AMSI VACATION RESEARCH SCHOLARSHIPS 2019–20

EXPLORE THE MATHEMATICAL SCIENCES THIS SUMMER

Study and Application of Concept-Extraction Algorithms in Statistical and Programming Sciences

Ana-Maria Vintila

Supervised by Professor Brenda Vo

University of New England, Australia

Vacation Research Scholarships are funded jointly by the Department of Education and Training and the Australian Mathematical Sciences Institute.

Abstract

Knowledge era today churns out and repeats information in the form of documents and media, rather than linking by concept and semantics. However, since concept-mapping enhances human cognition, students and AI-systems could exchange knowledge more successfully and readily using knowledge graphs instead of plain text. This project examines how recent natural language processing (NLP) algorithms use co-occurrences but also more advanced techniques like masking, autoregressive language models, and named entity recognition (NER) to extract lexical, syntactic, and semantic information from real-world text. Models use the word sense disambiguation (WSD) task to represent polysemy, and this is a key step towards concept extraction. Using linked concepts in this document and PyTorch code, we use a fine-grained approach to examine how architectures of state-of-the-art models like BERT, XLNet, and ERNIE 1.0 improve over previous attempts (Word2Vec) in capturing multiple word senses, towards better natural language understanding.

Contents

1 Introduction: Motivation for Text Processing

Vast amounts of knowledge are trapped in presentation media such as videos, html, pdfs, and paper as opposed to being concept-mapped, interlinked, addressable and reusable at fine grained levels. This defeats knowledge exchanges between humans, human cognition and AI-based systems. Especially in domain-specific areas of knowledge, better interlinking would be achieved if concepts would be extracted using context, polysemy, and key phrases. "You shall know a word by the company it keeps" (Firth, 1957).

Previous models like GloVe and Word2Vec motivated recent models like Transformer, ELMo, BERT, Transformer-XL, XLNet, and ERNIE 1.0 to move beyond simple co-occurrence counts to extract meaning. For instance, **ERNIE 2.0** "broadens the vision to include more lexical, syntactic and semantic information from training corpora in form of *named entities* (like person names, location names, and organization names), *semantic closeness* (proximity of sentences), *sentence order or discourse relations*" (Sun et al., 2019).

Aims of this Research

1. *"Study"* - I will inventory, study, and compare architectures and frameworks to learn how they leverage entities, polysemy and contextual meaning for future study in concept extraction and natural language understanding.
2. *"Application"* - Using state of the art NLP frameworks like PyTorch, AllenNLP, and Thinc, I aim to illustrate key model architecture while applying the model to machine translation (MT).

Statement of Authorship: This report is planned, coded and written entirely by me, and I cite authors where applicable.

2 Word Embeddings

2.1 Usage of Word Embeddings in Natural Language Processing

Word embeddings are fixed-length vector representations of words that have led to the success of many NLP systems in recent years, across tasks like named entity recognition (NER), semantic parsing (SP), part of speech tagging (POS), and semantic role labeling (SRL) (Luong et al. 2013, p. 1).

A key idea in NLP is suggests that information lives in text corpora and people and machines can use programs to collect and organize this information for use in NLP.

A second key idea in linguistics is that words used in similar ways have similar meanings (Firth, 1957). Therefore, a distributional view of word meaning arises when accounting for the full distribution of contexts in a corpus where the word is found. These "distributed representations of words in a vector space help learning algorithms to achieve better performance in natural language processing tasks by grouping similar words" (Mikolov et al. 2013a, p. 1). An example is generalization from one sentence to a class of similar sentences, such as "the wall is blue" to "the ceiling is red" (Smith, 2019, p. 4).

2.1.1 Key Concept: Distributed Representation

In a **distributed representation** of a word, information of that word is distributed across vector dimensions (Lenci, 2018). This is opposed to a local word representation, like the n-gram model which uses short context.

2.2 Intuitive Definition of Word Embeddings

In the world of natural language processing, word embeddings are a collection of unsupervised learning methods for capturing semantic and syntactic information about individual words in a compact low-dimensional vector representation. These learned representations are then useful for reasoning about word usage and meaning (Melamud et al. 2016, p. 1). Tokenization is a key step in segmenting text to create various kinds of word embeddings, like sentence, phrase, and character embeddings, as the difference between BERT and Transformer-XL will show.

2.2.1 Analogical Reasoning Property of Word Embeddings

Word embeddings can represent analogies. For example, gender differences can be represented by a constant difference vector, enabling mathematical operations between vectors based on ***vector space semantics*** (Colah, 2014). "Man is to woman as king is to queen" can be expressed using learned word vectors: $vector(man) - vector(woman) = vector(king) - vector(queen)$ (Smith, 2019). In machine translation (MT), this property suggests the two languages being translated have a similar 'shape' and by forcing them to line up at different points, they overlap and other points get pulled into the right positions" (Colah, 2014).

2.3 Mathematical Definition For Word Embeddings

A word embedding $W : [Words] \to R^n$ is a parametrized function mapping words in a language to an n-dimensional numeric vector.

From Rudolph et al. (2016), "each term in a vocabulary is associated with two latent vectors, an *embedding* and a *context vector*. These two types of vectors govern conditional probabilities that relate each word to its surrounding context." Rudolph and Blei (2017) explain that a word's conditional probability combines its *embedding* and *context vectors* of surrounding words, with different methods combining them differently. Subsequently, word embeddings are fitted to text by maximizing conditional probabilities of observed text.

2.4 Static Embeddings vs. Contextual Embeddings

2.4.1 What is Polysemy?

Polysemy means a word can have distinct meanings. A related concept in NLP we will also use is the **distributional hypothesis**, which states meaning depends on context, and words occurring in same contexts have similar meaning (Wiedemann et al. 2019).

2.4.2 The Problem With Context-Free, Static Embeddings

Classic word vectors, also called **static embeddings**, represent words in a low-dimensional continuous space by assigning a single vector per word, regardless of context (Ethayarajh, 2019). Skip-Gram (Mikolov et al., 2013a) and GloVe (Pennington et al., 2014) produce these "context-independent representations," as Peters et al. (2018) call them, since their word embedding matrix is trained to use co-occurring information in text, rather than the more dynamic language modeling approach (Batista, 2018). Although static embeddings still capture latent syntactic and semantic meaning by training over large corpora, they collapse all senses of a polysemous word into a single vector representation (Ethayarajh, 2019). For instance, the word "plant" would have an embedding that is the "average of its different contextual semantics relating to biology, placement, manufacturing, and power generation" (Neelakantan et al., 2015).

2.4.3 A Better Solution: Contextual Embeddings To Capture Polysemy

In different domains, context is defined differently. Rudolph et al. (2016) states that "each data point i has a *context* c, which is a set of indices of other data points." In linguistics, the data point is taken to be a word and the context is the sequence of surrounding words. In neural data, the data point is neuron activity at a specific time and context is surrounding neuron activity. In shopping data, the data point refers to a purchase and context can mean other items in a basket (Rudolph et al., 2016).

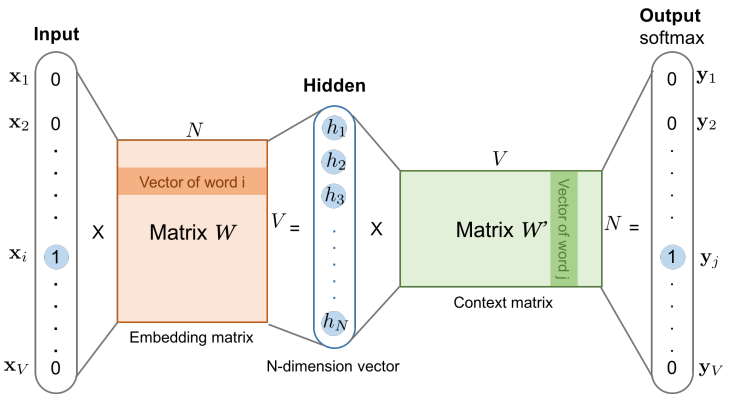


Figure 1: Simplified Skip-Gram Model with one input target word and one output context word. From *Learning Word Embeddings*, by Lilian Weng, 2017. https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html. Copyright 2017 by Weng.

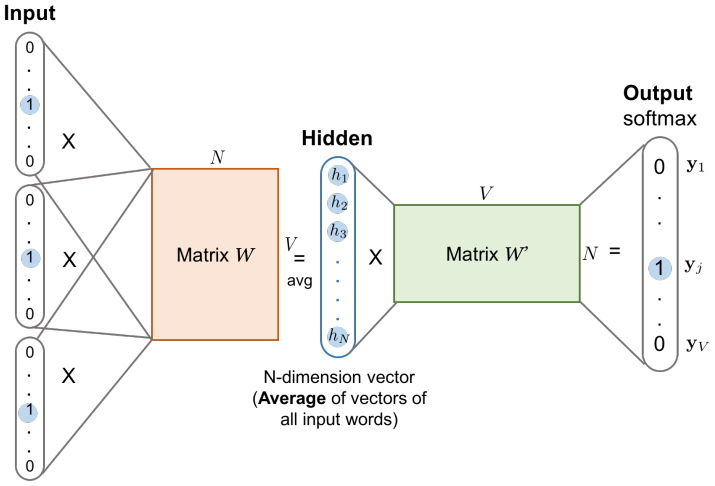


Figure 2: CBOW Model with several one-hot encoded context words at the input layer and one target word at the output layer. From *Learning Word Embeddings*, by Lilian Weng, 2017. https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html. Copyright 2017 by Weng.

A **contextual word embedding (CWE)** is usually obtained using a bidirectional language model (biLM) to capture forward and backward history from surrounding phrases of a word (Antonio, 2019). While static embeddings are simple "lookup tables", contextual embeddings contain word type information (Smith, 2019). CWEs avoid using a fixed word sense inventory, letting them: **(1)** create a vector per word, and **(2)** create a vector per word token in a context. Experimentally, CWEs capture word senses, letting Wiedemann et al. (2019) conclude CWEs are a more realistic model of natural language than static embeddings. Although contextualization models such as BERT, XLNet, and **ERNIE 2.0** differ widely, the field of NLP found that "sentence or context-level semantics together with word-level semantics proved to be a powerful innovation" (Wiedemann et al., 2019).

3 Word2Vec

3.1 Motivation for Word2Vec

Word2Vec is an unsupervised learning algorithm for obtaining word vector representations using a two-layer neural network. Existing word representations already capture linguistic patterns, allowing algebraic operations to be done on the word vectors in their semantic vector space. But Mikolov et al. (2013b) created **Word2Vec** to learn *efficient* representations from *large* data, as opposed to previous architectures that reduced quality of learned vectors by using smaller data and dimensionality.

Both the Skip-Gram and Continuous Bag of Words Model (CBOW) in Word2Vec are neural network language models with one hidden layer. Their input vector $\vec{x} = (x_1, \ldots, x_V)$ and output vector $\vec{y} = (y_1, \ldots, y_V)$ are both **one-hot encodings**, and the hidden layer of the neural network is a word embedding with dimension N.

3.1.1 Key Concept: One-Hot Encodings

A **one-hot vector encoding** is the simplest type of word embedding where each cell in the vector corresponds to a distinct vocabulary word. A 1 is placed in the cell marking the position of the word in the vocabulary, and 0 elsewhere. A problem with these is they lead to high-dimensional vectors for large vocabularies, raising computational costs. Secondly, they do not let similarity between words to be represented.

3.1.2 Skip-Gram

The Skip-Gram model predicts context words given a single target word. It uses a fixed sliding window to capture bidirectional context along a sentence, around a single target word. The target is input as a one-hot encoding to a neural network which updates the target vector with values near 1 in cells corresponding to predicted context words (Weng, 2017). The Skip-Gram is illustrated in fig. 1.

Consider the following sentence from Weng (2017): "The man who passes the sentence should swing the sword." Using context window size $c = 5$ and target word "swing", the Skip-Gram should learn to predict the context words `{"sentence", "should", "the","sword"}`, and so the target-context word pairs fed into the model for training are: `("swing","sentence")`, `("swing", "should")`, `("swing","the")`, and `("swing","sword")`.

3.1.3 Continuous Bag of Words Model (CBOW)

The **continuous bag of words model (CBOW)** is opposite to Skip-Gram since it predicts the *target* word based on a *context* word. Generally during training, CBOW receives a window of n context words around the target word w_t at each time step t to predict the target word (Mikolov et al., 2013b). The forward pass for CBOW is similar to Skip-Gram's, except CBOW averages context word vectors while multiplying INPUT $\vec{x}$ and the *input* $\rightarrow$ *hidden layer* matrix W. From Rong (2016), $\vec{h} = \frac{1}{c} W \cdot \left(\overrightarrow{x_1} + \overrightarrow{x_2} + \ldots + \overrightarrow{x_c}\right) = \frac{1}{c} \cdot \left(\overrightarrow{v_{w_1}} + \overrightarrow{v_{w_2}} + \ldots + \overrightarrow{v_{w_c}}\right)$, where c is the number of context words, $w_1, \ldots, w_c$ are the context words, and v_w is the input vector for general word w. According to Weng (2016), the fact that CBOW averages distributional information of the context vectors makes CBOW better suited for small datasets. The CBOW is shown in fig. 2.

3.2 Phrase-Learning in Word2Vec Using Skip-Gram

The problem with previous word vectors is their lack of phrase representation. "Canada" and "Air" could not be recognized as part of a larger concept and be combined into "Air Canada". Many phrases have meaning that is not just a composition of the meanings of its individual words and should be represented as a unique identity. In contrast, a bigram like "this is" should remain unchanged (Mikolov et al., 2013a, p. 5).

As an attempt to solve this, the **Phrase Skip-Gram** forms phrases based on the score $S_{phrase} = \frac{C(w_i w_j) - \delta}{C(w_i) C(w_j)}$, where $C(\cdot)$ is the count of a unigram w_i or bigram $w_i w_j$ and δ is a discounting threshold to avoid creating infrequent words and phrases. High values of S_{phrase} means the phrase is most likely a phrase rather than simple concatenation of two words. Mikolov et al. (2013a) found that this Phrase Skip-Gram with hierarchical softmax and subsampling outperformed the original model on large data.

Also, the Phrase Skip-Gram creates vectors exhibiting a linear structure called **additive compositionality**, which allows words to be combined meaningfully by adding their word vectors. Since learned context vectors can represent the overall distribution of context words in which the target word appears, and since the vectors in the loss function are logarithmically related to the probabilities from the output layer, the sum of two word vectors is related to the product of the context distributions (using the logarithm sum rule). This product of distributions acts like an AND function since words are weighted by probability. Consequently, if the key phrase "Volga River" appears many times in the same sentence along with "Russian" and "river", the sum $vector(\texttt{"Russian"}) + vector(\texttt{"river"})$ results in the phrase $vector(\texttt{"Volga River"})$ or at least a vector close to it (Mikolov et al., 2013a).

4 Transformer

The **Transformer model** introduced by Vaswani et al. (2017) for neural machine translation (NMT) proves more parallelizable than general seq-to-seq models with attention. Rather than recurrent neural networks (RNNs) combined with the **attention mechanism**, the Transformer sequence-to-sequence model is composed of only a **self attention mechanism** to attend to different input tokens for generating a sequence of contextual embeddings.

4.1 Self-Attention

4.1.1 Motivation for Self-Attention

"*The animal didn't cross the road because it was too tired.*"

What does "it" in this sentence refer to? Is "it" referring to the road or to the animal? This question may be simple to a human but not to a machine.

This is the motivation for using **self-attention**: when the Transformer processes the word "it", self-attention allows it to associate "it" with "animal". As the Transformer processes each word, self-attention allows it to look at other positions in the input sentence for clues on creating a better encoding for this word. In each layer, a part of the attention mechanism that focuses on "the animal" was *baked in* to a part of the representation of the word "it" (Trevett, 2020).

4.1.2 Query, Key, Value

Formally, "an ***attention function*** can be described as mapping a query and a set of key-value vectors pairs to an output. (Vaswani et al., 2017). The **Query matrix** Q contains row-wise information for which word to calculate self attention; the **Key matrix** K holds word vector representations on its rows, for *each* word in the sentence, and the **Value matrix** V contains vector row-wise information for the rest of the words in the sentence. Multiplying the query vector with the key vector of a particular word, stored in Q and K computes a result that indicates how much *value* vector V to consider.

For above example sentence, Q refers to "it"; V has vectors for words other than "it", and K has vectors for each word, including "it". The final calculated word embedding or **output** is a weighted sum of **value** vectors and softmax probabilities of the dot product between query and key vectors, as in eq. (1):

$$\texttt{Attention}\left(Q, K, V\right) = \texttt{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

Appendix: Self Attention Calculations in Transformer describes in complete detail the vector and matrix version calculations of self attention.

4.2 Multi-Head Attention

4.2.1 Motivation for Multi-Head Attention

A **multi-head attention mechanism** comprises of several self-attention heads, enabling the Transformer to "jointly attend to information from different representation subspaces at different positions." Instead of calculating attention once, multi-head attention does self attention many times in parallel on the projected dimensions, concatenates the independent attention outputs, and once again projects the result into the expected dimension to give a final value (Vaswani et al., 2017; Weng, 2018).

In the example sentence, when the Transformer encodes the word "it," one attention head may focus most on "the animal" while another head focuses on "tired", so the Transformer's "it" vector incorporates some representation of all the words in the sentence (Trevett, 2020).

