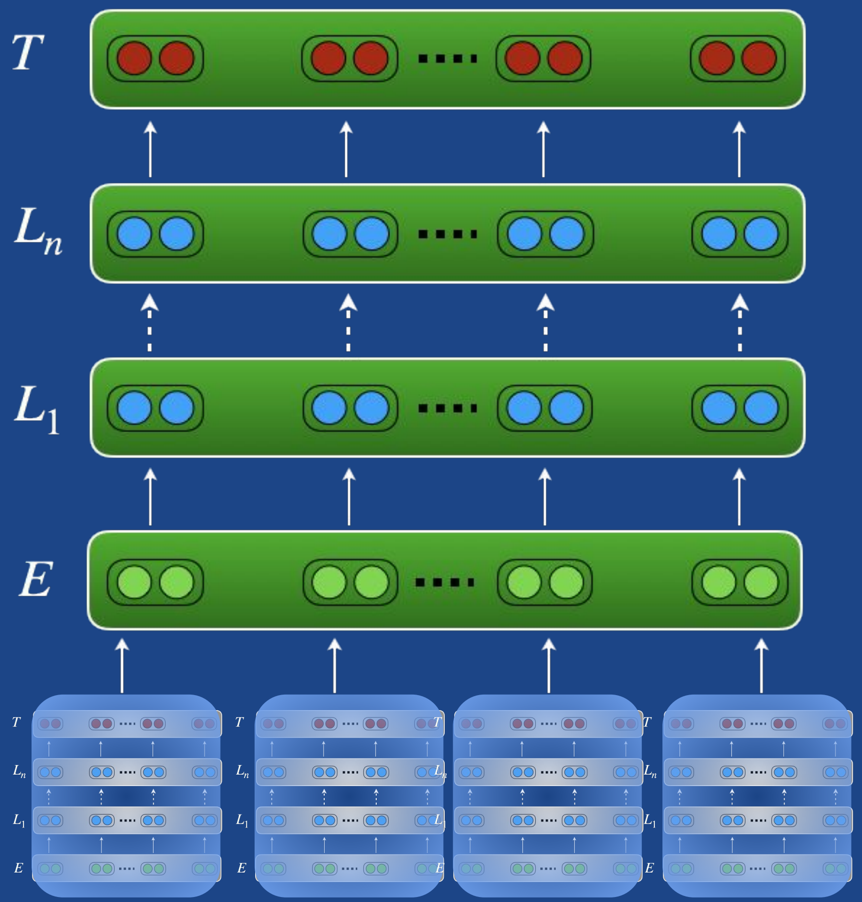


4.2.A – Optimization: Which weights?

Don't touch the pretrained weights!

Feature extraction:

- ❑ Alternatively, pretrained representations are **used as features** in downstream model

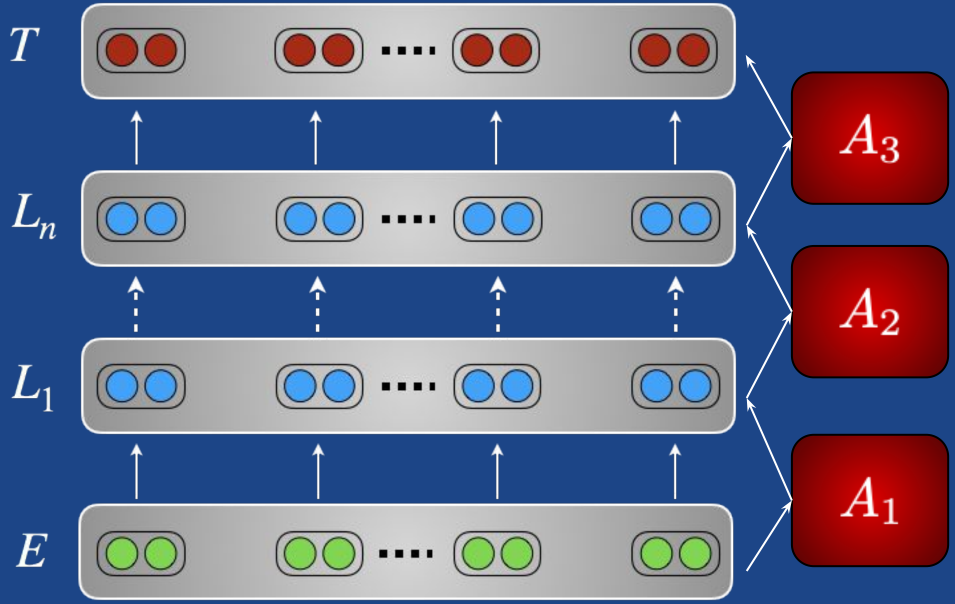


4.2.A – Optimization: Which weights?

Don't touch the pretrained weights!

Adapters

- ❑ Task-specific modules that are added **in between** existing layers

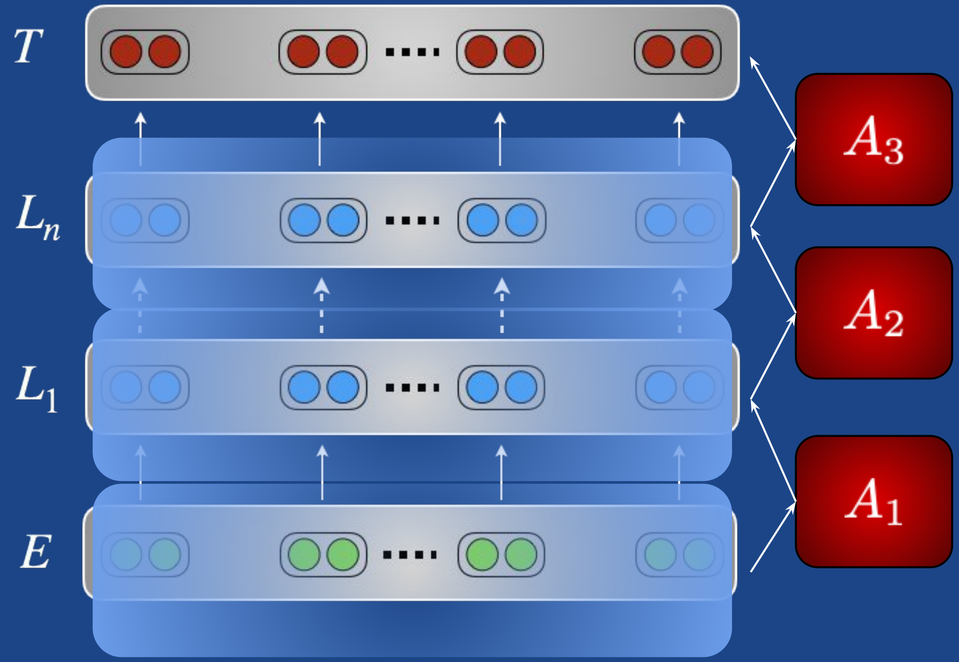


4.2.A – Optimization: Which weights?

Don't touch the pretrained weights!

Adapters

- ❑ Task-specific modules that are added **in between** existing layers
- ❑ Only adapters are trained

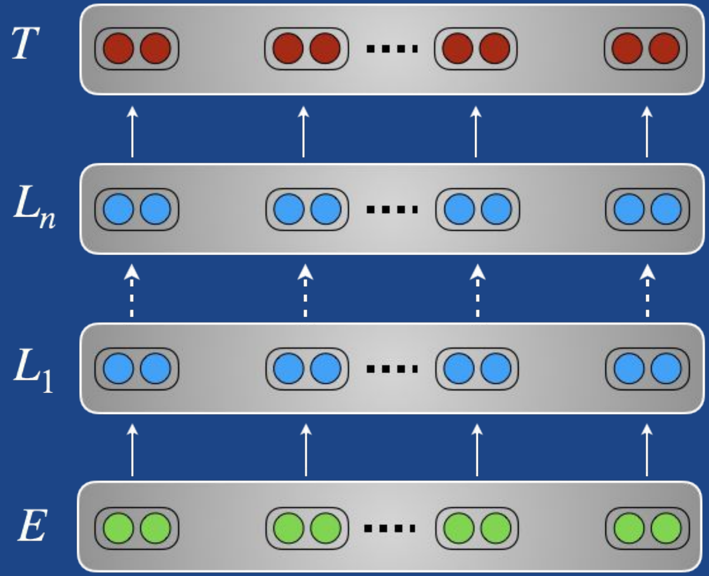


4.2.A – Optimization: Which weights?

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

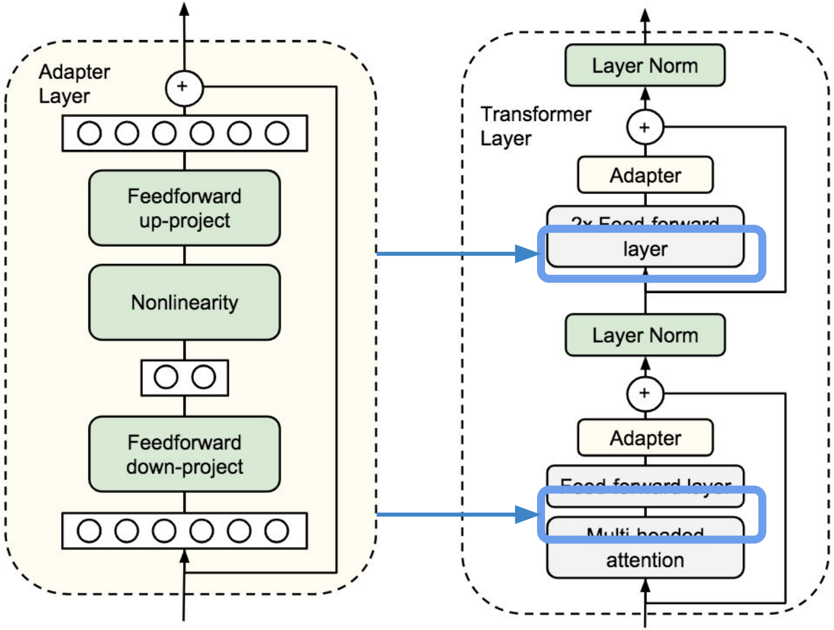
Image credit: Chanaky

Hands-on: Model adaptation

Second adaptation scheme: Using Adapters

- Modifications:
  - **add Adapters** inside the backbone model: *Linear* ➪ *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

Hands-on: Model adaptation

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

```python
class TransformerWithAdapters(Transformer):
    def __init__(self, adapters_dim, embed_dim, hidden_dim, num_embeddings, num_max_positions,
                 num_heads, num_layers, dropout, causal):
        """ Transformer with adapters (small bottleneck layers) """
        super().__init__(embed_dim, hidden_dim, num_embeddings, num_max_positions, num_heads, num_layers,
                         dropout, causal)
        self.adapters_1 = nn.ModuleList()
        self.adapters_2 = nn.ModuleList()
        for _ in range(num_layers):

            self.adapters_1.append(nn.Sequential(nn.Linear(embed_dim, adapters_dim),
                                                 nn.ReLU(),
                                                 nn.Linear(adapters_dim, embed_dim)))

            self.adapters_2.append(nn.Sequential(nn.Linear(embed_dim, adapters_dim),
                                                 nn.ReLU(),
                                                 nn.Linear(adapters_dim, embed_dim)))

    def forward(self, x, padding_mask=None):
        """ x has shape [seq length, batch], padding_mask has shape [batch, seq length] """
        positions = torch.arange(len(x), device=x.device).unsqueeze(-1)
        h = self.tokens_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, adapter_1, layer_norm_2, feed_forward, adapter_2)\
                          in zip(self.layer_norms_1, self.attentions,    self.adapters_1,
                                 self.layer_norms_2, self.feed_forwards, self.adapters_2):
            h = layer_norm_1(h)
            x, _ = attention(h, h, h, attn_mask=attn_mask, need_weights=False, key_padding_mask=padding_mask)
            x = self.dropout(x)

            x = adapter_1(x) + x  # Add an adapter with a skip-connection after attention module

            h = x + h

            h = layer_norm_2(h)
            x = feed_forward(h)
            x = self.dropout(x)

            x = adapter_2(x) + x  # Add an adapter with a skip-connection after feed-forward module

            h = x + h
        return h
```

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:

```
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},"
      f" i.e. {100 * trained_parameters/full_parameters:.2f}%")
```

```
We will train 1.284961e+07 parameters out of 5.125265e+07, i.e. 25.07%
```

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).

