ISO 29990:2010

NATIONAL INFOCOMM COMPETENCY FRAMEWORK

NUS National University of Singapore | ISS INSTITUTE OF SYSTEMS SCIENCE

Text Processing using Machine Learning

Sentence Representation

Liling Tan

05 Dec 2019

OVER 5,500 GRADUATE ALUMNI

OFFERING OVER 120 ENTERPRISE IT, INNOVATION & LEADERSHIP PROGRAMMES

TRAINING OVER 120,000 DIGITAL LEADERS & PROFESSIONALS

© 2018 National University of Singapore. All Rights Reserved

Overview

Lecture

- Sentence Representation
- Type of Learning
- Skipthoughts and Siamese Net
- InferSent and USE
- Generalized LM

Hands-on

- PyTorch LMs

© 2018 National University of Singapore. All Rights Reserved

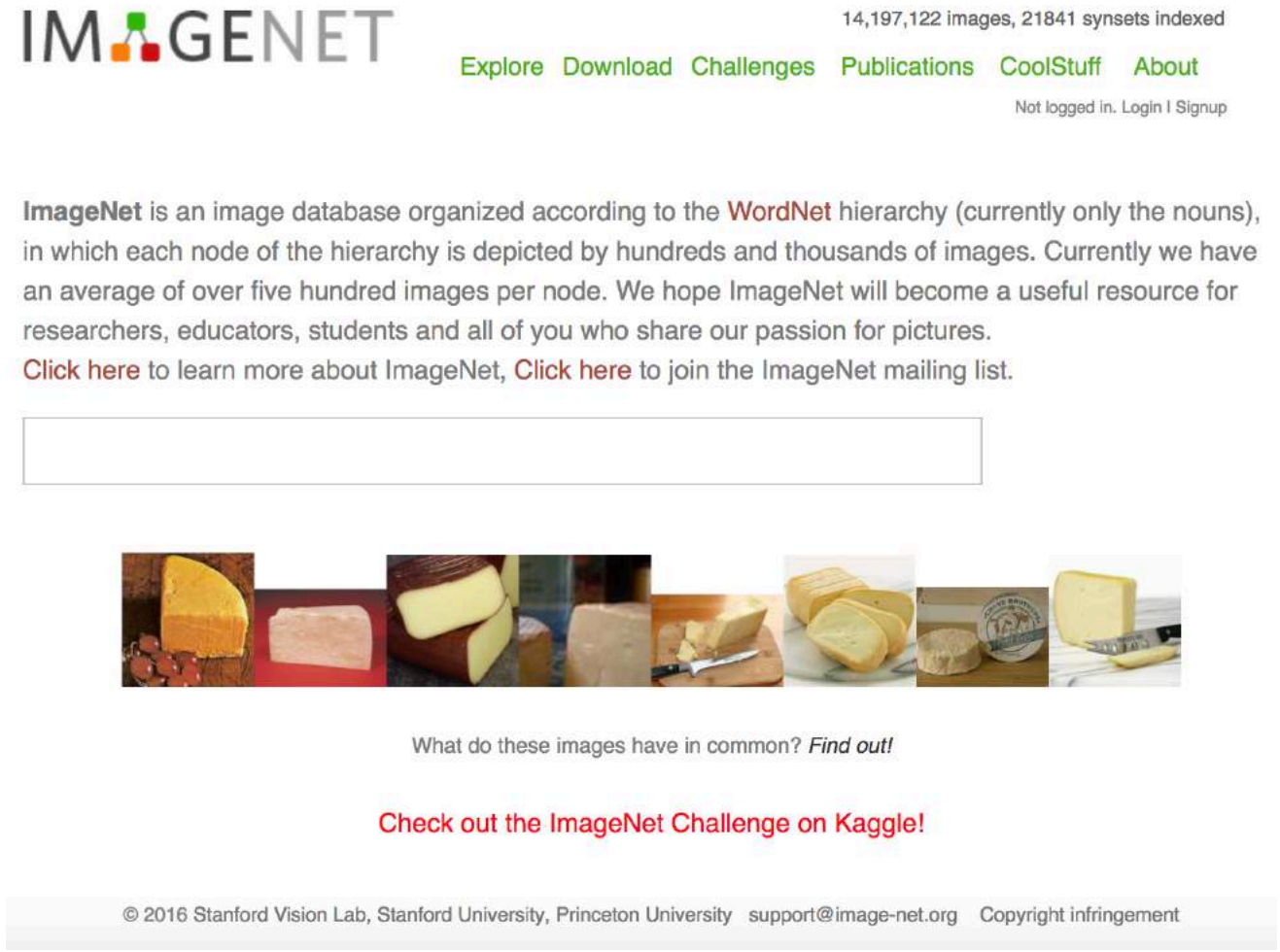
Sentence Representation

© 2018 National University of Singapore. All Rights Reserved

The "ImageNet" Moment

Various Computer Vision Challenges (ImageNet, MS Coco, etc.) started a wave of groups **training models and sharing pre-trained models.**

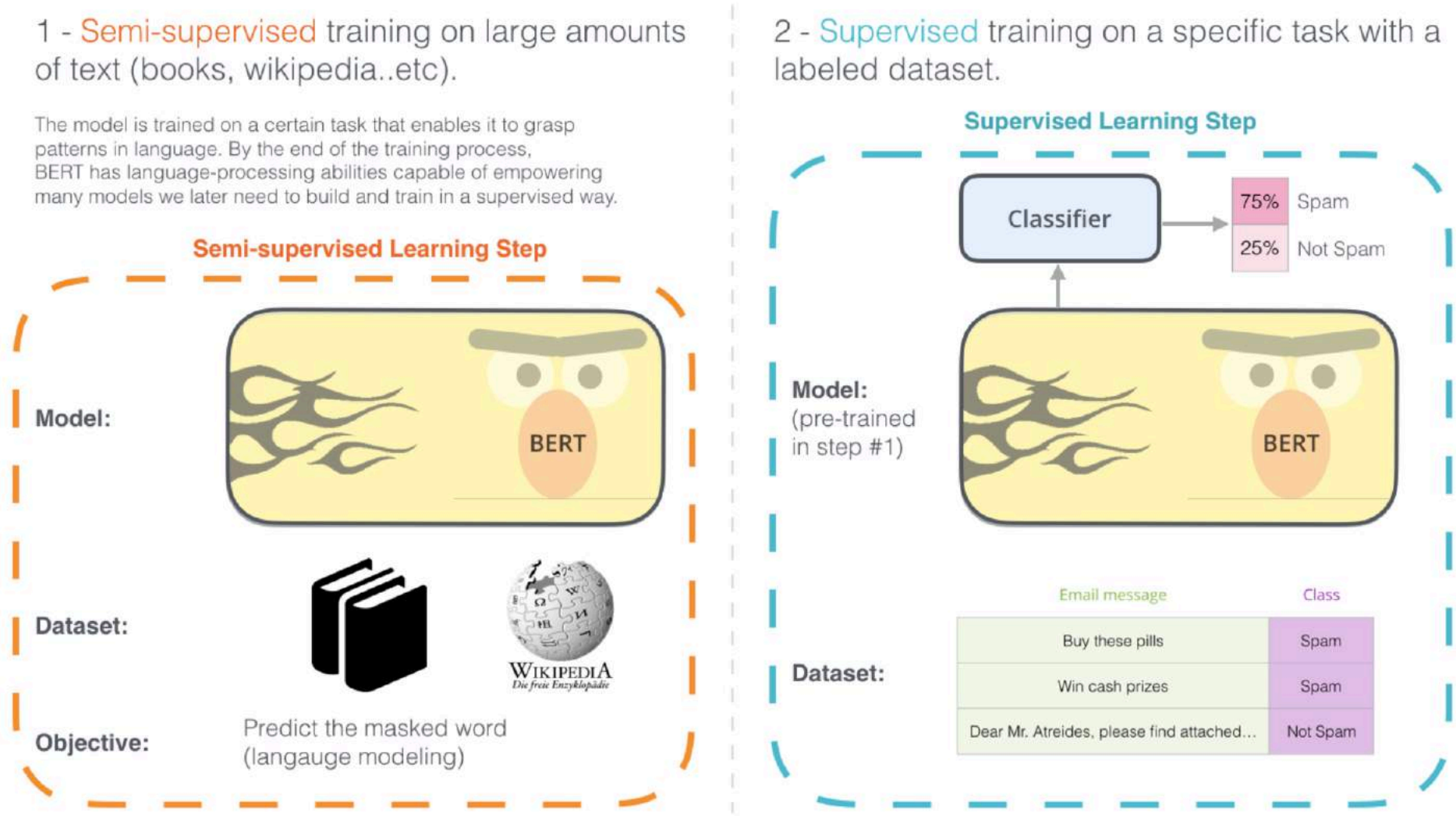
Fine-tuning / transfer learning for these pre-trained models are faster and *"usually better"* when training new models for other computer vision task.



(*Image from Stanford Vision Lab)

© 2018 National University of Singapore. All Rights Reserved

The "ImageNet" Moment for NLP



The two steps of how BERT is developed. You can download the model pre-trained in step 1 (trained on un-annotated data), and only worry about fine-tuning it for step 2. [Source for book icon].

(*Image from Jay Alammar's blog)

The "ImageNet" Moment for NLP

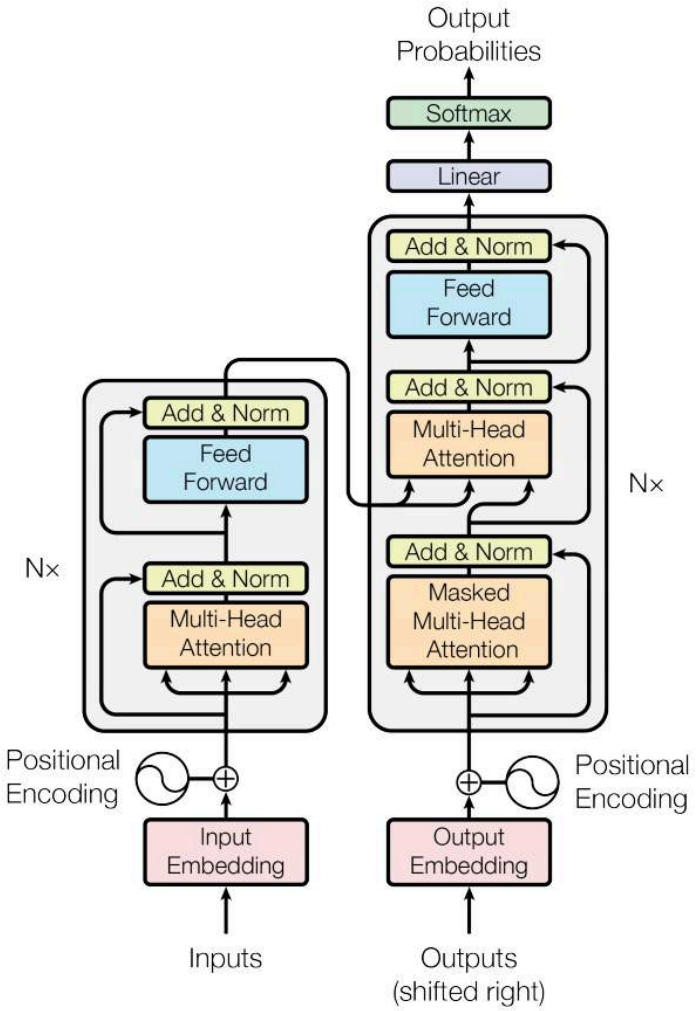


Figure 1: The Transformer - model architecture.

(*Image from Vaswani et al. 2017)

Major breakthrough of BERT came through the **self-attention network architecture, aka. Transformer** (Vaswani et al. 2017)

But do note the caveats with transfer-learning (aka. pre-training and fine-tuning):

- Could get **same results from random initialization** vs pre-training counterparts (He et al. 2018 on "*Rethinking ImageNet*")
- Understanding **why pre-training works in NLP still unclear** (Goldberg, 2018, see also Erhan, 2010)

© 2018 National University of Singapore. All Rights Reserved

Attention is not an Explanation (26 Feb 2019)

- **https://arxiv.org/abs/1902.10186**

- *Attention mechanisms have seen wide adoption in neural NLP models. In addition to improving predictive performance, these are often touted as affording transparency: models equipped with attention provide a distribution over attended-to input units, and this is often presented (at least implicitly) as communicating the relative importance of inputs. However,* ***it is unclear what relationship exists between attention weights and model outputs.*** *... Our findings show that* ***standard attention modules do not provide meaningful explanations and should not be treated as though they do****.*

© 2018 National University of Singapore. All Rights Reserved

Pay Less Attention (ICLR 2019)

- https://openreview.net/pdf?id=SkVhlh09tX

- *Self-attention is a useful mechanism to build generative models for language and images. It determines the importance of context elements by comparing each element to the current time step. In this paper, we show that a very lightweight convolution can perform competitively to the best reported self-attention results. …*

- *Next, we introduce **dynamic convolutions which are simpler and more efficient than self-attention**. We predict separate convolution kernels based solely on the current time-step in order to determine the importance of context elements. The number of operations required by this approach scales linearly in the input length, whereas **self-attention is quadratic**.*

© 2018 National University of Singapore. All Rights Reserved

Types of Learning

© 2018 National University of Singapore. All Rights Reserved

Types of Learning

- **Multi-Task Learning:** Training on multiple datasets/tasks

(Image from Burpple.com)

© 2018 National University of Singapore. All Rights Reserved

Types of Learning

- **Multi-Task Learning:** Training on multiple datasets/tasks

- **Transfer Learning:** Type of Multi-Task Learning, where learning is multi-task but evaluation focus on is on a "downstream" single task

(Image from Burpple.com)

© 2018 National University of Singapore. All Rights Reserved

Types of Learning

- **Multi-Task Learning:** Training on multiple datasets/tasks
- **Transfer Learning:** Type of Multi-Task Learning, where learning is multi-task but evaluation focus on is on a "downstream" single task
- **Domain Adaptation:** Type of Transfer Learning, where training is on generic and/or some in-domain datasets but evaluation is focus on in-domain dataset

(Image from sethlui.com)

© 2018 National University of Singapore. All Rights Reserved

Plethora of Tasks in NLP

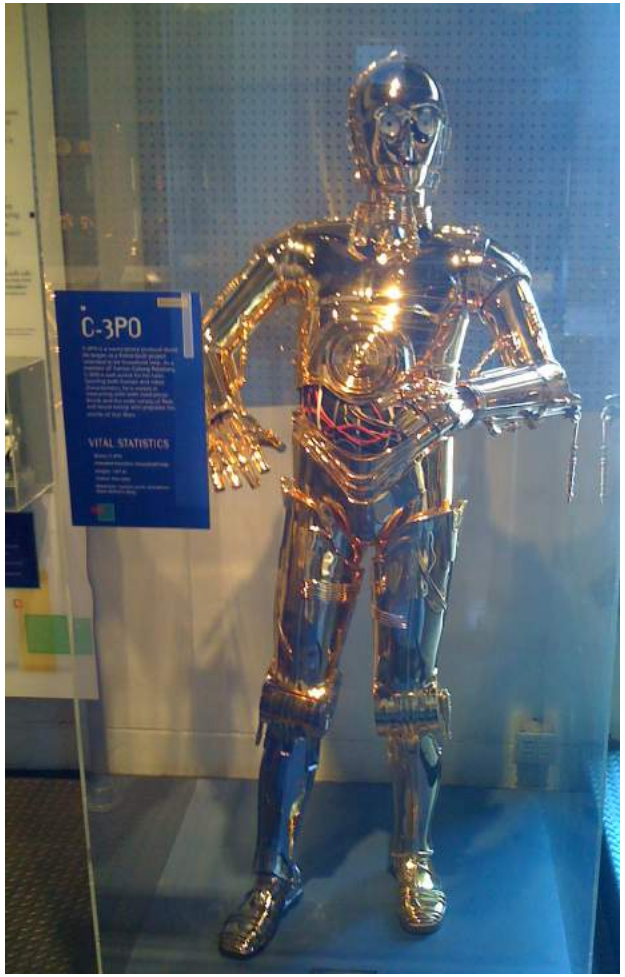
- In NLP, there are a plethora of tasks, each requiring different varieties of data
  - **Only text:** e.g. language modeling
  - **Naturally occurring data:** e.g. machine translation
  - **Hand-labeled data:** e.g. most analysis tasks
- And each in many languages, many domains!

This slide is taken from CMU NN4NLP (Neubig, 2019)

Why Multi-Task Learning?