4.2.2 Multi-Head Attention: Matrix Calculation

Using notation from Vaswani et al. (2017) and Alammar (2018b):

1. **Create Q, K, V matrices**: With multi-headed attention there are now separate Q, K, V weight matrices for each attention head h, in $1 \leq h \leq H$. Each row of input matrix X corresponds to a word in the input sentence. For each attention head, X is multiplied by trained parameter matrices to produce the separate query, key, value matrices:

$$Q_h = X \cdot W_h^Q \quad K_h = X \cdot W_h^K \quad V_h = X \cdot W_h^V$$

   where H = number of attention heads, and the dimensions of the parameter matrices are $W_h^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_h^K \in \mathbb{R}^{d_{model} \times d_k}$, and $W_h^V \in \mathbb{R}^{d_{model} \times d_v}$.

2. **Apply Softmax To Get Output Matrix**: Steps two through six in the vector calculation of self-attention can be condensed in a single matrix step to find the final output matrix Z_h for the h-th attention head for any self-attention layer:

$$Z_h := \texttt{softmax}\left(\frac{Q_h K_h^T}{\sqrt{d_k}}\right) \cdot V_h \tag{2}$$

3. **Concatenate Output Matrices**: Now there are H different output matrices Z for each attention head. But the feed-forward layer is only expecting a single matrix instead of H. So, this step concatenates the H matrices and multiplies the result by an additional weights matrix W^O to return to expected dimensions.

$$\texttt{MultiHeadAttention}(Q, K, V) = \texttt{Concat}(\texttt{head}_1, \ldots, \texttt{head}_H) \cdot W^O \tag{3}$$

   where $\texttt{head}_i = \texttt{Attention}(Q \cdot W_h^Q, K \cdot W_h^K, V \cdot W_h^V)$, where $\texttt{Attention}(Q, K, V) = \texttt{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V$ as before and $W^O \in \mathbb{R}^{H \cdot d_v \times d_{model}}$.

4.3 Positional Encodings

4.3.1 Motivation for Positional Encodings

Since the Transformer contains no recurrence mechanism it does not benefit from its resulting way of seeing order in the input sequence sentence. Instead, Vaswani et al. (2017) use **positional encodings** to inject absolute positional information of tokens into the sequence. Otherwise, the sentences "I like dogs more than cats" and "I like cats more than dogs" incorrectly would encode the same meaning (Raviraja, 2019).

A **positional encoding** follows a specific, learned pattern to identify word position or the distance between words in the sequence (Alammar, 2018b). They use sinusoidal waves to attend to relative positions since for any fixed offset k, the positional encoding $\texttt{PosEnc}_{pos+k}$ can be represented as a linear function of $\texttt{PosEnc}_{pos}$.

$$\texttt{PosEnc}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \qquad \texttt{PosEnc}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

4.4 Position-wise Feed Forward Layer

A **positionwise feed-forward layer** is a kind of feed-forward neural network (FNN) applied to each position separately and identically. The FFN contains two linear transformations with a $\texttt{ReLU}$ or $\texttt{max}$ activation function between them: $\texttt{FFN}(x) = \texttt{ReLU}(0, xW_1 + b_1)W_2 + b_2$

4.5 Residual Connection

A **residual connection** is a sub-layer in both Encoder and Decoder stacks which adds inputs to outputs of a sub-layer. This allows gradients during optimization to directly flow through a network rather than undergo transformation by nonlinear activations $\texttt{LayerNorm}(x + \texttt{Sublayer}(x))$, where $\texttt{Sublayer}(x)$ is a general function representing the sub-layer's operation. Each sub-layer in the stack, such as self-attention and positionwise feed forward layers, are surrounded by residual connection layers followed by layer normalization (Raviraja, 2019).

4.6 Masked Multi-Head Attention

The Decoder uses a sub-layer called **masked self attention**, which is an attention mechanism with **masking** to prevent positions from attending to subsequent positions. With masking, when the Decoder decodes embedding $\overrightarrow{w_i}$, it becomes unaware of words $\overrightarrow{w_{>i}}$ past position i and thus can only use words $\overrightarrow{w_{\leq i}}$. This protects the Transformer from cheating on predictions (Ta-Chun, 2018).

4.7 Encoder-Decoder Attention

This **encoder-decoder attention layer** is similar to multi-head self attention but it creates the query matrix $\boldsymbol{Q}$ from the layer below it (a Decoder self attention layer) and uses the key $\boldsymbol{K}$ and value $\boldsymbol{V}$ matrices from the Encoder stack's output (Alammar, 2018b).

4.8 Encoder

The Encoder is a **bidirectional recurrent network (RNN)**. The forward RNN reads the input sequence $\vec{x} = \{x_1, \ldots, x_{T_x}\}$ from left to right to produce a sequence of forward hidden states $\{\overrightarrow{h_1}, \ldots, \overrightarrow{h_{T_x}}\}$. The backward RNN reads the sequence in reverse order and returns a sequence of backward hidden states $\{\overleftarrow{h_1}, \ldots, \overleftarrow{h_{T_x}}\}$. Then, for each word x_t an annotation is obtained by concatenating the corresponding forward hidden state vector $\overrightarrow{h}_t$ with the backward one $\overleftarrow{h}_t$, such that $h_t = \left\{\overrightarrow{h}_t^T \; ; \; \overleftarrow{h}_t^T\right\}^T$, $t = 1, \ldots, T_x$. (Note: arrows here denote the direction of the network rather than vector notation.) This allows the annotation vector h_t for word x_t to contain bidirectional context (Bahdanau et al., 2016).

The Encoder is composed of N identical **Encoder layers**, which together are named the **Encoder stack**. A single **Encoder layer** is composed of the sub-layers **Multi-Head Attention** and **Position-wise Feed Forward Layer** (Trevett, 2020).

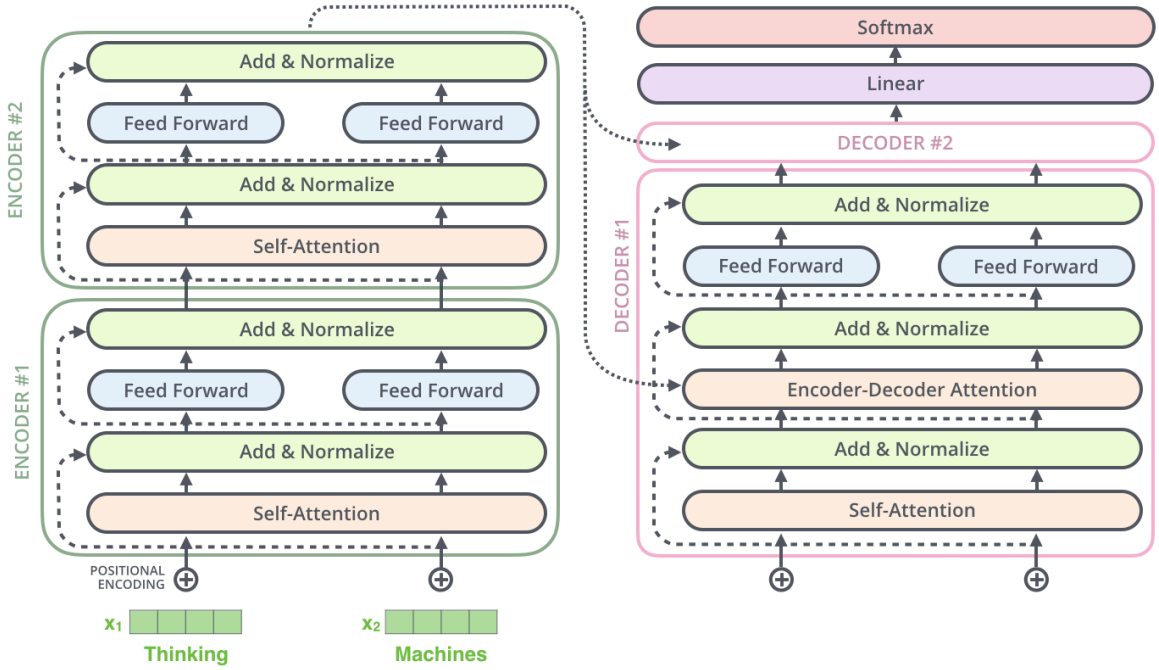


Figure 3: The layers inside Encoder and Decoder. From *The Illustrated Transformer*, by Alammar, 2018. https://jalammar.github.io/illustrated-transformer/. Copyright 2018 by Alammar.

4.9 Decoder

The Decoder neural network generates hidden states $s_t = \text{Decoder}\left(s_{t-1}, y_{t-1}, c_t\right)$ for time steps $t = 1, \ldots, m$ where the context vector $c_t = \sum_{i=1}^{n} \alpha_{ti} \cdot h_i$ is a sum of the hidden states of the input sentence, weighted by alignment scores, as for the Seq-to-Seq model (Weng, 2018).

Similarly to the Encoder, the Decoder contains a stack of N Decoder layers, each of which consists of three sub-layers: **Masked Multi-Head Attention**, and **Encoder-Decoder Attention**, and lastly **Position-wise Feed Forward Layer**.

4.10 Final Linear and Softmax Layer

The Decoder stack outputs a vector of floats. Then, the **Linear Layer**, a fully-connected neural network, projects the Decoder's output vector in a larger-dimensioned "logits vector" so each cell holds a score corresponding to each unique vocabulary word. After, a **Softmax Layer** then converts the Linear Layer's scores into probabilities using softmax function. To find the predicted word, the cell with highest probability is chosen, and corresponding word is called the predicted word, and is output for a particular time step.

5 ELMo

5.1 Combing Back To Polysemy

As previously explained, **polysemy** refers to a word's distinct meanings. Contextual embeddings outperform traditional embeddings on nlp tasks since they assign distinct word vectors per token given a context, as opposed to collapsing all senses within one vector (Wiedemann et al., 2019). Otherwise, homonyms like "book" (text) and "book" (reservation) would be assigned the same vector representation even though these are different words.

5.2 Motivation for ELMo

ELMo (Embeddings from Language Models) are "learned functions of the internal states of a bidirectional language model (biLM)" that is pretrained on a large corpus. ELMo representations are *deep* since they are derived from all the biLM's internal layers.

Higher-level LSTM layers in the ELMo model capture contextual meaning, useful for supervised word sense disambiguation (WSD), and lower layers capture syntax information, useful for part of speech tagging (POS). Mixing these signals allows the learned embeddings select the types of semi-supervision most needed for each end task, so ELMo embeddings end up richer than traditional embeddings (Peters et al., 2018).

5.3 Describing ELMo

"ELMo is a task-specific combination of the intermediate layer representations in the biLM" (Peters et al., 2018). In eq. (4), for each word token t_k, an L-layer biLM creates $2L + 1$ representations:

$$\begin{aligned} R_k &= \left\{ \mathbf{x}_k^{LM},\ \overrightarrow{\mathbf{h}}_{kj}^{LM},\ \overleftarrow{\mathbf{h}}_{kj}^{LM} \mid j = 1, \ldots, L \right\} \\ &= \left\{ \mathbf{h}_{kj}^{LM} \mid j = 1, \ldots, L \right\} \end{aligned} \tag{4}$$

where $\mathbf{h}_{k0}^{LM}$ is the token layer and the hidden state vector $\mathbf{h}_{kj}^{LM} = \left\{ \overrightarrow{\mathbf{h}}_{kj}^{LM},\ \overleftarrow{\mathbf{h}}_{kj}^{LM} \right\}$, a concatenation of backward and forward hidden states, for each bidirectional LSTM layer. ELMo model collapses all layers in the above vector into a single ELMo embedding ready for a specific task, by weighting the biLM layers in a task-specific way, as shown in eq. (5):

$$\mathbf{ELMo}_k^{task} = E\left(R_k; \theta^{task}\right) = \gamma^{task} \sum_{j=0}^{L} s_j^{task}\, \mathbf{h}_{kj}^{LM} \tag{5}$$

where the vector $\mathbf{s}^{task} = \left\{ s_j^{task} \right\}$ of softmax-normalized weights and task-dependent scalar parameter γ^{task} allow the model for the specific $task$ to scale the entire $\mathbf{ELMo}_k^{task}$ vector. The index k corresponds to a k-th word, and index j corresponds to the j-th layer out of L layers. Here, h_{kj}^{LM} is the output of the j-th LSTM for word k, and s_j is the weight of h_{kj}^{LM} used to compute the representation for word k (Peters et al., 2018). ELMo can be applied to specific tasks by concatenating the ELMo word embeddings with context-free word embeddings, such as those from GloVe, to be fed into a task-specific RNN for processing.

5.3.1 ELMo's Key Feature

	Source	Nearest Neighbors
GloVe	play	playing, game, games, played, players, plays, player, Play, football, multiplayer
biLM	Chico Ruiz made a spectacular play on Alusik 's grounder {...}	Kieffer , the only junior in the group , was commended for his ability to hit in the clutch , as well as his all-round excellent play .
	Olivia De Havilland signed to do a Broadway play for Garson {...}	{...} they were actors who had been handed fat roles in a successful play , and had talent enough to fill the roles competently , with nice understatement .

Table 1: Nearest neighbors to "play" using GloVe and the context embeddings from a biLM. From Table 4 in *Deep Contextualized Word Representations*, by Peters et al., 2018. https://arxiv.org/pdf/1802.05365.pdf. Copyright 2018 by Peters et al.

Peters et al. (2018) found that ELMo's biLM contextual vector outputs encode task-specific information that traditional word vectors do not capture.

To illustrate, consider table 1. The word "play" is highly polysemous since it has many different meanings. To contrast performance of traditional and ELMo embeddings on capturing polysemy, Peters et al. (2018) compared words nearest to "play" found using GloVe embeddings with those found using ELMo's biLM.

GloVe's neighbors include several different parts of speech, like verbs ("played", "playing"), and nouns ("player", "game") and only in the sport sense of the word. However, the bottom two rows show that the nearest neighbor sentences from the biLM's contextual embedding of "play" can disambiguate between *both* the parts of speech *and* word sense of "play". The last row's input sentence contains the noun / acting sense of "play" and this is matched in the nearest neighbor sentence, highlighting ELMo's strengths in part of speech tagging (POS) and word sense disambiguation (WSD).

6 BERT

6.1 Problem with ELMo

Unlike LSTMs, simple word embedding models cannot capture combinations of words, negation and polysemy. But language models have been effective at *sentence-level* nlp tasks like natural language inference and paraphrasing, which predict sentence relationships, and also *token-level* tasks like named entity recognition (NER) and question answering (QA), where a fine-grained approach is needed (Devlin et al., 2019).

Previous methods like **ULMFiT** and ELMo use a bidirectional language model (biLM) to account for left and right context. But the problem is that neither the forward LSTM nor the backward LSTM account for past and future tokens ***at the same time.*** As a result, information from the entire sentence is not used ***simultaneously*** regardless of token position, so the models do not perform that well.

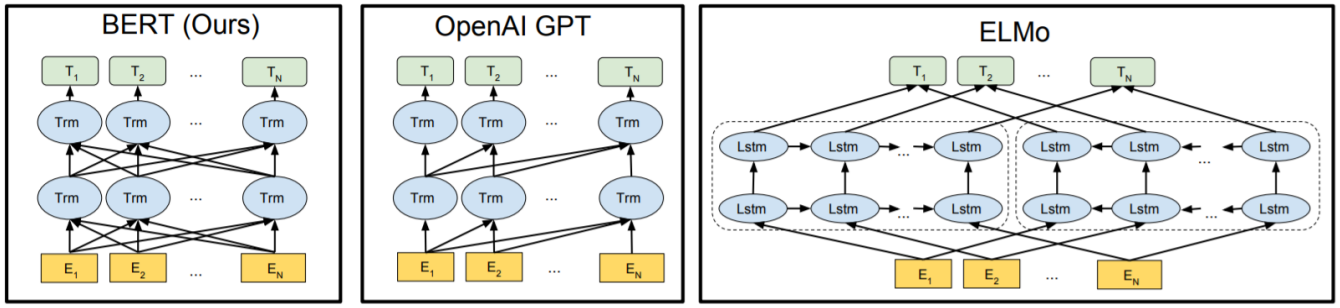


Figure 4: Comparing pre-training models: BERT uses a bidirectional Transformer. OpenAI GPT uses a forward Transformer. ELMo combines independently-trained forward and backward LSTMs. Among the three, only BERT embeddings are jointly conditioned on forward and backward context in all layers. Alongside architectural differences, BERT and OpenAI GPT are fine-tuning approaches, while ELMo is a feature-based approach (Devlin et al., 2019). *NOTE:* E_n *= the* n*-th token in the input sequence, and* T_n *= the corresponding output embedding.* From *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, by Devlin et al., 2019. https://arxiv.org/pdf/1810.04805.pdf. Copyright 2019 by Devlin et al.

6.2 Motivation for BERT

Instead of using ELMo's "shallow" combination of "independently-trained" biLMs, "BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on *both* left and right context in all layers" (Devlin et al., 2019). **BERT (Bidirectional Encoder Representations from Transformers)** combines a biLM, self attention, and Transformer (more powerful than LSTMs), and also a masked language model (MLM), yielding vast performance gains over previous models (Wiedemann et al., 2019).

6.3 Describing BERT

6.3.1 Input Embedding in BERT

The input embedding in BERT is created by summing three kinds of embeddings:

1. ***WordPiece* token embeddings:** The *WordPiece* tokenization strategy, instead of tokenizing by the natural separations between English words, subdivides words into smaller, basic units. For example, the WordPiece tokenization of "playing" might be "play" + "**ing". This allows BERT to handle rare, unknown words (Weng, 2019) and reduce vocabulary size while increasing amount of data available per word. For instance, if "play" and "**ing" and "**ed" are present in the vocabulary but "playing" and "played" are not, then these can be recognized by their sub-units.
2. **Segment embeddings:** are arbitrary spans of contiguous text made by packing sentence parts together. Contrary to BERT, Transformer-XL's **segment embeddings** respect sentence boundaries.
3. **Positional embeddings:** same as in the Transformer, to account for word ordering.