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

```
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.

4.2.B – Optimization: What schedule?

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.

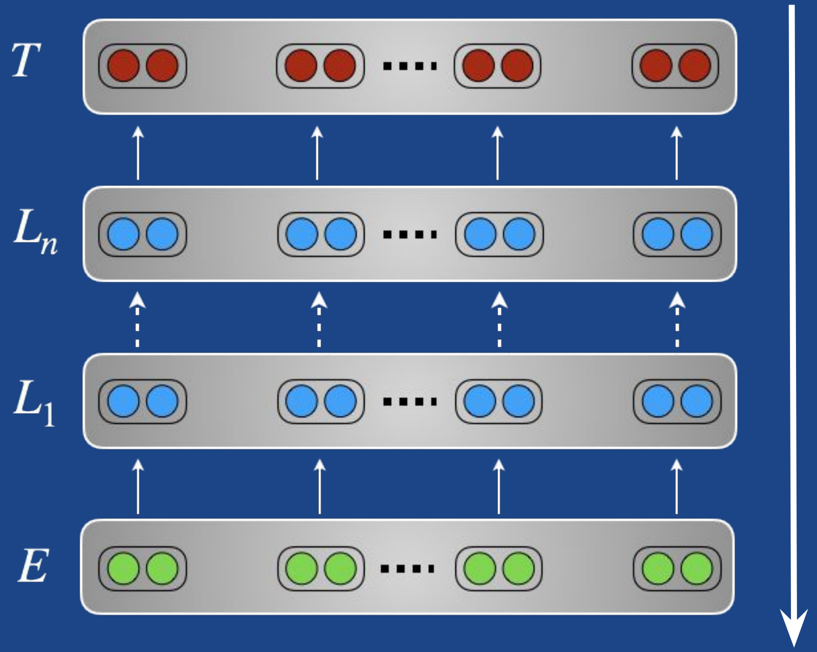
Image credit: Markus

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

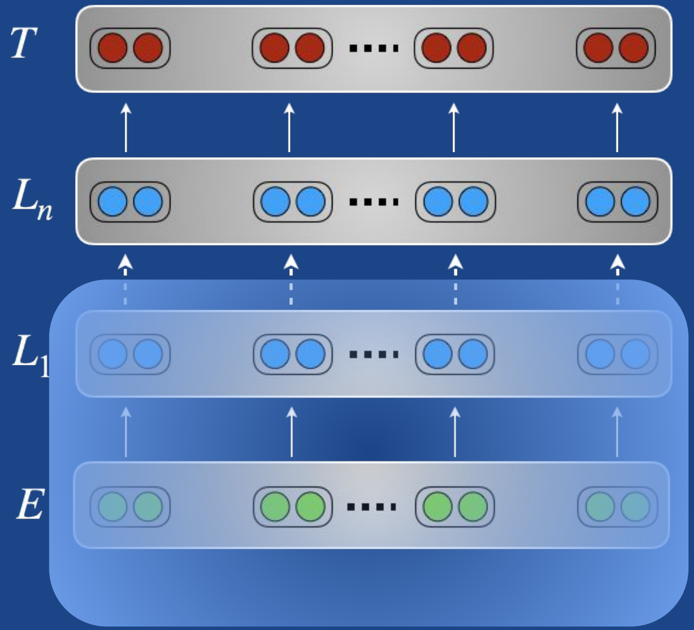


4.2.B – Optimization: Freezing

Main intuition: Training all layers at the same time on **data of a different distribution and task** may lead to instability and poor solutions.

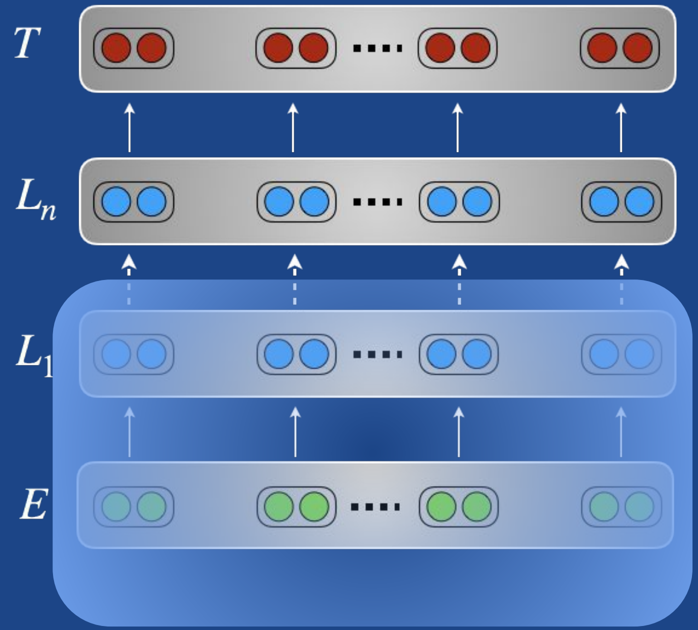
Solution: **Train layers individually** to give them time to adapt to new task and data.

Goes back to layer-wise training of early deep neural networks ([Hinton et al., 2006](); [Bengio et al., 2007]()).



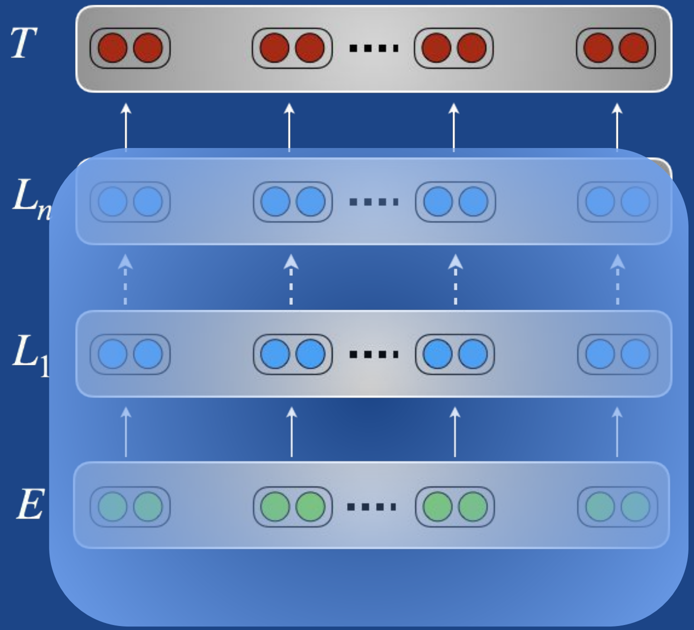
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)



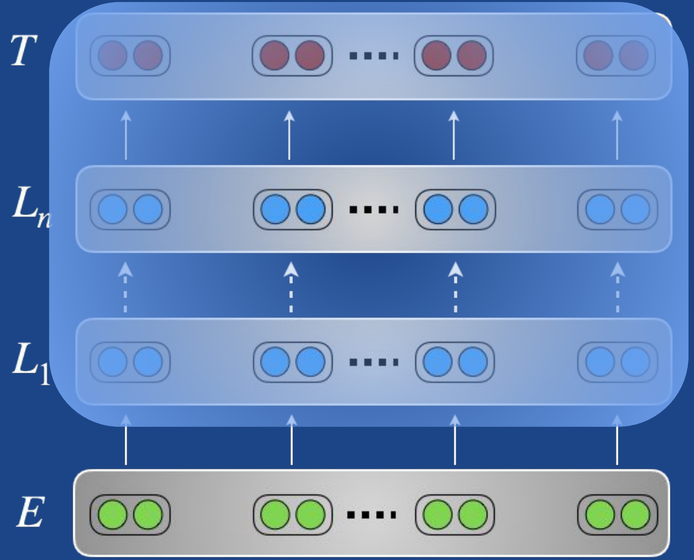
4.2.B – Optimization: Freezing

- 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



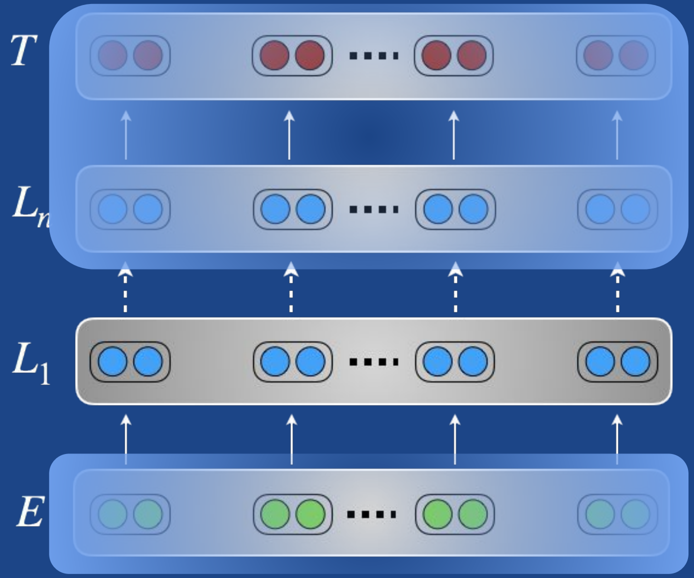
4.2.B – Optimization: Freezing

- 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



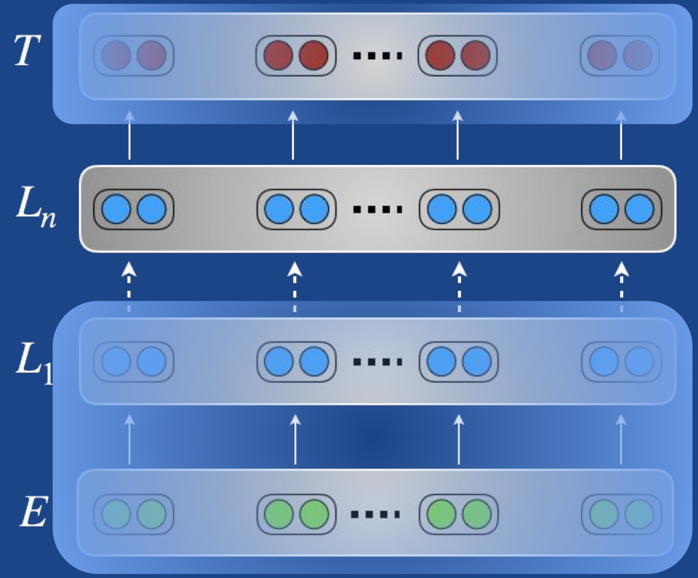
4.2.B – Optimization: Freezing

- 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



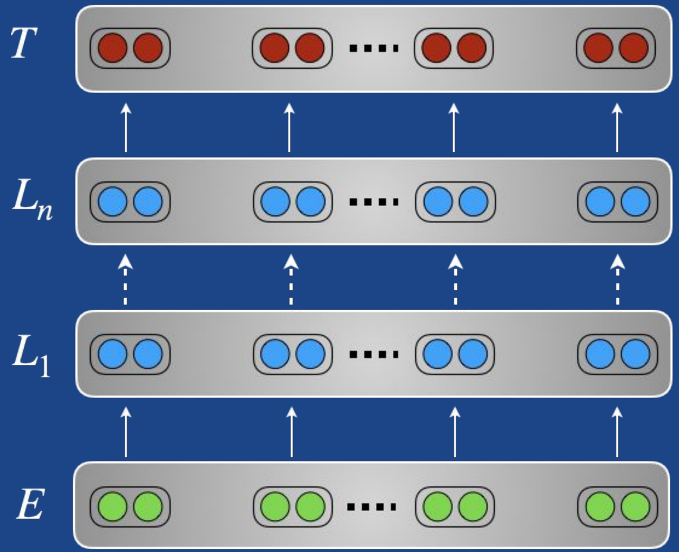
4.2.B – Optimization: Freezing

- 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



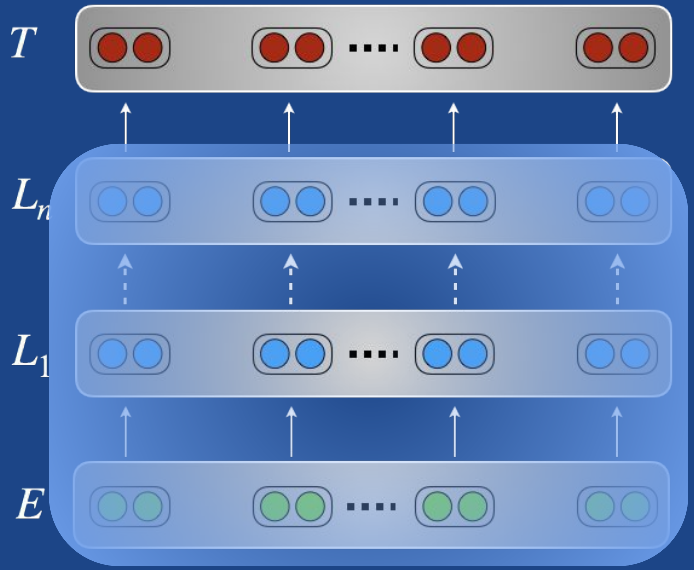
4.2.B – Optimization: Freezing

- 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



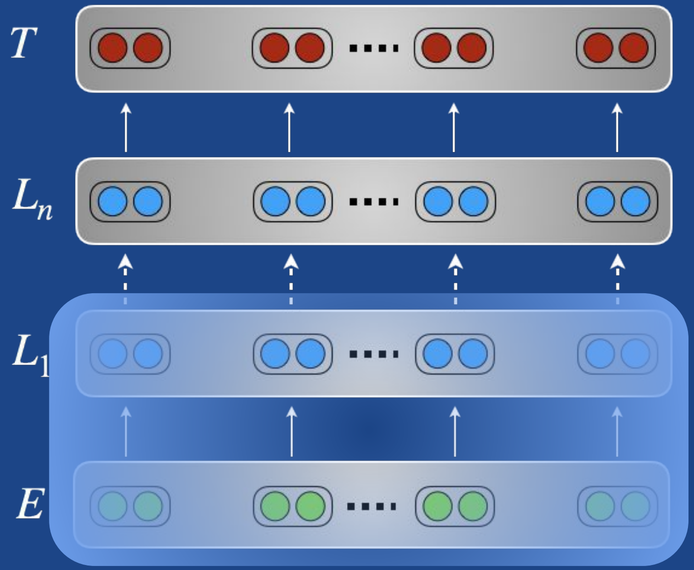
4.2.B – Optimization: Freezing

- 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



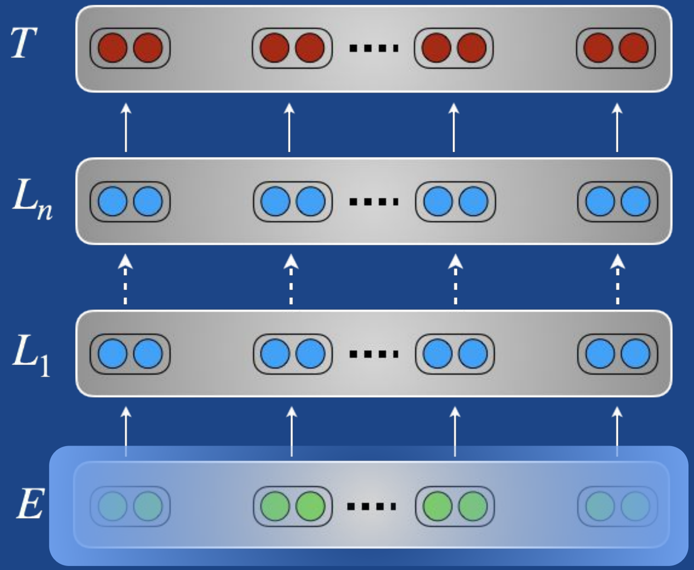
4.2.B – Optimization: Freezing

- 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



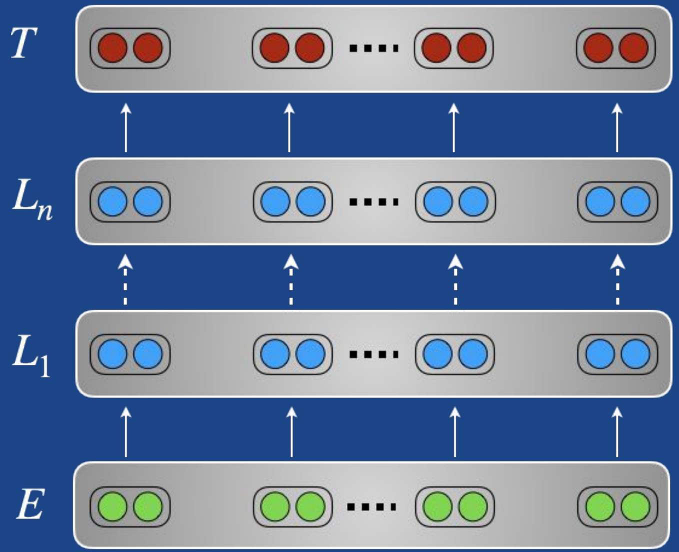
4.2.B – Optimization: Freezing

- 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



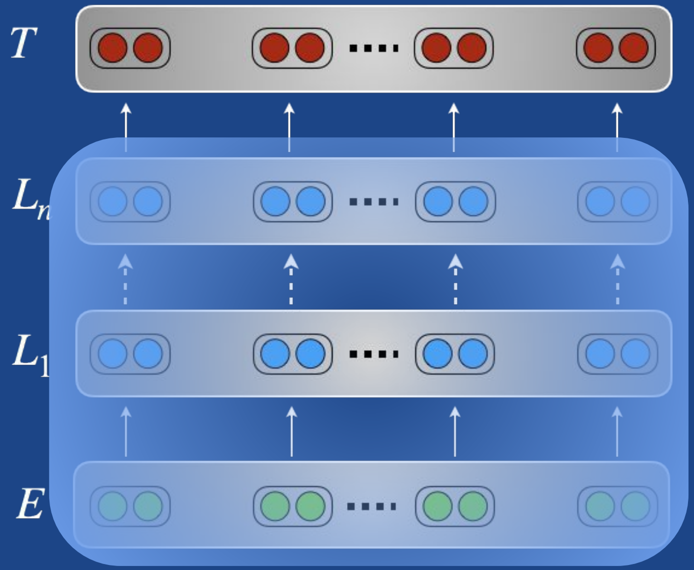
4.2.B – Optimization: Freezing

- ❏ 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



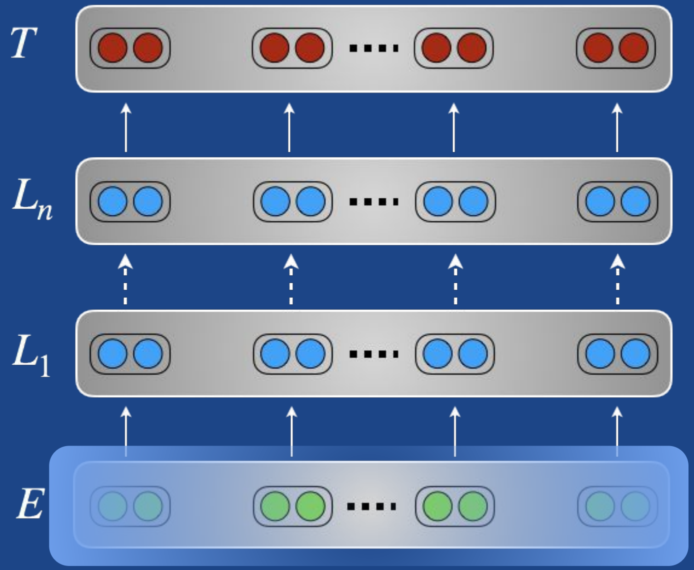
4.2.B – Optimization: Freezing

- 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



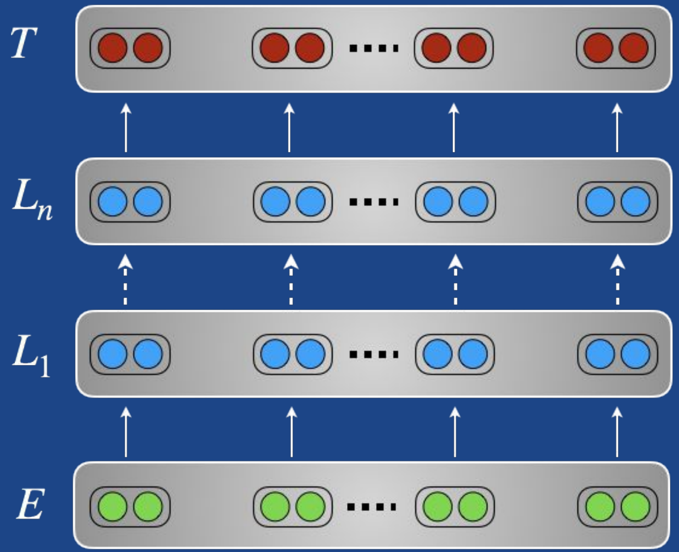
4.2.B – Optimization: Freezing

- Freezing all but the top layer (Long et al., ICML 2015)
- 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



4.2.B – Optimization: Freezing

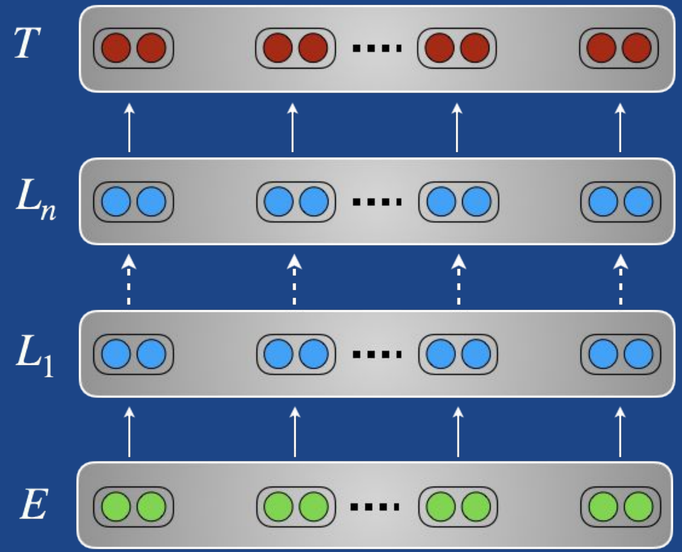
- Freezing all but the top layer (Long et al., ICML 2015)
- Chain-thaw (Felbo et al., EMNLP 2017): training one layer at a time
- Gradually unfreezing (Howard & Ruder, ACL 2018): unfreeze one layer after another
- Sequential unfreezing (Chronopoulou et al., NAACL 2019): hyper-parameters that determine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

Image credit: Chanaky

Hands-on: Adaptation

Gradual unfreezing is similar to our previous freezing process.
We start by freezing all the model except the newly added parameters:

```
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 will start by training 1.199579e+07 parameters out of 5.039883e+07, i.e. 23.80%
```

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

```
import re
# We will unfreeze blocks regularly along the training: one block every `unfreezing_interval` step
unfreezing_interval = int(len(train_loader) * adapt_args.n_epochs / (args.num_layers + 1))