Strong AI

- Aka. AGI (Artificial General Intelligence)
- Human-like or super-human abilities
- *"Can think and have a mind"*

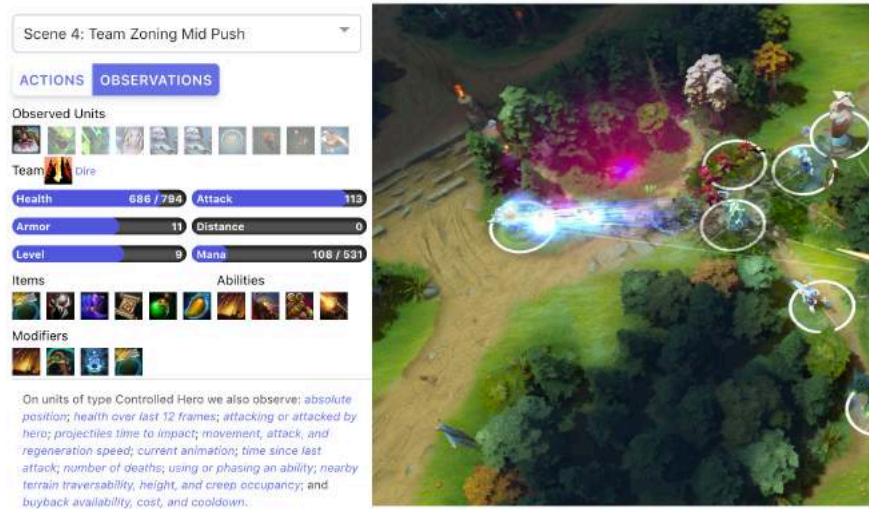


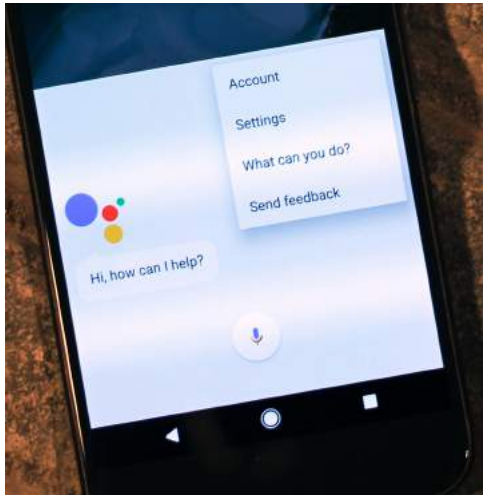
© 2018 National University of Singapore. All Rights Reserved

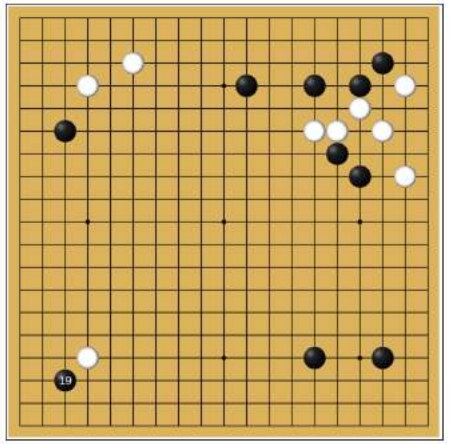
Why Multi-Task Learning?

Weak AI

- Vaguely, "computationally perform specific task(s)"
- Minimal awareness beyond task(s)
- "*Can only act like it thinks and has a mind*"





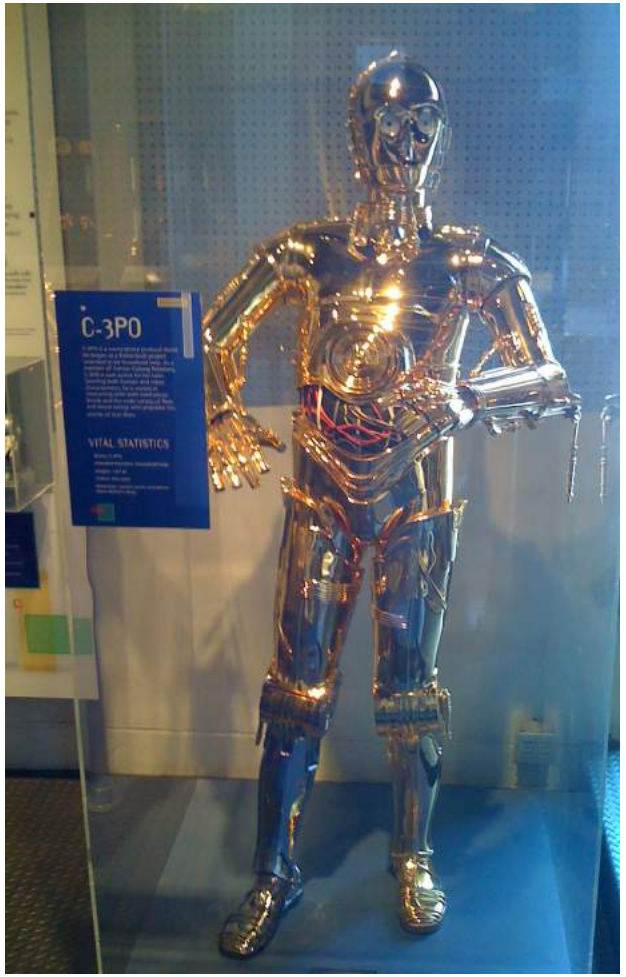


© 2018 National University of Singapore. All Rights Reserved

Why Multi-Task Learning?

Strong NLP

- Aka. AGI (Artificial General Intelligence)
- Human-like or super-human abilities
- *"Can think and have a mind"*
- *"Can understand and produce human languages*"

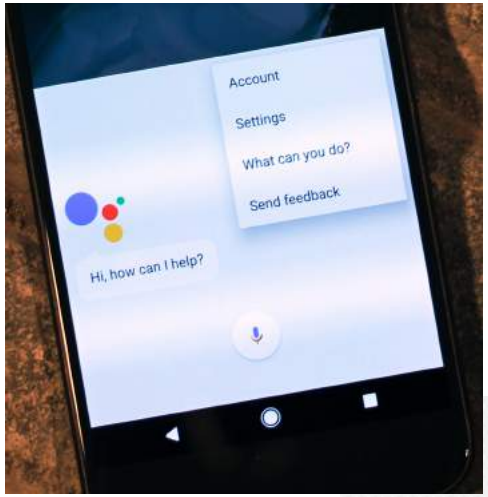


© 2018 National University of Singapore. All Rights Reserved

Why Multi-Task Learning?

Weak NLP

- Minimal awareness beyond task(s)
- "*Can only act like it thinks and has a mind*"
- "Can only perform specific human language(s) task"



© 2018 National University of Singapore. All Rights Reserved

Why Multi-Task Learning?

- Ideally, we want a "Strong NLP"
- Honestly, **making labelled dataset is time-consuming and requires lots of resources,** so the resultant dataset created are often small-ish
- By combining multiple tasks, we have more data =)

© 2018 National University of Singapore. All Rights Reserved

Why Transfer-Learning?

- Even with Multi-Task Learning **dataset with labels are still small relative to the size of text-only dataset** (e.g. Wikipedia, Common Crawl, News articles)

- What if we can learn a generalized language model then reuse it to fine-tune for downstream tasks when necessary?

- How do we represent a sentence?

© 2018 National University of Singapore. All Rights Reserved

Pretrained Language Models

© 2018 National University of Singapore. All Rights Reserved

Why Transfer-Learning?

- It started with the idea that we need to somehow represent sentence by a vector

  "*A sentence embedding can be a word embedding but a word embedding cannot be a sentence embedding.*"
  – Nat Gillin

- So multi-task and transfer learning becomes a means to train sentence embeddings

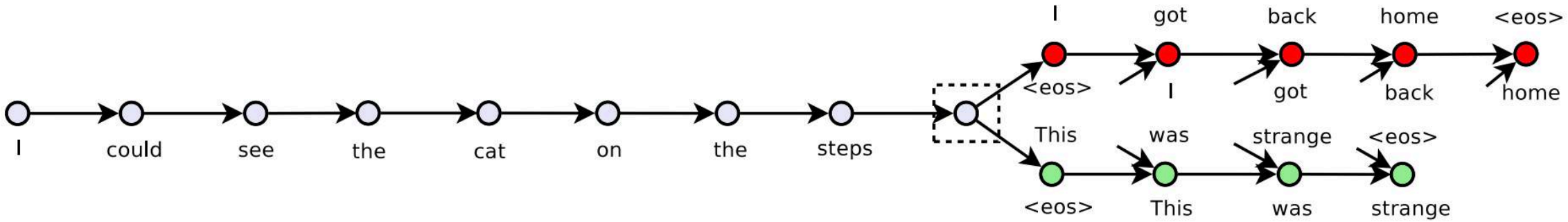
- Fast-forward to today, it's just a lot of *transformers with tricks*…

© 2018 National University of Singapore. All Rights Reserved

SkipThought, Siamese Net and Sentence Similarity

© 2018 National University of Singapore. All Rights Reserved

Skip-Thought Vectors (Kiros et al. 2017)



- Get a sentence triplets, from the focus sentence, generate both sentence before (red) and sentence after (green)
- Learn a Seq2Seq model, use the learnt encoder as a sentence encoder

Objective. Given a tuple (s_{i-1}, s_i, s_{i+1}), the objective optimized is the sum of the log-probabilities for the forward and backward sentences conditioned on the encoder representation:

$$\sum_t \log P(w_{i+1}^t | w_{i+1}^{<t}, \mathbf{h}_i) + \sum_t \log P(w_{i-1}^t | w_{i-1}^{<t}, \mathbf{h}_i) \quad (10)$$

© 2018 National University of Singapore. All Rights Reserved

Skip-Thought Vectors (Kiros et al. 2017)

Query and nearest sentence
he ran his hand inside his coat , double-checking that the unopened letter was still there . he slipped his hand between his coat and his shirt , where the folded copies lay in a brown envelope .
im sure youll have a glamorous evening , she said , giving an exaggerated wink . im really glad you came to the party tonight , he said , turning to her .
although she could tell he had n't been too invested in any of their other chitchat , he seemed genuinely curious about this . although he had n't been following her career with a microscope , he 'd definitely taken notice of her appearances .
an annoying buzz started to ring in my ears , becoming louder and louder as my vision began to swim . a weighty pressure landed on my lungs and my vision blurred at the edges , threatening my consciousness altogether .
if he had a weapon , he could maybe take out their last imp , and then beat up errol and vanessa . if he could ram them from behind , send them sailing over the far side of the levee , he had a chance of stopping them .
then , with a stroke of luck , they saw the pair head together towards the portaloos . then , from out back of the house , they heard a horse scream probably in answer to a pair of sharp spurs digging deep into its flanks .
" i 'll take care of it , " goodman said , taking the phonebook . " i 'll do that , " julia said , coming in .
he finished rolling up scrolls and , placing them to one side , began the more urgent task of finding ale and tankards . he righted the table , set the candle on a piece of broken plate , and reached for his flint , steel , and tinder .

© 2018 National University of Singapore. All Rights Reserved

Semantic Textual Similarity

Dataset	Domain	Score	Sent1	Sent2
STS2012-gold	surprise.OnWN	5.0	render one language in another language ...	restate (words) from one language into another ...
STS2012-gold	surprise.OnWN	3.25	nations unified by shared interests, history or ...	a group of nations having common interests. ...
STS2012-gold	surprise.OnWN	3.25	convert into absorbable substances, (as if) with ...	soften or disintegrate by means of chemical act ...
STS2012-gold	surprise.OnWN	4.0	devote or adapt exclusively to an skill, ...	devote oneself to a special area of work. ...
STS2012-gold	surprise.OnWN	3.25	elevated wooden porch of a house ...	a porch that resembles the deck on a ship. ...
STS2012-gold	surprise.OnWN	4.0	either half of an archery bow ...	either of the two halves of a bow from handle to ...
STS2012-gold	surprise.OnWN	3.333	a removable device that is an accessory to la ...	a supplementary part or accessory. ...
STS2012-gold	surprise.OnWN	4.75	restrict or confine	place limits on (extent or access). ...
STS2012-gold	surprise.OnWN	0.5	orient, be positioned	be opposite.
STS2012-gold	surprise.OnWN	4.75	Bring back to life, return from the dead ...	cause to become alive again. ...

Given a dataset of **pairs of sentences and a similarity score** assigned by humans, learn a model to assign a score given two sentences

Metric: Correlation score with human annotations

Famous datasets:

- SemEval STS
- SICK
- MS Paraphrase Corpus
- Quora Question Pairs
- Corpus of Linguistics Acceptability

Siamese Network for STS

Muller and Thyagarajan (2016) trained a network with two separate LSTM layers and the **last layer is a Mahattan distance between the output of the two LSTMs**

Works well on the SICK dataset, outperforms the Skip-Thought

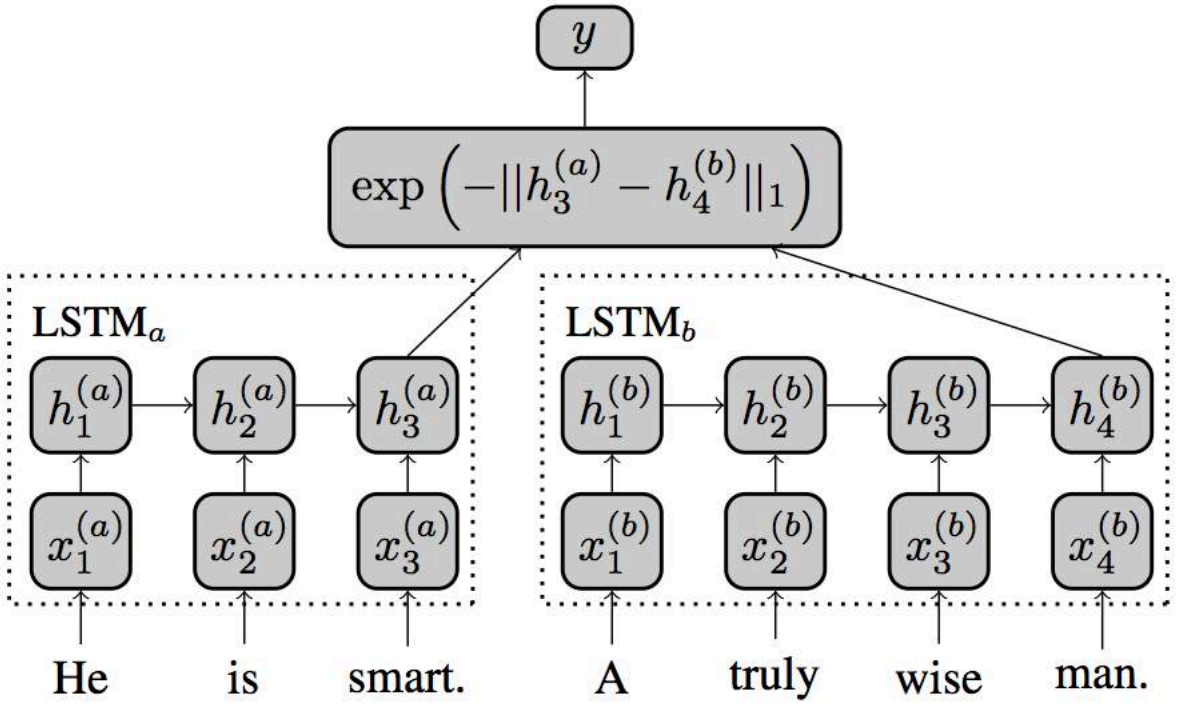


Figure 1: Our model uses an LSTM to read in word-vectors representing each input sentence and employs its final hidden state as a vector representation for each sentence. Subsequently, the similarity between these representations is used as a predictor of semantic similarity.

Stanford Natural Language Inference, InferSent, Deep Averaging Network and Universal Sentence Encoder

© 2018 National University of Singapore. All Rights Reserved

Stanford Natural Language Inference Dataset

Text	Judgments	Hypothesis
A man inspects the uniform of a figure in some East Asian country.	contradiction C C C C C	The man is sleeping
An older and younger man smiling.	neutral N N E N N	Two men are smiling and laughing at the cats playing on the floor.
A black race car starts up in front of a crowd of people.	contradiction C C C C C	A man is driving down a lonely road.
A soccer game with multiple males playing.	entailment E E E E E	Some men are playing a sport.
A smiling costumed woman is holding an umbrella.	neutral N N E C N	A happy woman in a fairy costume holds an umbrella.

The SNLI corpus (version 1.0) is a collection of 570k human-written English sentence pairs manually labeled for balanced classification with the labels ***entailment***, ***contradiction***, **and** ***neutral***, supporting the task of natural language inference (NLI), also known as recognizing textual entailment (RTE).