6.3.2 BERT's Framework

There are two steps in BERT's framework: **pre-training**, in which BERT trains on *unlabeled data* over different tasks, and **fine-tuning**, in which BERT is initialized with the pre-training parameters to train over *labeled data* for specific nlp tasks. Pre-training using the Masked Language Model (MLM) and Next Sentence Prediction (NSP) tasks allows BERT to learn **bidirectional context** and *sentence-level* information, as opposed to simple language models that predict subsequent tokens given context.

6.3.3 Masked Language Model (MLM)

It is a well-known problem that bidirectional conditioning causes lower layers to leak information about tokens, so each word can implicitly "see itself" letting the model trivially guess the target word in a multi-layered context (Devlin et al., 2019).

BERT's solution is to use a **masked language model (MLM)**, which randomly masks some input tokens to predict original tokens using only context. This fuses left and right context to get *deep* bidirectional context, unlike ELMo's shallow left-to-right language model (Devlin et al., 2019). However, the issue with masking is that it hampers BERT's performance. BERT only predicts the `[MASK]` token

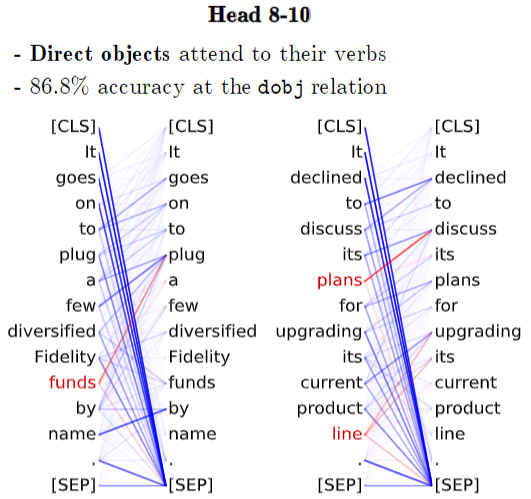


Figure 5: In heads 8-10, direct objects attend to their verbs. Line darkness indicates attention strength. Red indicates attention to/from red words, to highlight certain attentional behaviors. From *What Does BERT Look At? An Analysis of BERT's Attention*, by Clark et al., 2019. https://arxiv.org/abs/1906.04341. Copyright 2019 by Clark et al.

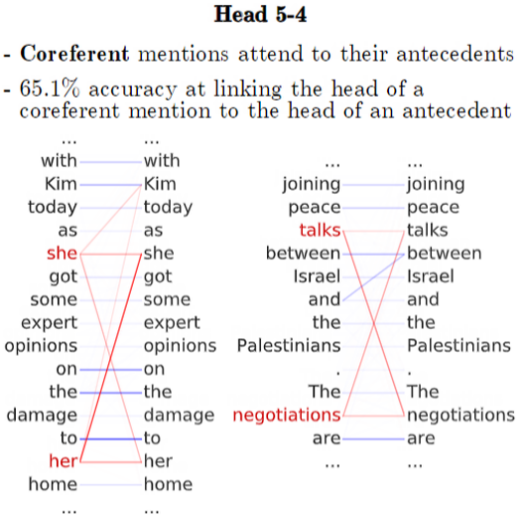


Figure 6: In heads 5-4, BERT does coreference resolution (CR) by linking the head of a coreferent mention to the head of an antecedent. From *What Does BERT Look At? An Analysis of BERT's Attention*, by Clark et al., 2019. https://arxiv.org/abs/1906.04341. Copyright 2019 by Clark et al.

when it is present in the input, even though BERT should predict correct tokens regardless of which tokens are present in input (Kurita, 2019a). As a solution, BERT varies its masking strategy: (1) replace the word to be masked with `[MASK]` only with 80% probability; (2) replace the word to be masked with a random word 10% of the time; and (3) keep the masked word 10% of the time.

6.3.4 Next Sentence Prediction (NSP)

Ordinary language models perform badly for tasks requiring sentence-level knowledge, like question answering (QA) and natural language inference (NLI). Thus, BERT pre-trains on a **next sentence prediction (NSP)** task to practice determining if one sentence is the next sentence of the other.

6.4 Experimental Results of BERT

Model	SST-2		SST-5	
	All	Root	All	Root
Avg word vectors [9]	85.1	80.1	73.3	32.7
RNN [8]	86.1	82.4	79.0	43.2
RNTN [9]	87.6	85.4	80.7	45.7
Paragraph vectors [2]	–	87.8	–	48.7
LSTM [10]	–	84.9	–	46.4
BiLSTM [10]	–	87.5	–	49.1
CNN [11]	–	87.2	–	48.0
$\text{BERT}_{\text{BASE}}$	94.0	91.2	83.9	53.2
$\text{BERT}_{\text{LARGE}}$	**94.7**	**93.1**	**84.2**	**55.5**

Table 2: Accuracy (%) of several models on sentiment classification (SC) SST dataset. BERT has highest accuracy scores. From *Table B.II in Fine-Grained Sentiment Classification Using BERT*, by Munikar et al., 2019. https://arxiv.org/pdf/1910.03474.pdf. Copyright 2019 by Munikar et al.

Using a simple accuracy measure, Munikar et al. (2019) found that a pre-trained BERT model fine-tuned for sentiment analysis (SA) task outperformed complex models such as RNNs and CNNs. The table 2 includes results on phrases and entire reviews.

This proves transfer learning is possible with BERT's deep contextual bidirectional language model.

6.5 Probing BERT

BERT has surpassed state-of-the-art performance in a wide range of nlp tasks but it is not known why. Clark et al. (2019) use an "attention-based probing classifier" to study BERT's internal vector representations to understand what kinds of linguistic features BERT learns from its self-supervised training on unlabeled data.

6.5.1 BERT Learns Dependency Syntax

Firstly, Clark et al. (2019) found that BERT's attention heads behave similarly, like focusing on positional offsets or attending broadly over an entire sentence.

Secondly, while individual attention heads do not capture syntax dependency structure as a whole, certain heads are better at detecting various syntax dependency relations. For example, the heads detect "direct objects of verbs, determiners of nouns, objects of prepositions, and objects of possessive pronouns with > 75% accuracy (Clark et al., 2019).

Attention heads 8-10 in fig. 5 learn how direct objects attend to their verbs, and achieves high accuracy at the direct object relation task. The fact that BERT learns this via self-supervision coupled with the heads' propensity for learning syntax may explain BERT's success. Attention heads 5-4 in fig. 6 perform coreference resolution (CR). This task is more challenging than syntax tasks since coreference links span longer than syntax dependencies, and even state-of-the-art models struggle at this task.

6.5.2 BERT's Contribution In Polysemy and Word Sense Disambiguation

Using a kNN classification, Wiedemann et al. (2019) compared the contextual embeddings of ELMo and BERT on word sense disambiguation (WSD) and found BERT places polysemic words into distinct regions according to their senses, while ELMo cannot since word senses in ELMo's embedding space are less sharply clustered.

Further, in table 3 Wiedemann et al. (2019) study how BERT matched word senses from the test set to given polysemic word from a training set. BERT predicted correctly when the text had *vocabulary overlap* like in example (2) "along the *bank* of the river" (the input text) and "along the *bank* of the river Greta" (nearest neighbor found by BERT). BERT also predicted correctly when text had *semantic overlap*, like in example (3) "little earthy bank" (input) "huge bank [of snow]" (nearest neighbor found by BERT).

However, BERT struggled when facing *vocabulary and semantic overlap in conjunction*. Example (5) in table 3 shows BERT predicted "land bank" as in a *supply or stock* but the correct sense of "land bank" was *financial institution*. Distinguishing verb senses was trickier: in example (12), the correct sense label of polysemic "watch" was *to look attentively* while BERT predicted its sense as *to follow with the eyes or the mind; observe*.

	Example sentence	Nearest neighbor
	SE-3 (train)	SE-3 (test)
(2)	Soon after setting off we came to a forested valley along the **banks**[Sloping Land] of the Gwaun.	In my own garden the twisted hazel, corylus avellana contorta, is underplanted with primroses, bluebells and wood anemones, for that is how I remember them growing, as they still do, along the **banks**[Sloping Land] of the rive Greta
(3)	In one direction only a little earthy **bank**[A Long Ridge] separates me from the edge of the ocean, while in the other the valley goes back for miles and miles.	The lake has been swept clean of snow by the wind, the sweepings making a huge **bank**[A Long Ridge] on our side that we have to negotiate.
(5)	He continued: assuming current market conditions do not deteriorate further, the group, with conservative borrowings, a prime land **bank**[A Financial Institution] and a good forward sales position can look forward to another year of growth.	Crest Nicholson be the exception, not have much of a land **bank**[Supply or Stock] and rely on its skill in land buying.
(12)	In between came lots of coffee drinking while **watching**[To Look Attentively] the balloons inflate and lots of standing around deciding who would fly in what balloon and in what order [...].	So Captain Jenks returned to his harbor post to **watch**[To Follow With the Eyes or the Mind; observe] the scouting plane put in five more appearances, and to feel the certainty of this dread rising within him.

Table 3: Example predictions by BERT based on nearest neighbor sentences. The polysemic word is **bolded**, and has a WordNet description tag describing its correct sense to be predicted. **True positives by BERT are green** while **false positives made by BERT are red**. From (adapted) Table 4 in *Does BERT Make Any Sense? Interpretable Word Sense Disambiguation with Contextualized Embeddings*, by Wiedemann et al., 2019. https://arxiv.org/pdf/1909.10430.pdf. Copyright 2019 by Wiedemann et al.

7 Transformer-XL

7.1 Problem With Transformer: Context-Fragmentation Problem

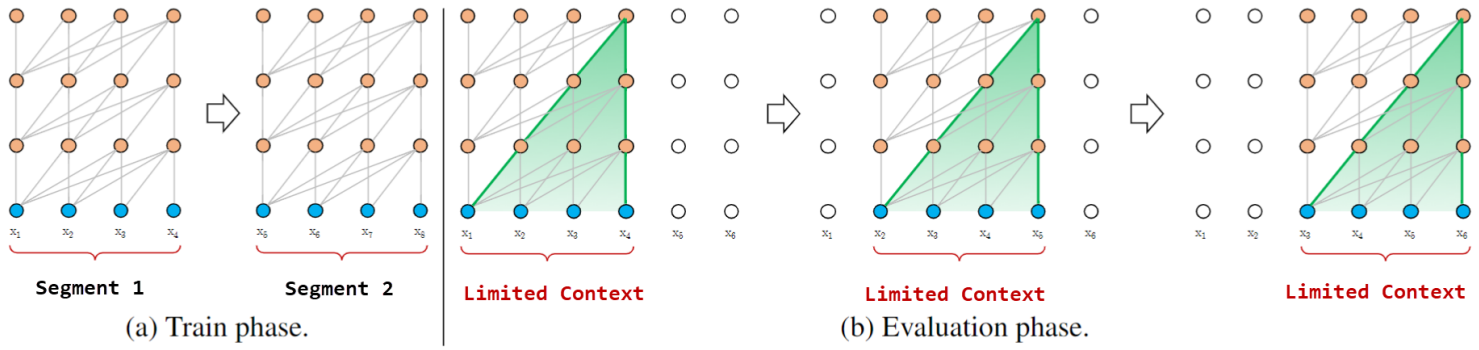


Figure 7: Vanilla Transformer with segment embedding length = 4. During each evaluation step, the Transformer consumes a segment embedding and makes a prediction at the last position. Next, the segment is shifted one position, so the new segment must be processed from scratch, breaking context dependency between first tokens of each segment and between segments. From *Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context*, by Dai et al., 2019. https://arxiv.org/pdf/1901.02860.pdf. Copyright 2019 by Dai et al.

Transformers cannot learn long-term dependencies because of **fixed-length context**, or context that is limited by input length (Dai et al., 2019). The *vanilla* (ordinary) Transformer model adapted by Al-Rfou et al. (2018) uses fixed-length context since the model is trained within its segment embeddings, blocking information from flowing across segment embeddings during forward and backward propagation passes. This result, visible in fig. 7, causes the Transformer to forget context from previous segments. This is termed the **context fragmentation problem**.

7.2 Motivation for Transformer-XL

Transformer-XL (extra long) avoids "disrupting temporal coherence." Instead of breaking up sequences into arbitrary **fixed lengths**, the Transformer-XL *respects natural language boundaries* like sentences and paragraphs, helping it gain richer context over these and longer texts like documents. Through the use of segment-level recurrence mechanism and relative positional encodings, the Transformer-XL can overcome the context-fragmentation problem and represent longer-spanning dependencies (Dai et al., 2019).

7.3 Describing Transformer-XL

7.3.1 Segment-Level Recurrence Mechanism

To resolve the context fragmentation problem, the Transformer-XL employs a **segment-level recurrence mechanism** to capture long-term dependencies using information from previous segments. When a segment is being processed, each hidden layer receives two inputs: (1) the previous hidden layer outputs of the *current segment* (akin to the vanilla model, shown as gray arrows in fig. 8), and (2) the previous hidden layer outputs of the *previous segment* (green arrows in fig. 8). So although gradient updates or training still occurs within a segment, this extended context feature lets historical information to be fully used, avoiding context fragmentation.

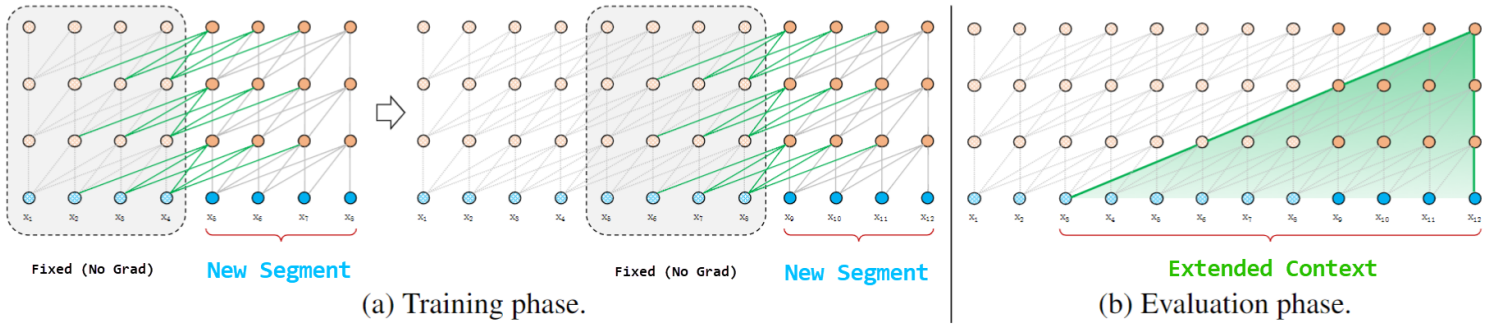


Figure 8: Segment level recurrence mechanism at work: the hidden state for previous segment is *fixed* and *stored* to later be reused as an extended context while the new segment is processed. Like in Transformer, gradient updates or training still occurs within a segment, but the extended context feature allows historical information to now be incorporated. From *Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context*, by Dai et al., 2019. https://arxiv.org/pdf/1901.02860.pdf. Copyright 2019 by Dai et al.

Intuitively, for the sentence "I went to the store. I bought some cookies," the model receives the sentence "I went to the store," caches the hidden states, and only *then* feeds the remaining sentence "I bought some cookies" and cached hidden states together into the model (Kurita, 2019b).

To formally describe the recurrence mechanism, let $\mathbf{s}_\tau = \{x_{\tau_1}, x_{\tau_2}, \ldots, x_{\tau_L}\}$ and $\mathbf{s}_{\tau+1} = \{x_{\tau+1_1}, x_{\tau+1_2}, \ldots, x_{\tau+1_L}\}$ denote two consecutive segment embeddings of length L. Let $\mathbf{h}_\tau^n \in \mathbb{R}^{L\times d}$ be the n-th layer's hidden state vector produced for the τ-th segment $\mathbf{s}_\tau$, where d is the hidden dimension. Then n-th layer's hidden state for next segment $\mathbf{s}_{\tau+1}$ is calculated in eq. (6) as:

$$
\begin{aligned}
\tilde{\mathbf{h}}_{\tau+1}^{n-1} &= \left\{\texttt{StopGradient}\left(\mathbf{h}_\tau^{n-1} \circ \mathbf{h}_{\tau+1}^{n-1}\right)\right\} \\
\mathbf{q}_{\tau+1}^n, \quad \mathbf{k}_{\tau+1}^n, \quad \mathbf{v}_{\tau+1}^n &= \mathbf{h}_{\tau+1}^{n-1}\cdot\mathbf{W}_q^T, \quad \tilde{\mathbf{h}}_{\tau+1}^{n-1}\cdot\mathbf{W}_k^T, \quad \tilde{\mathbf{h}}_{\tau+1}^{n-1}\cdot\mathbf{W}_v^T \\
\mathbf{h}_{\tau+1}^n &= \texttt{TransformerLayer}\left(\mathbf{q}_{\tau+1}^n, \quad \mathbf{k}_{\tau+1}^n \quad \mathbf{v}_{\tau+1}^n\right)
\end{aligned}
\tag{6}
$$

where the first line indicates that the two consecutive hidden states are concatenated; $\mathbf{W}$ denotes model parameters; and $\mathbf{q}, \mathbf{k}, \mathbf{v}$ are the query, key, and value vectors. The key feature here, compared to standard Transformer, is that the key $\mathbf{k}_{\tau+1}^n$ and value $\mathbf{v}_{\tau+1}^n$ are conditioned on the elongated context $\tilde{\mathbf{h}}_{\tau+1}^{n-1}$ *and* the previous segment's cached context $\mathbf{h}_\tau^{n-1}$. This is visible by the green lines in fig. 8.