@trainer.on(Events.ITERATION_COMPLETED)
def unfreeze_layer_if_needed(engine):
    if engine.state.iteration % unfreezing_interval == 0:
        # Which layer should we unfreeze now
        unfreezing_index = args.num_layers - (engine.state.iteration // unfreezing_interval)

        # Let's unfreeze it
        unfreezed = []
        for name, param in adaptation_model.named_parameters():
            if re.match(r"transformer\.[^\.]*\." + str(unfreezing_index) + r"\.", name):
                unfreezed.append(name)
                param.require_grad = True
        print(f"Unfreezing block {unfreezing_index} with {unfreezed}")
```

Hands-on: Adaptation

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

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

Epoch [1/3]  [307/307] 100%| , loss=7.56e-02 [00:57<00:00]
Unfreezing block 15 with ['transformer.attentions.15.in_proj_weight', 'transformer.attentions.15.in_proj_bias
Unfreezing block 14 with ['transformer.attentions.14.in_proj_weight', 'transformer.attentions.14.in_proj_bias
Unfreezing block 13 with ['transformer.attentions.13.in_proj_weight', 'transformer.attentions.13.in_proj_bias
Unfreezing block 12 with ['transformer.attentions.12.in_proj_weight', 'transformer.attentions.12.in_proj_bias
Unfreezing block 11 with ['transformer.attentions.11.in_proj_weight', 'transformer.attentions.11.in_proj_bias
Validation Epoch: 1 Error rate: 7.706422018348624
Epoch [2/3]  [307/307] 100%| , loss=2.27e-02 [00:59<00:00]
Unfreezing block 10 with ['transformer.attentions.10.in_proj_weight', 'transformer.attentions.10.in_proj_bias
Unfreezing block 9 with ['transformer.attentions.9.in_proj_weight', 'transformer.attentions.9.in_proj_bias',
Unfreezing block 8 with ['transformer.attentions.8.in_proj_weight', 'transformer.attentions.8.in_proj_bias',
Unfreezing block 7 with ['transformer.attentions.7.in_proj_weight', 'transformer.attentions.7.in_proj_bias',
Unfreezing block 6 with ['transformer.attentions.6.in_proj_weight', 'transformer.attentions.6.in_proj_bias',
Unfreezing block 5 with ['transformer.attentions.5.in_proj_weight', 'transformer.attentions.5.in_proj_bias',
Validation Epoch: 2 Error rate: 6.788990825688068
Epoch [3/3]  [307/307] 100%| , loss=5.05e-03 [00:56<00:00]
Unfreezing block 4 with ['transformer.attentions.4.in_proj_weight', 'transformer.attentions.4.in_proj_bias',
Unfreezing block 3 with ['transformer.attentions.3.in_proj_weight', 'transformer.attentions.3.in_proj_bias',
Unfreezing block 2 with ['transformer.attentions.2.in_proj_weight', 'transformer.attentions.2.in_proj_bias',
Unfreezing block 1 with ['transformer.attentions.1.in_proj_weight', 'transformer.attentions.1.in_proj_bias',
Unfreezing block 0 with ['transformer.attentions.0.in_proj_weight', 'transformer.attentions.0.in_proj_bias',
Unfreezing block -1 with []
Validation Epoch: 3 Error rate: 7.339449541284404
<ignite.engine.engine.State at 0x7ff4c61999e8>

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

Test Results - Error rate: 5.200
```

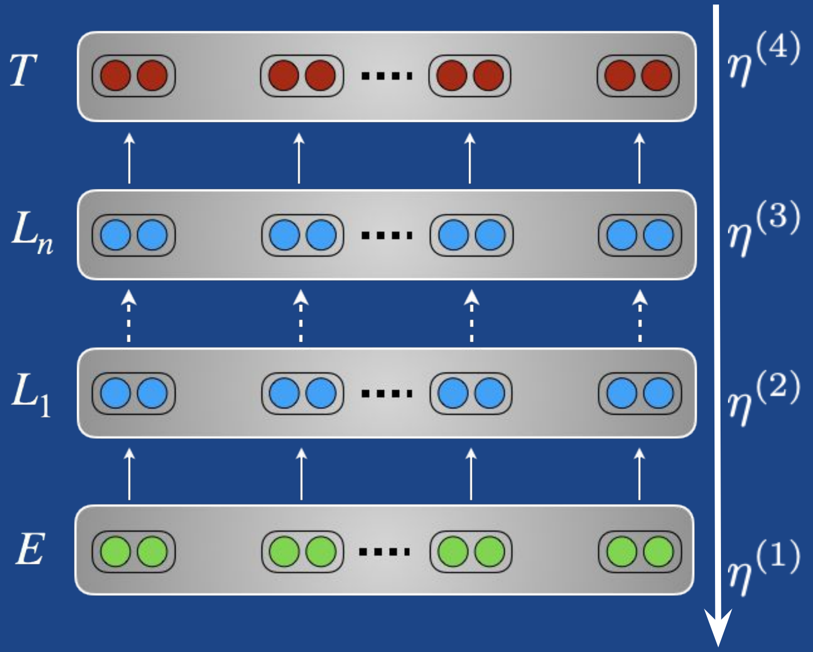
We show simple experiments in the Colab. Better hyper-parameters settings can probably be found.

4.2.B – Optimization: Learning rates

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

Where and when?

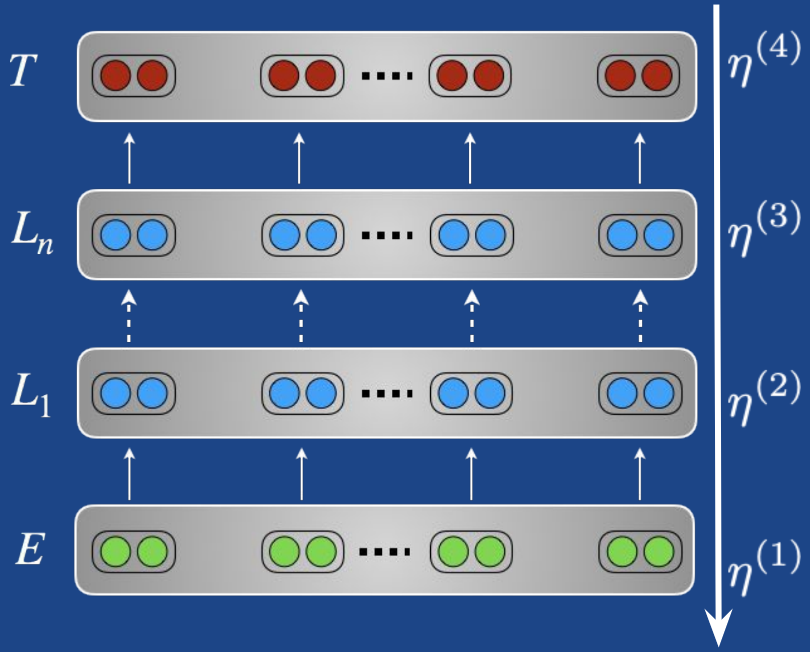
- ❑ **Lower layers** (capture general information)
- ❑ **Early** in training (model still needs to adapt to target distribution)
- ❑ **Late** in training (model is close to convergence)



4.2.B – Optimization: Learning rates

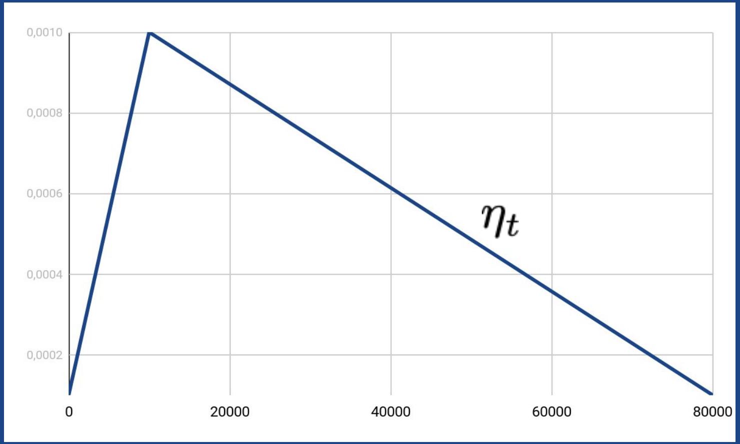
- Discriminative fine-tuning (Howard & Ruder, ACL 2018)
  - Lower layers capture general information
    → Use lower learning rates for lower layers

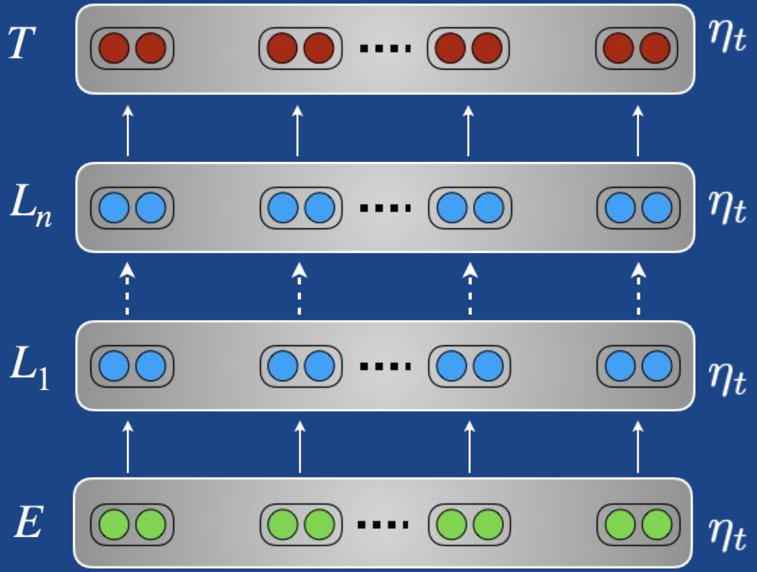
$$\eta^{(i)} = \eta \times d_f^{-i}$$



4.2.B – Optimization: Learning rates

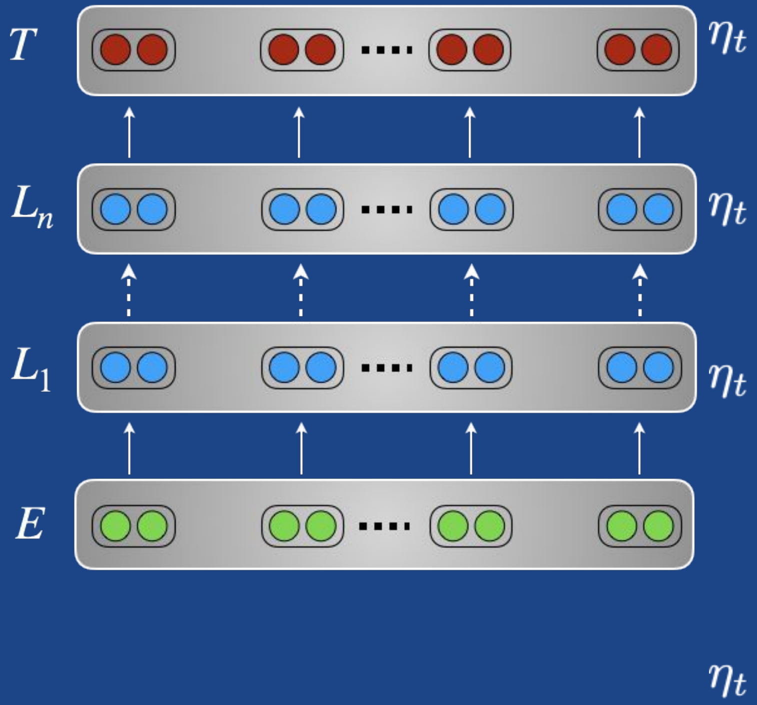
- Discriminative fine-tuning
- Triangular learning rates (Howard & Ruder, ACL 2018)
  - Quickly move to a suitable region, then slowly converge over time





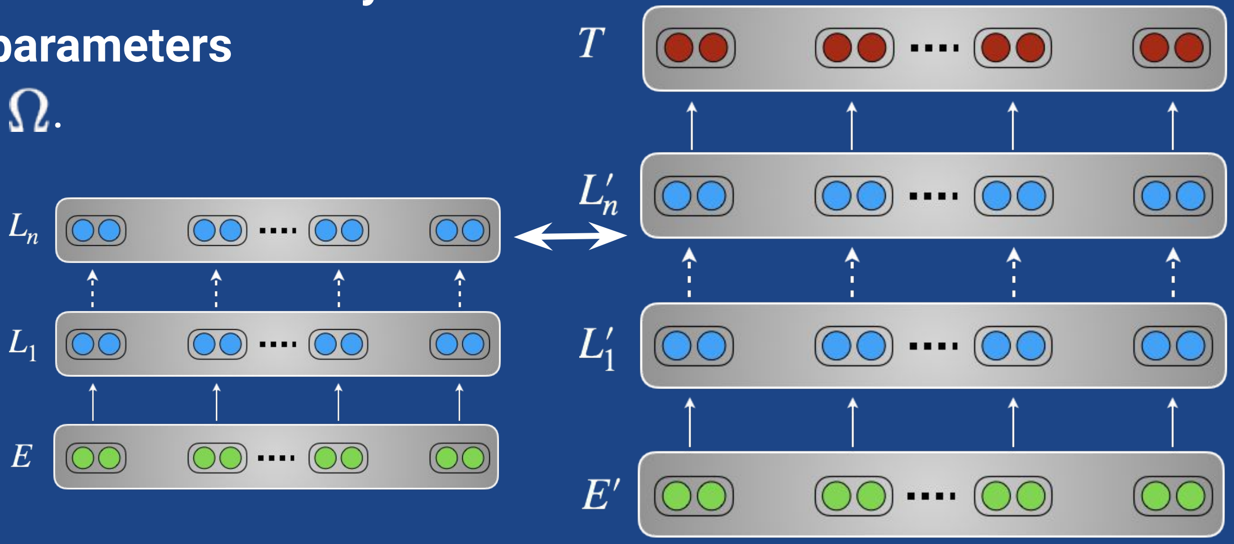
4.2.B – Optimization: Learning rates

- Discriminative fine-tuning
- Triangular learning rates (Howard & Ruder, ACL 2018)
  - 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) and Transformer-based methods (BERT, GPT)
  - Facilitates optimization; easier to escape suboptimal local minima



4.2.B – Optimization: Regularization

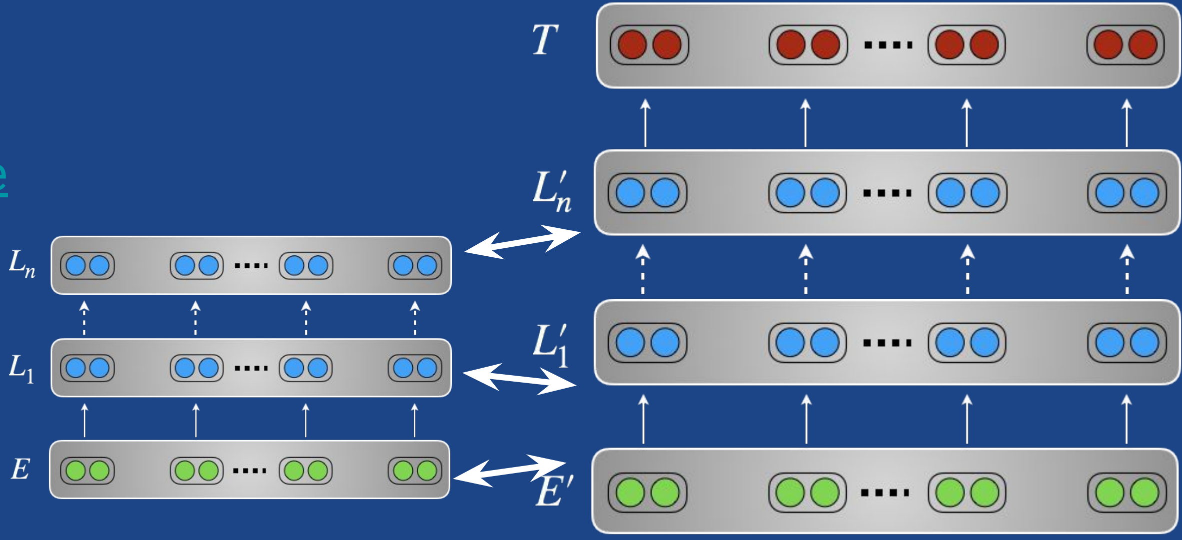
Main idea: minimize catastrophic forgetting by encouraging target model parameters to **stay close to pretrained model parameters** using a regularization term Ω.