InferSent: Supervised Learning of Universal Sentence Representations from Natural Language Inference

1. Initialize a model for sentence encoder,

2. Put the pair of sentences through the encoder and generate the sentence vector, u and v.

3. Compute the two distance / similarity metrics |u-v| and u*v

4. Concat output of 2 and 3 put it through feed-forward net,

5. End up with a 3 way softmax

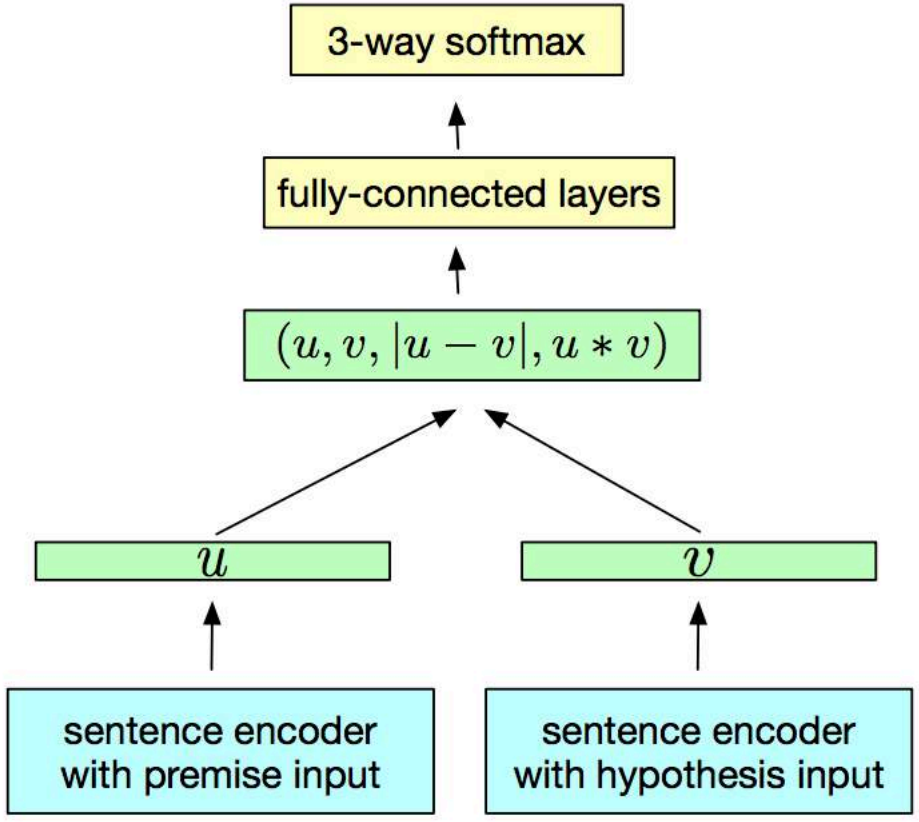


Figure 1: **Generic NLI training scheme.**

InferSent: Supervised Learning of Universal Sentence Representations from Natural Language Inference

y = [0, 0, 1]

z = FFN(u, v, | u-v | , u*v)

z has shape [1 x 3]

Last layer options:

(i) sigmoid(z) , multi-label, multi-class
(ii) softmax(z) , single-label, multi-class

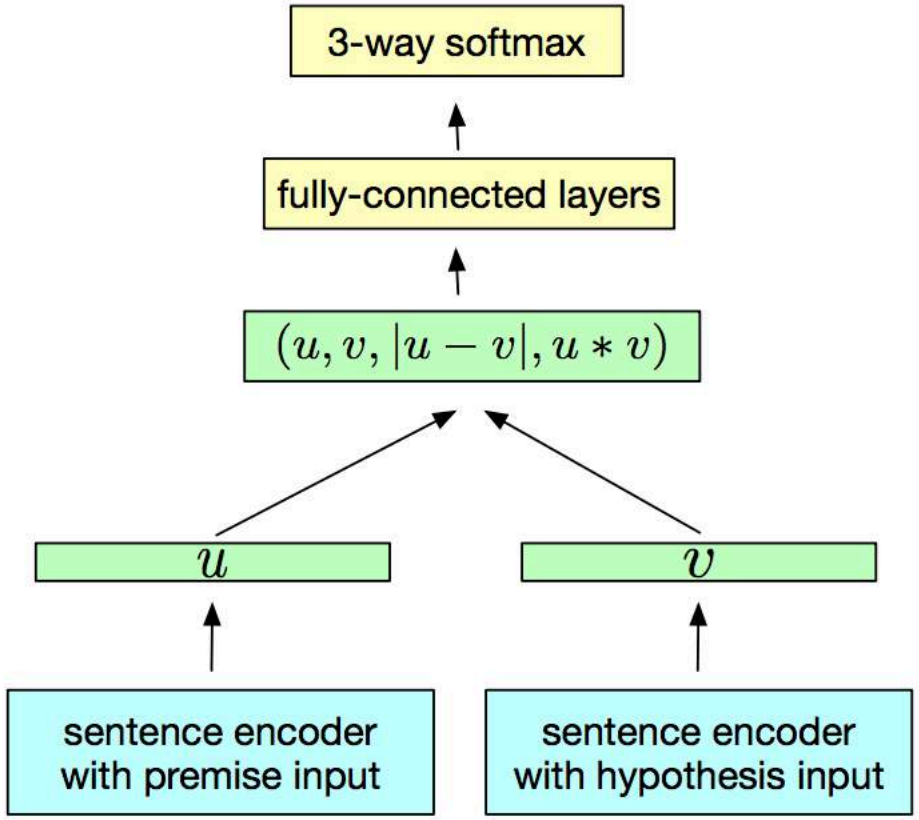


Figure 1: **Generic NLI training scheme.**

InferSent: Supervised Learning of Universal Sentence Representations from Natural Language Inference

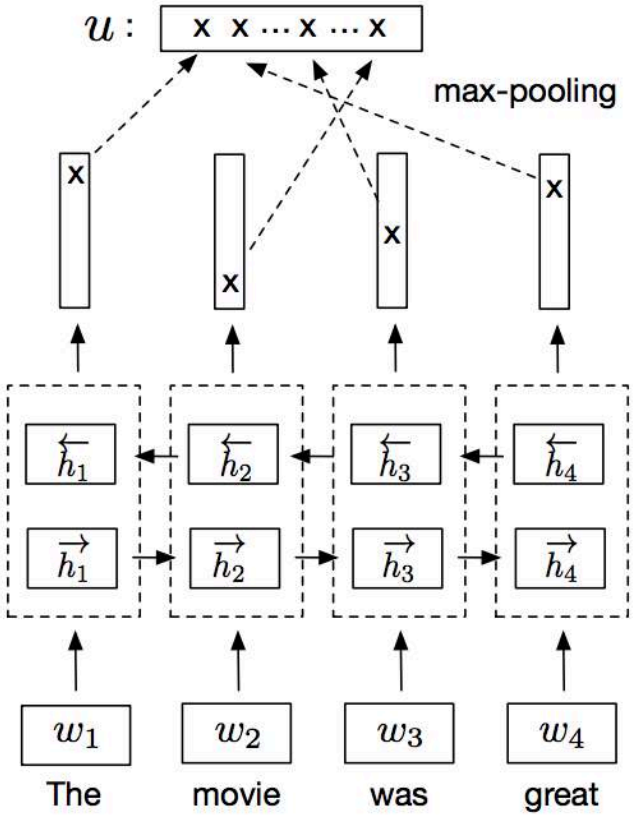


Figure 2: **Bi-LSTM max-pooling network.**

Model	dim	NLI dev	NLI test	Transfer micro	Transfer macro
LSTM	2048	81.9	80.7	79.5	78.6
GRU	4096	82.4	81.8	81.7	80.9
BiGRU-last	4096	81.3	80.9	82.9	81.7
BiLSTM-Mean	4096	79.0	78.2	83.1	81.7
Inner-attention	4096	82.3	82.5	82.1	81.0
HConvNet	4096	83.7	83.4	82.0	80.9
BiLSTM-Max	4096	**85.0**	**84.5**	**85.2**	**83.7**

Table 3: **Performance of sentence encoder architectures** on SNLI and (aggregated) transfer tasks. Dimensions of embeddings were selected according to best aggregated scores (see Figure 5).

InferSent: Supervised Learning of Universal Sentence Representations from Natural Language Inference

name	task	N	premise	hypothesis	label
SNLI	NLI	560k	"Two women are embracing while holding to go packages."	"Two woman are holding packages."	entailment
SICK-E	NLI	10k	A man is typing on a machine used for stenography	The man isn't operating a steno-graph	contradiction
SICK-R	STS	10k	"A man is singing a song and play-ing the guitar"	"A man is opening a package that contains headphones"	1.6
STS14	STS	4.5k	"Liquid ammonia leak kills 15 in Shanghai"	"Liquid ammonia leak kills at least 15 in Shanghai"	4.6

Table 2: **Natural Language Inference and Semantic Textual Similarity tasks**. NLI labels are contra-diction, neutral and entailment. STS labels are scores between 0 and 5.

Model	MR	CR	SUBJ	MPQA	SST	TREC	MRPC	SICK-R	SICK-E	STS14
Unsupervised representation training (unordered sentences)										
Unigram-TFIDF	73.7	**79.2**	90.3	82.4	-	85.0	73.6/81.7	-	-	.58/.57
ParagraphVec (DBOW)	60.2	66.9	76.3	70.7	-	59.4	72.9/81.1	-	-	.42/.43
SDAE	74.6	78.0	90.8	86.9	-	78.4	**73.7**/80.7	-	-	.37/.38
SIF (GloVe + WR)	-	-	-	-	82.2	-	-	-	**84.6**	**.69**/ -
word2vec BOW†	77.7	79.8	90.9	88.3	79.7	83.6	72.5/81.4	0.803	78.7	.65/.64
fastText BOW†	78.3	81.0	**92.4**	87.8	**81.9**	84.8	**73.9/82.0**	0.815	78.3	.63/.62
GloVe BOW†	**78.7**	78.5	91.6	87.6	79.8	83.6	72.1/80.9	0.800	78.6	.54/.56
GloVe Positional Encoding†	78.3	77.4	91.1	87.1	80.6	83.3	72.5/81.2	0.799	77.9	.51/.54
BiLSTM-Max (untrained)†	77.5	**81.3**	89.6	**88.7**	80.7	**85.8**	73.2/81.6	**0.860**	83.4	.39/.48
Unsupervised representation training (ordered sentences)										
FastSent	70.8	78.4	88.7	80.6	-	76.8	72.2/80.3	-	-	**.63/.64**
FastSent+AE	71.8	76.7	88.8	81.5	-	80.4	71.2/79.1	-	-	.62/.62
SkipThought	76.5	80.1	93.6	87.1	82.0	**92.2**	**73.0/82.0**	**0.858**	82.3	.29/.35
SkipThought-LN	**79.4**	**83.1**	**93.7**	**89.3**	82.9	88.4	-	**0.858**	79.5	.44/.45
Supervised representation training										
CaptionRep (bow)	61.9	69.3	77.4	70.8	-	72.2	73.6/81.9	-	-	.46/.42
DictRep (bow)	76.7	78.7	90.7	87.2	-	81.0	68.4/76.8	-	-	**.67**/**.70**
NMT En-to-Fr	64.7	70.1	84.9	81.5	-	82.8	69.1/77.1	-		.43/.42
Paragram-phrase	-	-	-	-	79.7	-	-	0.849	83.1	**.71**/ -
BiLSTM-Max (on SST)†	(*)	83.7	90.2	89.5	(*)	86.0	72.7/80.9	0.863	83.1	.55/.54
BiLSTM-Max (on SNLI)†	79.9	84.6	92.1	**89.8**	83.3	**88.7**	75.1/82.3	**0.885**	**86.3**	.68/.65
BiLSTM-Max (on AllNLI)†	**81.1**	**86.3**	**92.4**	**90.2**	**84.6**	88.2	**76.2/83.1**	**0.884**	**86.3**	**.70/.67**
Supervised methods (directly trained for each task – no transfer)										
Naive Bayes - SVM	79.4	81.8	93.2	86.3	83.1	-	-	-	-	-
AdaSent	83.1	86.3	95.5	93.3	-	92.4	-	-	-	-
TF-KLD	-	-	-	-	-	-	80.4/85.9	-	-	-
Illinois-LH	-	-	-	-	-	-	-	-	84.5	-
Dependency Tree-LSTM	-	-	-	-	-	-	-	0.868	-	-

Deep Averaging Network

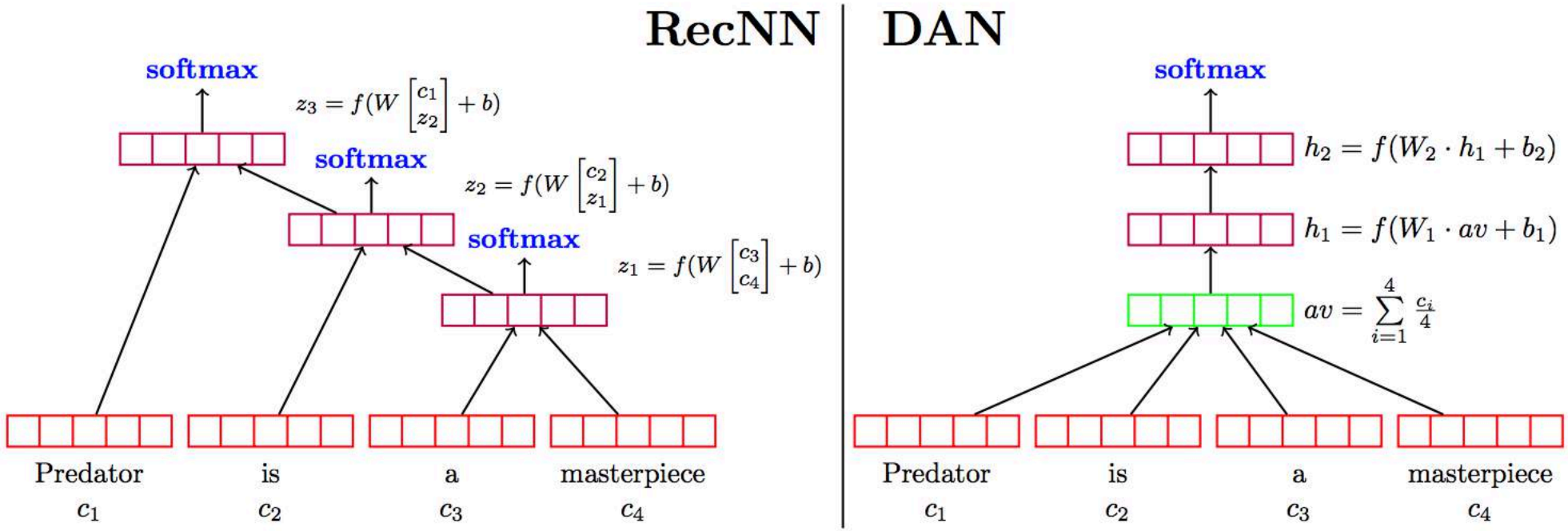


Figure 1: On the left, a **RecNN** is given an input sentence for sentiment classification. Softmax layers are placed above every internal node to avoid vanishing gradient issues. On the right is a two-layer **DAN** taking the same input. While the **RecNN** has to compute a nonlinear representation (purple vectors) for every node in the parse tree of its input, this **DAN** only computes two nonlinear layers for every possible input.

© 2018 National University of Singapore. All Rights Reserved

Deep Averaging Network

Sentence	DAN	DRecNN	Ground Truth
a lousy movie that's not merely unwatchable, but also unlistenable	negative	negative	negative
if you're not a prepubescent girl, you'll be laughing at britney spears' movie-starring debut whenever it does n't have you impatiently squinting at your watch	negative	negative	negative
blessed with immense physical prowess he may well be, but ahola is simply not an actor	positive	neutral	negative
who knows what exactly godard is on about in this film, but his words and images do n't have to add up to mesmerize you.	positive	positive	positive
it's so good that its relentless, polished wit can withstand not only inept school productions, but even oliver parker's movie adaptation	negative	positive	positive
too bad, but thanks to some lovely comedic moments and several fine performances, it's not a total loss	negative	negative	positive
this movie was not good	negative	negative	negative
this movie was good	positive	positive	positive
this movie was bad	negative	negative	negative
the movie was not bad	negative	negative	positive

© 2018 National University of Singapore. All Rights Reserved

Universal Sentence Encoder (Transfer Learning Datasets)

MR : Movie review snippet sentiment on a five star scale (Pang and Lee, 2005).

CR : Sentiment of sentences mined from customer reviews (Hu and Liu, 2004).

SUBJ : Subjectivity of sentences from movie reviews and plot summaries (Pang and Lee, 2004).

MPQA : Phrase level opinion polarity from news data (Wiebe et al., 2005).

TREC : Fine grained question classification sourced from TREC (Li and Roth, 2002).

SST : Binary phrase level sentiment classification (Socher et al., 2013).

STS Benchmark : Semantic textual similarity (STS) between sentence pairs scored by Pearson correlation with human judgments (Cer et al., 2017).