This new recurrence scheme allows more efficient computations. Also, it can cache multiple previous segments not just the previous one, making it more similar to an RNN's memory.

7.3.2 Relative Positional Encoding

Trying to graft the Transformer's positional encodings directly with the segment-level recurrence mechanism caused problems. Simply put, the Transformer-XL could not keep positional word order straight when reusing hidden states because tokens from different

segments ended up with the same positional encodings, even though their position and importance could differ. This confused the model.

To remedy this and correctly merge the recurrence scheme with positional word order, the authors invented **relative positional encodings** that work conjunction with attention scores of each layer, as opposed to only before the first layer, and which are based on *relative* not *absolute* distances between tokens (Horev, 2019). Formally speaking, the authors created a relative positional encodings matrix whose i-th row indicates a relative distance of i between two positions, and injected this into the attention module. As a result, the query vector can distinguish between two tokens using their different distances. Now, from Dai et al. and Horev (2019), the attention head calculation has four parts:

1. **Content-based addressing:** the original attention score without positional encoding.
2. **Content-dependent positional bias:** with respect to the current query. It uses a sinusoidal function that gets distance between tokens instead of *absolute* position of a current token.
3. **Learned global content bias:** is a learned vector that accounts for the other tokens in the key matrix.
4. **Learned global positional bias:** is a learned vector that adjusts the importance based only on distance between tokens, using the intuition that recent previous words are more relevant that a word from the previous paragraph.

7.4 Experimental Results: Ablation Study for Segment-Level Recurrence and Relative Positional Encodings

Remark	Recurrence	Encoding	Loss	PPL init	PPL best	Attn Len
Transformer-XL (128M)	✓	Ours	Full	**27.02**	**26.77**	**500**
-	✓	Shaw et al. (2018)	Full	27.94	27.94	256
-	✓	Ours	Half	28.69	28.33	460
-	✗	Ours	Full	29.59	29.02	260
-	✗	Ours	Half	30.10	30.10	120
-	✗	Shaw et al. (2018)	Full	29.75	29.75	120
-	✗	Shaw et al. (2018)	Half	30.50	30.50	120
-	✗	Vaswani et al. (2017)	Half	30.97	30.97	120
Transformer (128M)†	✗	Al-Rfou et al. (2018)	Half	31.16	31.16	120
Transformer-XL (151M)	✓	Ours	Full	23.43	**23.09**	**640**
					23.16	450
					23.35	300

Table 4: Ablation study for Segment-Level Recurrence Mechanism on the WikiText-103 data set. **PPL best** (model output) means perplexity score obtained using an optimal backpropagation training time length. **Attn Len** (model input) is the shortest possible attention length during evaluation to achieve the corresponding PPL best. From *Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context*, by Dai et al., 2019. https://arxiv.org/pdf/1901.02860.pdf. Copyright 2019 by Dai et al.

The authors seek to isolate the effects of Transformer-XL's segment-level recurrence mechanism using with different encoding schemes. In table 4, Shaw et al. (2018) uses relative encodings, and Vaswani and Al-Rfou use absolute encodings.

In table 4, segment-level recurrence and relative positional encoding must be used in conjunction for best performance, since when using the new recurrence with the new encodings, the Transformer-XL can generalize to larger attention sequences (with length 500) during evaluation while getting lowest perplexity score of 26.7.

However, if the Transformer-XL just uses the new encodings, even using shorter attention sequences of 260 and 120 with different backprop loss schemes cannot help lower its perplexity scores, as shown in the last two rows of table 4's Transformer-XL section. The standard Transformer does poorly in general, showing high overall perplexities even using short attention lengths.

8 XLNet

While most commonly in NLP models are in the form of neural networks that are pretrained on large, unlabeled data and then fine-tuned for specific tasks, different unsupervised pretraining loss functions have also been explored. From these, autoregressive language model (AR) and autoencoding language model (AE) have been the most powerful pretraining objectives.

8.1 Problems With BERT

From Yang et al. (2020), a *forward* autoregressive language model (AR) maximizes the likelihood of an input sequence by using a *forward* autoregressive decomposition, shown in eq. (7):

$$max_\theta \Big(log\, P_\theta(\mathbf{x}) \Big) = \sum_{t=1}^{T} log\, P_\theta\big(x_t \mid \mathbf{x}_{<t}\big) \tag{7}$$

Meanwhile, autoencoding language model (AE)s like BERT takes an input sequence $\mathbf{x}$, and corrupts it $\hat{\mathbf{x}}$ by masking some tokens. The autoencoding model recreates masked tokens $\overline{\mathbf{x}}$ from the corrupted input $\hat{\mathbf{x}}$, as in eq. (8):

$$max_\theta \Big(log\, P_\theta(\overline{\mathbf{x}} \mid \hat{\mathbf{x}}) \Big) \approx \sum_{t=1}^{T} m_t\, log\, P_\theta\big(x_t \mid \hat{\mathbf{x}}\big) \tag{8}$$

where $m_t = 1$ indicates that the input token x_t is masked. With this in mind, Yang et al. (2020) note BERT's problems as follows:

1. **Independence Assumption:** the $\approx$ approximation sign in eq. (7) indicates BERT factorizes its joint conditional probability $P_\theta(\bar{\mathbf{x}} \mid \hat{\mathbf{x}})$ assuming that all masked tokens $\bar{\mathbf{x}}$ are rebuilt independently of each other, even though texts are rife with long-range dependencies.
2. **Data Corruption**: Masking tokens from BERT's masked language modeling task do not appear in real data during fine-tuning, and since BERT does this in pre-training, a discrepancy arises between these two stages.

Kurita (2019b) gives an example to show how BERT predicts tokens independently. Consider the sentence: "I went to the `[MASK] [MASK]` and saw the `[MASK] [MASK] [MASK]`." Two ways to fill this are: (1) "I went to *New York* and saw the *Empire State building*," or (2) "I went to *San Francisco* and saw the *Golden Gate bridge*." But BERT might incorrectly mix things up and predict "I went to *San Francisco* and saw the *Empire State building*."

Clearly, since BERT predicts masked tokens simultaneously, it fails to learn tokens' interlocking dependencies, which is a major point against BERT since even simple language models learn at least unidirectional token dependencies (Kurita, 2019b).

8.2 Motivation for XLNet

Confronted with BERT's limitations, Yang et al. (2020) conceived **XLNet** to keep the benefits of *both* autoencoding and autoregressive language modeling while avoiding their issue. Firstly, the autoregressive language model (AR) objective in XLNet lets the probability $P_\theta(\mathbf{x})$ (from eq. (7)) be factored using the universally-true probability product rule, side-stepping BERT's false independence assumption. Intuitively this means XLNet can predict tokens sequentially and autoregressively rather than simultaneously like BERT. Secondly, XLNet's **permutation language model** captures bidirectional context like an autoencoding model. It uses a built-in two-stream attention mechanism to create target-aware predictions.

8.3 Describing XLNet

8.3.1 Permutation Language Model

Can a model be trained to use **bidirectional context** while avoiding masking and its resulting problem of independent predictions? Like language models, a **permutation language model** predicts unidirectionally, but instead of predicting in order, the model predicts tokens in a random order. Simply put, a permutation language model must accumulate bidirectional context by finding dependencies between all possible input combinations (Yang et al., 2020).

Consider the sentence: "I like cats more than dogs." A traditional language model predicts individual words sequentially from previous context words. But a permutation language model randomly samples prediction order, such as: "cats", "than", "I", "more", "dogs", "like", where "than" could be conditioned on "cats" and "I" might be conditioned on seeing "cats, than", and so on (Kurita, 2019b).

8.3.2 Need for Target-Aware Representations

Trying to merge permutation language model and Transformer blinded XLNet's target predictions to the permutation positions generated by the permutation language model.

The fault lies in the nature of Transformers: while predicting a token at a position, Transformer masks the token's embedding but *unfortunately, also* its positional encoding, so it cannot see the position of the target it must predict. Intuitively, this means a sentence cannot be accurately represented, since positions like the beginning of a sentence have different distributions from other positions in the sentence.

Formally, let $\mathcal{Z}_T$ = all possible permutations of the T-length sequence $[1, 2, \ldots, T]$; z_t = the t-th element and $\mathbf{z}_{<t}$ = the vector of the first $t-1$ elements of a permutation $\mathbf{z} \in \mathcal{Z}_T$; $\mathbf{x}$ = a text sequence of length T; x_{z_t} = the content token to be predicted in the text sequence $\mathbf{x}_{\mathbf{z}_t}$ that is indexed by the permutations in $\mathbf{z}_{<t}$.

Yang et al. (2020) began by parameterizing the next-token predictive distribution using the softmax:

$$P_\theta\left(X_{z_t} = x_i \mid \mathbf{x}_{\mathbf{z}_t}\right) = \frac{\exp\left(e(x_i)^T \cdot h_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}}\right)\right)}{\sum_{i=1} \exp\left(e(x_i)^T \cdot h_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}}\right)\right)} \tag{9}$$

where $e(x_i)$ = the embedding of the i-th token, x_i, and $h_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}}\right)$ = the hidden state vector for $\mathbf{x}_{\mathbf{z}_t}$ created by the Transformer after masking. But $h_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}}\right)$ does not depend on the position z_t of the target it must predict, so XLNet is limited to predicting the same distribution regardless of target position. To remedy this, the authors replaced the hidden state h_θ in eq. (9) by $g_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}}, z_t\right)$ which takes the target position z_t as an argument.

8.3.3 Two-Stream Self-Attention

When formulating $g_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}}, z_t\right)$ the Transformer architecture produced yet another contradiction: (1) to predict token x_{z_t}, $g_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}}, z_t\right)$ needs only the *position* z_t and not the *content* x_{z_t}, or else the prediction becomes trivial, and (2) to predict all other tokens x_{z_j} with $j > t$, $g_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}}, z_t\right)$ also needs x_{z_t} to incorporate the entire contextual information. Thus, Dai et al. (2019) invented the **two-stream atten-**

tion mechanism which creates an overall hidden state from two separate sets of hidden states: the **content stream** and **query stream**.

The **content stream** $h_\theta\left(\mathbf{x}_{\mathbf{z}_{\leq t}}\right)$ encodes *context* $\mathbf{x}_{\mathbf{z}\leq t}$ like ordinary hidden states in the Transformer, while also encoding the *content* token x_{z_t}. For the t-th prediction in the sentence, the content stream (using both z_t and x_{z_t}) is computed for attention layers $m = 1, \ldots, M$ as $h_{z_t}^{(m)} = \texttt{Attention}\left(\mathbf{Q} = h_{z_t}^{(m-1)},\ \mathbf{KV} = \mathbf{h}_{\mathbf{z}_{\leq t}}^{(m-1)};\ \theta\right)$.

Query stream $g_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}},\ z_t\right)$ encodes *context* $\mathbf{x}_{\mathbf{z}\leq t}$ with *position* z_t simultaneously, but not the *content* token x_{z_t}, to evade the contradiction. For the t-th prediction in a sentence, the query stream (using z_t but not x_{z_t}) is computed as $g_{z_t}^{(m)} = \texttt{Attention}\left(\mathbf{Q} = g_{z_t}^{(m-1)},\ \mathbf{KV} = \mathbf{h}_{\mathbf{z}_{<t}}^{(m-1)};\ \theta\right)$.

Intuitively, the purpose of the content and query stream is as follows. Suppose we must predict the word "like" in *"I like cats more than dogs"* where the previous words in the permutation were "more" and "dogs." The **content stream** would encode this information while the **query stream** would hold positional knowledge about "like" as well as content stream information, so XLNet can predict "like" (Kurita, 2019b).

8.3.4 Relative Segment Encodings

XLNet builds on Transformer-XL's relative encodings to model relationships between positions. While BERT learns a segment embedding to distinguish *which specific segments the (words) positions are from*, XLNet learns an embedding that encodes if two words are *within the same segment embedding*. As a benefit, XLNet can be applied to tasks that take arbitrarily many input sequences (Kurita, 2019b).

8.4 Conceptual Difference Between XLNet and BERT

To further illustrate the conceptual difference between XLNet and BERT from a model training standpoint, take the list of words $\left[\texttt{New}, \texttt{York}, \texttt{is}, \texttt{a}, \texttt{city}\right]$. Let the two tokens $\left[\texttt{New}, \texttt{York}\right]$ be selected for prediction, so the models must maximize the log-likelihood: $\log P(\texttt{New York | is a city})$. Assuming XLNet uses the factorization order $\left[\texttt{is}, \texttt{a}, \texttt{city}, \texttt{New}, \texttt{York}\right]$, then BERT and XLNet have the following loss functions:

$$\begin{aligned} \mathcal{J}_{\text{BERT}} &= \log P\left(\texttt{New} \mid \texttt{is a city}\right) + \log P\left(\texttt{York} \mid \texttt{is a city}\right) \\ \mathcal{J}_{\text{XLNet}} &= \log P\left(\texttt{New} \mid \texttt{is a city}\right) + \log P\left(\texttt{York} \mid \texttt{New}, \texttt{is a city}\right) \end{aligned} \tag{10}$$

While BERT can learn some kind of dependency between the pairs New and York, eq. (10) clearly shows that XLNet learns stronger dependency or a "denser training signal" (Dai et al., 2019).

9 ERNIE 1.0

9.1 Motivation for ERNIE 1.0

Co-occurrence count models like Word2Vec, GloVe, and BERT create word vector representations only via surrounding contexts, not also through prior knowledge in the sentence, and thus fail to capture relations between entities in a sentence. Consider the following training sentence:

"Harry Potter is a series of fantasy novels written by J. K. Rowling."

Using co-occurring words "J.", "K.", and "Rowling", BERT is limited to predicting the token "K." but utterly fails at recognizing the whole entity *J. K. Rowling*. A model could use mere co-occurrences to predict missing entity *Harry Potter* but it would not be making use of the relationship between the novel and its writer.

ERNIE 1.0 comes to the rescue here. **ERNIE (Enhanced Representation through Knowledge Integration)** extrapolates the relationship between the *Harry Potter* entity and *J. K. Rowling* entity using implicit knowledge of words and entities, and then uses this relationship to predict that *Harry Potter* is a series written by *J. K. Rowling* (Sun et al., 2019a).

Sentence	Harry	Potter	is	a	series	of	fantasy	novels	written	by	British	author	J.	K.	Rowling
Basic-level Masking	[mask]	Potter	is	a	series	[mask]	fantasy	novels	[mask]	by	British	author	J.	[mask]	Rowling
Entity-level Masking	Harry	Potter	is	a	series	[mask]	fantasy	novels	[mask]	by	British	author	[mask]	[mask]	[mask]
Phrase-level Masking	Harry	Potter	is	[mask]	[mask]	[mask]	fantasy	novels	[mask]	by	British	author	[mask]	[mask]	[mask]

Figure 9: ERNIE 1.0 uses basic masking to get word representations, followed by phrase-level and entity-level masking. From *ERNIE: Enhanced Representation Through Knowledge Integration*, by Sun et al., 2019. https://arxiv.org/pdf/1904.09223.pdf. Copyright 2019 by Sun et al.

9.1.1 phrase-level masking

A phrase is a "small group of words of characters together acting as a conceptual unit" (Sun et al., 2019a). ERNIE chunks sentences by phrase boundaries, then does phrase-masking by randomly selecting phrases, masking them, and training to predict subpieces of the phrase. This way, phrase information gets built into ERNIE's embeddings.

9.1.2 entity-level masking

From Sun et al. (2019a), name entities contain "persons, locations, organizations, products" that are denoted with a proper name, and can be abstract or have physical existence. Entities often contain important information within a sentence, so are regarded as conceptual units. ERNIE parses a sentence for its named entities, then masks and predicts all slots within the entities, as in fig. 9.

9.2 Experimental Results of ERNIE 1.0

pre-train dataset size	mask strategy	dev Accuracy	test Accuracy
10% of all	word-level(chinese character)	77.7%	76.8%
10% of all	word-level&phrase-level	78.3%	77.3%
10% of all	word-level&phrase-leve&entity-level	78.7%	77.6%
all	word-level&phrase-level&entity-level	79.9 %	78.4%

Table 5: Ablation study for ERNIE's phrase-level masking and entity-level masking. From Table 2 in *ERNIE: Enhanced Representation Through Knowledge Integration*, by Sun et al., 2019. https://arxiv.org/pdf/1904.09223.pdf. Copyright 2019 by Sun et al.

Sun et al. (2019a) show that ERNIE 1.0 outperforms BERT by more than 1% absolute accuracy on five Chinese NLP tasks: natural language inference (NLI) (XNLI data), semantic textual similarity (STS) (LCQMC data), named entity recognition (NER) (MSRA-NER data), sentiment analysis (SA) (ChnSentiCorp data), question answering (QA) (NLPCC-DBQA data).

Evidence of the superiority of ERNIE's knowledge integration strategy is visible in table 5, which shows that adding phrase masking to basic word-level masking improved ERNIE's performance almost a full percent, and adding entity masking to this combination resulted in still higher gains for larger data.

Additionally, the authors tested ERNIE's knowledge learning ability using fill-in-the-blanks on named entities in paragraphs. In case 1 from table 6, ERNIE predicts the correct father name entity based on prior knowledge in the article while BERT simply memorizes one of the sons' name, completely ignoring any relationship between mother and son. In case 2, BERT outputs some characters from contextual patterns it learned, but cannot even fill the slot with an entity, while ERNIE fills the slots with the *correct* entity. In cases 3,4,6 BERT once more fills the slots with characters related to the sentences but not with the semantic concept, while ERNIE again predicts the correct entities. However in case 4, even though ERNIE predicts the wrong city name, it still understands the semantic type is a city. Evidently, ERNIE's contextual knowledge understanding is far superior to BERT's predictions (Sun et al., 2019a).