4.2.B – Optimization: Regularization

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

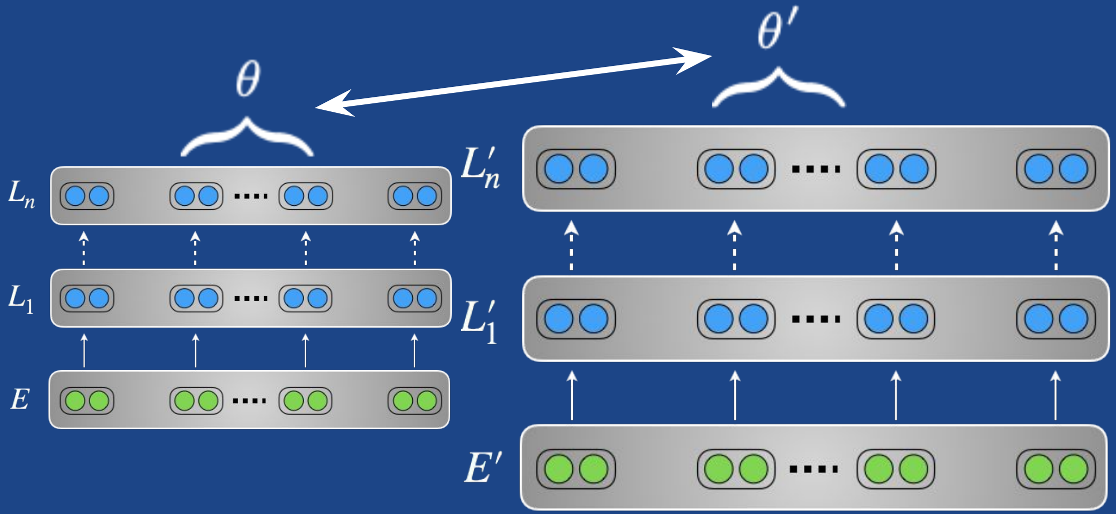
$$\Omega = \Sigma_1 ||L_i - L'_i||_2$$



4.2.B – Optimization: Regularization

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

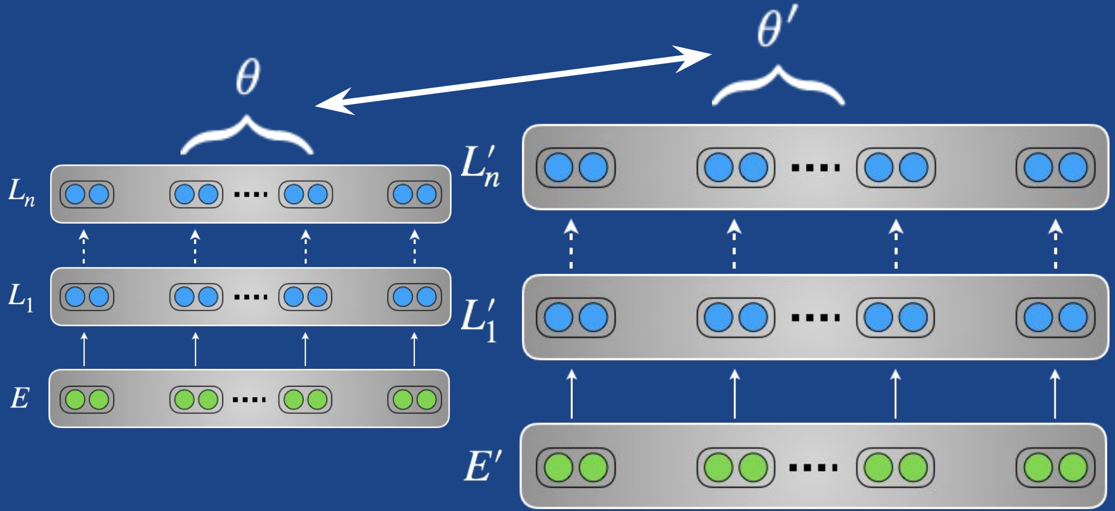
$$\Omega = \sum_i \frac{\lambda}{2} F_i(\theta'_i - \theta_i)^2$$



4.2.B – Optimization: Regularization

EWC has downsides in continual learning:

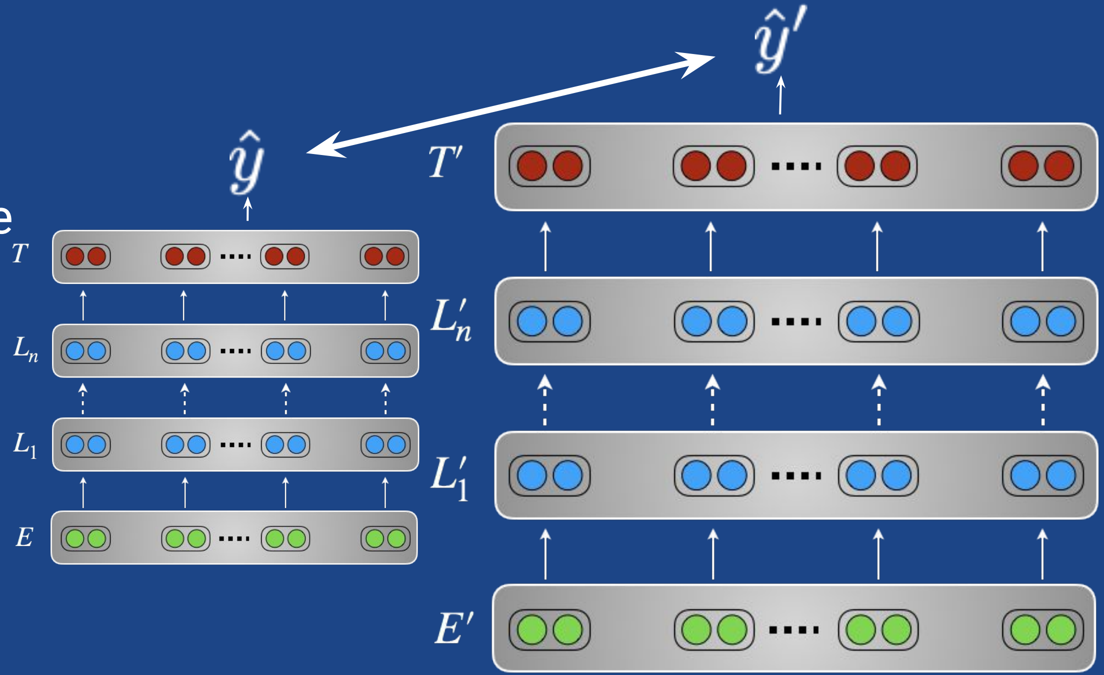
- May over-constrain parameters
- Computational cost is linear in the number of tasks ([Schwarz et al., ICML 2018](#))



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:

$$\Omega = \mathcal{H}(\hat{y}, \hat{y}')$$

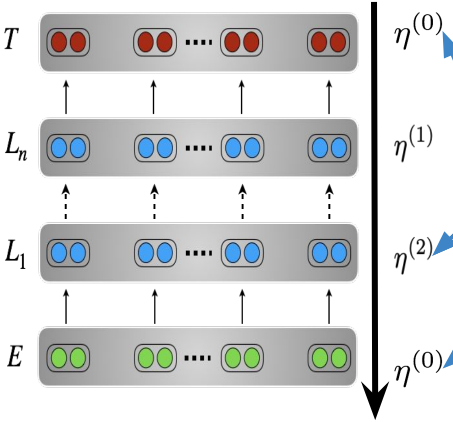


Hands-on #5:
Using discriminative learning

Image credit: Chanaky

Hands-on: Model adaptation

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\.[^\.]*\." + 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\.[^\.]*\.\d*\."
group = {'name': '0',
         'params': [p 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'])\
    == 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

```python
@trainer.on(Events.ITERATION_STARTED)
def update_layer_learning_rates(engine):
    for param_group in optimizer.param_groups:
        layer_index = int(param_group["name"])
        param_group["lr"] = param_group["lr"] / (adapt_args.decreasing_factor ** layer_index)
```

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**

Image credit: Alfredo

4.2.C – Optimization trade-offs: Space

Task-specific modifications

Feature extraction **Adapters** **Fine-tuning**

Many ↔ Few

Additional parameters

Feature extraction **Adapters** **Fine-tuning**

Many ↔ Few

Parameter reuse

Feature extraction **Adapters** **Fine-tuning**

All ↔ None

4.2.C – Optimization trade-offs: Time

Training time

Feature extraction **Adapters** **Fine-tuning**

Slow ⟷ Fast

4.2.C – Optimization trade-offs: Performance

- Rule of thumb: If task source and target tasks are **dissimilar***, 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

Image credit: Naveen

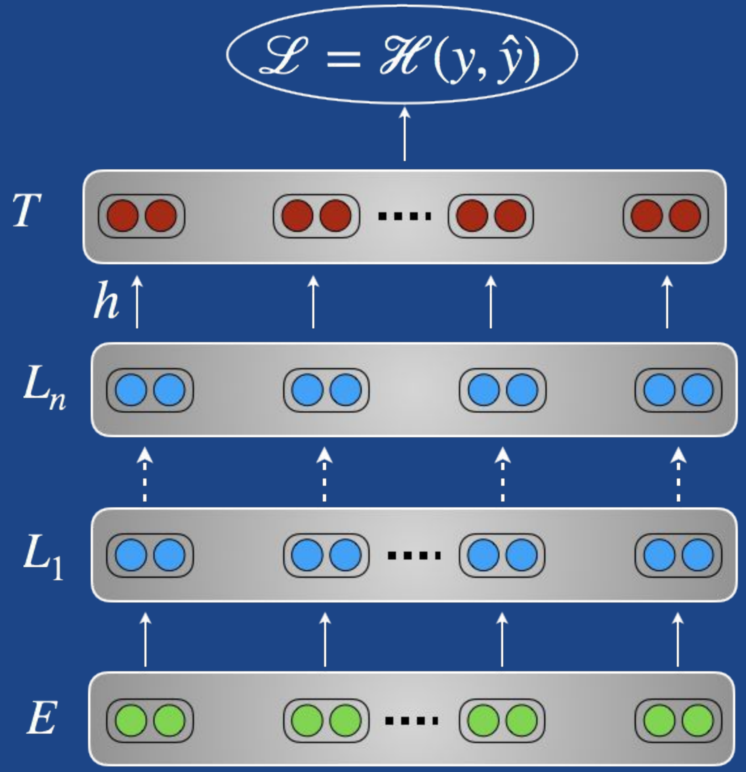
4.3.A – Getting more signal: Basic fine-tuning

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



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

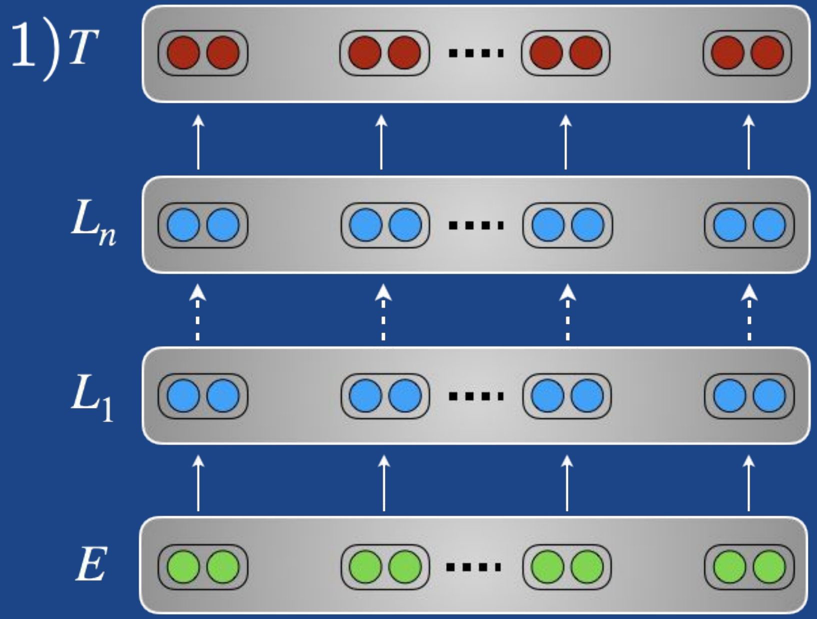
C. Dataset Slicing
When the model consistently underperforms on particular slices of the data

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

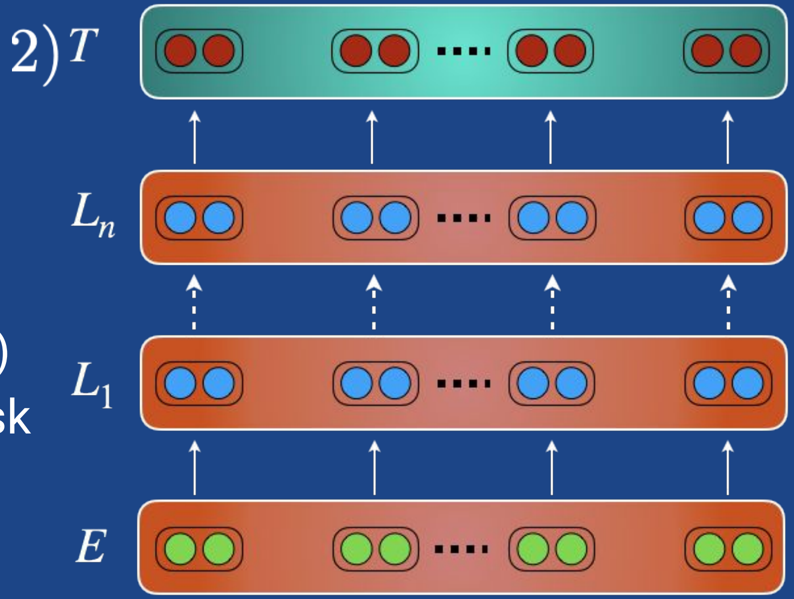


4.3.B – Getting more signal: Sequential adaptation

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)



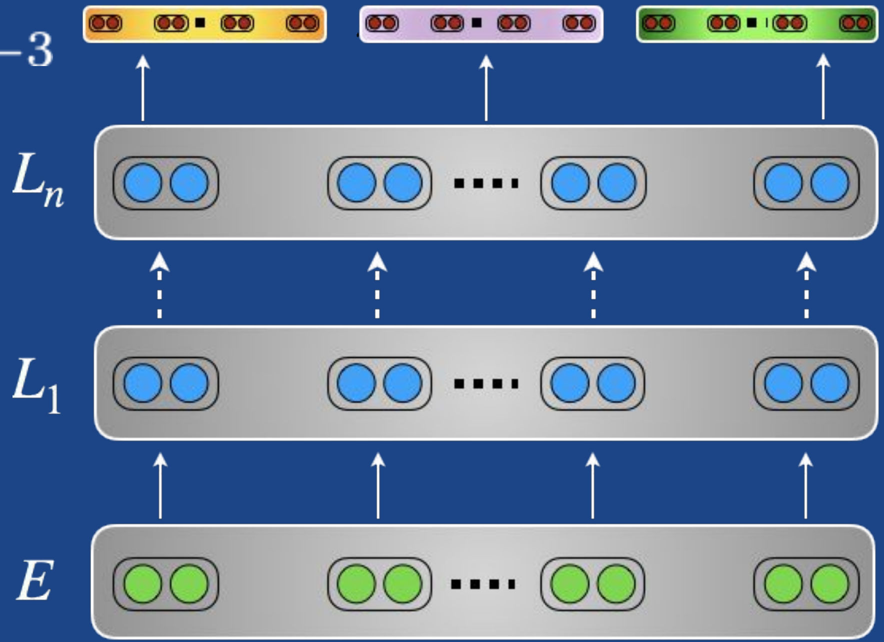
4.3.B – Getting more signal: Multi-task fine-tuning

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.

$$\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3$$

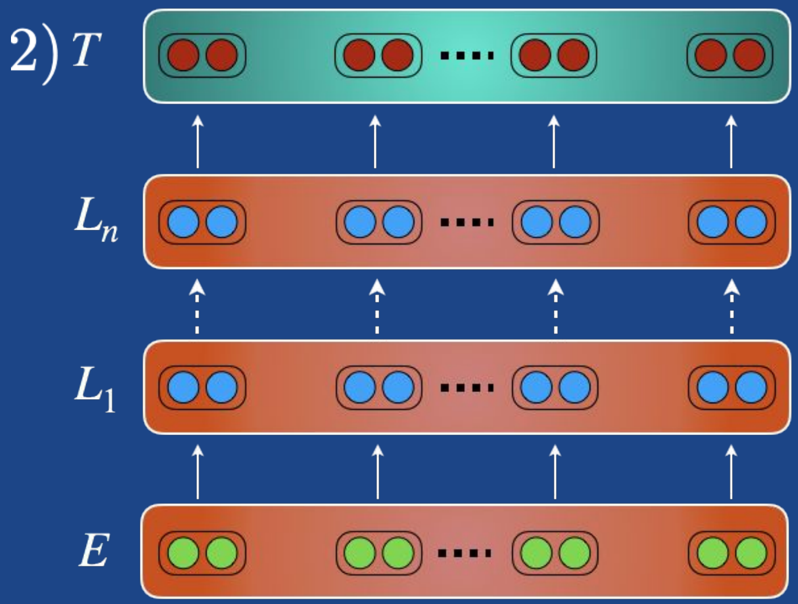
1) T_{1-3}



4.3.B – Getting more signal: Multi-task fine-tuning

Fine-tune the model jointly on related tasks

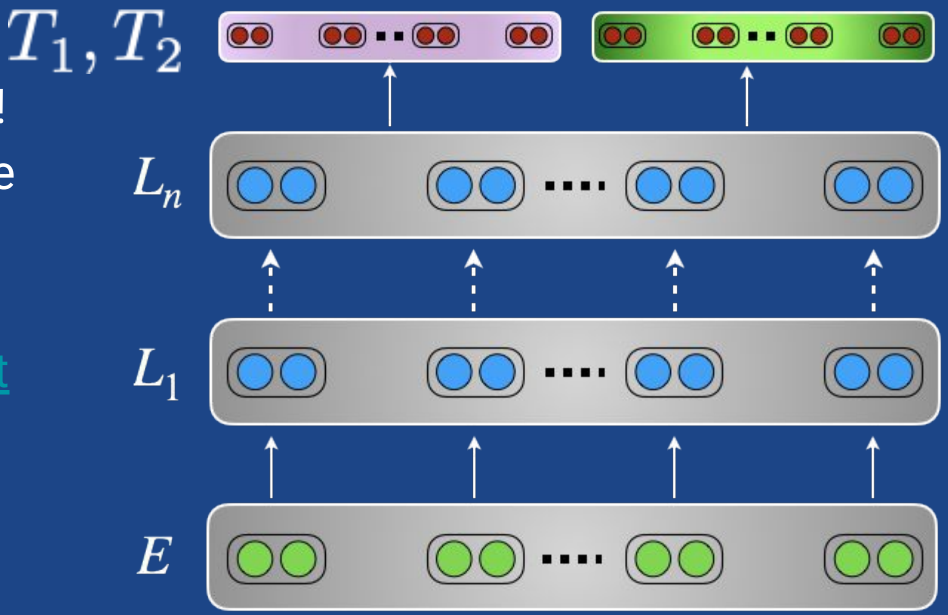
- ❑ For each optimization step, sample a task and a batch for training.
- ❑ Train via multi-task learning for a couple of epochs.
- ❑ Fine-tune on the target task only for a few epochs at the end.