Dataset	Train	Dev	Test
SST	67,349	872	1,821
STS Bench	5,749	1,500	1,379
TREC	5,452	-	500
MR	-	-	10,662
CR	-	-	3,775
SUBJ	-	-	10,000
MPQA	-	-	10,606

Table 1: Transfer task evaluation sets

© 2018 National University of Singapore. All Rights Reserved

Universal Sentence Encoder (Results)

Model	MR	CR	SUBJ	MPQA	TREC	SST	STS Bench (dev / test)
Sentence & Word Embedding Transfer Learning							
USE_D+DAN (w2v w.e.)	77.11	81.71	93.12	87.01	94.72	82.14	–
USE_D+CNN (w2v w.e.)	78.20	82.04	93.24	85.87	97.67	85.29	–
USE_T+DAN (w2v w.e.)	81.32	86.66	93.90	88.14	95.51	86.62	–
USE_T+CNN (w2v w.e.)	81.18	87.45	93.58	87.32	98.07	86.69	–
Sentence Embedding Transfer Learning							
USE_D	74.45	80.97	92.65	85.38	91.19	77.62	0.763 / 0.719 (r)
USE_T	81.44	87.43	93.87	86.98	92.51	85.38	0.814 / 0.782 (r)
USE_D+DAN (lrn w.e.)	77.57	81.93	92.91	85.97	95.86	83.41	–
USE_D+CNN (lrn w.e.)	78.49	81.49	92.99	85.53	97.71	85.27	–
USE_T+DAN (lrn w.e.)	81.36	86.08	93.66	87.14	96.60	86.24	–
USE_T+CNN (lrn w.e.)	81.59	86.45	93.36	86.85	97.44	87.21	–
Word Embedding Transfer Learning							
DAN (w2v w.e.)	74.75	75.24	90.80	81.25	85.69	80.24	–
CNN (w2v w.e.)	75.10	80.18	90.84	81.38	97.32	83.74	–
Baselines with No Transfer Learning							
DAN (lrn w.e.)	75.97	76.91	89.49	80.93	93.88	81.52	–
CNN (lrn w.e.)	76.39	79.39	91.18	82.20	95.82	84.90	–

© 2018 National University of Singapore. All Rights Reserved

Generalized Language Models

Disclaimer: Almost the rest of the content for this lecture would have came from https://lilianweng.github.io/lil-log/2019/01/31/generalized-language-models.html and http://jalammar.github.io/illustrated-bert/
(Both are good summaries of the latest sentence embeddings)

CoVe: Contextual Word Vectors

CoVe

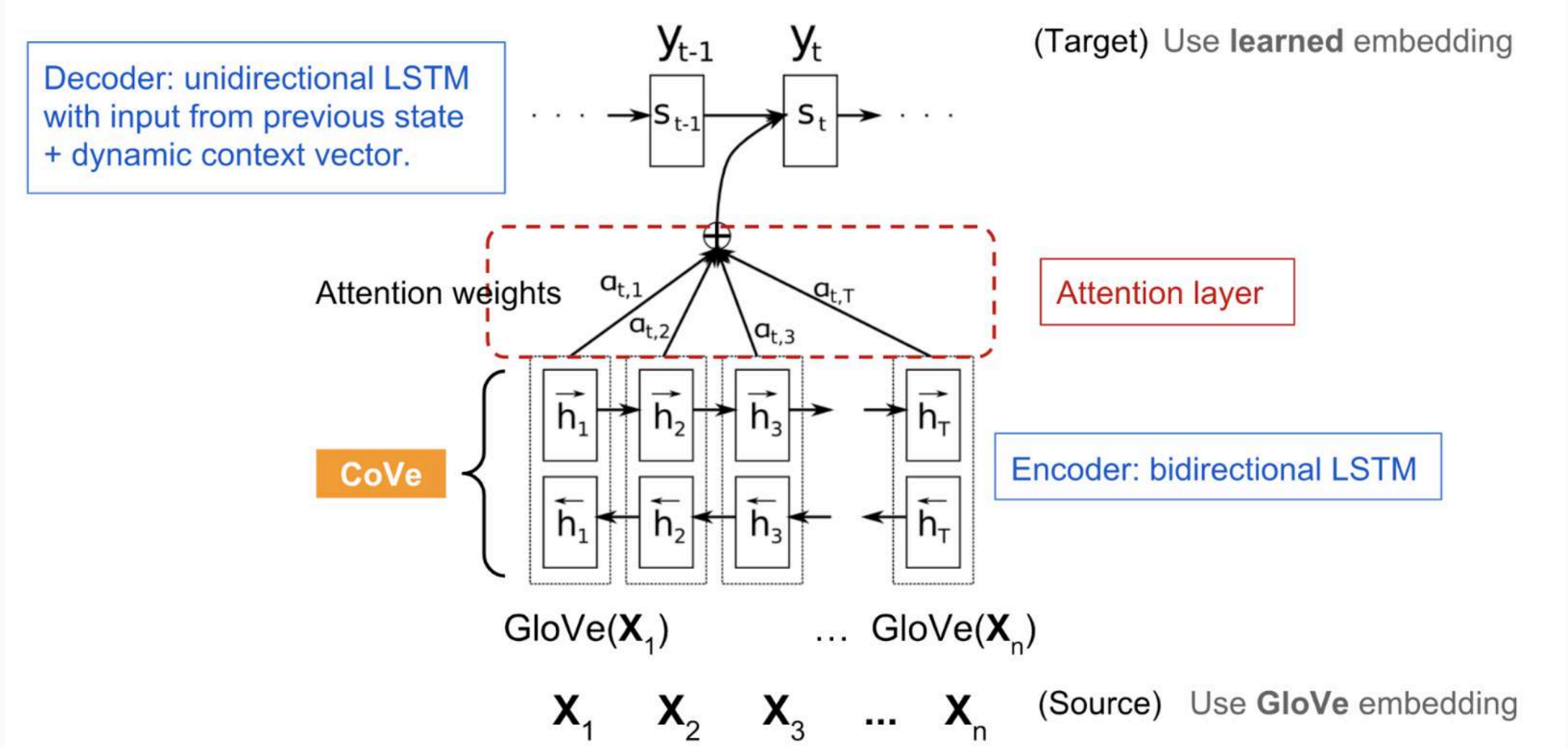


Fig. 1. The NMT base model used in CoVe.

CoVe

- A sequence of n words in source language (English): $x = [x_1, \ldots, x_n]$.
- A sequence of m words in target language (German): $y = [y_1, \ldots, y_m]$.
- The GloVe vectors of source words: $\text{GloVe}(x)$.
- Randomly initialized embedding vectors of target words: $z = [z_1, \ldots, z_m]$.
- The biLSTM encoder outputs a sequence of hidden states:

  $h = [h_1, \ldots, h_n] = \text{biLSTM}(\text{GloVe}(x))$ and $h_t = [\overrightarrow{h}_t; \overleftarrow{h}_t]$ where the forward LSTM computes $\overrightarrow{h}_t = \text{LSTM}(x_t, \overrightarrow{h}_{t-1})$ and the backward computation gives us $\overleftarrow{h}_t = \text{LSTM}(x_t, \overleftarrow{h}_{t-1})$.
- The attentional decoder outputs a distribution over words: $p(y_t \mid H, y_1, \ldots, y_{t-1})$ where H is a stack of hidden states $\{h\}$ along the time dimension:

$$\text{decoder hidden state: } s_t = \text{LSTM}([z_{t-1}; \tilde{h}_{t-1}], s_{t-1})$$
$$\text{attention weights: } \alpha_t = \text{softmax}(H(W_1 s_t + b_1))$$
$$\text{context-adjusted hidden state: } \tilde{h}_t = \tanh(W_2[H^\top \alpha_t; s_t] + b_2)$$
$$\text{decoder output: } p(y_t \mid H, y_1, \ldots, y_{t-1}) = \text{softmax}(W_{\text{out}} \tilde{h}_t + b_{\text{out}})$$

Using CoVe

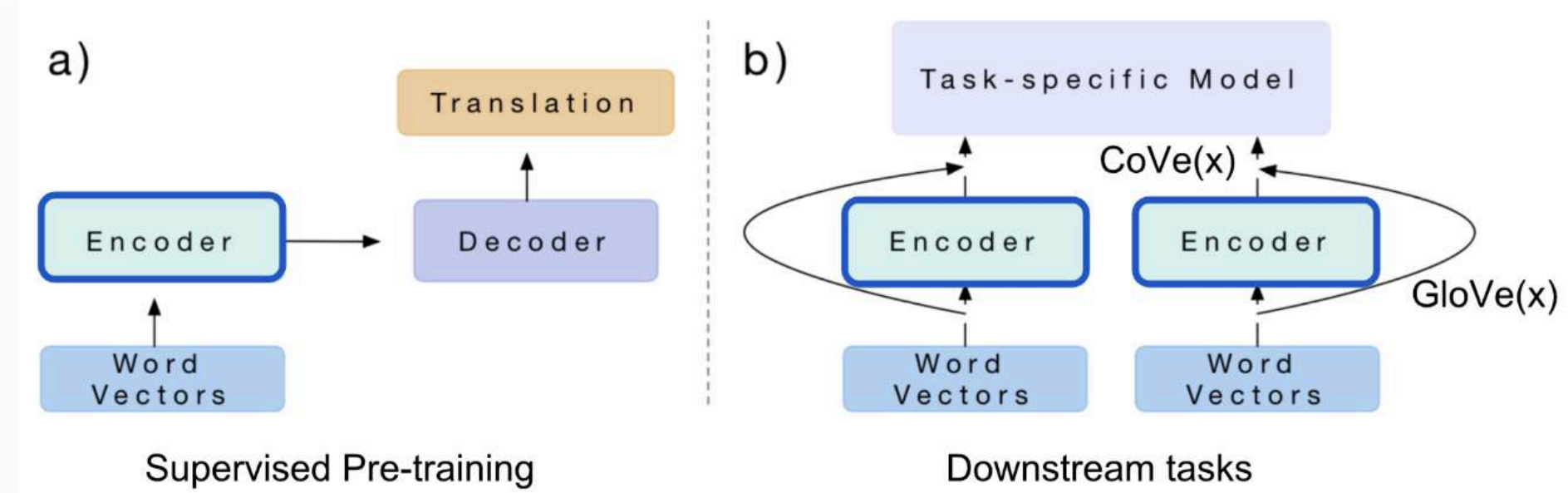
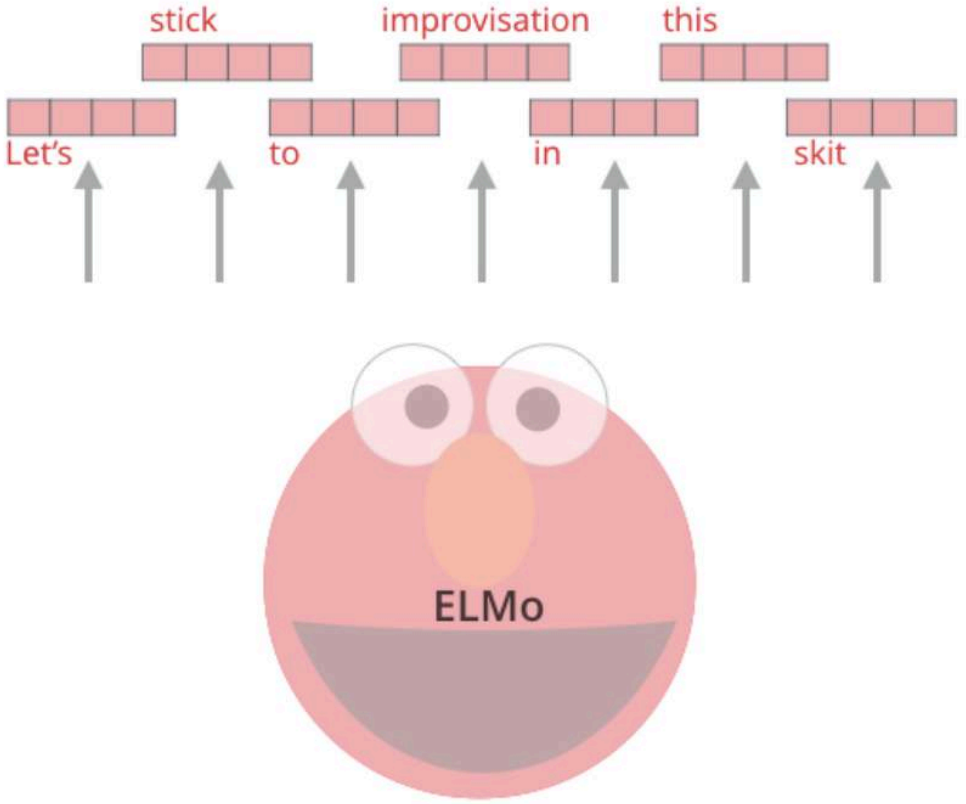


Fig. 2. The CoVe embeddings are generated by an encoder trained for machine translation task. The encoder can be plugged into any downstream task-specific model. (Image source: original paper)

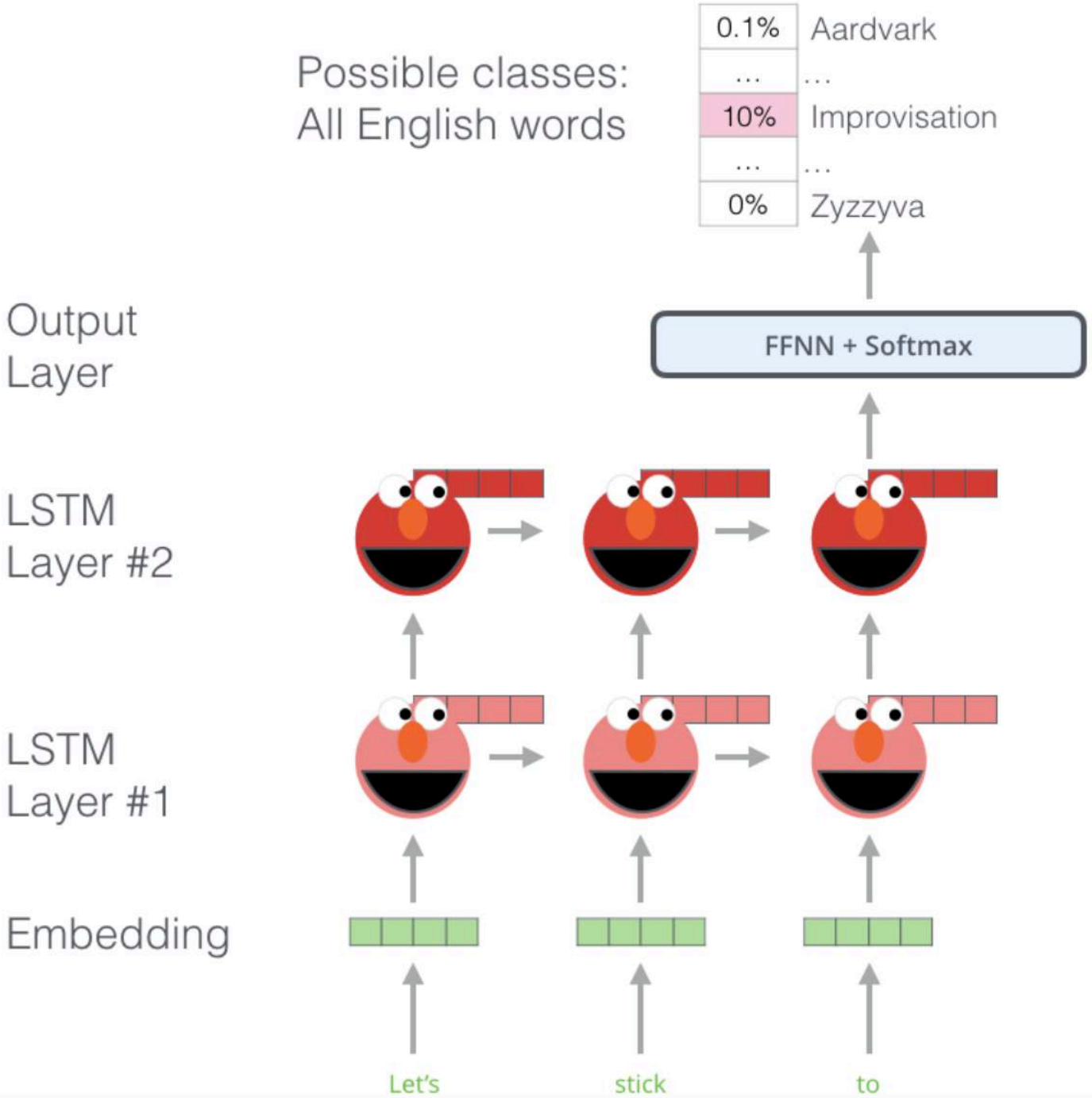
ELMo: Embeddings from Language Model

ELMo

ELMo Embeddings



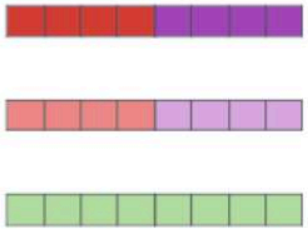
ELMo


Possible classes:
All English words
0.1% Aardvark
... ...
10% Improvisation
... ...
0% Zyzzyva
Output Layer
FFNN + Softmax
LSTM Layer #2
LSTM Layer #1
Embedding
Let's
stick
to