Case	Text	ERNIE	BERT	Answer
1	"In September 2006, ___ married Cecilia Cheung. They had two sons, the older one is Zhenxuan Xie and the younger one is Zhennan Xie."	Tingfeng Xie	Zhenxuan Xie	Tingfeng Xie
4	"Australia is a highly developed capitalist country with ___ as its capital. As the most developed country in the Southern Hemisphere, the 12th largest economy in the world and the fourth largest exporter of agricultural products in the world, it is also the world's largest exporter of various minerals."	Melbourne	(Not a city name)	Canberra
6	"Relativity is a theory about space-time and gravity, which was founded by ___."	Einstein	(Not a word in Chinese)	Einstein

Table 6: Comparing ERNIE to BERT on Cloze Chinese Task. From Figure 4 in *ERNIE: Enhanced Representation Through Knowledge Integration*, by Sun et al., 2019a. Copyright 2019 by Sun et al.

10 Conclusion and Future Work

In the future, my aim is to apply NLP algorithms in the real world by doing: domain adaptation, transfer learning, model fine-tuning to avoid costs of training models from scratch with huge data. *Transfer learning* or domain adaptation describes how a model created for one task is reused for a different task. Pre-trained models like BERT are used as starting points for downstream tasks that require much time and resources in order to train (Brownlee, 2017). This will enable me to research further into **concept embeddings** and key phrase extraction.

Secondly, I would like to explore Bayesian approaches to NLP. Brazinkas et al. (2018) use a **Bayesian** Skip-Gram model to encode words as probability densities rather than simple vectors like in Word2Vec, enabling it to capture polysemy, and thus strengthening it on tasks like lexical substitution and word sense disambiguation (WSD). As an example insight from the paper: "when 'kiwi' appears in a context suggesting the 'bird' sense, the posterior [of the word 'kiwi'] becomes more 'peaky' and moves towards the representation of word 'bird' " (Brazinkas et al., 2018).

Thirdly, I want to study and apply multi-task and continual learning from ERNIE 2.0 model, which performs better than even top models like BERT, XLNet, and ERNIE 1.0. I especially want to implement all these models using a domain-specific language for NLP to separate technical details from actual neural network logic, to promote modularity and model understanding.

A Appendix: Language Models

A language model takes a sequence of word vectors and outputs a sequence of predicted word vectors by learning a probability distribution over words in a vocabulary. In representation terms, the "vector representation of a word depends on the context vector representation" (Ibrahim, 2019). Many tasks such as machine translation (MT), spell correction, text summarization, question answering (QA), and sentiment analysis (SA) all use language models to convert text into machine-interpretable language (Chromiak, 2017).

Intuitively, language models predict words in a blank. For instance, given the following context: "The ___ sat on the mat" where ___ is the word to predict, a language model might suggest the word "cat" should fill the blank a certain percentage of the time and the word "dog" would fill the blank with lower probability (Kurita, 2019).

Formally, language models compute the conditional probability of a word w_t given a context, such as its previous $n-1$ words: $P(w_t|w_{t-1},...,w_{t-n+1})$ (Ruder, 2016). The joint probability of a T-sized sequence of word tokens $w_1,...,w_T$ from a sentence S is expressed as:

$$\begin{aligned} P(S) &= P\left(w_1,...,w_T\right) \\ &= P(w_1)\cdot P(w_2 \mid w_1)\cdot ... \cdot P(w_n \mid w_1,...,w_{T-1}) \\ &= \prod_{t=1}^{T} P\left(w_t \mid w_1,...,w_{t-1}\right) \end{aligned} \tag{11}$$

Typically, the **Markov Assumption**, which states that the probability of a word depends only on its previous word, reduces context history and intake of model data. Thus the joint probability in eq. (11) is estimated using the n previous words as: $P\left(w_1,...,w_T\right) \approx \prod_{t=1}^{T} P\left(w_t \mid w_{t-1},...,w_{t-n+1}\right)$.

A.1 n-gram language model

An n-gram is a sequence of n words. The simple n-gram language model assigns probabilities to sentences and word sequences. It calculates a word's probability based on the frequencies of its constituent n-grams, taking just the preceding $n-1$ words as context instead of the entire corpus (Ruder, 2016): $P\left(w_t \mid w_{t-1},...,w_{t-n+1}\right) = \frac{count(w_{t-n+1},...,w_{t-1},w_t)}{count(w_{t-n+1},...,w_{t-1})}$

A.2 neural network language model

A.2.1 Curse of Dimensionality

Bengio et al. (2003) defines the *curse of dimensionality* in NLP as how a word sequence may differ from the training set of word sequences. This appears when modeling the joint distribution between discrete random variables (like words in a sentence).

A.2.2 Key Concept: Neural Network Representation

A **neural network** is a function from vectors to vectors. All neural network representations rely on a structure called a neuron, a linear formula: $W\cdot x+b$. By applying a nonlinear function $f(\cdot)$ to the neuron and by incorporating many hidden layers and by stacking neurons together, a neural network can model any function.

Many NLP applications using neural networks feed in word tokens as parameters that are *optimized* to fit text by minimizing a continuous loss function (Smith, 2019). There are three components to a neural network:

1. **Embedding Layer**: this layer creates word embeddings by multiplying an index vector with a word vector matrix.
2. **Intermediate Layer(s)**: multiple layers are used to create a fully-connected layer that applies a non-linearity function (like hyperbolic tangent or sigmoid) to the concatenation of word embeddings.
3. **Softmax Layer**: the last layer normalizes the word embedding matrix to using a **softmax function** to create a probability distribution over words w_t in the vocabulary: $P\left(w_t \mid w_{t-1},...,w_{t-n+1}\right) = \frac{\exp\left(h^T\cdot v'_{w_t}\right)}{\sum_{w_i\in V}\exp\left(h^T\cdot v'_{w_i}\right)}$, where V = vocabulary of a corpus, h = output vector of the hidden layer, and v'_w = the output embedding of word w.

A.2.3 Solution to Curse of Dimensionality: Neural Model and Continuous Vector Representations

The n-gram model seeks to remedy the *curse of dimensionality* by combining short overlapping word sequences seen in the training set.

However, Bengio et al. (2003) developed a *neural probabilistic language model* to learn a distributed representation for words to allow the model to generalize to unseen data. The neural model does two tasks simultaneously; (1) it learns a distributed representation for each word, and also (2) it learns the probability distribution of word sequences *as a function of* the distributed representations. Since the neural model uses continuous word vector representations, the learned probability function's parameters increase linearly, rather than exponentially, with the vocabulary size and vector dimension, thus resolving the curse of dimensionality (Bengio et al., 2003).

A.3 bidirectional language model (biLM)

A.3.1 forward language model

A general language model predicts a next word given its context words, $P(\textit{Word} \mid \textit{Context})$.

A **forward language model** takes this context to be previous words. From Peters et al. (2018), given a sequence of N tokens $(t_1, t_2, \ldots, t_N)$, a forward language model calculates the probability of the tokenized sentence assuming the probability of a word token t_k is conditional on its history tokens: $P(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} P(t_k \mid t_1, t_2, \ldots, t_{k-1})$.

A.3.2 backward language model

A **backward language model** is similar to a forward model excepts it predicts the current token t_k conditional on future context tokens: $P(t_1, t_2, \ldots, t_N) = \prod_{k=1}^{N} P(t_k \mid t_{k+1}, t_{k+2}, \ldots, t_N)$.

A.3.3 Combining Forward and Backward

A **bidirectional language model (biLM)** combines the forward and backward language models during maximum likelihood estimation to *jointly* maximize the log likelihood of the forward and backward directions:

$$\sum_{k=1}^{N} \left(\log\left(P\left(t_k \mid t_1, \ldots, t_{k-1}; \vec{\theta}\right)\right) + \log\left(P\left(t_k \mid t_{k+1}, t_{k+2}, \ldots, t_N; \vec{\theta}\right)\right)\right) \tag{12}$$

where $\vec{\theta}$ represents additional parameters (Peters et al., 2018).

Akbik et al. (2018) use the hidden states of a bidirectional recurrent neural network to contextualize words. Consider the sentences "Mary accessed the bank account" and "The swan waded to the bank of the river." In the first sentence, a unidirectional contextual model would represent the target word "bank" based on 'I accessed the' but not 'account,' thus failing to capture polysemy of 'bank.' But a bidirectional model represents "bank" using both past and future context to capture its multiple senses.

A.4 autoregressive language model (AR)

From Yang et al. (2020), an **autoregressive language model (AR)** *autoregressively* estimates the probability distribution of a text sequence $\mathbf{x} = \{x_1, \ldots, x_T\}$. The AR model factorizes the likelihood into a forward product, $P(\mathbf{x}) = \prod_{t=1}^{T} P(x_t \mid \mathbf{x}_{<t})$, using tokens before a timestep, or a backward product, $P(\mathbf{x}) = \prod_{t=T}^{1} P(x_t \mid \mathbf{x}_{>t})$, using tokens after a timestep. Then, a neural network is trained to model either conditional distribution. But due to AR's unidirectionality, it cannot model bidirectional contexts, and thus performs poorly for downstream nlp tasks.

A.5 autoencoding language model (AE)

An **autoencoding language model (AE)** recreates original data from corrupted input, like BERT. Given a input sequence, some tokens are randomly masked and the model guesses correct tokens. Since AE modeling does not estimate densities, it is able to learn bidirectional contexts.

B Appendix: Preliminary Models for State-of-the-Art NLP Algorithms

B.1 Recurrent Neural Networks (RNN)

B.1.1 Motivation for RNNs

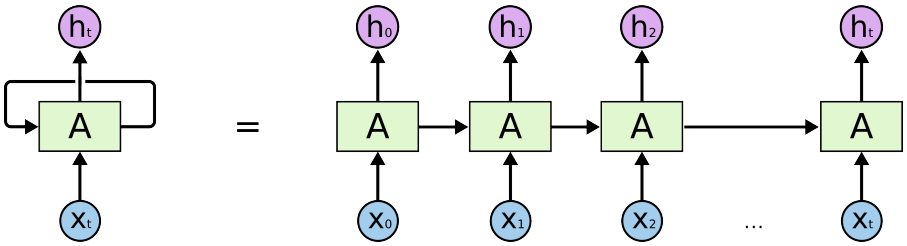


Figure 10: Unrolled view of Recurrent Neural Network with Hidden States h_i and inputs x_i. From *Understanding LSTMs*, by Colah., 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Copyright 2015 by Colah.

Traditional neural networks cannot persist information. As a comparison, while humans do not start thinking from scratch each time they learn something new, neural networks lack memory. Inherently related to sequences, recurrent neural networks use a recurrence or looping mechanism to introduce data persistence in the model to overcome this problem (Colah, 2015). This looping mechanism acts like a "highway" to flow from one step to the next by passing inputs and modified hidden states along until computing a final prediction (Nguyen, 2018a).

B.1.2 Describing RNNs

An RNN is a unidirectional language model since it uses context words left of the target word. It is a neural network that takes in a sequence (sentence) of input symbols (words) $\vec{x} = \{x_1, \ldots, x_T\}$, and for each time t, the current hidden state h_t is updated via the formula $h_t = f\left(h_{t-1}, x_t\right)$ where $f(\cdot)$ is a nonlinear activation function. The RNN's intermediate task is to predict a probability distribution over an input sequence (sentence) by predicting the next word symbol x_t in the sequence sentence, using left context, so the output at time t is the conditional distribution $P\left(x_t \mid x_{t-1}, \ldots, x_1\right)$. The probability of the entire sequence sentence $\vec{x}$ is the product of all the probabilities of the individual words, $P(\vec{x}) = \prod_{t=1}^{T} P\left(x_t \mid x_{t-1}, \ldots, x_1\right)$ (Cho, 2014).

B.2 Long-Short Term Memory Networks (LSTM)

B.2.1 Motivation for LSTM: Vanishing Gradient Problem

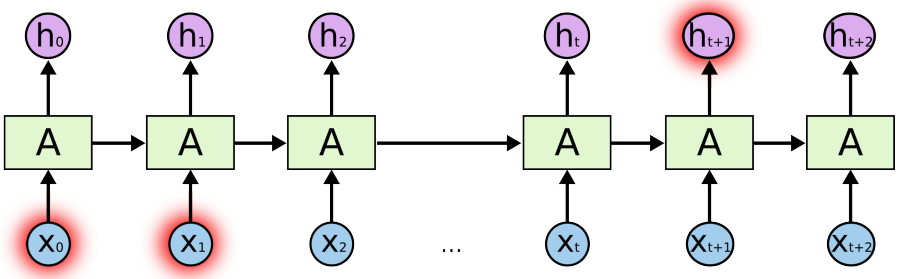


Figure 11: Long-Term Dependency Problem in RNNs (widening gap between inputs x_i and hidden states h_j. From *Understanding LSTMs*, by Colah, 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Copyright 2015 by Colah.

RNNs suffer from the well known **long-term dependency problem**. In some prediction tasks, longer context is needed to predict a target word. For instance to predict the last word in the sentence "I grew up in France ... I speak fluent *French*", a model would need the earlier context word "France." When this gap between target and context words becomes too large, RNNs cannot learn their relationship (Colah, 2015).

Mathematically speaking, this is due to the **vanishing gradient problem**. During backward propagation of errors through the RNN, the gradient shrinks as it back propagates through time and becomes too small to update the parameter weights significantly.

This is compounded by the fact that since inputs at any timestep t are dependent on previous $t-1$ outputs, longer sequences require more gradient calculations. Adjustments to earlier layers thus become smaller, causing gradients to shrink exponentially as they are backpropagated through to earlier layers of the RNN. As a result, RNNs "forget" older history, resulting in short-term memory as shown in fig. 11 (Nguyen, 2018b).

B.2.2 Describing LSTMs

A **long-short term memory network (LSTM)** is a type of RNN that learns long-term dependencies by design, unlike RNNs. LSTMs use gates or neural networks that decide how to add or remove information from the cell state, explicitly letting the LSTM "remember" or "forget" (Nguyen, 2018b). Since LSTMs can accumulate increasingly richer information while parsing the sentence, by the time the last word is reached, the hidden layer of the network provides a **semantic representation** of the entire sentence (Palangi et al., 2016).

A core idea in an LSTM is the **cell state**, shown in fig. 14 as the topmost line with the gates merging into it. For example, for a language predicting the next word based on previous ones, the appropriate behavior of its cell state might be to include the gender of the present subject so the model uses correct pronouns. When a new subject is observed, the cell state should forget the old subject's gender and retain the new one (Colah, 2015).

From Nguyen (2018b), these gates regulate information flow as follows:

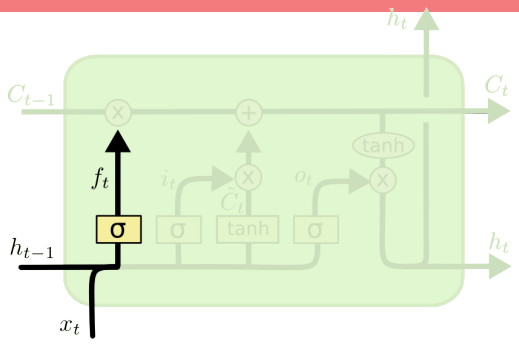


Figure 12: Forget Gate Calculation. From *Understanding LSTMs*, by Colah, 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Copyright 2015 by Colah.

Forget Gate: the forget gate, shown in fig. 12, decides information to discard or keep. The previous hidden state h_{t-1} and current input x_t are passed through the sigmoid nonlinearity function. The forget gate outputs a number between 0 and 1 for each number in the cell state C_{t-1}; values closer to 0 indicate the forget gate should discard the information and values closer to 1 should be kept.

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{13}$$

where f_t = forget gate for time t, $\sigma(\cdot)$ = sigmoid, W_f = the weight matrix at the forget layer, and b_f = forget gate's bias term.

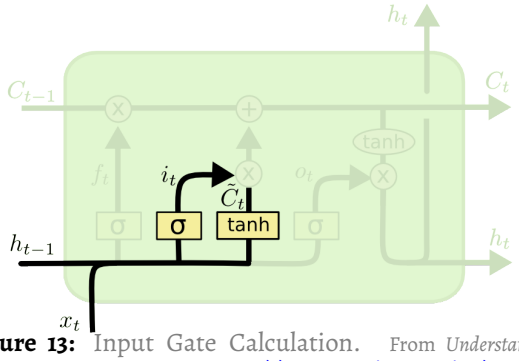


Figure 13: Input Gate Calculation. From *Understanding LSTMs*, by Colah, 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Copyright 2015 by Colah.

Input Gate: the input gate i_t, shown in fig. 13, updates the cell state C_t. Previous hidden state h_{t-1} and current input x_t are passed though a signmoid function normalize vector cells between 0 and 1. The input gate is later used with the cell state to decide how values are updated.

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \tag{14}$$

where i_t is the input gate for time t, $\sigma(\cdot)$ is the sigmoid, W_i is the weight matrix at the input layer, and b_i is the input gate's bias term.

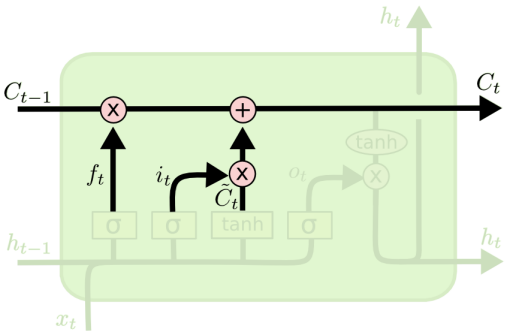


Figure 14: Cell State Calculation. From *Understanding LSTMs*, by Colah, 2015.https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Copyright 2015 by Colah.