4.3.B – Getting more signal: Multi-task fine-tuning

Fine-tune the model with an unsupervised auxiliary task

$$\mathcal{L} = \mathcal{L}_1 + \lambda \mathcal{L}_2$$



- 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 λ (Chronopoulou et al., NAACL 2019)
- Used as a separate step in ULMFiT

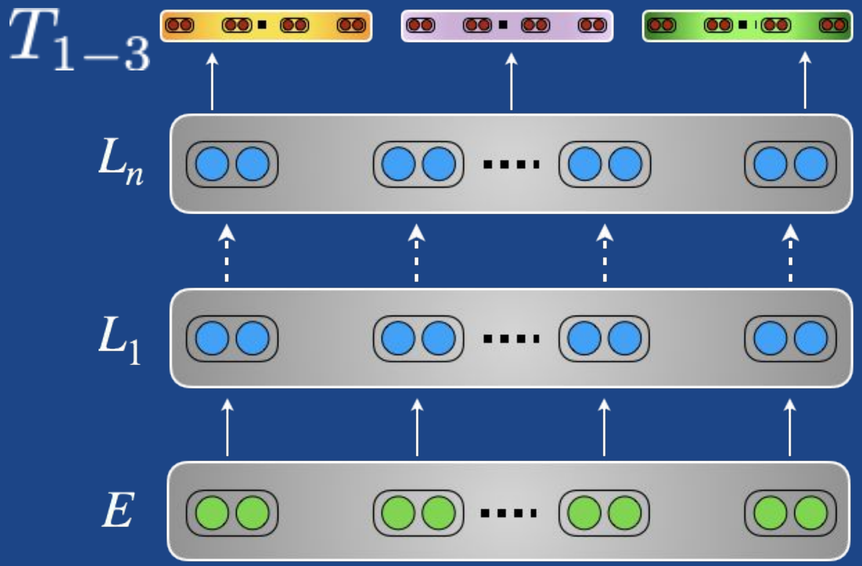
4.3.B – Getting more signal: Dataset slicing

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]

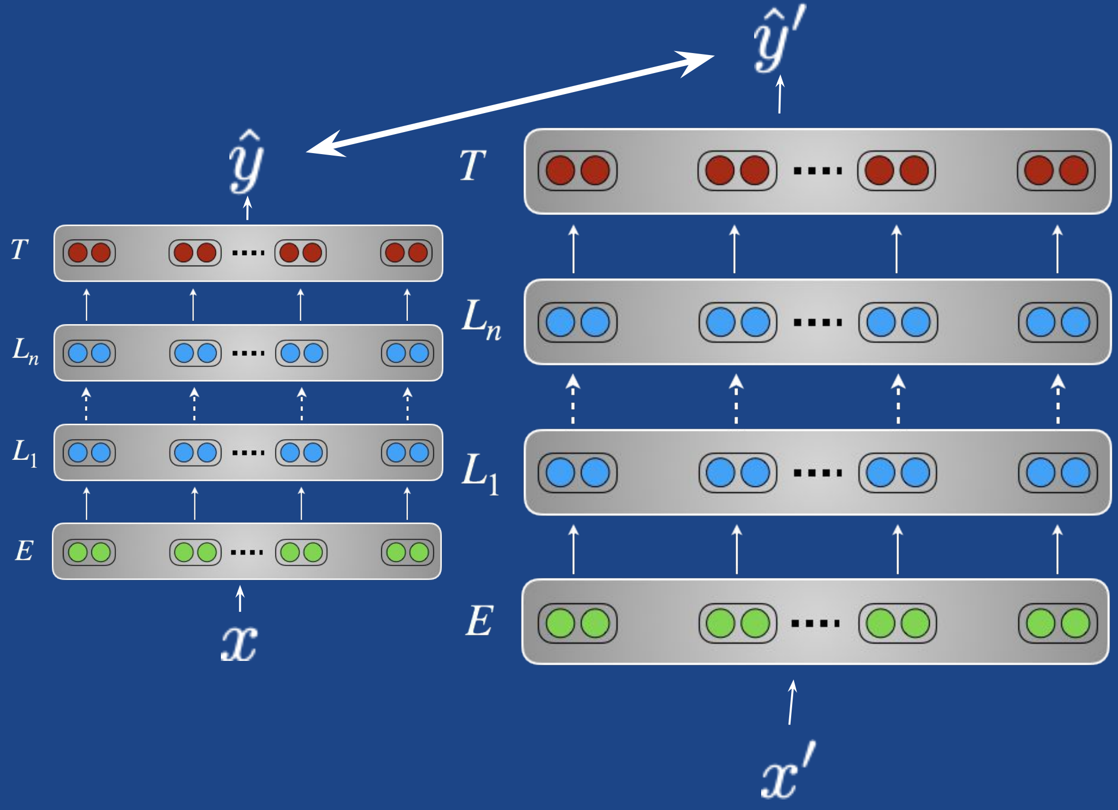
$$\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3$$



4.3.B – Getting more signal: Semi-supervised learning

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

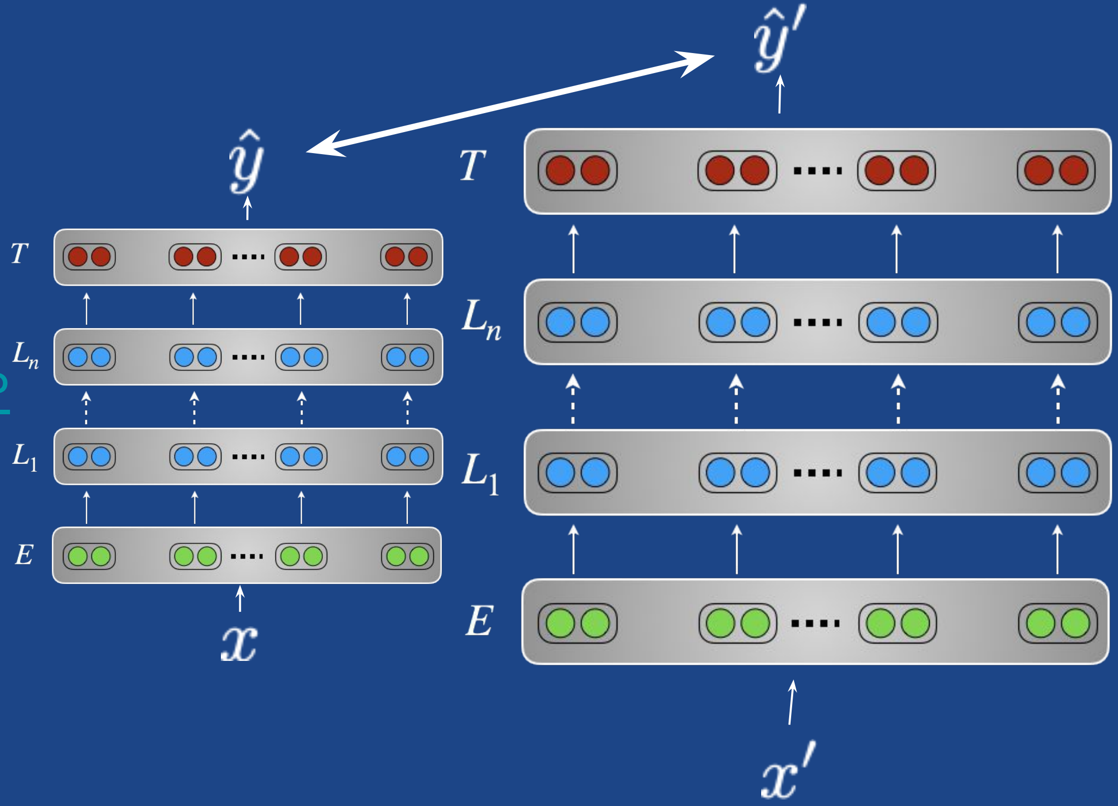
- ❑ Main idea: Minimize distance between predictions on original input x and perturbed input x'



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

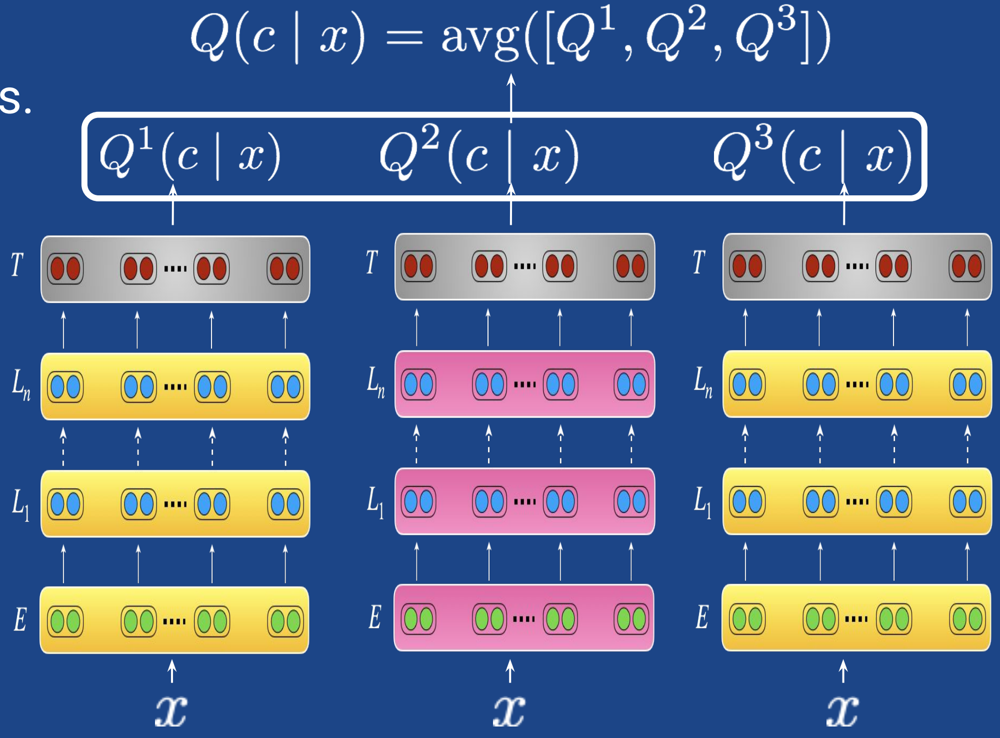
- **Knowledge distillation**
  Distill an ensemble of fine-tuned models in a single smaller model

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)



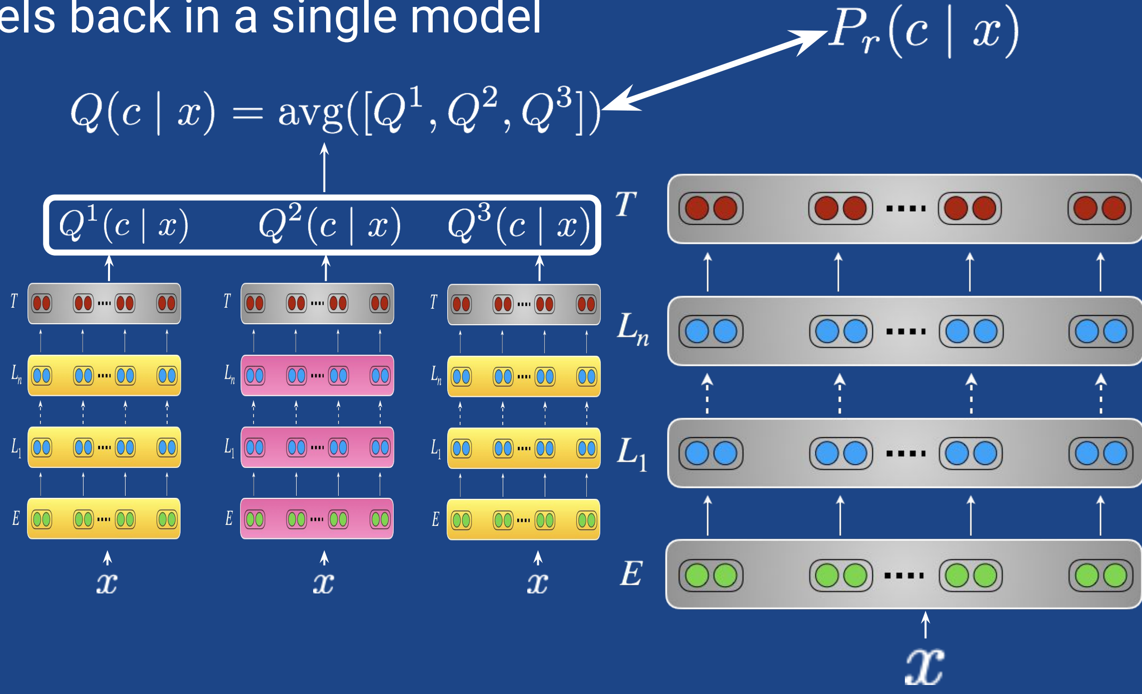
4.3.C – Getting more signal: Distilling

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



Hands-on #6:
Using multi-task learning

Image credit: Chanaky

Hands-on: 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

```python
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 [seq length, batch]

    _, losses = adaptation_model(inputs,
                                 clf_tokens_mask=(inputs == tokenizer.vocab['[CLS]']),
                                 clf_labels=labels,
                                 lm_labels=inputs,
                                 padding_mask=(batch == tokenizer.vocab['[PAD]']))

    clf_loss, lm_loss = losses
    loss = (adapt_args.clf_loss_coef * clf_loss
          + adapt_args.lm_loss_coef  * lm_loss) / adapt_args.gradient_accumulation_steps

    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()
```

```python
class TransformerWithClfHeadAndLMHead(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,
                                       config.dropout, causal=not config.mlm)

        self.lm_head = nn.Linear(config.embed_dim, config.num_embeddings, bias=False)
        self.classification_head = nn.Linear(config.embed_dim, fine_tuning_config.num_classes)

        self.apply(self.init_weights)
        self.tie_weights()

    def tie_weights(self):
        self.lm_head.weight = self.transformer.tokens_embeddings.weight

    def init_weights(self, module):
        if isinstance(module, (nn.Linear, nn.Embedding, nn.LayerNorm)):
            module.weight.data.normal_(mean=0.0, std=self.config.initializer_range)
        if isinstance(module, (nn.Linear, nn.LayerNorm)) and module.bias is not None:
            module.bias.data.zero_()

    def forward(self, x, clf_tokens_mask, lm_labels=None, clf_labels=None, padding_mask=None):
        """ x and clf_tokens_mask have shape [seq length, batch] padding_mask has shape [batch, seq length] """
        hidden_states = self.transformer(x, padding_mask)

        lm_logits = self.lm_head(hidden_states)
        clf_tokens_states = (hidden_states * clf_tokens_mask.unsqueeze(-1).float()).sum(dim=0)
        clf_logits = self.classification_head(clf_tokens_states)

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

        if lm_labels is not None:
            shift_logits = lm_logits[:-1] if self.transformer.causal else lm_logits
            shift_labels = lm_labels[1:] if self.transformer.causal else lm_labels
            loss_fct = nn.CrossEntropyLoss(ignore_index=-1)
            loss.append(loss_fct(shift_logits.view(-1, shift_logits.size(-1)), shift_labels.view(-1)))

        if len(loss):
            return (lm_logits, clf_logits), loss

        return lm_logits, clf_logits
```

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).