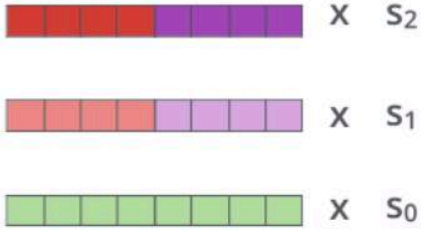
ELMo

Embedding of “stick” in “Let’s stick to” - Step #2

1- Concatenate hidden layers

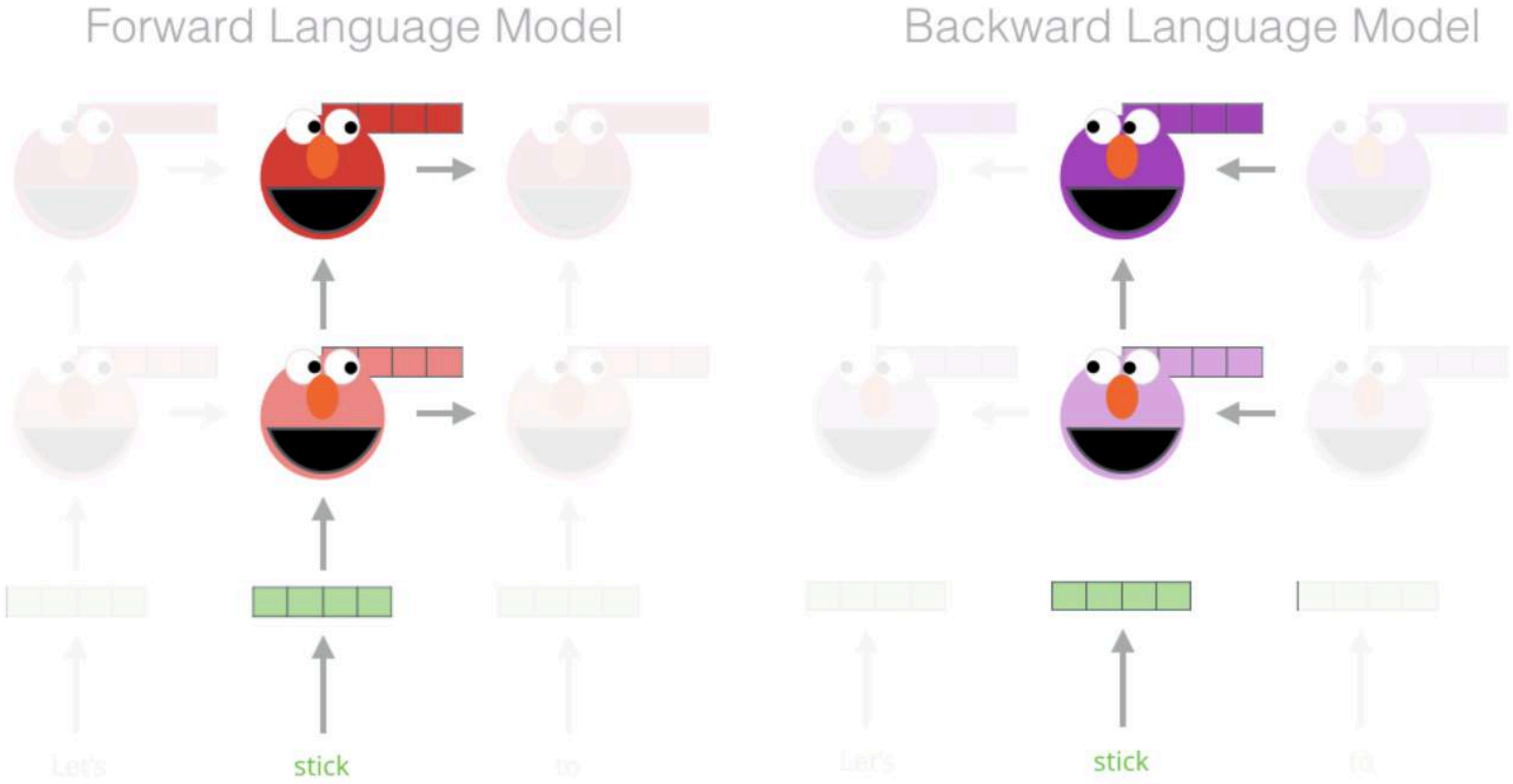


2- Multiply each vector by a weight based on the task



3- Sum the (now weighted) vectors

ELMo embedding of “stick” for this task in this context



ULMFit:Universal Language Model Fine-tuning for Text Classification

ULMFit

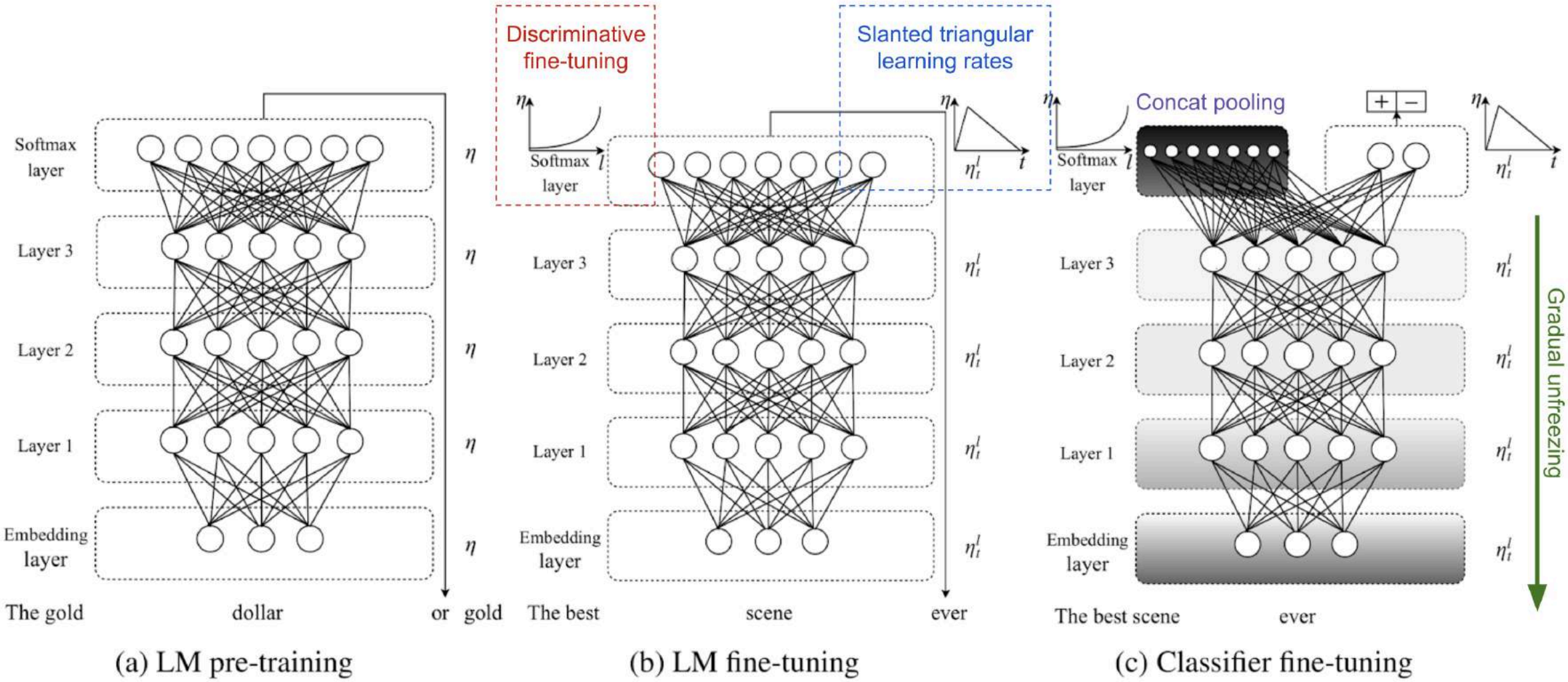
1) **General LM Pre-training:** on Wikipedia

2) **Target task LM fine-tuning**
   - **Discriminative fine-tuning:** Tune different layers with different LR
   - **Slanted triangular LR:** Use a customized cyclic LR

3) **Target task classifier fine-tuning:** 2 layers FFN + softmax
   - **Concat pooling:** extract max and mean over history of hidden states and concat them with final hidden states.
   - **Gradual unfreezing:** unfreeze layers one epoch at a time

© 2018 National University of Singapore. All Rights Reserved

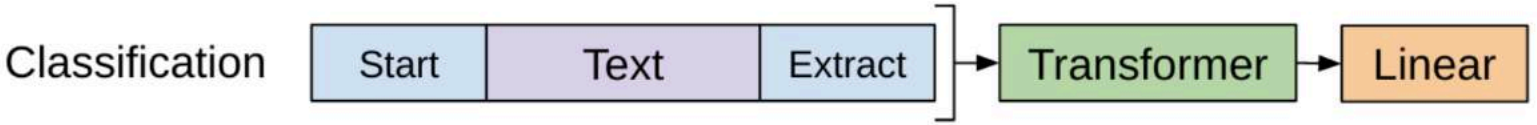
ULMFit


Discriminative fine-tuning
Slanted triangular learning rates
Concat pooling
Softmax layer
Layer 3
Layer 2
Layer 1
Embedding layer
Gradual unfreezing
The gold
dollar
or
gold
The best
scene
ever
The best scene
ever
(a) LM pre-training
(b) LM fine-tuning
(c) Classifier fine-tuning

OpenAI GPT: Improving Language Understanding by Generative Pre-Training

OpenAI GPT

There's no post-training FFN!!!



$$P(y \mid x_1, \ldots, x_n) = \text{softmax}(\mathbf{h}_L^{(n)} \mathbf{W}_y)$$

The loss is to minimize the negative log-likelihood for true labels. In addition, adding the LM loss as an auxiliary loss is found to be beneficial, because:

- (1) it helps accelerate convergence during training and
- (2) it is expected to improve the generalization of the supervised model.

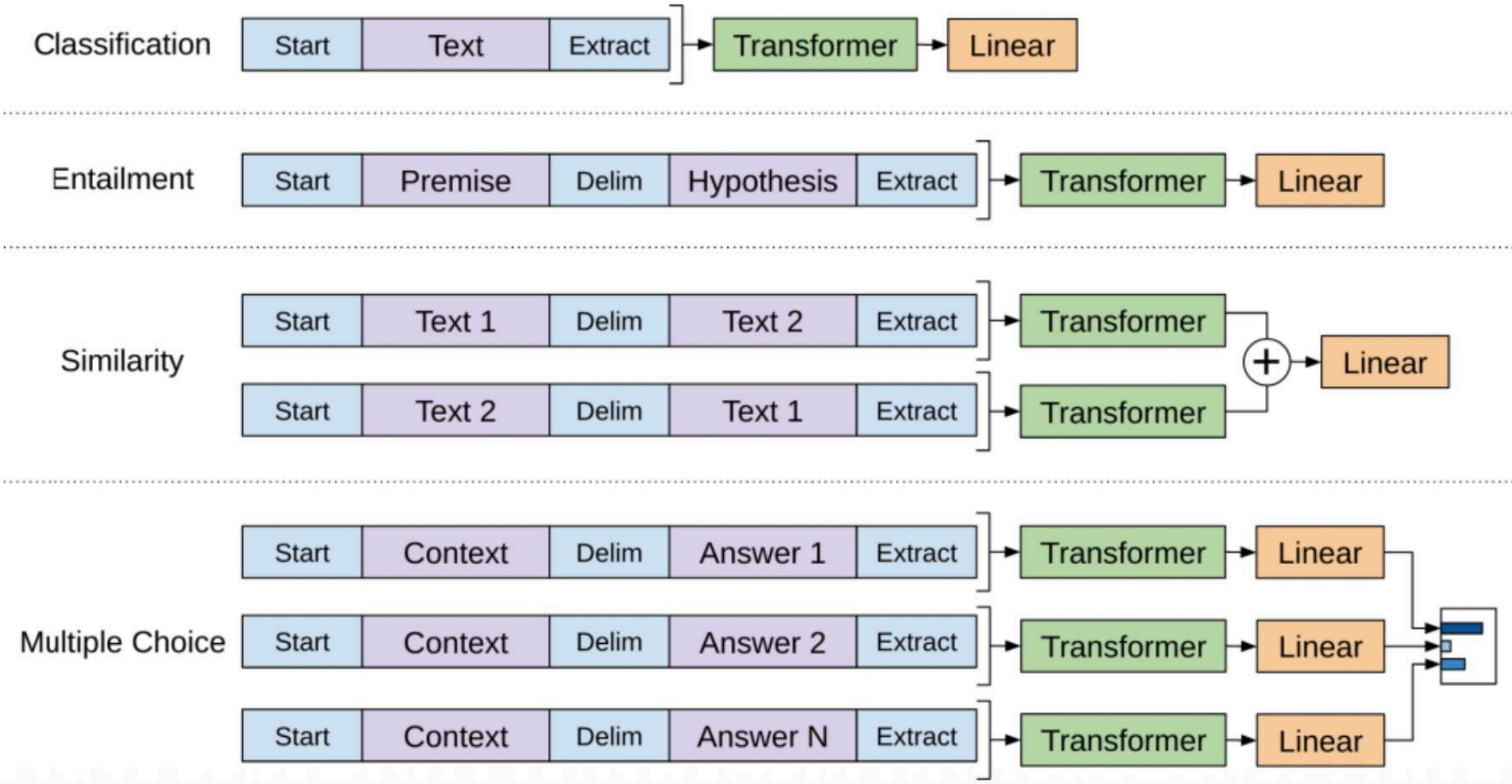
$$\mathcal{L}_{\text{cls}} = \sum_{(\mathbf{x},y)\in\mathcal{D}} \log P(y \mid x_1, \ldots, x_n) = \sum_{(\mathbf{x},y)\in\mathcal{D}} \log \text{softmax}(\mathbf{h}_L^{(n)}(\mathbf{x})\mathbf{W}_y)$$

$$\mathcal{L}_{\text{LM}} = -\sum_i \log p(x_i \mid x_{i-k}, \ldots, x_{i-1})$$

$$\mathcal{L} = \mathcal{L}_{\text{cls}} + \lambda \mathcal{L}_{\text{LM}}$$

© 2018 National University of Singapore. All Rights Reserved

OpenAI GPT


Classification
Start
Text
Extract
Transformer
Linear
Entailment
Start
Premise
Delim
Hypothesis
Extract
Transformer
Linear
Similarity
Start
Text 1
Delim
Text 2
Extract
Start
Text 2
Delim
Text 1
Extract
Transformer
Transformer
Linear
Multiple Choice
Start
Context
Delim
Answer 1
Extract
Start
Context
Delim
Answer 2
Extract
Start
Context
Delim
Answer N
Extract
Transformer
Transformer
Transformer
Linear
Linear
Linear

BERT: Bidirectional Encoder Representations from Transformers

BERT: Masked Language Model

It is unsurprising to believe that a representation that learns the context around a word rather than just after the word is able to better capture its meaning, both syntactically and semantically. BERT encourages the model to do so by training on the *"mask language model" task*:

1. Randomly mask 15% of tokens in each sequence. Because if we only replace masked tokens with a special placeholder `[MASK]`, the special token would never be encountered during fine-tuning. Hence, BERT employed several heuristic tricks:
   - (a) with 80% probability, replace the chosen words with `[MASK]`;
   - (b) with 10% probability, replace with a random word;
   - (c) with 10% probability, keep it the same.
2. The model only predicts the missing words, but it has no information on which words have been replaced or which words should be predicted. The output size is only 15% of the input size.

BERT: Next Sentence Prediction

Task 2: Next sentence prediction

Motivated by the fact that many downstream tasks involve the understanding of relationships between sentences (i.e., QA, NLI), BERT added another auxiliary task on training a *binary classifier* for telling whether one sentence is the next sentence of the other:

1. Sample sentence pairs (A, B) so that:
    - (a) 50% of the time, B follows A;
    - (b) 50% of the time, B does not follow A.
2. The model processes both sentences and output a binary label indicating whether B is the next sentence of A.

The training data for both auxiliary tasks above can be trivially generated from any monolingual corpus. Hence the scale of training is unbounded. The training loss is the sum of the mean masked LM likelihood and mean next sentence prediction likelihood.

BERT: Input Hacking

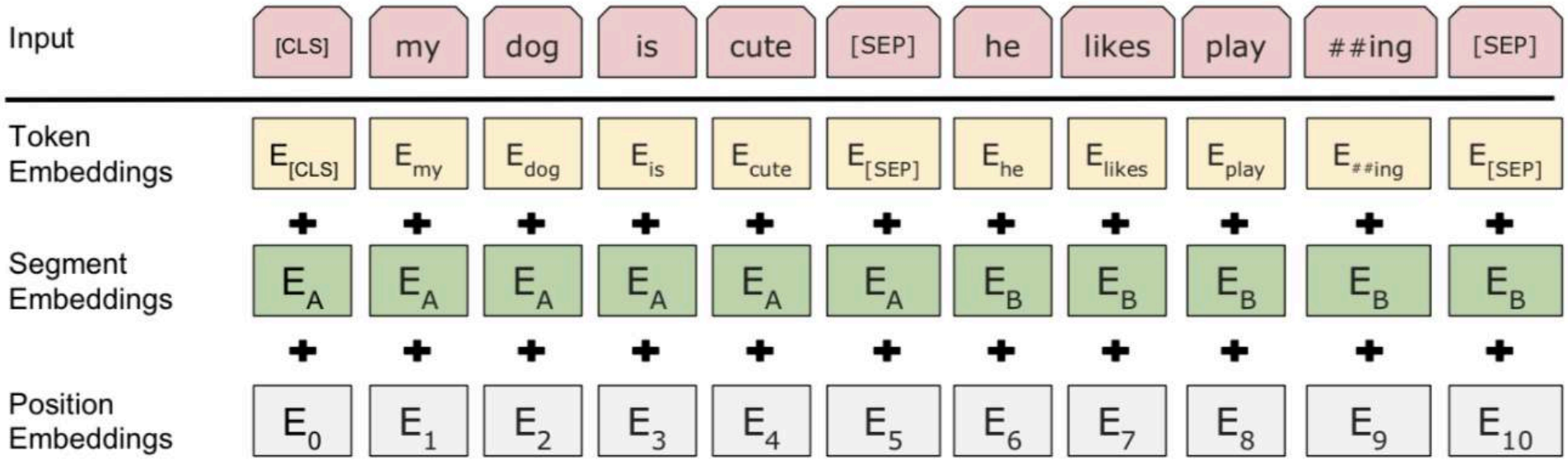
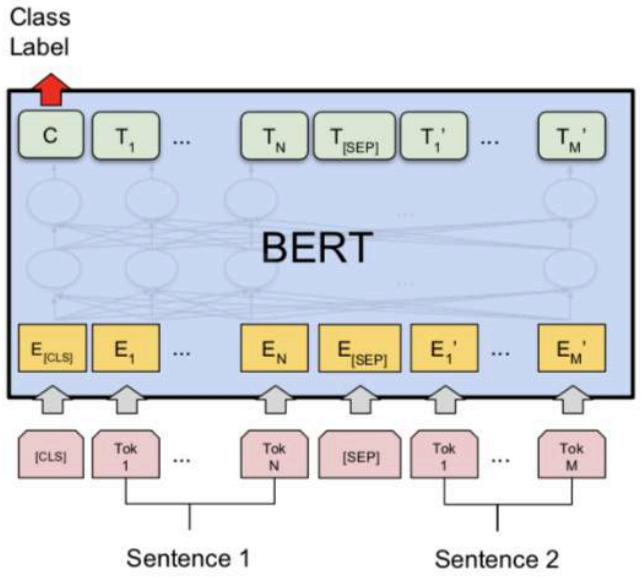


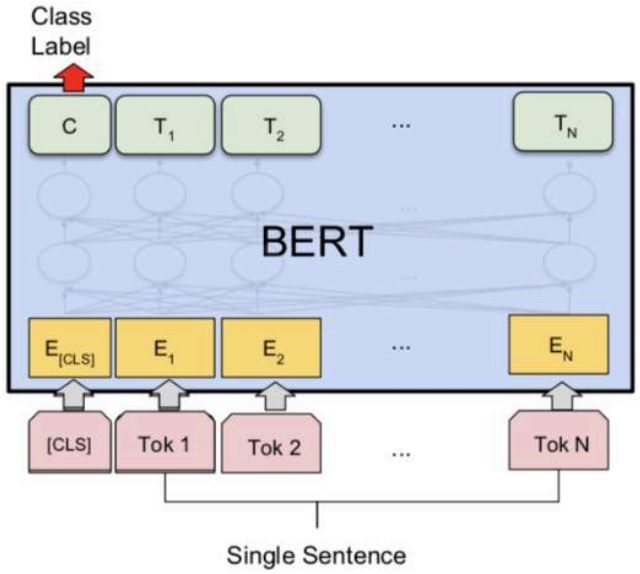
Fig. 11. BERT input representation. (Image source: original paper)

Note that the first token is always forced to be `[CLS]` — a placeholder that will be used later for prediction in downstream tasks.