Cell State: Current cell state C_t, shown in fig. 14, takes h_{t-1} and x_t and normalizes them to be between -1 and 1 via a hyperbolic tangent nonlinearity:

$$C_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{15}$$

where C_t is the cell state for time t, $\tanh(\cdot)$ is the hyperbolic tangent, W_C is the weight matrix at the cell state layer, and b_C is the cell state's bias term. Next, pointwise multiplications occur to regulate memory in LSTM:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot C_t \tag{16}$$

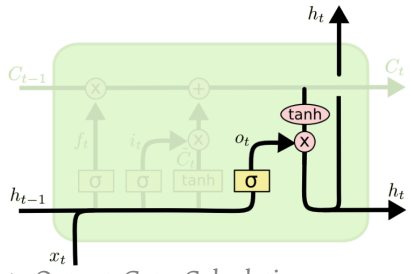


Figure 15: Output Gate Calculation. From *Understanding LSTMs*, by Colah, 2015.https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Copyright 2015 by Colah.

Output Gate: the output gate, shown in fig. 15, determines the next hidden state by multiplying the previous output state by the cell state that is filtered by the hyperbolic tangent.

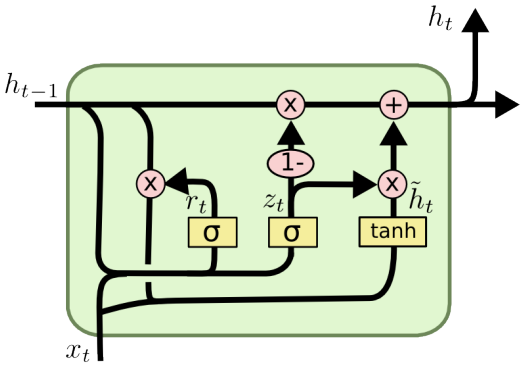
$$\begin{aligned} o_t &= \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \\ h_t &= o_t \cdot \tanh(C_t) \end{aligned} \tag{17}$$

B.3 Gated Recurrent Networks (GRU)

The gated recurrent unit (GRU) from Cho et al. (2014) is a type of LSTM that "combines the forget and input gates into a single **update gate**" and "merges the cell state and hidden state" (Colah, 2015), resulting with only the reset gate r_t and update gate z_t (Nguyen, 2018b).

The **update gate** z_t controls how much information from previous hidden state contributes to current hidden state, acting like a memory cell in the LSTM to remember long-term dependencies (Cho et al., 2014).

The **reset gate** r_t signals the hidden state on how to forget previous information. When the reset gate is close to 0, the initialized hidden state $\tilde{h}_t$ must ignore previous hidden state h_{t-1} and reset with the current input x_t only. Intuitively, this allows the activation hidden state h_t to forget any irrelevant information for the future (Cho et al., 2014).



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 16: The GRU Cell with Gates. From *Understanding LSTMs*, by Colah, 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Copyright 2015 by Colah.

Nguyen (2018b) notes that since GRU's have fewer tensor operations, they are more efficient than LSTMs during training. The GRU adaptively remembers and forgets because each hidden unit has separate reset and update gates so each hidden unit can capture dependencies over different time scales. Frequently active reset gates help the GRU remember short-term dependencies while frequently active update gates help the GRU note long-term dependencies (Cho et al., 2014).

C GloVe

The **Global Vectors for Word Representation (GloVe)** model is an unsupervised learning algorithm that aims to capture meaning in a semantic vector space using global count statistics instead of local context (Pennington et al., 2014). GloVe uses probability ratios to represent meaning as vector offsets.

C.1 Problem with Word2Vec: Secret in the Loss Function

Word2Vec uses only local and not global context. So even though words "the" and "cat" might be used together frequently, Word2Vec will not discern if this is because "the" is a common word or because "the" and "cat" are actually correlated (Kurita, 2018).

From Pennington et al. (2014), Word2Vec implicitly optimizes over a co-occurrence matrix while streaming over input sentences. In doing so, it optimizes its log likelihood loss of seeing words *simultaneously* in the same context windows, resulting in an alternative way in eq. (18) of expressing Word2Vec's loss function.

$$J = -\sum_i X_i \sum_j P_{ij} \log(Q_{ij}) \tag{18}$$

In eq. (18), $X_i = \sum_k X_{ik}$ is the total number of words appearing in the context of word i and Q_{ij} is the probability that word j appears in context of word i and is estimated as $Q_{ij} = \text{softmax}\left(w_i \cdot w_j\right)$. Evidently, Word2Vec's loss function is form of cross entropy between the predicted and actual word distributions found in context of word i. The problem with this is twofold: (1) cross entropy models long-tailed distributions poorly, and (2) the cross-entropy here is weighted by X_i, causing equal-streaming over data, so a word appearing n times contributes to the loss n times (Pennington et al., 2014). Recognizing there is no inherent justification for streaming across all words equally, GloVe instead computes differences between unnormalized probabilities, contrary to Word2Vec (Kurita, 2018a).

C.2 Motivation for GloVe

Previous global count models like Latent Semantic Analysis (LSA) produced word embeddings lacking vector analogical properties and lacking *dimensions of meaning* such as gender, grammar tense, and plurality, disabling downstream models from easily extracting meaning from those word vectors (Kurita, 2018a). Building from past failures while avoiding Word2Vec's local context problems, GloVe instead uses a principled and explicit approach for learning these *dimensions of meaning*.

C.3 Describing GloVe

C.3.1 Meaning Extraction Using Co-Occurrence Counts

GloVe uses a co-occurrence matrix that describes how words co-occur within a fixed sliding window, assuming co-occurrence counts reveal word meaning. Words **co-occur** when they appear together within this fixed window (Kurita, 2018a). GloVe takes the matrix as input, rather than the entire corpus, so GloVe disregards sentence-level information while taking into account only corpus-wide co-occurrence.

From Weng (2017), the co-occurrence probability is defined as:

$$p_{\text{co}}\left(w_k \mid w_i\right) = \frac{C(w_i, w_k)}{C(w_i)}$$

where $C(w_i, w_k)$ counts the co-occurrence between words w_i and w_k.

GloVe's insight is that word meanings are captured by ratios of co-occurrence probabilities rather than the probabilities themselves. To illustrate how GloVe uses these counts, consider two words $w_i =$ "ice" and $w_j =$ "steam".

- **Case 1:** For context words related to `ice` but not `steam` (like `solid`), the co-occurrence probability $p_{\text{co}}\left(\texttt{solid} \mid \texttt{ice}\right)$ should be much larger than $p_{\text{co}}\left(\texttt{solid} \mid \texttt{steam}\right)$, therefore the ratio $\frac{p_{\text{co}}\left(\texttt{solid} \mid \texttt{ice}\right)}{p_{\text{co}}\left(\texttt{solid} \mid \texttt{steam}\right)}$ gets very large.
- **Case 2:** Conversely, for words related to `steam` but not `ice` (like `gas`) ⇒ the co-occurrence ratio $\frac{p_{\text{co}}\left(\texttt{gas} \mid \texttt{ice}\right)}{p_{\text{co}}\left(\texttt{gas} \mid \texttt{steam}\right)}$ should be small.
- **Case 3:** For words w related to both `ice` and `steam` (like `water`) or related to neither (like `fashion`), the ratio $\frac{p_{\text{co}}\left(w \mid \texttt{ice}\right)}{p_{\text{co}}\left(w \mid \texttt{steam}\right)}$ should be near one.

D Sequence To Sequence Model

D.1 Describing Seq-to-Seq Model

A **sequence-to-sequence (Seq-to-Seq) model** processes a sequence of embeddings. An **Encoder** squashes the source sequence $\vec{x} = \{x_1, \ldots, x_{T_x}\}$ of individual word vectors x_t in the input sentence X into a *fixed-length* **context vector** that is sent to a **Decoder** which outputs a target sequence one word at a time (Alammar, 2018a).

Formally, an Encoder gated recurrent unit (GRU) outputs a hidden state given a previous hidden state and the current input $h_t = \texttt{EncoderGRU}\left(x_t, h_{t-1}\right)$, where the context vector named z is assigned the last hidden state: $z = h_{T_x}$. The context vector is then passed to the Decoder along with a target token y_t and previous Decoder hidden state s_{t-1} to return a current hidden state, $s_t = \texttt{DecoderGRU}\left(y_t, s_{t-1}, z\right)$. The context vector z does not have a time step t subscript, signifying the same context vector from the Encoder is reused each time step in the Decoder.

D.2 Problem with Seq-to-Seq Models

The context vector z does not have a time step t subscript, meaning this same context vector from the Encoder is reused each time step in the Decoder. Essentially, compressing the inputs into such a **fixed-length** vector leads to a **long-term dependency problem** since the Decoder uses only the last hidden state of the Encoder, not all hidden states.

D.3 The Attention Mechanism

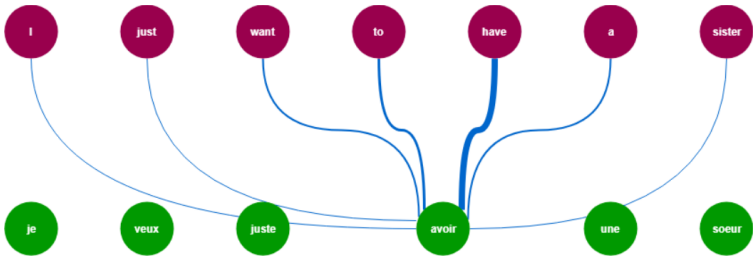


Figure 17: Attention Mechanism: How words are considered for contextual evidence. From *Intuitive Understanding of Seq2seq model and Attention Mechanism in Deep Learning*, by Medium, 2019. https://medium.com/analytics-vidhya. Copyright n.d by n.d.

The **attention mechanism** was proposed in neural machine translation (NMT) task to memorize longer sentences by "selectively focusing on parts of the source sentence" as required (Luong et al., 2015). Instead of creating a single context vector z from the Encoder's last hidden state h_{T_x}, the *attention architecture* creates a context vector for each input word or timestep t, reducing the information compression problem by letting the Decoder use all Encoder hidden states. The attention mechanism essentially creates links between the context vector and entire source input. A general illustration is in fig. 17.

D.4 Seq-to-Seq Model Using Attention

The key components of a Seq-to-Seq model with attention are its Attention forward pass, which computes attention scores, and Decoder forward pass, which outputs the context vector.

D.4.1 Forward Pass of Attention

The attention mechanism is viewed as a layer in the Seq-to-Seq model with a forward pass that updates parameters. The steps for the forward pass to calculate attention scores α_{ti} is as follows:

1. First, an **alignment model** $\texttt{align}$ calculates **energy scores** e_{ti} that measure how well the "inputs around position i and the output at position t match" (Bahdanau et al., 2016). The energy scores $e_{ti} = \texttt{align}\left(s_{t-1}, h_i\right)$ are weights specifying how much of the Decoder hidden state s_{t-1} and the Encoder hidden state h_i of the source sentence should be considered for each output (Ta-Chun, 2018; Bahdanau et al., 2016).
2. Next, the energy score e_{ti} is passed through a feed-forward neural network and softmax to calculate the attention scores in eq. (19), ensuring the attention vector $\vec{\alpha} = \left\{\alpha_{ti}\right\}$ has values normalized between 0 and 1:

$$\alpha_{ti} = \frac{\exp\left(e_{ti}\right)}{\sum_{k=1}^{T_x} \exp\left(e_{ik}\right)} \tag{19}$$

D.4.2 Forward Pass of Decoder

Once the attention scores have been calculated, the Decoder calculates the context vector, $c_t = \sum_{i=1}^{T_x} \alpha_{ti} \cdot h_i$, which is a sum of Encoder hidden states h_i weighted by attention scores $\vec{\alpha} = \left\{\alpha_{ti}\right\}$ (Ta-Chun, 2018). Intuitively, the context vector is an **expectation**.

Through this attention mechanism, the Seq-To-Seq model relieves the Encoder from having to compress all source sentence information into a fixed-length vector, allowing the information to spread through the hidden state sequence $\vec{h} = \left\{h_1, \ldots, h_T\right\}$ and later be detected by the Decoder (Trevett, 2020).

E Appendix: Self Attention Calculations in Transformer

Remember the following example sentence from the Transformer section: *"The animal didn't cross the road because it was too tired."*

Each word has an associated **query, key, value** vector which are created by multiplying the words embeddings with parameter weight matrices W^Q, W^K, W^V that are associated with the query, key, and value matrices, respectively. For the example, let input matrix be $X = \{\overrightarrow{x_1}, \overrightarrow{x_2}, ..., \overrightarrow{x_n}\}$, where vector $\overrightarrow{x_i}$ corresponds to word $\overrightarrow{w_i}$, and there are n words. Then the input word vectors are: $\overrightarrow{x_1}$ = "The", $\overrightarrow{x_2}$ = "animal", $\overrightarrow{x_3}$ = "didn't", $\overrightarrow{x_4}$ = "cross", $\overrightarrow{x_5}$ = "the", $\overrightarrow{x_6}$ = "road", $\overrightarrow{x_7}$ = because", $\overrightarrow{x_8}$ = "it", $\overrightarrow{x_9}$ = "was", $\overrightarrow{x_{10}}$ = "too", $\overrightarrow{x_{11}}$ = "tired", $\overrightarrow{x_{12}}$ = "." and the corresponding word embedding vectors are denoted $\{\overrightarrow{w_1}, \overrightarrow{w_2}, ..., \overrightarrow{w_n}\}$ and the **query, key, value** matrices are denoted $Q = \{\overrightarrow{q_1}, \overrightarrow{q_2}, ..., \overrightarrow{q_n}\}$, $K = \{\overrightarrow{k_1}, \overrightarrow{k_2}, ..., \overrightarrow{k_n}\}$, $V = \{\overrightarrow{v_1}, \overrightarrow{v_2}, ..., \overrightarrow{v_n}\}$ respectively.

E.0.1 Self-Attention: Vector Calculation

Using notation from Vaswani et al. (2017) and Alammar (2018b),

1. **Create Query, Key, Value Vectors** from each of the Encoder's input word embeddings $\overrightarrow{w_i}$ by multiplying them by appropriate rows in the three parameter matrices.
2. **Calculate a Score**: to determine how much *focus to place on other parts of the input sentence* while encoding a word at a certain position. The score is calculated by taking the dot product of the **query** and **key** vectors of the word being scored. Thus for word $\overrightarrow{w_i}$, the scores are:
$$\texttt{scores}_{w_i} = \left\{\overrightarrow{q_i} \cdot \overrightarrow{k_1}, \overrightarrow{q_i} \cdot \overrightarrow{k_2}, ..., \overrightarrow{q_i} \cdot \overrightarrow{k_n}\right\}$$
3. **Scale The Score**: The scores are scaled using d_k, the dimension of the key vector. From Vaswani et al. (2017), "for large values of d_k, the dot products grow large in magnitude, forcing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$." Thus the scores for $\overrightarrow{w_i}$ are:
$$\texttt{scores}_{\overrightarrow{w_i}} = \left\{\frac{\overrightarrow{q_i} \cdot \overrightarrow{k_1}}{\sqrt{d_k}}, \frac{\overrightarrow{q_i} \cdot \overrightarrow{k_2}}{\sqrt{d_k}}, ..., \frac{\overrightarrow{q_i} \cdot \overrightarrow{k_n}}{\sqrt{d_k}}\right\}$$
4. **Apply Softmax**: The softmax function normalizes the scores into probabilities.
$$\texttt{scores}_{\overrightarrow{w_i}} = \texttt{softmax}\left(\left\{\frac{\overrightarrow{q_i} \cdot \overrightarrow{k_1}}{\sqrt{d_k}}, \frac{\overrightarrow{q_i} \cdot \overrightarrow{k_2}}{\sqrt{d_k}}, ..., \frac{\overrightarrow{q_i} \cdot \overrightarrow{k_n}}{\sqrt{d_k}}\right\}\right)$$
5. **Compute the Weights**: The weighted values for word embedding $\overrightarrow{w_i}$ are calculated by multiplying each value vector in matrix V by the softmax scores. Intuitively, this cements the values of words to focus on while drowning out irrelevant words.
$$\texttt{weights}_{\overrightarrow{w_i}} = \texttt{scores}_{\overrightarrow{w_i}} * (\overrightarrow{v_1}, ..., \overrightarrow{v_n})$$
6. **Compute Output Vector**: The weight vector's cells are summed to produce the **output vector** of the self-attention layer for word embedding $\overrightarrow{w_i}$:
$$\overrightarrow{\texttt{output}_{w_i}} = \texttt{softmax}\left(\frac{\overrightarrow{q_i} \cdot \overrightarrow{k_1}}{\sqrt{d_k}}\right) \cdot \overrightarrow{v_1} + \texttt{softmax}\left(\frac{\overrightarrow{q_i} \cdot \overrightarrow{k_1}}{\sqrt{d_k}}\right) \cdot \overrightarrow{v_2} + ... + \texttt{softmax}\left(\frac{\overrightarrow{q_i} \cdot \overrightarrow{k_1}}{\sqrt{d_k}}\right) \cdot \overrightarrow{v_n}$$

E.1 Self-Attention: Matrix Calculation

Using notation from Vaswani et al. (2017) and Alammar (2018b),

1. **Calculate Query, Key, Value Matrices**: The word embeddings are packed into the rows of input matrix X and this is multiplied by each of the trained parameter matrices W^Q, W^K, W^V to produce the Q, K, V matrices:
$$\begin{aligned} Q &= X \cdot W^Q \\ K &= X \cdot W^K \\ V &= X \cdot W^V \end{aligned}$$
2. **Calculate Self Attention**: Steps 2 through 6 of the vector calculation for self attention can be condensed into a single matrix step where $Q = \{\overrightarrow{q_1}, \overrightarrow{q_2}, ..., \overrightarrow{q_n}\}$, $K = \{\overrightarrow{k_1}, \overrightarrow{k_2}, ..., \overrightarrow{k_n}\}$, $V = \{\overrightarrow{v_1}, \overrightarrow{v_2}, ..., \overrightarrow{v_n}\}$:
$$\texttt{Attention}(Q, K, V) = \texttt{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V$$

F Appendix: Glossary of NLP Tasks

Most of these definitions are from Collobert et al. (2011).