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

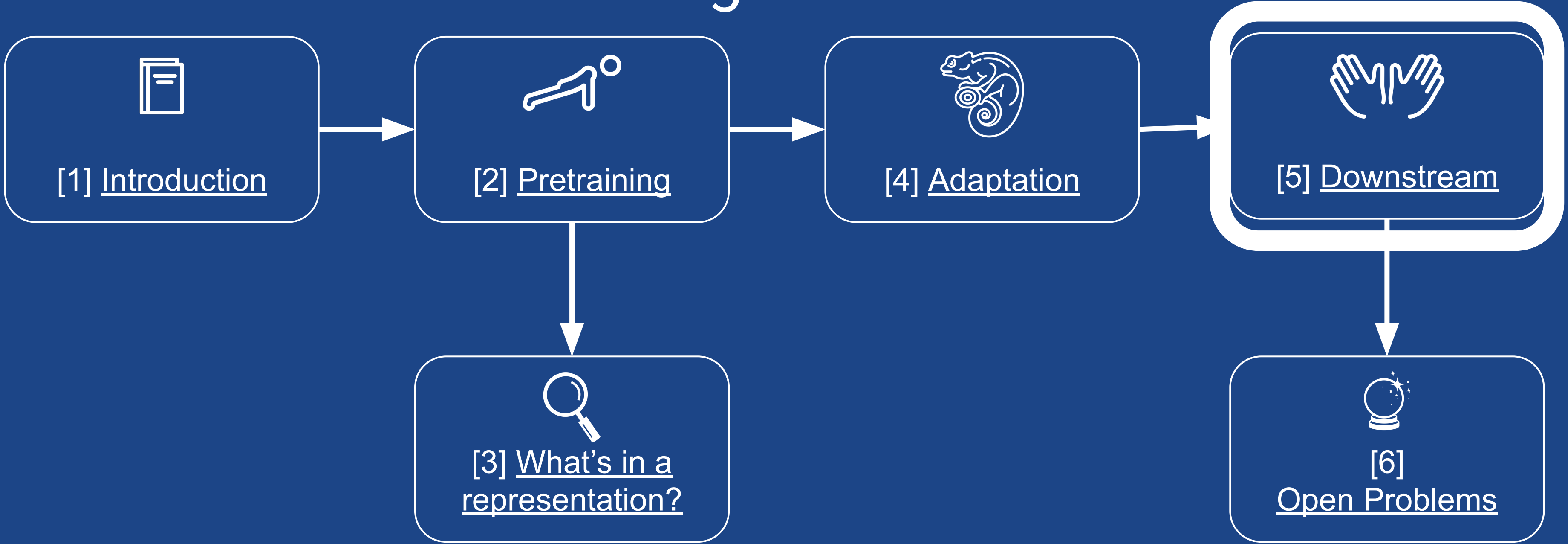
```
Epoch [1/6]     [307/307] 100%| , loss=1.07e+00 [01:21<00:00]
Validation Epoch: 1 Error rate: 9.35779816513761
Epoch [2/6]     [307/307] 100%| , loss=7.08e-01 [01:21<00:00]
Validation Epoch: 2 Error rate: 7.522935779816509
Epoch [3/6]     [307/307] 100%| , loss=5.46e-01 [01:22<00:00]
Validation Epoch: 3 Error rate: 5.688073394495408
Epoch [4/6]     [307/307] 100%| , loss=4.66e-01 [01:21<00:00]
Validation Epoch: 4 Error rate: 5.321100917431187
Epoch [5/6]     [307/307] 100%| , loss=4.22e-01 [01:21<00:00]
Validation Epoch: 5 Error rate: 5.688073394495408
Epoch [6/6]     [307/307] 100%| , loss=3.98e-01 [01:21<00:00]
Validation Epoch: 6 Error rate: 5.321100917431187
<ignite.engine.engine.State at 0x7ff4c9357e80>
```

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

```
Test Results - Error rate: 3.400
```

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

Agenda


[1] Introduction
[2] Pretraining
[4] Adaptation
[5] Downstream
[3] What's in a representation?
[6] Open Problems

5. Downstream applications Hands-on examples

Image credit: Fahmi

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

Consumption	**CO_2e (lbs)**
Air travel, 1 passenger, NY↔SF	1984
Human life, avg, 1 year	11,023
American life, avg, 1 year	36,156
Car, avg incl. fuel, 1 lifetime	126,000
Training one model	
SOTA NLP model (tagging)	13
w/ tuning & experimentation	33,486
Transformer (large)	121
w/ neural architecture search	394,863

"Energy and Policy Considerations for Deep Learning in NLP" - Strubell, Ganesh, McCallum - ACL 2019

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)

Icons credits: David, Susannanova, Flatart, ProSymbols

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.
  http://ai.stanford.edu/~amaas/data/sentiment/
- 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

5.A – Document level classification using fast.ai

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:

```
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.

```
path = untar_data(URLs.IMDB_SAMPLE)
print("Path:", path)
df = pd.read_csv(path/'texts.csv')
df.head()
```

Path: /root/.fastai/data/imdb_sample

	label	text	is_valid
0	negative	Un-bleeping-believable! Meg Ryan doesn't even ...	False
1	positive	This is a extremely well-made film. The acting...	False
2	negative	Every once in a long while a movie will come a...	False
3	positive	Name just says it all. I watched this movie wi...	False
4	negative	This movie succeeds at being one of the most u...	False

DataBunch for the language model and the classifier

```
data_lm = TextLMDataBunch.from_csv(path, 'texts.csv')
data_clas = TextClasDataBunch.from_csv(path, 'texts.csv',
                    vocab=data_lm.train_ds.vocab, bs=42)
```

Load an AWD-LSTM (Merity et al., 2017) pretrained on WikiText-103 & fine-tune it on IMDB using the language modeling loss.

```
moms = (0.8,0.7)
learn = language_model_learner(data_lm, AWD_LSTM)
learn.unfreeze()
learn.fit_one_cycle(4, slice(1e-2), moms=moms)
learn.save_encoder('enc')
```

epoch	train_loss	valid_loss	accuracy	time
0	4.723435	3.968737	0.283498	00:15
1	4.416326	3.874095	0.286878	00:15
2	4.148463	3.836543	0.290434	00:16
3	3.951989	3.828021	0.291311	00:16

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 *nn.BatchNorm1d* ⇨ *nn.Dropout* ⇨ *nn.Linear* ⇨ *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/NAACLTransferFastAiColab

```
learn = text_classifier_learner(data_clas, AWD_LSTM)
learn.load_encoder('enc')
learn.fit_one_cycle(4, moms=moms)
```

epoch	train_loss	valid_loss	accuracy	time
0	0.663383	0.682115	0.572139	00:10
1	0.623683	0.609520	0.651741	00:10
2	0.597989	0.582999	0.666667	00:10
3	0.580533	0.555404	0.666667	00:09

```
learn.unfreeze()
learn.fit_one_cycle(8, slice(1e-5,1e-3), moms=moms)
```

epoch	train_loss	valid_loss	accuracy	time
0	0.555569	0.557091	0.681592	00:20
1	0.566048	0.541689	0.721393	00:21
2	0.554564	0.543157	0.736318	00:20
3	0.556879	0.526971	0.756219	00:20
4	0.552898	0.522964	0.751244	00:19
5	0.541698	0.514611	0.756219	00:19
6	0.535575	0.514330	0.756219	00:19
7	0.529567	0.515582	0.746269	00:19

5.B – Token level classification: BERT & Tensorflow

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

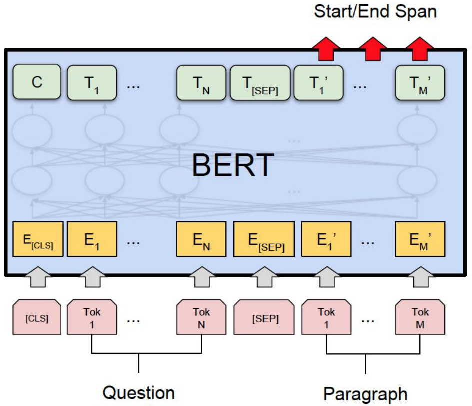
- ❑ Target task:
  SQuAD: a question answering dataset.
  https://rajpurkar.github.io/SQuAD-explorer/
- ❑ In this example we will directly use a Tensorflow checkpoint
  - ❑ Example: https://github.com/google-research/bert
  - ❑ We use the usual Tensorflow workflow: create model graph comprising the core model and the added/modified elements
  - ❑ Take care of variable assignments when loading the checkpoint

5.B – SQuAD with BERT & Tensorflow

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.




```
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 assignement_map from the variables of the checkpoint to the model variable, keeping only the variables in the model.

```
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[0], x[1])
    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)
```

And we can use *tf.train.init_from_ckeckpoint*

```
(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)
```

5.B – SQuAD with BERT & Tensorflow

TensorFlow-Hub

Working directly with TensorFlow requires to have access to–and include in your code– the *full code of the pretrained model.*

```
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()
```

TensorFlow Hub is a library for **sharing** machine learning models as *self-contained pieces of TensorFlow graph with their weights and assets.*

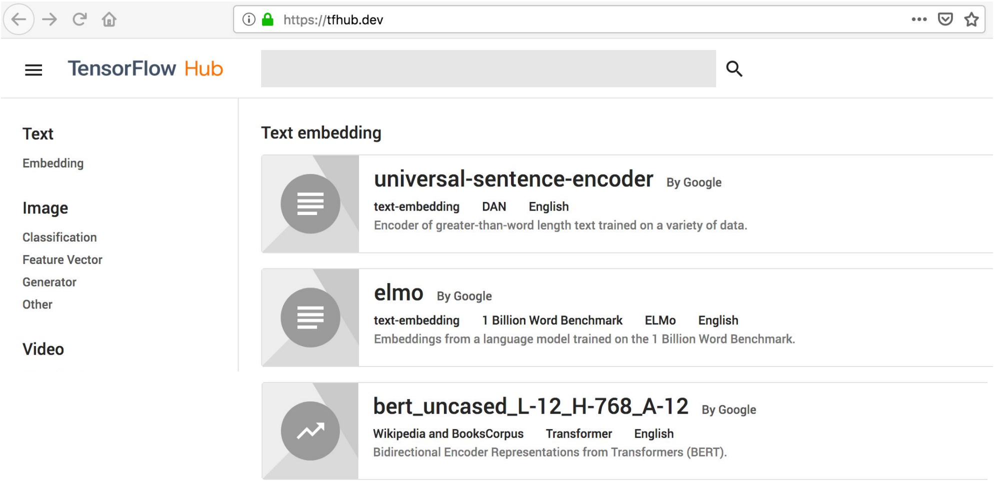
Modules are automatically downloaded and cached when instantiated.

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*.

```
!pip install "tensorflow_hub==0.4.0"

import tensorflow_hub as hub

def create_model(is_predicting, input_ids, input_mask, segment_        els,
                 num_labels):
  """Creates a classification model."""

  bert_module = hub.Module(
      BERT_MODEL_HUB,
      trainable=True)
  bert_inputs = dict(
      input_ids=input_ids,
      input_mask=input_mask,
      segment_ids=segment_ids)
  bert_outputs = bert_module(
      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"]
```

5.B – SQuAD with BERT & Tensorflow

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*)

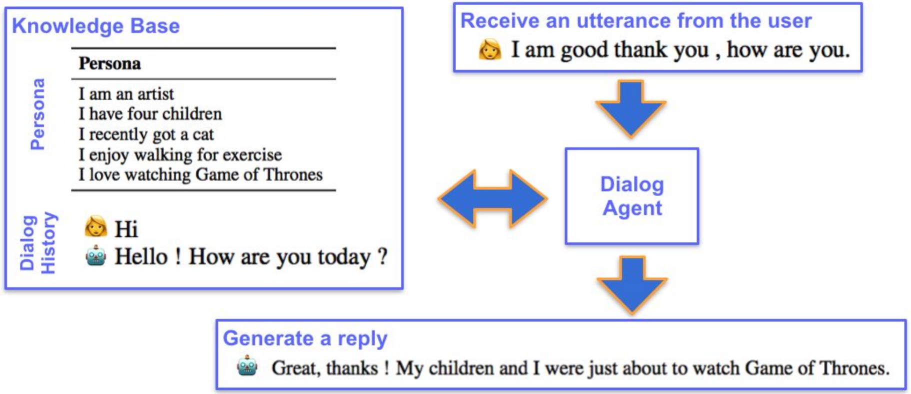
5.C – Language Generation: OpenAI GPT & PyTorch

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
- ❑ 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

5.C – Chit-chat with OpenAI GPT & PyTorch

A dialog generation task:



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

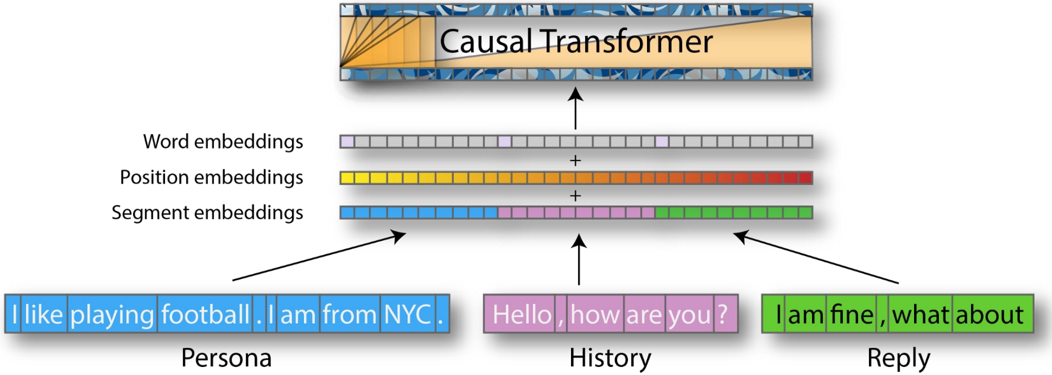
- ❑ Language modeling pre-training involves a single input: *a sequence of words*.
- ❑ In a dialog setting: several type of contexts are provided to generate an output sequence:
  - ❑ *knowledge base*: persona sentences,
  - ❑ *history of the dialog*: at least the last utterance from the user,
  - ❑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

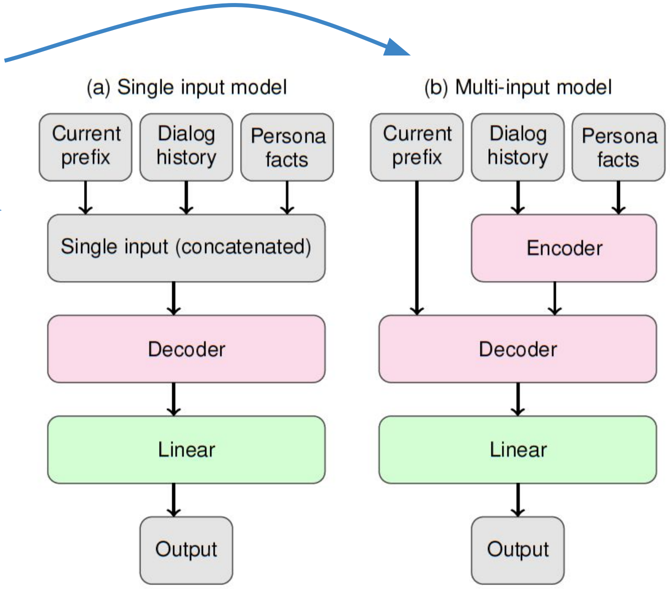
How should we adapt the model?

5.C – Chit-chat with OpenAI GPT & PyTorch

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

Golovanov, Kurbanov, Nikolenko, Truskovskyi, Tselousov and Wolf, ACL 2019

5.C – Chit-chat with OpenAI GPT & PyTorch

Let's import pretrained versions of OpenAI GPT tokenizer and model.

```
from pytorch_pretrained_bert import OpenAIGPTLMHeadModel, OpenAIGPTTokenizer

model = OpenAIGPTLMHeadModel.from_pretrained('openai-gpt')
tokenizer = OpenAIGPTTokenizer.from_pretrained('openai-gpt')
```

And add a few new tokens to the vocabulary

```
# We use 5 special tokens: <bos>, <eos>, <speaker1>, <speaker2>, <pad>
# to indicate start/end of the input sequence, tokens from user/bot and padding
SPECIAL_TOKENS = ["<bos>", "<eos>", "<speaker1>", "<speaker2>", "<pad>"]

# Add these special tokens to the vocabulary and the embeddings of the model:
tokenizer.set_special_tokens(SPECIAL_TOKENS)
model.set_num_special_tokens(len(SPECIAL_TOKENS))
```

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.

```
from itertools import chain

# Let's define our contexts and special tokens
persona_string = ["i like football", "i am from NYC"]
history_string = ["how are you ?", "pretty fine"]
reply_string = "great !"
bos, eos, speaker1, speaker2 = "<bos>", "<eos>", "<speaker1>", "<speaker2>"

persona = [tokenizer.tokenize(s) for s in persona_string]
history = [tokenizer.tokenize(s) for s in history_string]
reply = tokenizer.tokenize(reply_string)

def build_inputs(persona, history, reply):
    # Build our sequence by adding delimiters and concatenating
    sequence = [[bos] + list(chain(*persona))] + history + [reply + [eos]]
    sequence = [sequence[0]] + [ [speaker2 if (len(sequence)-i) % 2 else speaker1] + s
                                for i, s in enumerate(sequence[1:])]
    # Build our word, segments and position inputs from the sequence
    words = list(chain(*sequence))                          # word tokens
    segments = [speaker2 if i % 2 else speaker1             # segment tokens
                for i, s in enumerate(sequence) for _ in s]
    position = list(range(len(words)))                      # position tokens
    return words, segments, position, sequence

words, segments, position, sequence = build_inputs(persona, history, reply)

# Tokenize words and segments embeddings:
words = tokenizer.convert_tokens_to_ids(words)
segments = tokenizer.convert_tokens_to_ids(segments)
lm_targets = ([-1] * sum(len(s) for s in sequence[:-1])) \
             + [-1] + tokenizer.convert_tokens_to_ids(sequence[-1][1:])
```