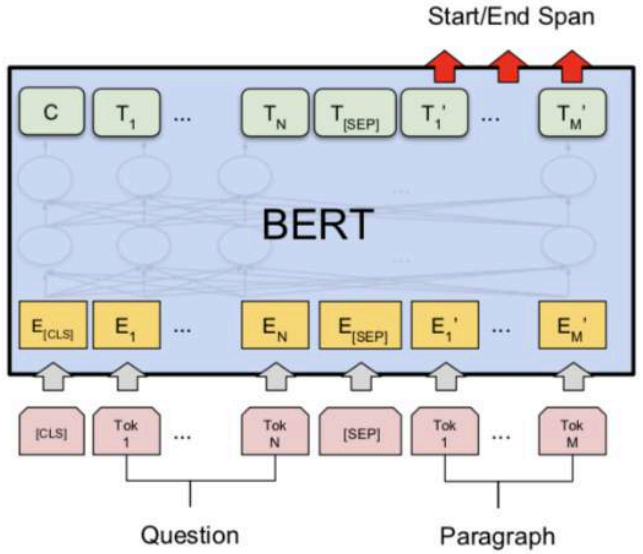
BERT



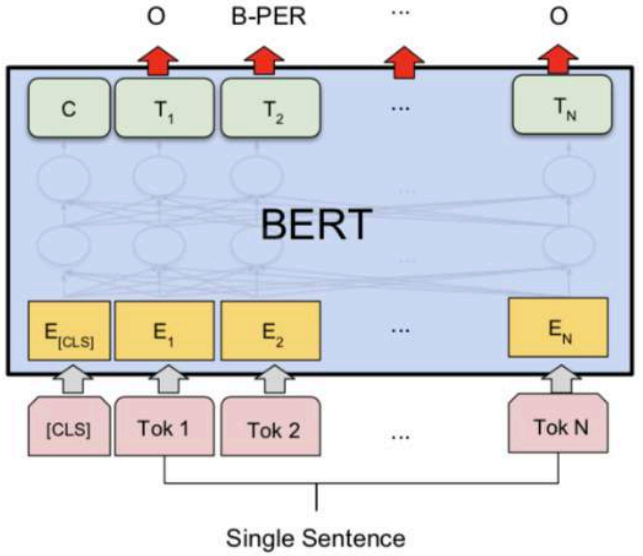
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tasks: SST-2, CoLA

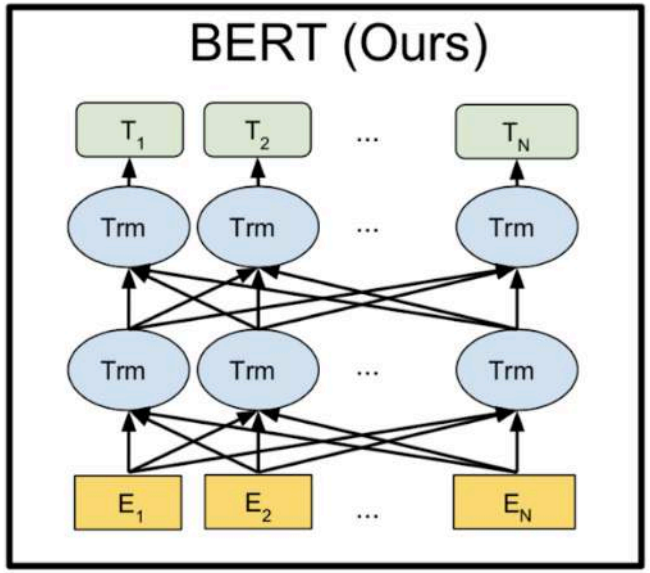


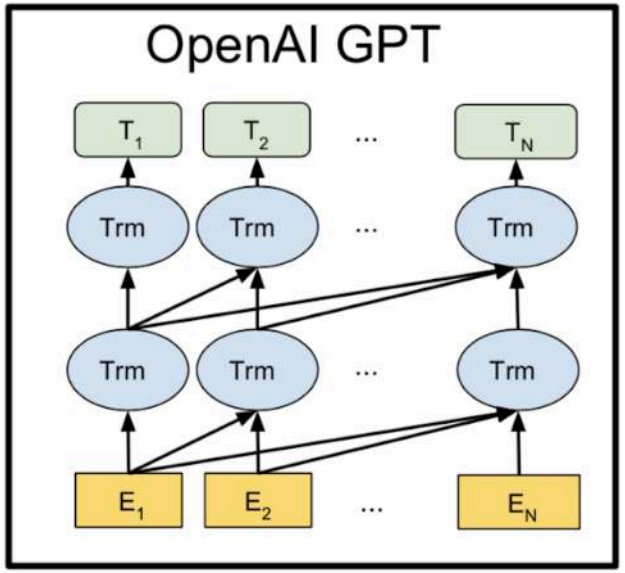
(c) Question Answering Tasks: SQuAD v1.1

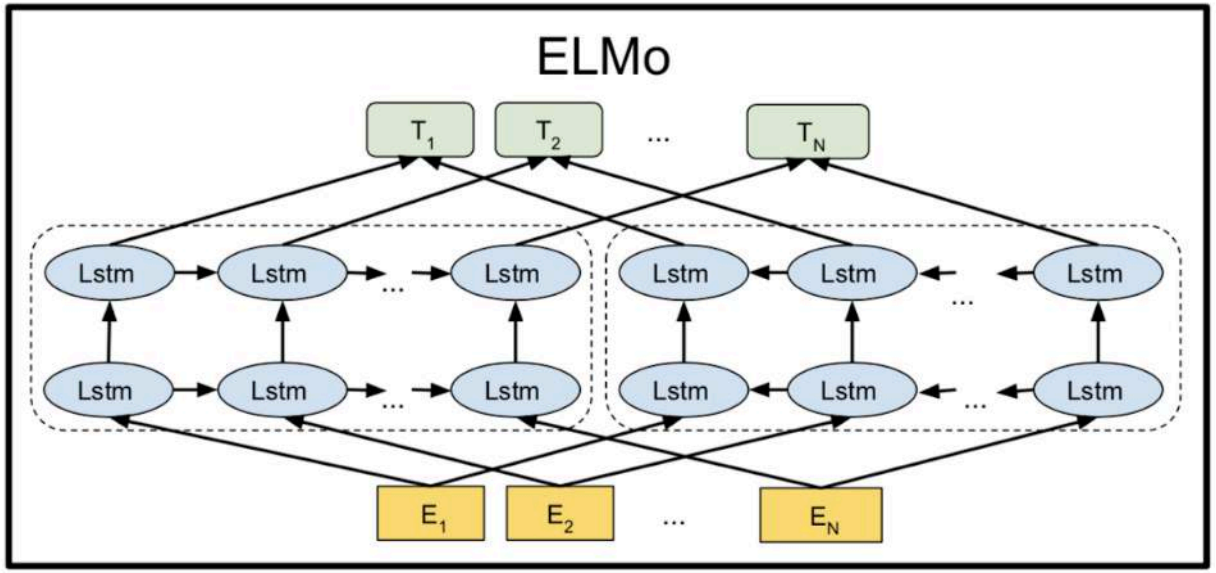


(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

BERT vs OpenAI GPT vs ELMo





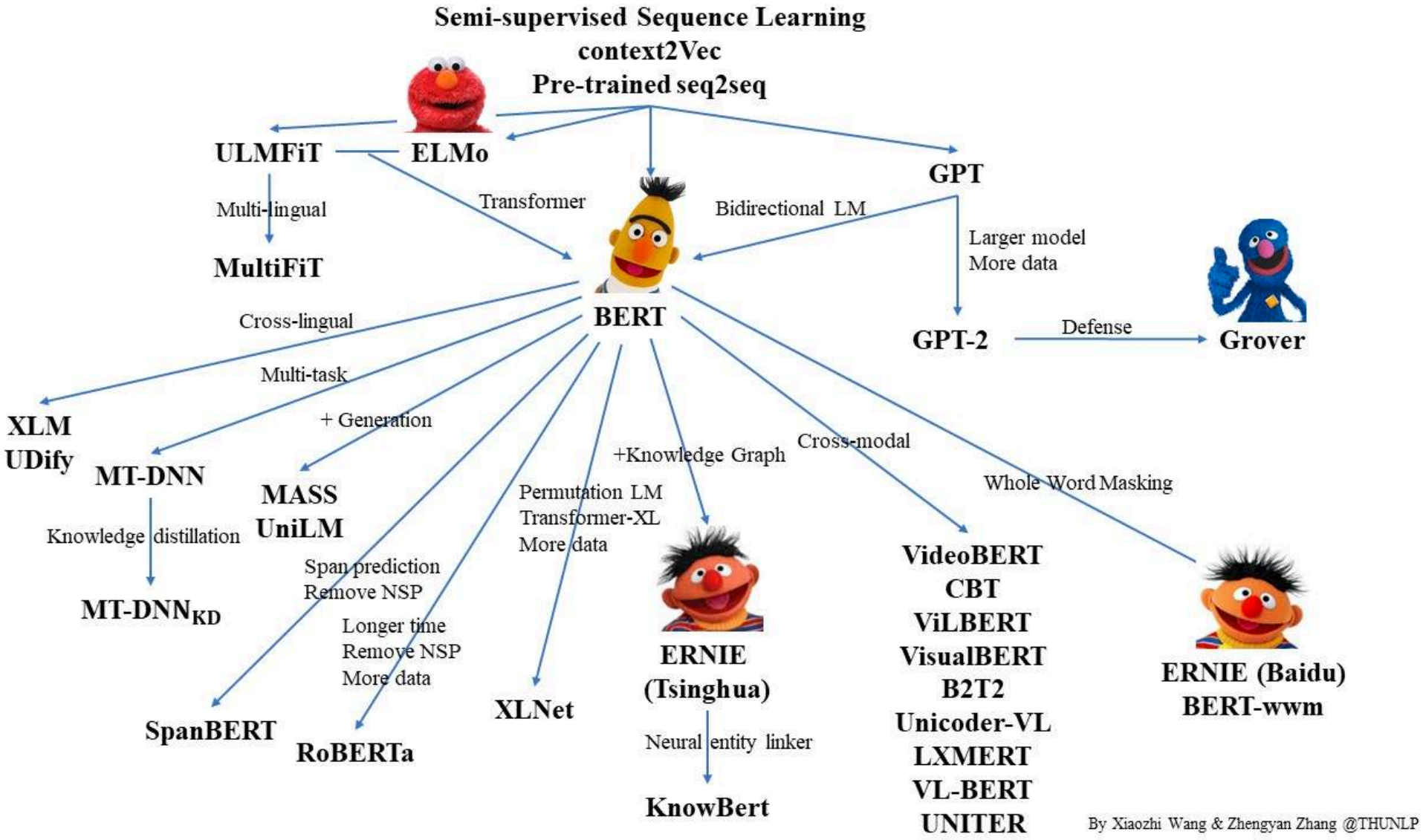


Summary: Sentence Representation

Summary: Generalized LM (Weng, 2019)

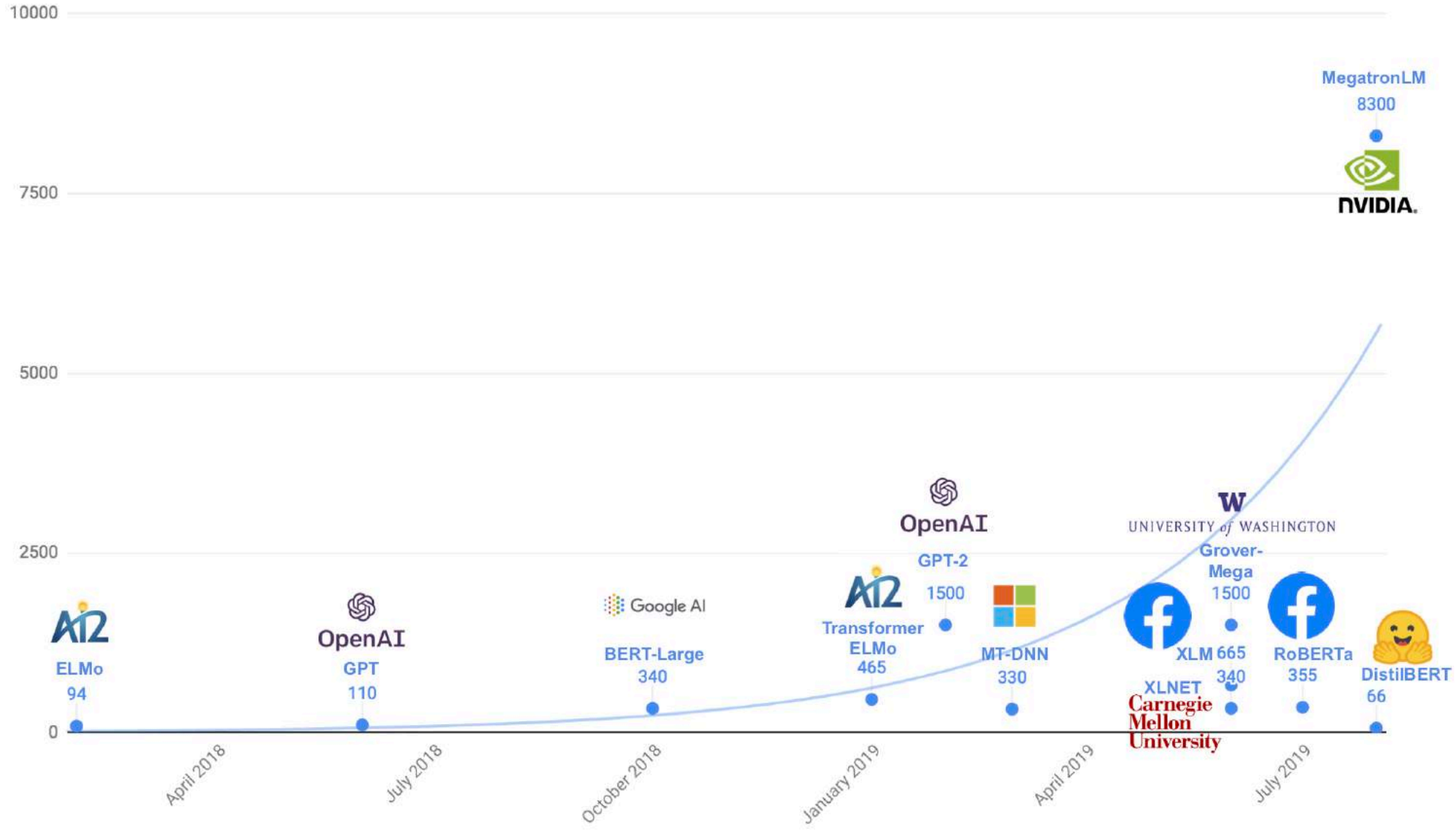
	Base model	pre-training	Downstream tasks	Downstream model	Fine-tuning
CoVe	seq2seq NMT model	supervised	feature-based	task-specific	/
ELMo	two-layer biLSTM	unsupervised	feature-based	task-specific	/
CVT	two-layer biLSTM	semi-supervised	model-based	task-specific / task-agnostic	/
ULMFiT	AWD-LSTM	unsupervised	model-based	task-agnostic	all layers; with various training tricks
GPT	Transformer decoder	unsupervised	model-based	task-agnostic	pre-trained layers + top task layer(s)
BERT	Transformer encoder	unsupervised	model-based	task-agnostic	pre-trained layers + top task layer(s)
GPT-2	Transformer decoder	unsupervised	model-based	task-agnostic	pre-trained layers + top task layer(s)

Summary: Pre-trained LM (THU-NLP)



Source: https://github.com/thunlp/PLMpapers

DistilBERT (Sanh et al. 2019)