F.1 semantic parsing (SP)

Semantic parsing (SP) converts a natural language representation into machine-understanding form. Types include machine translation (MT) and question answering (QA).

F.2 key phrase extraction

Key phrase extraction task uses morphological, syntactic information, and typed grammar dependencies to learn relevant phrases, and is an important tool for concept extraction. Systems may use part of speech tagging (POS) and named entity recognition (NER) and named entity recognition (NER) to recognize that ``Air Canada" is a single entity composed of separate words which only when combined give new meaning (Mikolov et al., 2013a). In some cases, part of speech tagging (POS) uses noun phrases to extract key phrases.

Key phrase extraction is used in the real world to automate data collection; it results in benefits like data scalability and consistent criteria, so concepts can become used and interlinked at fine-grained levels ("Keyword Extraction", 2019).

F.3 machine translation (MT)

Machine translation (MT) task translates an input text in a given language to a target language. BERT uses the attention mechanism to account for contextual meaning across a sentence and not just translate word by word.

F.4 neural machine translation (NMT)

Neural machine translation (NMT) is machine translation (MT) applied in neural network language models.

F.5 question answering (QA)

Question answering (QA) is a task for machines to answer questions posed by humans in a natural language.

F.6 semantic role labeling (SRL)

Also called "shallow" semantic parsing (SP) , **semantic role labeling (SRL)** is often described as answering the question "Who did what to whom?" (Peters et al. 2018). It tries to give a semantic role or tag to a syntactic constituent of a sentence (Collobert et al., 2011). Examples of semantic roles are *agent, goal* or *result*. Specifically, it detects semantics associated to a syntactic sentence feature like predicate or verb and then assigns them semantic roles. SRL can give multiple labels depending on the usage of the syntactic constituent in the sentence.

F.7 named entity recognition (NER)

Named entity recognition (NER) is a kind of information extraction task that labels known entities in text into categories like "Person", "Location," and "Organization." An **entity** is a proper noun such as a person, place, or product. A proper noun is more specific than general nouns, which represent more ambigious concepts; for example, "Emma Watson", "Eiffel Tower" and "Second Cup" are entites but the corresponding nouns "actress", "architecture" and "store" are themes.

F.8 part of speech tagging (POS)

Part-of-speech tagging (POS) is the process of labeling each word in a sentence with its part of speech. Every word token is labeled with a tag that identifies its syntactic role (noun, verb, adverb, adjective, ...).

F.9 chunking

Chunking labels entire pieces of a sentence with tags that indicate their part of speech or syntactic role. For example a phrase can be labeled *noun phrase (NP)*, or *verb phrase (VP)* or even *begin-chunk (B-NP)* and *inside-chunk (I-NP)*. Each *phrase* token is assigned one distinct part of speech tag.

Chunking operates on the *phrase level* while part of speech tagging (POS) operates on the *word level*.

F.10 word sense disambiguation (WSD)

Word sense disambiguation (WSD) identifies the correct word usage from a collection of senses. For a sentences containing polysemous words, models use contextual evidence to determine the correct word sense.

F.11 lexical substitution

Lexical substitution substitutes a word given its contextual meaning. For example, the word "bright" in the phrase "bright child" can be replaced with "smart" or "gifted" rather than "shining" (Brazinkas et al., 2018).

F.12 entailment recognition (ER)

The **entailment recognition** task is a kind of lexical entailment task or hyponymy detection. Given a pair of words, the task is to predict if the first word w_1 entails the second one w_2. For ("kiwi", "fruit"), the task would be to confirm that "kiwi" entails "fruit" since it is its hyponym.

F.13 textual entailment (TE)

Textual entailment (TE) is the task of determining if a "hypothesis" is true given a "premise" (Peters et al., 2018).

F.14 sentiment classification (SC)

See sentiment analysis (SA).

F.15 sentiment analysis (SA)

Sentiment analysis (SA) evaluates the sentiment expressed towards an entity (noun or pronoun) based on its proximity to positive or negative words (adjectives and adverbs). For example, a model may classify a movie review as positive, negative or neutral. Generally, SA systems find several attributes of the expression alongside its *polarity*, including the *subject*, the thing being talked about, and *opinion holder*, the entity holding the opinion.

F.16 word similarity

The **word similarity** task determines a similarity score for two input texts. Numerical measures such as cosine similarity compute angular distance between words, based on textual evidence.

F.17 word analogy

The **word analogy** task completes an analogy. For instance, given the analogy "meteor" is to "sky" as "dolphin" is to <blank>, the task would be to predict a word representing a body of water.

F.18 coreference resolution (CR)

Coreference resolution (CR) is the task of collecting all expressions in a text that refer to the same entity in a text. that refer to the same underlying real world entities. For the example input text: "The monkey clambered up the baobab tree and he grabbed a banana and ate it there. The hairy ape screeched while watching the setting sun over the river", and example output would be to tag the coreferent phrases "he" and "the hairy ape" with "the monkey"; and also tag "it" with the same label as "banana."

F.19 semantic textual similarity (STS)

Semantic textual similarity determines similarity for two input texts by assigning a score. This measures semantic similarity, so text meaning rather than syntactic similarity. STS differs from both textual entailment (TE) and paraphrase detection because it detects meaning overlap rather than using a discrete classification of particular relationships. According to Maheshwari et al. (2018), although semantic similarity is characterized by a "graded semantic relationship", it may be not specify the nature of the relationship since contradictory words still may score highly. For instance, "night" and "day" are highly related but contradictory in nature.

F.20 tokenization

Tokenization or **segmentation** is the task-specific process of segmenting text into machine-understandable language. The term *tokens* describes words but also punctuation, hyperlinks, and possessive markers, such as apostrophes (Mohler, 2018). For example, lemma-based tokenization would specify that the tokens "cat" and plural "cats" would mean one word with the same stem or core meaning-bearing unit. Other forms of tokenization exist to differentiate word form, so those would be distinct tokens. Sentences and even characters can be tokenized out of a paragraph (Chromiak, 2017). Types of tokenization are **subword tokenization** and **sentence-piece tokenization**, a key feature in Transformer-XL.

F.21 transfer learning

Transfer learning or **domain adaptation** describes how a model created for one task is reused for a different task. Popularly used in machine learning where pre-trained models like BERT are used as starting points for downstream tasks that require much time and resources in order to train. Basically, knowledge of the first task is transferred to the second, often more specific, task (Brownlee, 2017).

F.22 natural language inference (NLI)

Natural language inference (NLI) is the task of predicting whether a *hypothesis* is true, false, or undetermined when the model is given a *premise*. An adapted example from Ruder (2020) is:

G Appendix: POS-Tagging with LSTM in AllenNLP

G.1 The Problem

Given a sentence like *"The dog ate the apple"* we want to predict part-of-speech tags for each word, like [DET, NN, V, DET, NN]. This is the nlp task called part of speech tagging (POS). This experiment will do this using the typed AllenNLP framework, which is built on top of PyTorch.

The basic workflow of what we will do is:

1. Embed each word in a low-dimensional vector space.
2. Pass each numericalized word as vector through a Long-Short Term Memory Networks (LSTM) to get a sequence of encodings.
3. Use a feedforward layer in the LSTM to transform the encodings into a sequence of logits, corresponding to the possible part-of-speech tags.

G.2 Prepare Inputs for the Model

```
from typing import Iterator, List, Dict

import torch
import torch.tensor as Tensor
import torch.optim as optim
import numpy as np
```

Each training example is represented in AllenNLP as an `Instance` containing `Field` s of various types. Each example (`Instance`) will be composed of two things: a `TextField` containing the sentence, and a `SequenceLabelField` containing the corresponding part of speech tags.

```
from allennlp.data import Instance
from allennlp.data.fields import TextField, SequenceLabelField
```

We must implement two classes, one of which is the `DatasetReader` , which contains the logic for reading a file of data and producing a stream of `Instance` s.

```
from allennlp.data.dataset_readers import DatasetReader
```

Frequently we need to load datasets or models from URLs. The `cached_path` helper downloads such files, then caches them locally, and then returns the local path. It also accepts local file paths (which it just returns as is).

```
from allennlp.common.file_utils import cached_path
```

There are various ways to represent a word as one or more indices. For example, there might be a vocabulary of unique words where each word is assigned a corresponding id. Or there might be one id per character in the word and so each word is represented as a sequence of ids. AllenNLP uses a `TokenIndexer` abstraction for this representation. Thus, the `TokenIndexer` abstraction represents a rule for converting a token (word) into indices.

```
from allennlp.data.token_indexers import TokenIndexer, SingleIdTokenIndexer
from allennlp.data.tokenizers import Token
```

While the `TokenIndexer` represents a rule for converting a token into indices, a `Vocabulary` is a dictionary of corresponding mappings from strings to integers. For instance, say the `TokenIndexer` specifies that a token should be represented as a sequence of character ids. This implies the `Vocabulary` contains the dictionary mapping `{character -> id}` .

For now, we use a `SingleIdTokenIndexer` that assigns each token a unique id, and so the `Vocabulary` will just contain a mapping `{token -> id}` as well as the reverse mapping.

```
from allennlp.data.vocabulary import Vocabulary
```

Along with `DatasetReader` the other class we would typically need to implement in AllenNLP is `Model` , which is a PyTorch `Module` that takes tensor inputs and produces a `dict` of tensor outputs and the training `loss` to be optimized.

```
from allennlp.models import Model
```

The POS tagger model we are building consists of an embedding layer, LSTM model, and feed forward layer in this order. AllenNLP includes abstractions for all of these components (imported as below) that handle padding and batching and various other utilities.

```
from allennlp.modules.text_field_embedders import TextFieldEmbedder, BasicTextFieldEmbedder
from allennlp.modules.token_embedders import Embedding
from allennlp.modules.seq2seq_encoders import Seq2SeqEncoder, PytorchSeq2SeqWrapper
from allennlp.nn.util import get_text_field_mask, sequence_cross_entropy_with_logits
```

`CategoricalAccuracy` is for tracking accuracy on training and validation datasets.

```
from allennlp.training.metrics import CategoricalAccuracy
```

In our training, we will need a `DataIterator` that can intelligently batch the data.

```
from allennlp.data.iterators import BucketIterator
```

The `Trainer` trains the model.

```
from allennlp.training.trainer import Trainer
```

The `SentenceTaggerPredictor` is for making predictions on new inputs.

```
from allennlp.predictors import SentenceTaggerPredictor
```

Now we can set the seed for reproducibility:

```
torch.manual_seed(1)
```

```
<torch._C.Generator at 0x7f0e8481ed10>
```

Step 1: Create the `DatasetReader` for POS Tagging

The first step is to create the `DatasetReader` for our particular POS tagging task. The following methods are essential to this class:

- `__init__()`: the only parameter `DatasetReader` needs is a dict of `TokenIndexer`s that specify how to convert tokens into indices. By default we generate a single index for each token (which we also call "tokens") that is a unique id for each distinct token. This is just the standard "word to index" mapping used in most NLP tasks.
- `text_to_instance()`: the `DatasetReader.text_to_instance` takes the inputs corresponding to a training example (in this case, the tokens of the sentence and corresponding part-of-speech tags), and instantiates the corresponding `Field`s: a `TextField` for the sentence, and a `SequenceLabelField` for its tags. Then `text_to_instance()` returns the `Instance` containing those fields. The tags are optional since we should have the option of creating `Instance`s from unlabeled data to make predictions on them.
- `_read()`: Takes a filename and produces a stream of `Instance`s, by harnessing the `text_to_instance()` method.

```
class PosDatasetReader(DatasetReader):
    def __init__(self, tokenIndexers: Dict[str, TokenIndexer] = None) -> None:
        super().__init__(lazy = False)
        self.tokenIndexers = tokenIndexers or {"tokens": SingleIdTokenIndexer()}

    def text_to_instance(self, tokens: List[Token], tags: List[str] = None) -> Instance:
        sentenceField = TextField(tokens = tokens,
                                  token_indexers= self.tokenIndexers)
        fields = {"sentence": sentenceField}
        if tags:
            labelField = SequenceLabelField(labels = tags,
                                            sequence_field= sentenceField)
            fields["labels"] = labelField

        return Instance(fields = fields)

    def _read(self, filePath: str) -> Iterator[Instance]:
        with open(filePath) as f:
            for line in f:
                pairs = line.strip().split()
                sentence, tags = zip(*(pair.split("###") for pair in pairs))
                yield self.text_to_instance([Token(word) for word in sentence], tags)
```

Step 2: Create the `LstmTagger` Class

In general for AllenNLP, we always must implement classes inheriting from `DatasetReader` and `Model` class. Here, the `LstmTagger` class inherits from the `Model` class. Since `Model` is a subclass of `torch.nn.Module`, it must implement a `forward` method that takes tensor inputs and produces a dictionary of tensor outputs that include the loss for training the model. The components of `LstmTagger` are:

- `__init__()`: One thing that might seem unusual is that we're going pass in the embedder and the sequence encoder as constructor parameters. This allows us to experiment with different embedders and encoders without having to change the model code. The arguments in the `__init__()` method are as follows:
  1. `wordEmbeddings: TextFieldEmbedder`: the embedding layer is specified as an AllenNLP `TextFieldEmbedder` which represents a general way of turning tokens into tensors. (Here we know that we want to represent each unique word with a learned tensor, but using the general class allows us to easily experiment with different types of embeddings, for example ELMo.)
  2. `encoder: Seq2SeqEncoder`: the encoder is similarly specified as a general `Seq2SeqEncoder` even though we know we want to use an LSTM. Again, this makes it easy to experiment with other sequence encoders, for example a Transformer.
  3. `vocab: Vocabulary`: Every AllenNLP model also expects a `Vocabulary`, which contains the namespaced mappings of tokens to indices and labels to indices.
- `forward()`: Actual calculations over the computational graph happen in this method. Each `Instance` in the data set will get batched with other `Instance`s and fed into `forward`. It takes in `dict`s of tensors, with names equal to the names of the fields in the `Instance`. NOTE: In this case we have a sentence field and possibly a labels field so we will construct the `forward` method accordingly.
- `get_metrics()`: We included an accuracy metric that gets updated each forward pass. That means we need to override a `get_metrics` method that pulls the data out of it. Behind the scenes, the `CategoricalAccuracy` metric is storing the number of predictions and the number of correct predictions, updating those counts during each call to `forward`. Each call to `get_metrics()` returns the calculated accuracy and (optionally) resets the counts, allowing us to track accuracy anew for each epoch.

```
class LstmTagger(Model):
    def __init__(self,
                 wordEmbeddings: TextFieldEmbedder,
                 encoder: Seq2SeqEncoder,
                 vocab: Vocabulary) -> None:

        # Notice: we have to pass the vocab to the base class constructor
        super().__init__(vocab)

        self.wordEmbeddings: TextFieldEmbedder = wordEmbeddings
        self.encoder: Seq2SeqEncoder = encoder

        # The feed forward layer is not passed in as parameter.
        # Instead we construct it here.
        # It gets encoder's output dimension as the feedforward layer's input dimension
        # and uses vocab's size as the feedforward layer's output dimension.
        self.hiddenToTagLayer = torch.nn.Linear(in_features = encoder.get_output_dim(),
                                                out_features= vocab.get_vocab_size(namespace =
                                                    ↪  'labels'))

        # Instantiate an accuracy metric to track it during training
        # and validation epochs.
        self.accuracy = CategoricalAccuracy()


    def forward(self,
                sentence: Dict[str, Tensor],
                labels: Tensor = None) -> Dict[str, Tensor]:

        # Step 1: Create the masks

        # AllenNLP is designed to operate on batched inputs, but
        # different input sequences have different lengths. Behind the scenes AllenNLP is
        # padding the shorter inputs so that the batch has uniform shape, which means our
        # computations need to use a mask to exclude the padding. Here we just use the utility
        # function get_text_field_mask, which returns a tensor of 0s and 1s corresponding to
        # the padded and unpadded locations.
        mask: Tensor = get_text_field_mask(text_field_tensors= sentence)
```

```
        # Step 2: create the tensor embeddings
        # We start by passing the sentence tensor (each sentence a sequence of token ids)
        # to the word_embeddings module, which converts each sentence into a sequence
        # of embedded tensors.
        # Does forward pass of word embeddings layer
        embeddings: Tensor = self.wordEmbeddings(sentence)

        # Step 3: Encode the embeddings using mask
        # We next pass the embedded tensors (and the mask) to the LSTM,
        # which produces a sequence of encoded outputs.
        # Does forward pass of encoder layer
        encoderOutputs: Tensor = self.encoder(embeddings, mask)

        # Step 4: Finally, we pass each encoded output tensor to the feedforward
        # layer to produce logits corresponding to the various tags.
        # Does forward pass of the linear layer
        tagLogits = self.hiddenToTagLayer(encoderOutputs)
        output = {"tagLogits": tagLogits}

        # As before, the labels were optional, as we might want to run this model to
        # make predictions on unlabeled data. If we do have labels, then we use them
        # to update our accuracy metric and compute the "loss" that goes in our output.
        if labels is not None:
            self.accuracy(predictions = tagLogits, gold_labels = labels, mask = mask)
            output["loss"] = sequence_cross_entropy_with_logits(logits = tagLogits,
                                                                targets = labels,
                                                                weights = mask)

        return output

    def get_metrics(self, reset: bool = False) -> Dict[str, float]:
        return {"accuracy": self.accuracy.get_metric(reset)}
```

Step 3: Start Training

Now that we've implemented a `DatasetReader` and `Model`, we're ready to train.