PyTorch Hub

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

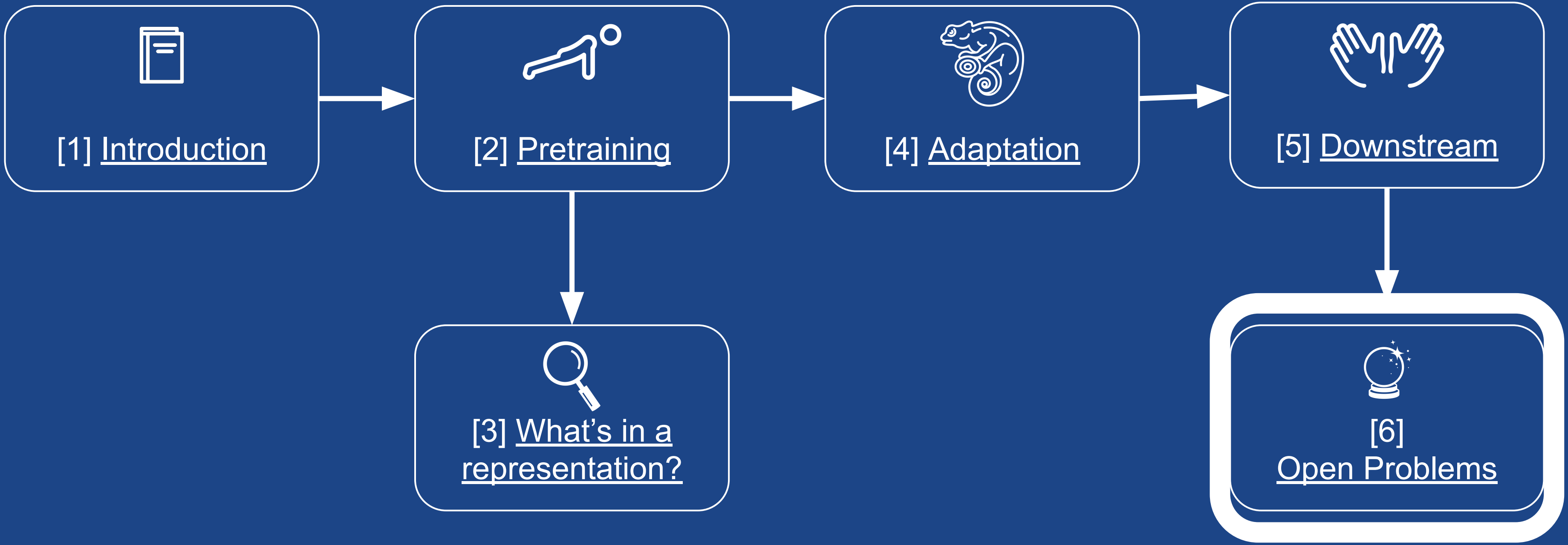
- ❏ PyTorch Hub is based on **GitHub repositories**
- ❏ A model is shared by adding a *hubconf.py* script to the root of a GitHub repository
- ❏ Both **model definitions** and **pre-trained weights** can be shared
- ❏ More details: https://pytorch.org/hub and https://pytorch.org/docs/stable/hub.html