Source: https://medium.com/huggingface/distilbert-8cf3380435b5

Fin

BERT Basics: Convert text to `array(float)`

[16, 83, 42] [19, 73, 21] [79, 12, 54] [27, 9, 54]

Sushi → is → yummy → </s>

is
[19, 73, 21]

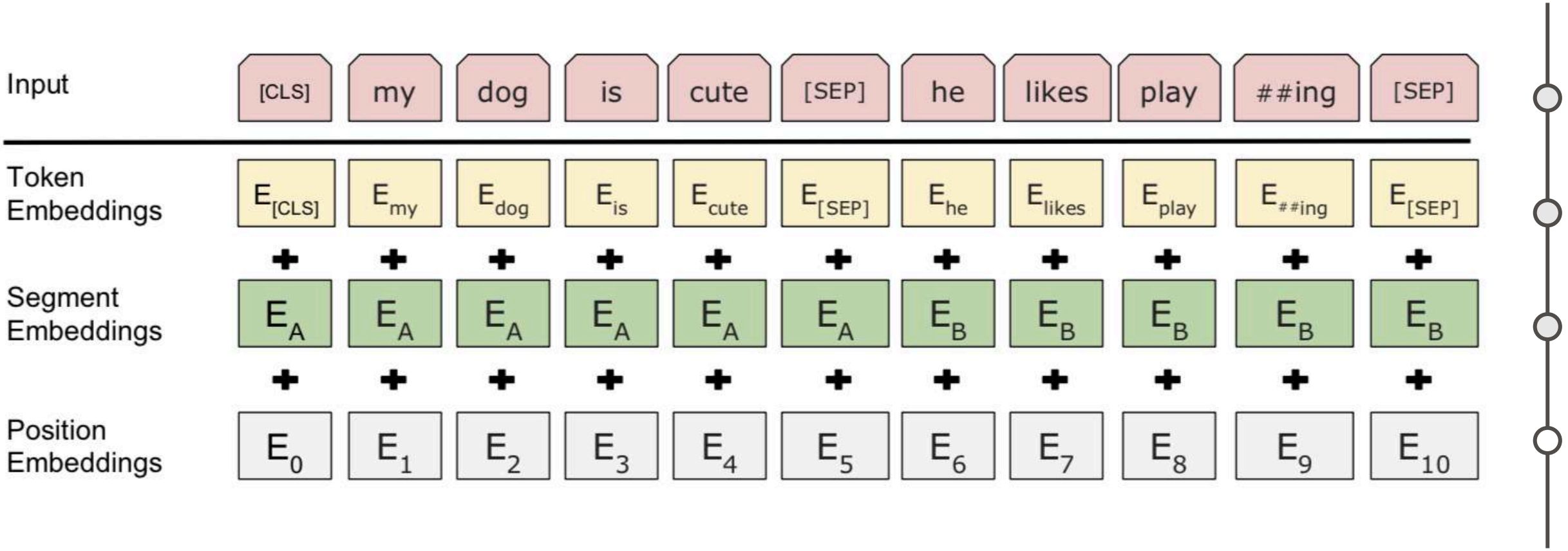
spicy
[79, 32, 87]

egg
[72, 43, 25]

yummy
[79, 12, 54]

hot
[91, 33, 62]

BERT Basics: Inputs



BERT Basics: WordPiece Inputs

```
from transformers import BertTokenizer, BertModel, BertForMaskedLM

# Load pre-trained model tokenizer (vocabulary)
# A tokenizer will split the text into the appropriate sub-parts (aka. tokens).
# Depending on how the pre-trained model is trained, the tokenizers defers.
tokenizer = BertTokenizer.from_pretrained('bert-large-uncased')

# Example of a tokenized input after WordPiece Tokenization.
text = "[CLS] my dog is cute [SEP] he likes playing [SEP]"
print(tokenizer.tokenize(text))
```

```
['[CLS]', 'my', 'dog', 'is', 'cute', '[SEP]', 'he', 'likes', 'playing', '[SEP]']
```

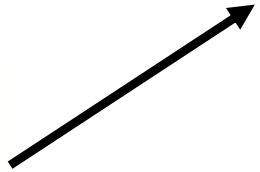
BERT Basics: WordPiece Inputs

```
from transformers import BertTokenizer, BertModel, BertForMaskedLM

# Load pre-trained model tokenizer (vocabulary)
# A tokenizer will split the text into the appropriate sub-parts (aka. tokens).
# Depending on how the pre-trained model is trained, the tokenizers defers.
tokenizer = BertTokenizer.from_pretrained('bert-large-uncased')

# Example of a tokenized input after WordPiece Tokenization.
text = "[CLS] my dog is cute [SEP] he likes playing [SEP]"
print(tokenizer.tokenize(text))
```

```
['[CLS]', 'my', 'dog', 'is', 'cute', '[SEP]', 'he', 'likes', 'playing', '[SEP]']
```



Meh… Don’t look like the example you have above -_-|||

BERT Basics: WordPiece Inputs

```
"playing" in tokenizer.wordpiece_tokenizer.vocab
```

:

```
True
```

```
print("slacking" in tokenizer.wordpiece_tokenizer.vocab)
```

```
False
```

```
text = "[CLS] my dog is cute [SEP] he likes slacking [SEP]"
tokenized_text = tokenizer.tokenize(text)  # There, we see the ##ing token!
print(tokenized_text)
```

```
['[CLS]', 'my', 'dog', 'is', 'cute', '[SEP]', 'he', 'likes', 'slack', '##ing', '[SEP]']
```

BERT Basics: Segment Indices

```
import numpy as np

# We need to create an array that indicates the end of sentences, delimited by [SEP]
text = "[CLS] my dog is cute [SEP] he likes slacking [SEP]"
tokenized_text = tokenizer.tokenize(text)  # There, we see the ##ing token!

# First we find the indices of `[SEP]`, and incrementally adds it up.
# Here's some Numpy gymnastics...
# Thanks to @divakar https://stackoverflow.com/a/58316889/610569
m = np.asarray(tokenized_text) == "[SEP]"
segments_ids = m.cumsum()-m
segments_ids
```

:

```
array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
```

We do some numpy gymnastics to get the **segement indices** to indicate the span of the sentences in the text.

BERT Basics: Segment Indices

```
import numpy as np

# We need to create an array that indicates the end of sentences, delimited by [SEP]
text = "[CLS] my dog is cute [SEP] he likes slacking [SEP]"
tokenized_text = tokenizer.tokenize(text)  # There, we see the ##ing token!

# First we find the indices of `[SEP]`, and incrementally adds it up.
# Here's some Numpy gymnastics...
# Thanks to @divakar https://stackoverflow.com/a/58316889/610569
m = np.asarray(tokenized_text) == "[SEP]"
segments_ids = m.cumsum()-m
segments_ids
```

```
array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
```

We do some numpy gymnastics to get the **segement indices** to indicate the span of the sentences in the text.

BERT Basics: PyTorch Tensors

```
import numpy as np

# We need to create an array that indicates the end of sentences, delimited by [SEP]
text = "[CLS] my dog is cute [SEP] he likes slacking [SEP]"
tokenized_text = tokenizer.tokenize(text)  # There, we see the ##ing token!

# First we find the indices of `[SEP]`, and incrementally adds it up.
# Here's some Numpy gymnastics...
# Thanks to @divakar https://stackoverflow.com/a/58316889/610569
m = np.asarray(tokenized_text) == "[SEP]"
segments_ids = m.cumsum()-m

tokens_tensor, segments_tensors = torch.tensor([token_indices]), torch.tensor([segments_ids])

# See the type change?
print(tokens_tensor.shape, type(token_indices), type(tokens_tensor))
print(segments_tensors.shape, type(segments_ids), type(segments_tensors))
```

```
torch.Size([1, 11]) <class 'list'> <class 'torch.Tensor'>
torch.Size([1, 11]) <class 'numpy.ndarray'> <class 'torch.Tensor'>
```

BERT Basics: PyTorch Tensors

```
import numpy as np

# We need to create an array that indicates the end of sentences, delimited by [SEP]
text = "[CLS] my dog is cute [SEP] he likes slacking [SEP]"
tokenized_text = tokenizer.tokenize(text)  # There, we see the ##ing token!

# First we find the indices of `[SEP]`, and incrementally adds it up.
# Here's some Numpy gymnastics...
# Thanks to @divakar https://stackoverflow.com/a/58316889/610569
m = np.asarray(tokenized_text) == "[SEP]"
segments_ids = m.cumsum()-m

tokens_tensor, segments_tensors = torch.tensor([token_indices]), torch.tensor([segments_ids])

# See the type change?
print(tokens_tensor.shape, type(token_indices), type(tokens_tensor))
print(segments_tensors.shape, type(segments_ids), type(segments_tensors))
```

```
torch.Size([1, 11]) <class 'list'> <class 'torch.Tensor'>
torch.Size([1, 11]) <class 'numpy.ndarray'> <class 'torch.Tensor'>
```

BERT Basics: Model Loading

```
import torch

device = 'cuda' if torch.cuda.is_available() else 'cpu'

# When using the BERT model for "encoding", i.e. convert string to array of floats,
# we use the `BertModel` object from pytorch transformer library.
model = BertModel.from_pretrained('bert-base-uncased')
model.eval(); model.to(device)
```

:

```
BertModel(
  (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
    (token_type_embeddings): Embedding(2, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
    (layer): ModuleList(
```

`model.eval()` indicates that you want to *use* the model

BERT Basics: Model Loading

```
import torch

device = 'cuda' if torch.cuda.is_available() else 'cpu'

# When using the BERT model for "encoding", i.e. convert string to array of floats,
# we use the `BertModel` object from pytorch transformer library.
model = BertModel.from_pretrained('bert-base-uncased')
model.eval(); model.to(device)
```

model.to()

sends the model into the CPU/GPU

:

```
BertModel(
  (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
    (token_type_embeddings): Embedding(2, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
    (layer): ModuleList(
```

BERT Basics: Encode Inputs

Similarly, we send our Tensor objects

.to(device)

```
# Predict hidden states features for each layer
with torch.no_grad():
    encoded_layers, _ = model(tokens_tensor.to(device), segments_tensors.to(device))

print(encoded_layers)
```

```
tensor([[[ 0.0737,  0.1229, -0.1612,  ...,  0.1235,  0.2381,  0.0880],
         [ 0.5442,  0.6884, -0.0744,  ..., -0.0346,  0.1388, -0.4474],
         [ 0.7117,  0.6298, -0.0084,  ...,  0.1541,  0.1260, -0.1969],
         ...,
         [ 0.6583,  0.4991, -0.1022,  ...,  0.2715, -0.2863, -0.4839],
         [-0.1440, -0.3067, -0.3331,  ...,  0.4313,  0.1080,  0.0214],
         [ 0.9325,  0.2506,  0.0805,  ...,  0.1434, -0.8708, -0.4659]]],
       device='cuda:0')
```

BERT Basics: Encode Inputs

When using the model,
i.e. `model.eval()`
we don't need
gradients information

```
# Predict hidden states features for each layer
with torch.no_grad():
    encoded_layers, _ = model(tokens_tensor.to(device), segments_tensors.to(device))

print(encoded_layers)
```

```
tensor([[[ 0.0737,  0.1229, -0.1612,  ...,  0.1235,  0.2381,  0.0880],
         [ 0.5442,  0.6884, -0.0744,  ..., -0.0346,  0.1388, -0.4474],
         [ 0.7117,  0.6298, -0.0084,  ...,  0.1541,  0.1260, -0.1969],
         ...,
         [ 0.6583,  0.4991, -0.1022,  ...,  0.2715, -0.2863, -0.4839],
         [-0.1440, -0.3067, -0.3331,  ...,  0.4313,  0.1080,  0.0214],
         [ 0.9325,  0.2506,  0.0805,  ...,  0.1434, -0.8708, -0.4659]]],
       device='cuda:0')
```

BERT Basics: Encode Inputs

model(tokens_tensor, segments_tensor)

"encodes" the inputs

```
# Predict hidden states features for each layer
with torch.no_grad():
    encoded_layers, _ = model(tokens_tensor.to(device), segments_tensors.to(device))

print(encoded_layers)
```

```
tensor([[[ 0.0737,  0.1229, -0.1612,  ...,  0.1235,  0.2381,  0.0880],
         [ 0.5442,  0.6884, -0.0744,  ..., -0.0346,  0.1388, -0.4474],
         [ 0.7117,  0.6298, -0.0084,  ...,  0.1541,  0.1260, -0.1969],
         ...,
         [ 0.6583,  0.4991, -0.1022,  ...,  0.2715, -0.2863, -0.4839],
         [-0.1440, -0.3067, -0.3331,  ...,  0.4313,  0.1080,  0.0214],
         [ 0.9325,  0.2506,  0.0805,  ...,  0.1434, -0.8708, -0.4659]]],
       device='cuda:0')
```

BERT Basics: Encode Inputs

```
print(encoded_layers)
```

```
tensor([[[ 0.0737,  0.1229, -0.1612,  ...,  0.1235,  0.2381,  0.0880],
         [ 0.5442,  0.6884, -0.0744,  ..., -0.0346,  0.1388, -0.4474],
         [ 0.7117,  0.6298, -0.0084,  ...,  0.1541,  0.1260, -0.1969],
         ...,
         [ 0.6583,  0.4991, -0.1022,  ...,  0.2715, -0.2863, -0.4839],
         [-0.1440, -0.3067, -0.3331,  ...,  0.4313,  0.1080,  0.0214],
         [ 0.9325,  0.2506,  0.0805,  ...,  0.1434, -0.8708, -0.4659]]],
       device='cuda:0')
```

```
encoded_layers.shape
```

```
torch.Size([1, 11, 768])
```

The output tensor shape is 3-Dimension, i.e.

(batch_size, sequence_length, hidden_dimension)

BERT Basics: Encode Inputs

```
print(encoded_layers)
```

```
tensor([[[ 0.0737,  0.1229, -0.1612,  ...,  0.1235,  0.2381,  0.0880],
         [ 0.5442,  0.6884, -0.0744,  ..., -0.0346,  0.1388, -0.4474],
         [ 0.7117,  0.6298, -0.0084,  ...,  0.1541,  0.1260, -0.1969],
         ...,
         [ 0.6583,  0.4991, -0.1022,  ...,  0.2715, -0.2863, -0.4839],
         [-0.1440, -0.3067, -0.3331,  ...,  0.4313,  0.1080,  0.0214],
         [ 0.9325,  0.2506,  0.0805,  ...,  0.1434, -0.8708, -0.4659]]],
       device='cuda:0')
```

```
encoded_layers.shape
```

```
torch.Size([1, 11, 768])
```

The output tensor shape is 3-Dimension, i.e.

(no. of sentences, lengths of sentences, model “hidden” info)

BERT Basics: Encode Inputs

```
# We need to create an array that indicates the end of sentences, delimited by [SEP]
text = "[CLS] my dog is cute [SEP] he likes slacking [SEP]"
tokenized_text = tokenizer.tokenize(text)  # There, we see the ##ing token!
m = np.asarray(tokenized_text) == "[SEP]"
segments_ids = m.cumsum()-m

tokens_tensor, segments_tensors = torch.tensor([token_indices]), torch.tensor([segments_ids])
print(tokens_tensor.shape, type(token_indices), type(tokens_tensor))
print(segments_tensors.shape, type(segments_ids), type(segments_tensors))
```

```
torch.Size([1, 11]) <class 'list'> <class 'torch.Tensor'>
torch.Size([1, 11]) <class 'numpy.ndarray'> <class 'torch.Tensor'>
```

```
encoded_layers.shape
```

```
torch.Size([1, 11, 768])
```

The inputs shape is 3-Dimension, i.e.

(no. of sentences, lengths of sentences, model “hidden” info)

BERT Basics: Convert text to `array(float)`

[16, 83, 42] [19, 73, 21] [79, 12, 54] [27, 9, 54]

Sushi → is → yummy → </s>

is
[19, 73, 21]

spicy
[79, 32, 87]

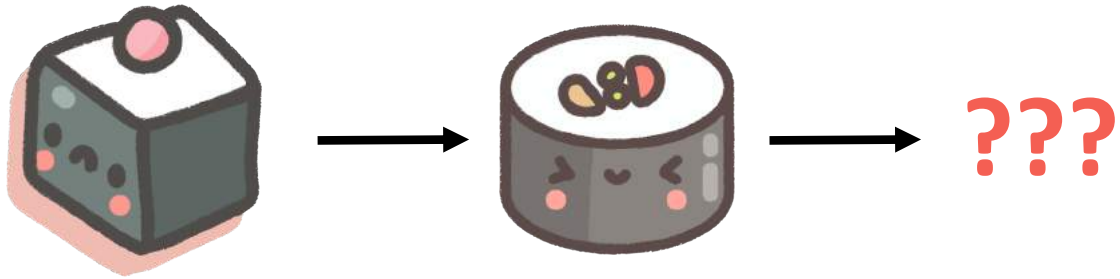
egg
[72, 43, 25]

yummy
[79, 12, 54]

hot
[91, 33, 62]

BERT Basics: How about Text Generaton?

[16, 83, 42] [19, 73, 21]



Transfered model learns

next dish to put on the conveyor

BERT Basics: What if we can "fill in the blanks"?

[16, 83, 42]

Sushi

???

???

[79, 12, 54]

yummy

[27, 9, 54]

</s>

is
[19, 73, 21]

spicy
[79, 32, 87]

egg
[72, 43, 25]

yummy
[79, 12, 54]

hot
[91, 33, 62]

BERT Basics: Masked Language Model

```
# Load the model.
model = BertForMaskedLM.from_pretrained('bert-base-uncased')
model.eval(); model.to(device)
```

```
BertForMaskedLM(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
```

Previously, we used **BertModel()** to *"encode tensors"*, here we use **BertForMaskedLM()**

BERT Basics: Masked Language Model

```
# We need to create an array that indicates the end of sentences, delimited by [SEP]
text = "[CLS] please don't let the [MASK] out of the [MASK] . [SEP]"
tokenized_text = tokenizer.tokenize(text)
token_indices = tokenizer.convert_tokens_to_ids(tokenized_text)

# Create the segment indices.
m = np.asarray(tokenized_text) == "[SEP]"
segments_ids = m.cumsum()-m

# Convert them to the arrays to pytorch tensors.
tokens_tensor, segments_tensors = torch.tensor([token_indices]), torch.tensor([segments_ids])

# Apply the model to the inputs.
with torch.no_grad(): # You can take this context manager to mean that we're not training.
    outputs, *_ = model(tokens_tensor.to(device),
                        token_type_ids=segments_tensors.to(device))

outputs.shape
```

:

```
torch.Size([1, 14, 30522])
```

BERT Basics: Masked Language Model

```
# We need to create an array that indicates the end of sentences, delimited by [SEP]
text = "[CLS] please don't let the [MASK] out of the [MASK] . [SEP]"
tokenized_text = tokenizer.tokenize(text)
token_indices = tokenizer.convert_tokens_to_ids(tokenized_text)

# Create the segment indices.
m = np.asarray(tokenized_text) == "[SEP]"
segments_ids = m.cumsum()-m

# Convert them to the arrays to pytorch tensors.
tokens_tensor, segments_tensors = torch.tensor([token_indices]), torch.tensor([segments_ids])

# Apply the model to the inputs.
with torch.no_grad(): # You can take this context manager to mean that we're not training.
    outputs, *_ = model(tokens_tensor.to(device),
                        token_type_ids=segments_tensors.to(device))

outputs.shape
```

```
torch.Size([1, 14, 30522])
```

Instead of having a full sentence, we have not two **[MASK]** words, i.e.“blanks” that needs to be filled

BERT Basics: Masked Language Model

```
# We need to create an array that indicates the end of sentences, delimited by [SEP]
text = "[CLS] please don't let the [MASK] out of the [MASK] . [SEP]"
tokenized_text = tokenizer.tokenize(text)
token_indices = tokenizer.convert_tokens_to_ids(tokenized_text)

# Create the segment indices.
m = np.asarray(tokenized_text) == "[SEP]"
segments_ids = m.cumsum()-m

# Convert them to the arrays to pytorch tensors
tokens_tensor, segments_tensors = torch.tensor([token_indices]), torch.tensor([segments_ids])

# Apply the model to the inputs.
with torch.no_grad(): # You can take this context manager to mean that we're not training.
    outputs, *_ = model(tokens_tensor.to(device),
                        token_type_ids=segments_tensors.to(device))

outputs.shape
```

Similar to **BertModel()** we do the tensors processing

:

```
torch.Size([1, 14, 30522])
```

BERT Basics: Masked Language Model

```
# We need to create an array that indicates the end of sentences, delimited by [SEP]
text = "[CLS] please don't let the [MASK] out of the [MASK] . [SEP]"
tokenized_text = tokenizer.tokenize(text)
token_indices = tokenizer.convert_tokens_to_ids(tokenized_text)

# Create the segment indices.
m = np.asarray(tokenized_text) == "[SEP]"
segments_ids = m.cumsum()-m

# Convert them to the arrays to pytorch tensors.
tokens_tensor, segments_tensors = torch.tensor([token_indices]), torch.tensor([segments_ids])

# Apply the model to the inputs.
with torch.no_grad(): # You can take this context manager to mean that we're not training.
    outputs, *_ = model(tokens_tensor.to(device),
                        token_type_ids=segments_tensors.to(device))

outputs.shape
```

Then we put the tensors through **`BertForMaskedLM()`** model

:

```
torch.Size([1, 14, 30522])
```

BERT Basics: Masked Language Model

```
# We need to create an array that indicates the end of sentences, delimited by [SEP]
text = "[CLS] please don't let the [MASK] out of the [MASK] . [SEP]"
tokenized_text = tokenizer.tokenize(text)
token_indices = tokenizer.convert_tokens_to_ids(tokenized_text)

# Create the segment indices.
m = np.asarray(tokenized_text) == "[SEP]"
segments_ids = m.cumsum()-m

# Convert them to the arrays to pytorch tensors.
tokens_tensor, segments_tensors = torch.tensor([token_indices]), torch.tensor([segments_ids])

# Apply the model to the inputs.
with torch.no_grad(): # You can take this context manager to mean that we're not training.
    outputs, *_ = model(tokens_tensor.to(device),
                        token_type_ids=segments_tensors.to(device))

outputs.shape
```

Et voila!!!

And output torch `Tensor` object

```
torch.Size([1, 14, 30522])
```

BERT Basics: Masked Language Model

```
# Apply the model to the inputs.
with torch.no_grad(): # You can take this context manager to mean that we're not training.
    outputs, *_ = model(tokens_tensor.to(device),
                        token_type_ids=segments_tensors.to(device))

outputs.shape
```

```
torch.Size([1, 14, 30522])
```

The inputs shape is also 3-Dimension as `BertModel()`,
but the last item is different for `BertForMaskedLM()` i.e.
(no. of sentences, lengths of sentences, *vocabulary size*)

BERT Basics: Masked Language Model

```
# Lets remember our original masked sentence.
print(tokenized_text)
# We have to check where the masked token is from the original text.
mask_index = tokenized_text.index('[MASK]')
assert mask_index == 7 # The 7th token.

# Then we fetch the vector for the 7th value,
# The [0, mask_index] refers to accessing vector of vocab_size for
# the 0th sentence, mask_index-th token.
output_value = outputs[0, mask_index]

# As a sanity check we can see that the shape of the output_value
# is the same as the `vocab_size` from the outputs' shape.
assert int(output_value.shape[0]) == len(tokenizer.wordpiece_tokenizer.vocab)
```

```
['[CLS]', 'please', 'don', "'", 't', 'let', 'the', '[MASK]', 'out', 'of', 'the', '[MASK]', '.', '[SEP]']
```

BERT Basics: Masked Language Model

```
# Lets remember our original masked sentence.
print(tokenized_text)
# We have to check where the masked token is from the original text.
mask_index = tokenized_text.index('[MASK]')
assert mask_index == 7 # The 7th token.

# Then we fetch the vector for the 7th value,
# The [0, mask_index] refers to accessing vector of vocab_size for
# the 0th sentence, mask_index-th token.
output_value = outputs[0, mask_index]

# As a sanity check we can see that the shape of the output_value
# is the same as the `vocab_size` from the outputs' shape.
assert int(output_value.shape[0]) == len(tokenizer.wordpiece_tokenizer.vocab)
```

Sanity check on the inner most *output dimensions* and the vocabulary size of the tokenizer

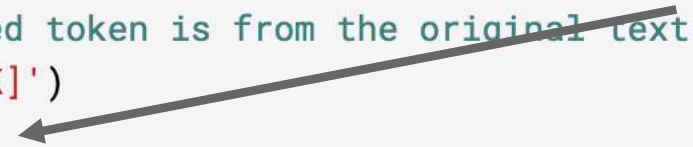
```
['[CLS]', 'please', 'don', "'", 't', 'let', 'the', '[MASK]', 'out', 'of', 'the', '[MASK]', '.', '[SEP]']
```

BERT Basics: Masked Language Model

```
# Lets recap the original sentence with the masked word.
print(text)

# We have to check where the first masked token is from the original text.
mask_index = tokenized_text.index('[MASK]')
output_value = outputs[0, mask_index]

## We use torch.argmax to get the index with the highest value.
mask_word_in_vocab = int(torch.argmax(output_value))
print(tokenizer.convert_ids_to_tokens([mask_word_in_vocab]))
```



Pick the last dimension of the output tensor, that's the same size of the *vocabulary size*

```
[CLS] please don't let the [MASK] out of the [MASK] . [SEP]
```

BERT Basics: Masked Language Model

```
# Lets recap the original sentence with the masked word.
print(text)

# We have to check where the first masked token is from the original text.
mask_index = tokenized_text.index('[MASK]')
output_value = outputs[0, mask_index]

## We use torch.argmax to get the index with the highest value.
mask_word_in_vocab = int(torch.argmax(output_value))
print(tokenizer.convert_ids_to_tokens([mask_word_in_vocab]))
```

Find the index from the vocabulary that has the largest value

```
[CLS] please don't let the [MASK] out of the [MASK] . [SEP]
```

BERT Basics: Masked Language Model

```
# Lets recap the original sentence with the masked word.
print(text)

# We have to check where the first masked token is from the original text.
mask_index = tokenized_text.index('[MASK]')
output_value = outputs[0, mask_index]

## We use torch.argmax to get the index with the highest value.
mask_word_in_vocab = int(torch.argmax(output_value))
print(tokenizer.convert_ids_to_tokens([mask_word_in_vocab]))
```

Convert the index back to string by looking into the tokenizer's vocab

```
[CLS] please don't let the [MASK] out of the [MASK] . [SEP]
```

BERT Basics: Masked Language Model

```
# Lets recap the original sentence with the masked word.
print(text)

# We have to check where the first masked token is from the original text.
mask_index = tokenized_text.index('[MASK]')
output_value = outputs[0, mask_index]

## We use torch.argmax to get the index with the highest value.
mask_word_in_vocab = int(torch.argmax(output_value))
print(tokenizer.convert_ids_to_tokens([mask_word_in_vocab]))
```

```
[CLS] please don't let the [MASK] out of the [MASK] . [SEP]
```

And the first [MASK] word is …

BERT Basics: Masked Language Model

```
# Lets recap the original sentence with the masked word.
print(text)

# We have to check where the first masked token is from the original text.
mask_index = tokenized_text.index('[MASK]')
output_value = outputs[0, mask_index]

## We use torch.argmax to get the index with the highest value.
mask_word_in_vocab = int(torch.argmax(output_value))
print(tokenizer.convert_ids_to_tokens([mask_word_in_vocab]))
```

```
[CLS] please don't let the [MASK] out of the [MASK] . [SEP]
['baby']
```

And the first [MASK] word is *"baby"*

BERT Basics: Masked Language Model

```
# Lets recap the original sentence with the masked word.
print(text)

# We have to check where the masked tokens are from the original text.
for mask_index, token in enumerate(tokenized_text):
    if token == '[MASK]':
        output_value = outputs[0, mask_index]
        mask_word_in_vocab = int(torch.argmax(output_value))
        print(tokenizer.convert_ids_to_tokens([mask_word_in_vocab]))
```

```
[CLS] please don't let the [MASK] out of the [MASK] . [SEP]
```

Lets loop through the tokens and fill in the `[MASK]`

BERT Basics: Masked Language Model

```
# Lets recap the original sentence with the masked word.
print(text)

# We have to check where the masked tokens are from the original text.
for mask_index, token in enumerate(tokenized_text):
    if token == '[MASK]':
        output_value = outputs[0, mask_index]
        mask_word_in_vocab = int(torch.argmax(output_value))
        print(tokenizer.convert_ids_to_tokens([mask_word_in_vocab]))
```

```
[CLS] please don't let the [MASK] out of the [MASK] . [SEP]
['baby']
```

And the second `[MASK]` word is ...

BERT Basics: Masked Language Model

```
# Lets recap the original sentence with the masked word.
print(text)

# We have to check where the masked tokens are from the original text.
for mask_index, token in enumerate(tokenized_text):
    if token == '[MASK]':
        output_value = outputs[0, mask_index]
        mask_word_in_vocab = int(torch.argmax(output_value))
        print(tokenizer.convert_ids_to_tokens([mask_word_in_vocab]))
```

```
[CLS] please don't let the [MASK] out of the [MASK] . [SEP]
['baby']
['way']
```

And the second [MASK] word is *"way"*

BERT Basics: Masked Language Model

```
def fill_in_the_blanks(text, model, tokenizer, return_str=False):
    tokenized_text = tokenizer.tokenize(text)
    token_indices = tokenizer.convert_tokens_to_ids(tokenized_text)
    # Create the segment indices.
    m = np.asarray(tokenized_text) == "[SEP]"
    segments_ids = m.cumsum()-m
    # Convert them to the arrays to pytorch tensors.
    tokens_tensor = torch.tensor([token_indices]).to(device)
    segments_tensors = torch.tensor([segments_ids]).to(device)

    # Apply the model to the inputs.
    with torch.no_grad(): # You can take this context manager to mean that we're not training.
        outputs, *_ = model(tokens_tensor, token_type_ids=segments_tensors)

    output_tokens = []
    for mask_index, token_id in enumerate(token_indices):
        token = tokenizer.convert_ids_to_tokens([token_id])[0]
        if token == '[MASK]':
            output_value = outputs[0, mask_index]
            # The masked word index in the vocab.
            mask_word_in_vocab = int(torch.argmax(output_value))
            token = tokenizer.convert_ids_to_tokens([mask_word_in_vocab])[0]
        output_tokens.append(token)

    return " ".join(output_tokens).replace(" ##", "").replace(" ' t ", "'t ") if return_str else output_tokens
```

仕組化 !!

(*shikumika*)

Lets put it together in one function!!

BERT Basics: Masked Language Model

```
# Load the model.
model = BertForMaskedLM.from_pretrained('bert-base-uncased')
model.eval(); model.to(device)

text = "[CLS] please don't let the [MASK] out of the [MASK] . [SEP]"
print(fill_in_the_blanks(text, model, tokenizer, return_str=True))
```

```
[CLS] please don't let the baby out of the way . [SEP]
```

BERT Basics: Masked Language Model

```
# Load the model.
model = BertForMaskedLM.from_pretrained('bert-base-uncased')
model.eval(); model.to(device)

text = "[CLS] please don't let the [MASK] out of the [MASK] . [SEP]"
print(fill_in_the_blanks(text, model, tokenizer, return_str=True))
```

```
[CLS] please don't let the baby out of the way . [SEP]
```

```
text = "[CLS] i like to drink beer and eat [MASK] . [SEP]"
print(fill_in_the_blanks(text, model,tokenizer, return_str=True))
```

BERT Basics: Masked Language Model

```
# Load the model.
model = BertForMaskedLM.from_pretrained('bert-base-uncased')
model.eval(); model.to(device)

text = "[CLS] please don't let the [MASK] out of the [MASK] . [SEP]"
print(fill_in_the_blanks(text, model, tokenizer, return_str=True))
```

```
[CLS] please don't let the baby out of the way . [SEP]
```

```
text = "[CLS] i like to drink beer and eat [MASK] . [SEP]"
print(fill_in_the_blanks(text, model,tokenizer, return_str=True))
```

```
[CLS] i like to drink beer and eat meat . [SEP]
```

BERT Basics: Masked Language Model

```
# Load the model.
model = BertForMaskedLM.from_pretrained('bert-base-uncased')
model.eval(); model.to(device)

text = "[CLS] please don't let the [MASK] out of the [MASK] . [SEP]"
print(fill_in_the_blanks(text, model, tokenizer, return_str=True))
```

```
[CLS] please don't let the baby out of the way . [SEP]
```

```
text = "[CLS] i like to drink beer and eat [MASK] . [SEP]"
print(fill_in_the_blanks(text, model,tokenizer, return_str=True))
```

```
[CLS] i like to drink beer and eat meat . [SEP]
```

```
text = "[CLS] i like to drink coffee and eat [MASK] . [SEP]"
print(fill_in_the_blanks(text, model, tokenizer, return_str=True))
```

```
[CLS] i like to drink coffee and eat it . [SEP]
```

BERT Basics: What if we can "fill in the blanks"?

[16, 83, 42]

Sushi

→ ??? →

[79, 12, 54]

[MASK]

→

[27, 9, 54]

yummy → </s>

is
[19, 73, 21]

spicy
[79, 32, 87]

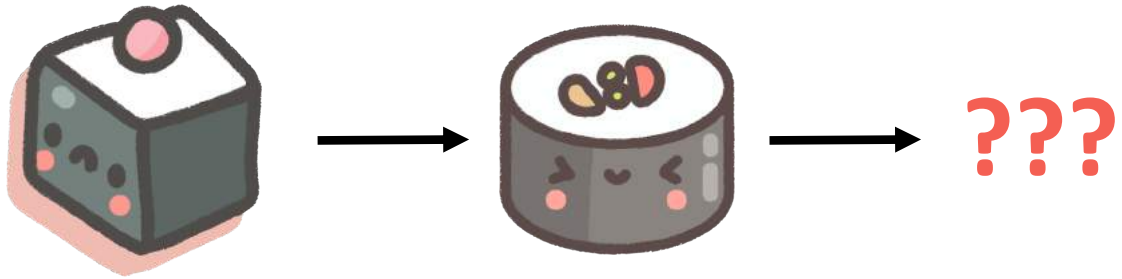
egg
[72, 43, 25]

yummy
[79, 12, 54]

hot
[91, 33, 62]

BERT Basics: How about Text Generaton?

[16, 83, 42] [19, 73, 21]



BERT Basics: How about Text Generaton?

[16, 83, 42] [19, 73, 21]

[MASK]

Simplest method, just fill in a [MASK] !!!