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:

```
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
[3] What's in a representation?
[6] Open Problems

6. Open problems and future directions

Image credit: Yazmin Alanis

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

Image credit: Yazmin Alanis

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

Pretraining tasks

More diverse self-supervised objectives

- ❑ Taking inspiration from computer vision
- ❑ Self-supervision in language mostly based on word co-occurrence (Ando and Zhang, 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 (Doersch et al., ICCV 2015)

Image colorization (Zhang et al., ECCV 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

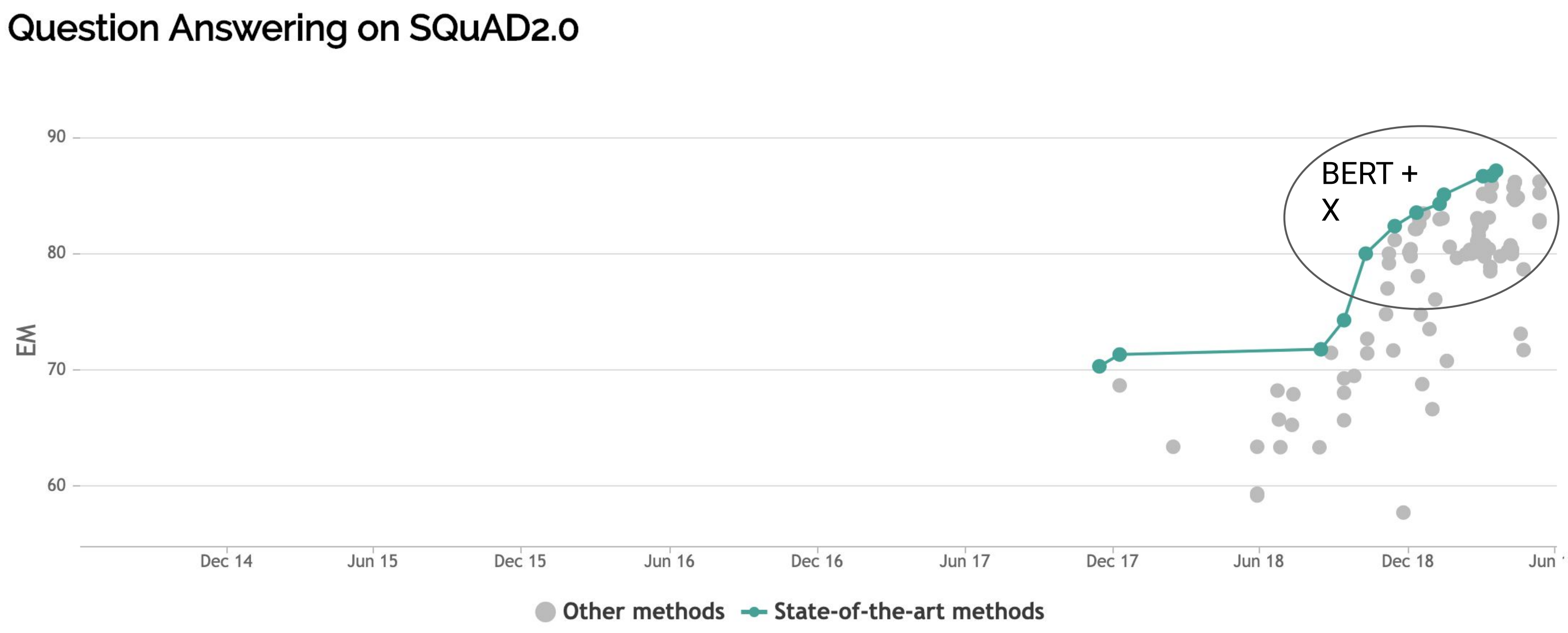
Questions?

- 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

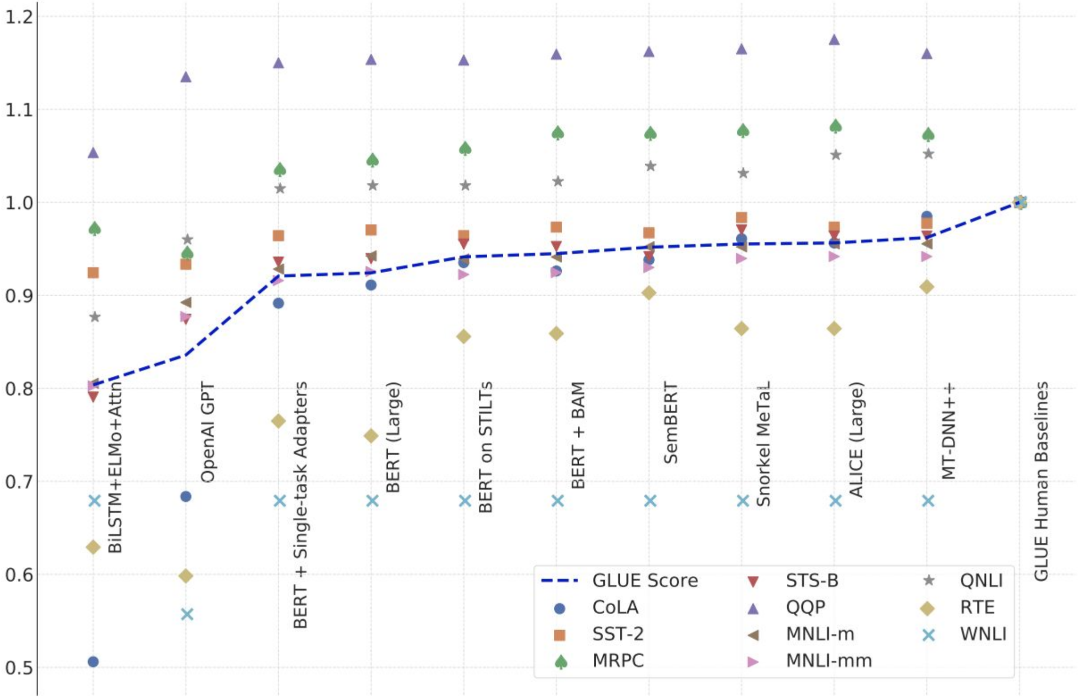
Extra slides

Why transfer learning in NLP? (Empirically)



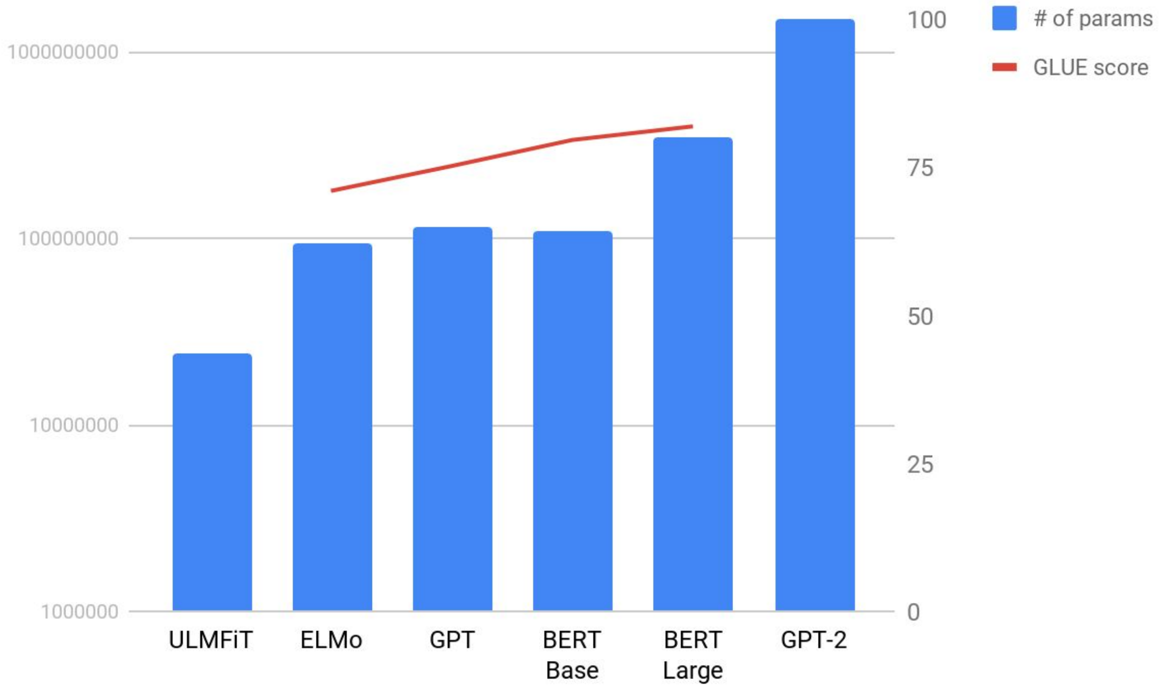
https://paperswithcode.com/sota/question-answering-on-squad20

GLUE* performance over time



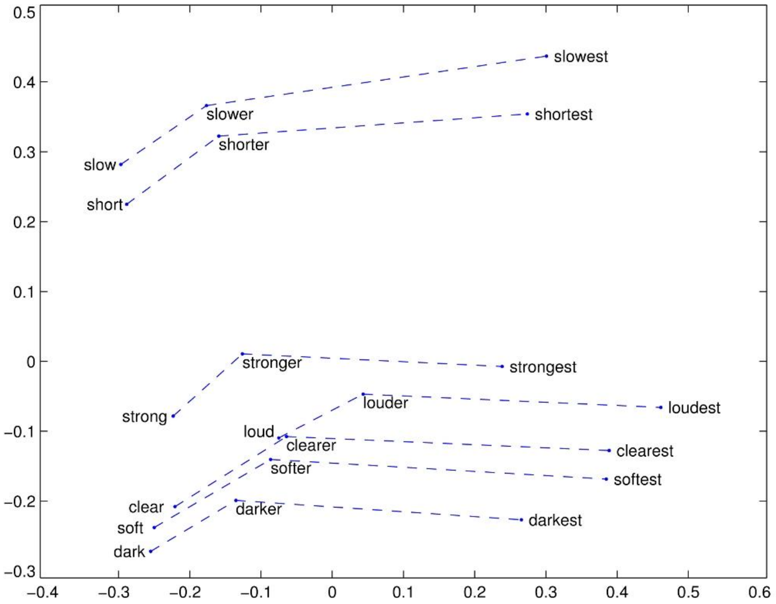
*General Language Understanding Evaluation (GLUE; [Wang et al., 2019](#)): includes 11 diverse NLP tasks

Pretrained Language Models: More Parameters



More word vectors

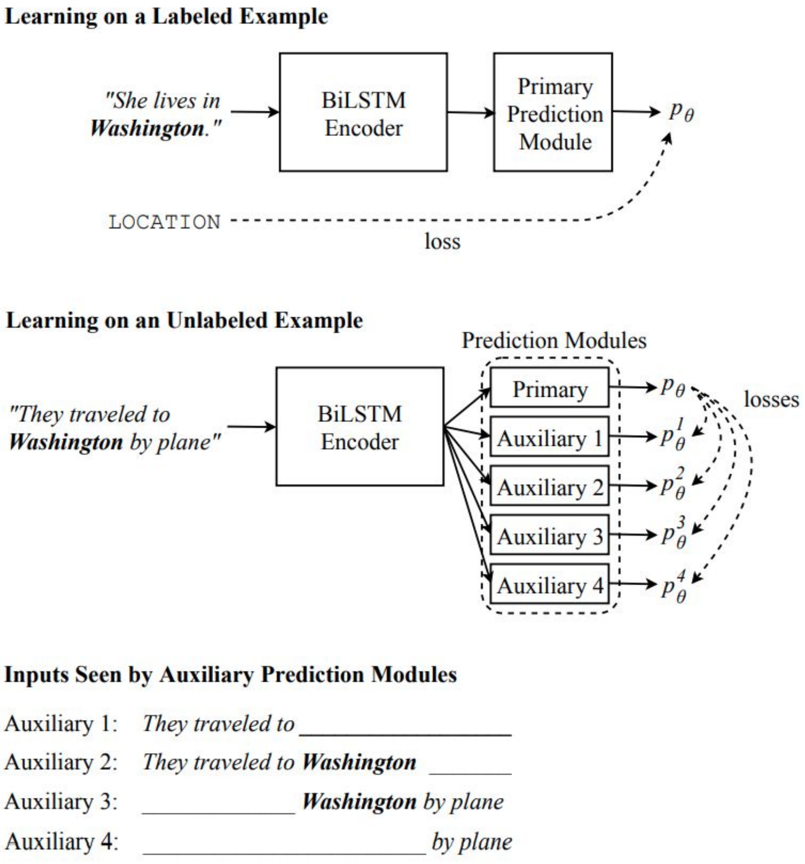
GLoVe: very large scale (840B tokens), co-occurrence based. Learns linear relationships (SOTA word analogy) (Pennington et al., 2014)



fastText: incorporates subword information (Bojanowski et al., 2017)

query	tiling	tech-rich	english-born	micromanaging	eateries	dendritic
fastText	tile flooring	tech-dominated tech-heavy	british-born polish-born	micromanage micromanaged	restaurants eaterie	dendrite dendrites
skipgram	bookcases built-ins	technology-heavy .ixic	most-capped ex-scotland	defang internalise	restaurants delis	epithelial p53

Semi-supervised Sequence Modeling with Cross-View Training



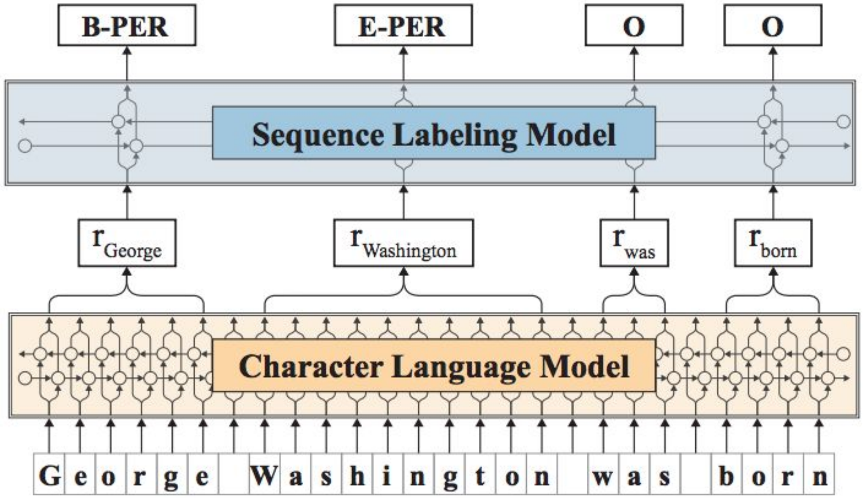
Method	CCG Acc.	Chunk F1	NER F1	FGN F1	POS Acc.	Dep. Parse UAS	Dep. Parse LAS	Translate BLEU
Shortcut LSTM (Wu et al., 2017)	95.1				97.53			
ID-CNN-CRF (Strubell et al., 2017)			90.7	86.8				
JMT† (Hashimoto et al., 2017)		95.8			97.55	94.7	92.9	
TagLM* (Peters et al., 2017)		96.4	91.9					
ELMo* (Peters et al., 2018)			92.2					
Biaffine (Dozat and Manning, 2017)						95.7	94.1	
Stack Pointer (Ma et al., 2018)						95.9	94.2	
Stanford (Luong and Manning, 2015)								23.3
Google (Luong et al., 2017)								26.1
Supervised	94.9	95.1	91.2	87.5	97.60	95.1	93.3	28.9
Virtual Adversarial Training*	95.1	95.1	91.8	87.9	97.64	95.4	93.7	–
Word Dropout*	95.2	95.8	92.1	88.1	97.66	95.6	93.8	29.3
ELMo (our implementation)*	95.8	96.5	92.2	88.5	97.72	96.2	94.4	29.3
ELMo + Multi-task*†	95.9	96.8	92.3	88.4	**97.79**	96.4	94.8	–
CVT*	95.7	96.6	92.3	88.7	97.70	95.9	94.1	**29.6**
CVT + Multi-task*†	96.0	96.9	92.4	88.4	97.76	96.4	94.8	–
CVT + Multi-task + Large*†	**96.1**	**97.0**	**92.6**	**88.8**	97.74	**96.6**	**95.0**	–

SOTA sequence modeling results

(Clark et al. EMNLP 2018)

Contextual String Embeddings

Pretrain bidirectional character level model, extract embeddings from first/last character

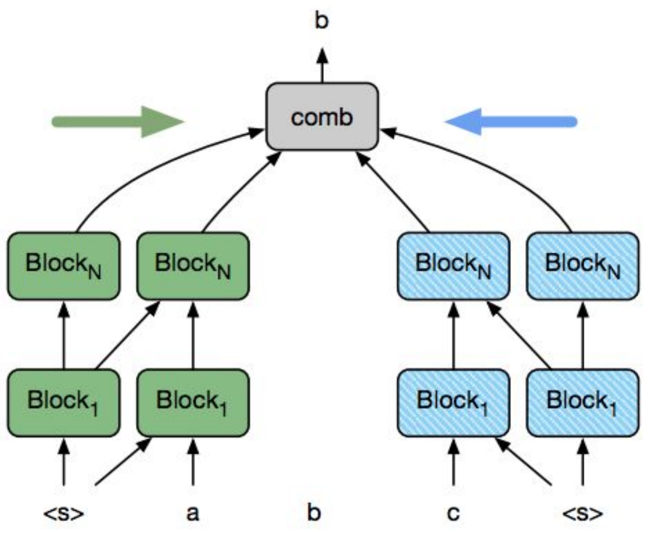


SOTA CoNLL 2003 NER results

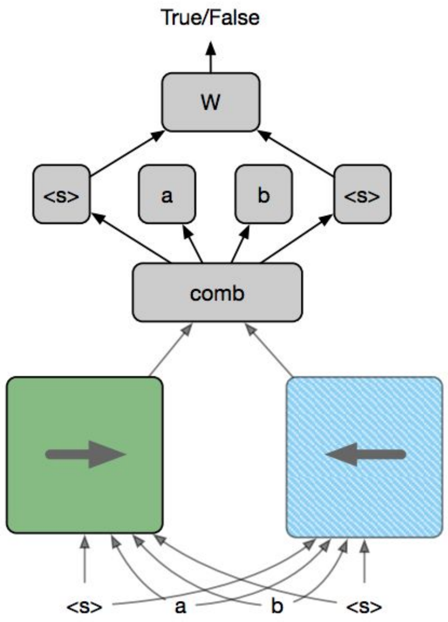
Task	PROPOSED	Previous best
NER English	**93.09**±0.12	*92.22*±0.1 (Peters et al., 2018)
NER German	**88.32**±0.2	*78.76* (Lample et al., 2016)
Chunking	**96.72**±0.05	*96.37*±0.05 (Peters et al., 2017)
PoS tagging	**97.85**±0.01	*97.64* (Choi, 2016)

(Akbik et al., COLING 2018) (see also Akbik et al., NAACL 2019)

Cloze-driven Pretraining of Self-attention Networks



Pretraining

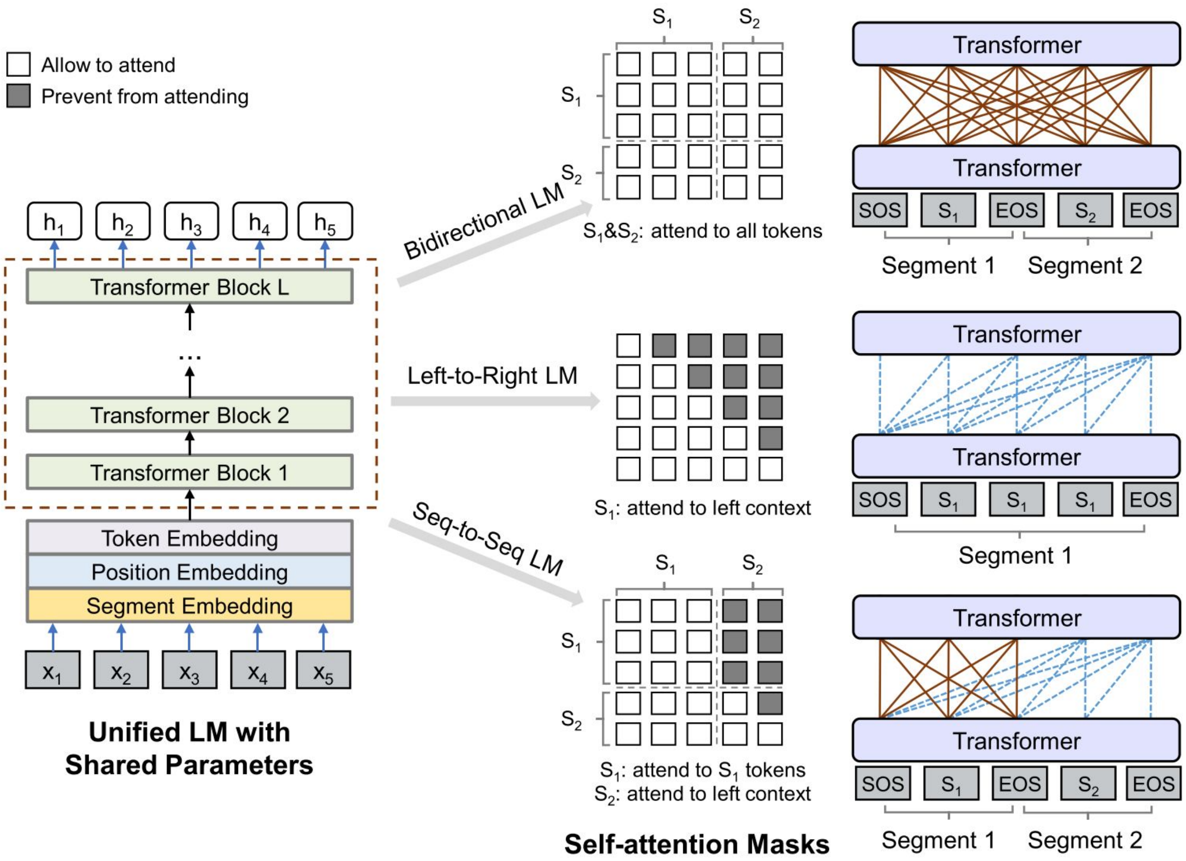


Fine-tuning

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

Baevski et al. (2019)

UniLM - Dong et al., 2019

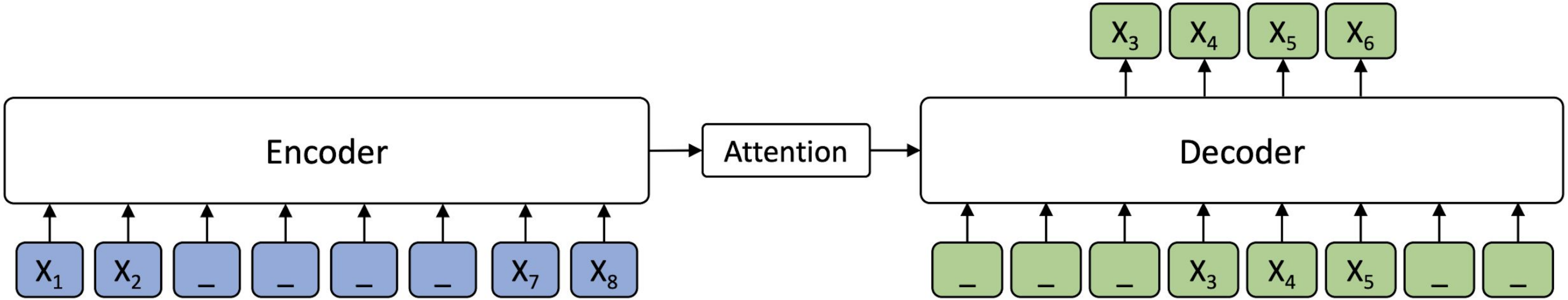


Model is jointly pretrained on three variants of LM (bidirectional, left-to-right, seq-to-seq)

SOTA on three natural language generation tasks

Masked Sequence to Sequence Pretraining (MASS)

Pretrain encoder-decoder



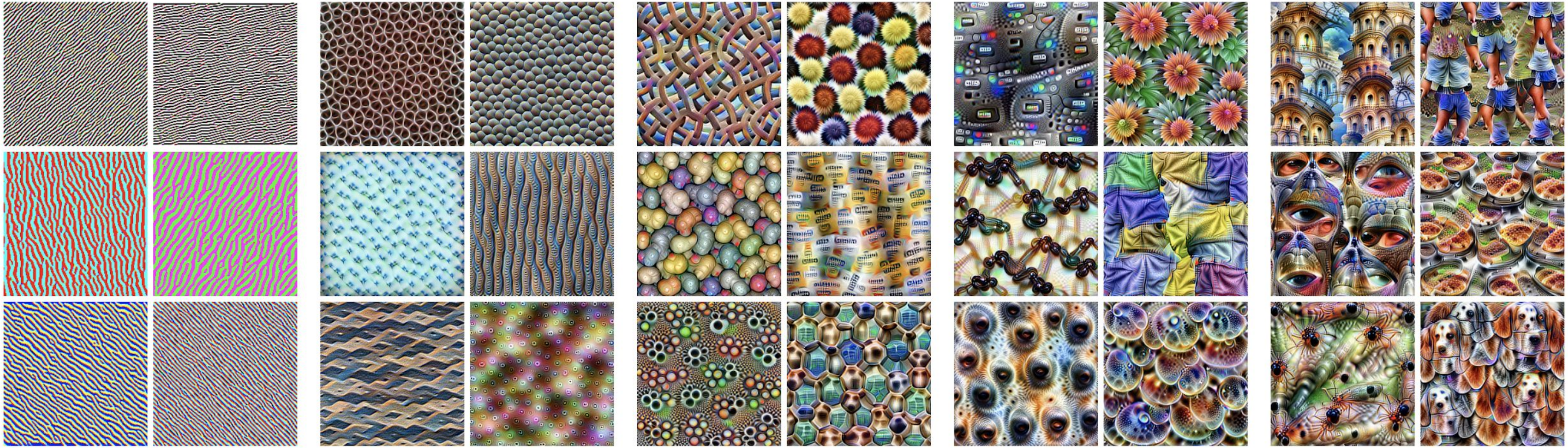
(Song et al., ICML 2019)

What matters: Pretraining Objective, Encoder

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.)

An inspiration from Computer Vision



Edges (layer conv2d0) **Textures** (layer mixed3a) **Patterns** (layer mixed4a) **Parts** (layers mixed4b & mixed4c) **Objects** (layers mixed4d & mixed4e)

From **lower to higher layers**, information goes from **general to task-specific**.

Image credit: [Distill]

Other methods for analysis

- Textual omission and multi-modal: Kadar et al. , 16
- 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

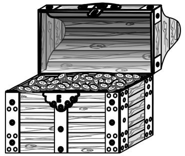
What to analyze?

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

- Layers
- Pretraining Objectives

What to look for?

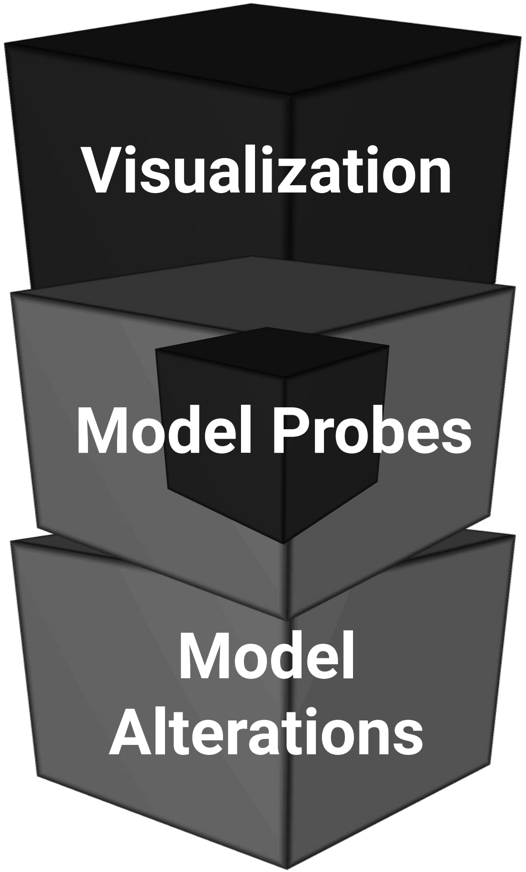
- Surface-level features
- Lexical features
  - E.g. POS tags
- Morphology
- Syntactic Structure
  - Word-level
  - Sentence-level
- Semantic Structure
  - E.g. Roles, Coreference



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

Analysis / Evaluation : Adversarial Methods

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.

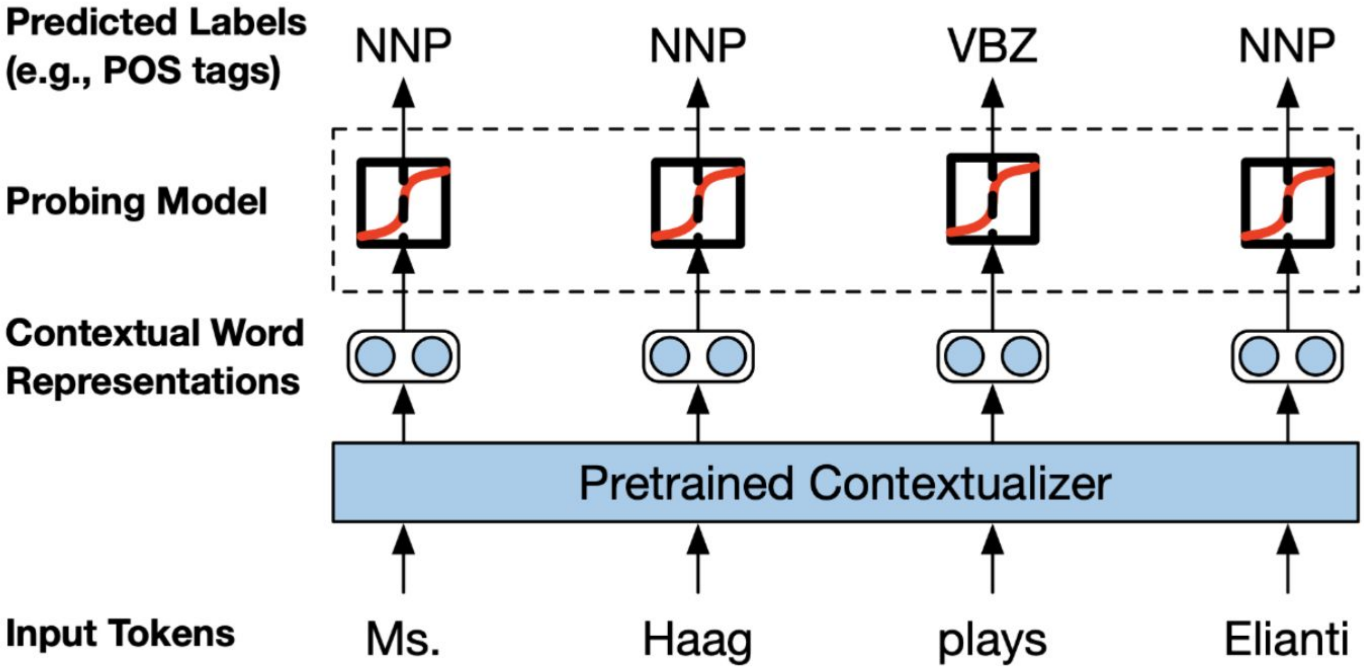
The number of old Acadian colonists declined after the year 1675.

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...

Credits: Jia & Liang (2017) and Percy Liang. AI Frontiers. 2018

Probes are simple linear / neural layers



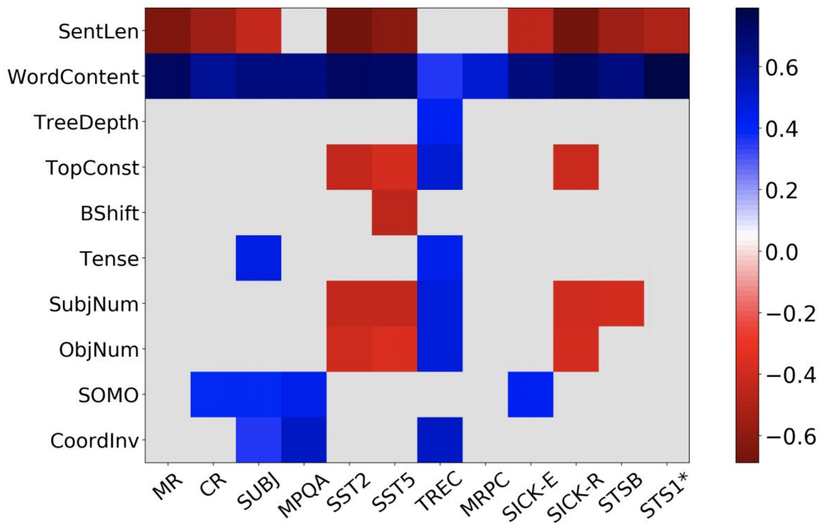
Liu et al., NAACL 2019

What is still left unanswered?

- **Interpretability** is difficult Lipton et al., 2016
  - 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!



Conneau et al., 2018

Transferability to downstream tasks