Instantiate a `DatasetReader` for POS tagging

We first need an instance of our `DatasetReader`.

```
reader = PosDatasetReader()
```

Download the Data

We can use the `PosDatasetReader` to read in the training data and validation data. Here we read them in from a URL, but you could read them in from local files if your data was local. We use `cached_path` to cache the files locally (and to hand `reader.read` the path to the local cached version.)

```
trainDataset: List[Instance] = reader.read(cached_path(
    'https://raw.githubusercontent.com/allenai/allennlp'
    '/master/tutorials/tagger/training.txt'))
validationDataset: List[Instance] = reader.read(cached_path(
    'https://raw.githubusercontent.com/allenai/allennlp'
    '/master/tutorials/tagger/validation.txt'))
trainDataset
validationDataset
```

```
2it [00:00, 5566.43it/s]
2it [00:00, 717.53it/s]

[<allennlp.data.instance.Instance at 0x7f0e1007ba20>,
 <allennlp.data.instance.Instance at 0x7f0dfe1b0828>]
```

Viewing what is inside the Training and Validation Data sets: it contains two `Instance`s, each equipped with `sentence` which has training `label`s that correspond to the parts of speech of the `sentence`.

```
for i in range(len(trainDataset)):
  print("Step {}:\n".format(i))
  print(str(trainDataset[i]))
```

```
Step 0:

Instance with fields:
    sentence: TextField of length 5 with text:
        [The, dog, ate, the, apple]
        and TokenIndexers : {'tokens': 'SingleIdTokenIndexer'}
    labels: SequenceLabelField of length 5 with labels:
        ('DET', 'NN', 'V', 'DET', 'NN')
        in namespace: 'labels'.

Step 1:

Instance with fields:
    sentence: TextField of length 4 with text:
        [Everybody, read, that, book]
        and TokenIndexers : {'tokens': 'SingleIdTokenIndexer'}
    labels: SequenceLabelField of length 4 with labels:
        ('NN', 'V', 'DET', 'NN')
        in namespace: 'labels'.
```

```
for i in range(len(validationDataset)):
  print("Step {}:\n".format(i))
  print(str(validationDataset[i]))
```

```
Step 0:

Instance with fields:
    sentence: TextField of length 5 with text:
        [The, dog, read, the, apple]
        and TokenIndexers : {'tokens': 'SingleIdTokenIndexer'}
    labels: SequenceLabelField of length 5 with labels:
        ('DET', 'NN', 'V', 'DET', 'NN')
        in namespace: 'labels'.

Step 1:

Instance with fields:
    sentence: TextField of length 4 with text:
        [Everybody, ate, that, book]
        and TokenIndexers : {'tokens': 'SingleIdTokenIndexer'}
    labels: SequenceLabelField of length 4 with labels:
        ('NN', 'V', 'DET', 'NN')
        in namespace: 'labels'.
```

Create the `Trainer`

Instantiating the `Trainer` and running it.

Setting the `patience = 10` means we run for 1000 epochs and stop training early if it ever spends 10 epochs without the validation metric improving. Note that the default validation metric is the loss, which improves by getting smaller, but we can also specify a different metric and direction (like accuracy, which should increase).

```
trainer = Trainer(model = posTagModel,
                  optimizer = optimizer,
                  iterator = iterator,
                  train_dataset = trainDataset,
                  validation_dataset = validationDataset,
                  patience = 10,
                  num_epochs = 1000,
                  cuda_device = cudaDevice)
trainer
```

```
<allennlp.training.trainer.Trainer at 0x7f0e1673a400>
```

When we launch it it will print a progress bar for each epoch that includes both the "loss" and the "accuracy" metric. If our model is good, the loss should go down and the accuracy up as we train.

```
trainer.train()
```

```
accuracy: 0.3333, loss: 1.1685 ||: 100%|██████████| 1/1 [00:00<00:00,  2.42it/s]
accuracy: 0.3333, loss: 1.1592 ||: 100%|██████████| 1/1 [00:00<00:00, 219.68it/s]
accuracy: 0.3333, loss: 1.1604 ||: 100%|██████████| 1/1 [00:00<00:00, 133.26it/s]
accuracy: 0.3333, loss: 1.1516 ||: 100%|██████████| 1/1 [00:00<00:00, 258.70it/s]
accuracy: 0.3333, loss: 1.1529 ||: 100%|██████████| 1/1 [00:00<00:00, 107.31it/s]
.....
accuracy: 0.4444, loss: 1.0128 ||: 100%|██████████| 1/1 [00:00<00:00, 95.56it/s]
accuracy: 0.4444, loss: 1.0110 ||: 100%|██████████| 1/1 [00:00<00:00, 205.28it/s]
....
accuracy: 0.4444, loss: 0.9270 ||: 100%|██████████| 1/1 [00:00<00:00, 145.82it/s]
accuracy: 0.4444, loss: 0.9242 ||: 100%|██████████| 1/1 [00:00<00:00, 259.24it/s]
accuracy: 0.5556, loss: 0.9248 ||: 100%|██████████| 1/1 [00:00<00:00, 135.84it/s]
....
accuracy: 0.6667, loss: 0.6577 ||: 100%|██████████| 1/1 [00:00<00:00, 261.36it/s]
accuracy: 0.6667, loss: 0.6565 ||: 100%|██████████| 1/1 [00:00<00:00, 125.30it/s]
accuracy: 0.6667, loss: 0.6541 ||: 100%|██████████| 1/1 [00:00<00:00, 383.32it/s]
...
accuracy: 0.7778, loss: 0.5787 ||: 100%|██████████| 1/1 [00:00<00:00, 231.13it/s]
accuracy: 0.7778, loss: 0.5768 ||: 100%|██████████| 1/1 [00:00<00:00, 143.38it/s]
accuracy: 0.8889, loss: 0.5751 ||: 100%|██████████| 1/1 [00:00<00:00, 241.66it/s]
accuracy: 0.8889, loss: 0.5732 ||: 100%|██████████| 1/1 [00:00<00:00, 96.03it/s]
....
accuracy: 0.8889, loss: 0.5479 ||: 100%|██████████| 1/1 [00:00<00:00, 121.25it/s]
accuracy: 0.8889, loss: 0.5464 ||: 100%|██████████| 1/1 [00:00<00:00, 267.37it/s]
accuracy: 1.0000, loss: 0.5443 ||: 100%|██████████| 1/1 [00:00<00:00, 128.88it/s]
.....
accuracy: 1.0000, loss: 0.0182 ||: 100%|██████████| 1/1 [00:00<00:00, 139.92it/s]
accuracy: 1.0000, loss: 0.0181 ||: 100%|██████████| 1/1 [00:00<00:00, 300.34it/s]
accuracy: 1.0000, loss: 0.0181 ||: 100%|██████████| 1/1 [00:00<00:00, 154.40it/s]
accuracy: 1.0000, loss: 0.0181 ||: 100%|██████████| 1/1 [00:00<00:00, 281.33it/s]
accuracy: 1.0000, loss: 0.0181 ||: 100%|██████████| 1/1 [00:00<00:00, 111.57it/s]
accuracy: 1.0000, loss: 0.0180 ||: 100%|██████████| 1/1 [00:00<00:00, 252.99it/s]
```

```
{'best_epoch': 999,
 'best_validation_accuracy': 1.0,
 'best_validation_loss': 0.01803998276591301,
 'epoch': 999,
 'peak_cpu_memory_MB': 3402.284,
 'peak_gpu_0_memory_MB': 813,
 'training_accuracy': 1.0,
 'training_cpu_memory_MB': 3402.284,
 'training_duration': '0:01:26.298209',
 'training_epochs': 999,
 'training_gpu_0_memory_MB': 813,
 'training_loss': 0.01809469424188137,
 'training_start_epoch': 0,
 'validation_accuracy': 1.0,
 'validation_loss': 0.01803998276591301}
```

G.3 Evaluate Model Outputs

Step 4: Generate Model Predictions

AllenNLP contains a `Predictor` abstraction that takes inputs, converts them to instances, and feeds them through the model and returns JSON-serializable results. Often we must implement a custom `Predictor` but here we can use `SentenceTaggerPredictor`. It takes in as parameters a `DatasetReader` for creating data instances, and our model, for making predictions.

```
predictor = SentenceTaggerPredictor(posTagModel, dataset_reader=reader)
predictor
```

```
<allennlp.predictors.sentence_tagger.SentenceTaggerPredictor at 0x7f0e1673af98>
```

The predictor object It has a `predict` method that just needs a sentence and returns (a JSON-serializable version of) the output `dict` from `forward()`. Here `tagLogits` will be an array of shape (5, 3) containing logits, corresponding to the 3 possible tags for each of the 5 words.

```
tagLogits = predictor.predict("The dog ate the apple")['tagLogits']
tagLogits
```

```
[[-0.9461401104927063, 3.022111415863037, -1.4651241302490234],
 [4.288924217224121, 0.07474756240844727, -4.382613182067871],
 [-3.532292604446411, 0.008625388145446777, 3.768585205078125],
 [-1.2684307098388672, 3.1266491413116455, -1.1786810159683228],
 [4.214295864105225, 0.0023859143257141113, -4.201425552368164]]
```

To get the actual "predictions" we can just take the `argmax`, as explained in Final Linear and Softmax Layer.

```
tagIndices = np.argmax(tagLogits, axis=-1)
```

Using the `Vocabulary` we can find the predicted tags of the inputted sentence.

```
print([posTagModel.vocab.get_token_from_index(i, 'labels') for i in tagIndices])
```

```
['DET', 'NN', 'V', 'DET', 'NN']
```

Above, we see the parts of speech have been correctly predicted:

1. `DET` is mapped correctly to articles like "the".
2. `NN` is mapped correctly to the nouns "dog" and "apple".
3. `V` is mapped correctly to the verb "ate".

References

1. Smith, and A., N. (2019, February 19). Contextual Word Representations: A Contextual Introduction. Retrieved from https://arxiv.org/abs/1902.06006
2. Melamud, O., Goldberger, et al. (2016). context2vec: Learning Generic Context Embedding with Bidirectional LSTM. *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*. doi: 10.18653/v1/k16-1006
3. Devlin, Jacob, et al. (2019, May 24). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Retrieved from https://arxiv.org/abs/1810.04805
4. Wiedemann, Gregor, et al. (2019, September 23). Does BERT Make Any Sense? Interpretable Word Sense Disambiguation with Contextualized Embeddings. Retrieved from https://arxiv.org/abs/1909.10430v1
5. Munikar, M., et al. (2019). Fine-grained Sentiment Classification using BERT. 2019 *Artificial Intelligence for Transforming Business and Society (AITB)*. doi: 10.1109/aitb48515.2019.8947435
6. Clark, K., et al. (2019). What Does BERT Look at? An Analysis of BERT's Attention. *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. doi: 10.18653/v1/w19-4828
7. Ethayarajh, K. (2019). How Contextual are Contextualized Word Representations? Comparing the Geometry of BERT, ELMo, and GPT-2 Embeddings. *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. doi: 10.18653/v1/d19-1006
8. Batista, D. (n.d.). Language Models and Contextualised Word Embeddings. Retrieved from http://www.davidsbatista.net/blog/2018/12/06/Word_Embeddings/
9. Neelakantan, A., et al. (2014). Efficient Non-parametric Estimation of Multiple Embeddings per Word in Vector Space. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. doi: 10.3115/v1/d14-1113
10. Antonio, M. (2019, September 5). Word Embedding, Character Embedding and Contextual Embedding in BiDAF - an Illustrated Guide. Retrieved from https://towardsdatascience.com/the-definitive-guide-to-bidaf-part-2
11. Mikolov, T., Sutskever, I., et al. (2013a, October 16). Distributed Representations of Words and Phrases and their Compositionality. Retrieved from https://arxiv.org/pdf/1310.4546.pdf
12. Mikolov, Tomas, et al. (2013b, September 7). Efficient Estimation of Word Representations in Vector Space. Retrieved from https://arxiv.org/abs/1301.3781
13. Weng, L. (2017, October 15). Learning Word Embedding. Retrieved from https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html
14. Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global Vectors for Word Representation.*Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.doi: 10.3115/v1/d14-1162
15. Sutskever, I., et al. (2014, December 14). Sequence to Sequence Learning with Neural Networks. Retrieved from https://arxiv.org/abs/1409.3215
16. Vaswani, Ashish, et al. (2017, December 6). Attention Is All You Need. Retrieved from https://arxiv.org/abs/1706.03762
17. G, R. (2019, March 18). Transformer Explained - Part 1. Retrieved from https://graviraja.github.io/transformer/
18. Ta-Chun. (2018, October 3). Seq2seq pay Attention to Self Attention: Part 1. Retrieved from https://medium.com/@bgg/seq2seq-pay-attention-to-self-attention-part-1-d332e85e9aad
19. Alammar, Jay. "Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)." *Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention) – Jay Alammar – Visualizing Machine Learning One Concept at a Time*, 2018a, jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq
20. Alammar, J. (2018b, June 27). The Illustrated Transformer. Retrieved from http://jalammar.github.io/illustrated-t
21. Peters, et al. "Deep Contextualized Word Representations." *ArXiv.org*, (22 Mar. 2018), arxiv.org/abs/1802.05365.
22. Dai, Zihang, et al. "Transformer-XL: Attentive Language Models beyond a Fixed-Length Context." *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, (2019), https://arxiv.org/pdf/1901.02860.pdf.
23. Yang, Z, et al. "XLNet: Generalized Autoregressive Pretraining for Language Understanding." *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, (2020), https://arxiv.org/pdf/1906.08237.pdf.
24. Sun, Y, et al. "ERNIE: Enhanced Representations Through Knowledge Integration." *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, (2019a), https://arxiv.org/pdf/1904.09223.pdf.
25. Sun, Y, et al. "ERNIE 2.0: A Continual Pre-Training Framework for Language Understanding." *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, (2019b), https://arxiv.org/pdf/1907.12412.pdf.
26. Colah. "Deep Learning, NLP, and Representations." *Deep Learning, NLP, and Representations - Colah's Blog*, 2014, colah.github.io/posts/2014-07-NLP-RNNs-Representations/.

27. Colah. "Understanding LSTM Networks." *Understanding LSTM Networks – Colah's Blog*, 2015, colah.github.io/posts/2015-08-Understanding-LSTMs/.
28. Mohler. "The 7 Basic Functions of Text Analytics." *Lexalytics*, 20 Jan. 2019, www.lexalytics.com/lexablog/text-analyt
29. Chromiak, Michał. "NLP: Explaining Neural Language Modeling." *Michał Chromiak's Blog*, 30 Nov. 2017, mchromiak.github.io/articles/2017/Nov/30/Explaining-Neural-Language-Modeling/.
30. Rudolph, M., and Blei, D. (2018). Dynamic Embeddings for Language Evolution. *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW 18*. Retrieved from https://arxiv.org/abs/1703.08052
31. Rudolph, and Maja R., R. (2016, November 21). Exponential Family Embeddings. Retrieved from https://arxiv.org/abs/1608.00778
32. Lenci, A. (2018). Distributional Models of Word Meaning. *Annual Review of Linguistics*, 4(1), 151–171. Retrieved from https://www.annualreviews.org/doi/abs/10.1146/annurev-linguistics-030514-125254
33. Kurita, K. (2018a, May 4). Paper Dissected: "Glove: Global Vectors for Word Representation." *Machine Learning Explained*. Retrieved from http://mlexplained.com/2018/04/29/paper-dissected-glove-global-vectors-for-wor
34. Kurita, K. (2018b, June 15). Paper Dissected: "Deep Contextualized Word Representations." *Machine Learning Explained*. Retrieved from http://mlexplained.com/2018/06/15/paper-dissected-deep-contextualized-word-repr
35. Kurita, K. (2019a, January 30). Paper Dissected: "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." *Machine Learning Explained*. Retrieved from https://mlexplained.com/2019/01/07/paper-dissected-
36. Kurita, K. (2019b, July 7). Paper Dissected: "XLNet: Generalized Autoregressive Pretraining for Language Understanding." *Machine Learning Explained*. Retrieved from https://mlexplained.com/2019/06/30/paper-dissected-xlnet-gene
37. Ruder, S. (2019, October 6). On word embeddings - Part 1. Retrieved from https://ruder.io/word-embeddings-1/
38. Ibrahim, A. H. (2019, June 30). A Study on CoVe, Context2Vec, ELMo, ULMFiT and BERT. Retrieved from https://ahmedhanibr wordpress.com/2019/07/01/a-study-on-cove-context2vec-elmo-ulmfit-and-bert/
39. Bengio, Y. (2013, March 2). A Neural Probabilistic Language Model. Retrieved from http://www.jmlr.org/papers/volume3/bengio03a/
40. Rong, X. (2016, June 5). word2vec Parameter Learning Explained. Retrieved from https://arxiv.org/abs/1411.2738
41. Nguyen, M. (2018a, July 10). Illustrated Guide to Recurrent Neural Networks. Retrieved from https://towardsdatascience com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9
42. Nguyen, M. (201b8, July 10). Illustrated Guide to LSTM's and GRU's: A step by step explanation. Retrieved from https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-
43. Cho, et al. (2014, June 3). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. Retrieved from https://arxiv.org/abs/1406.1078v1
44. Trevett, B. (2020, February 25). bentrevett/pytorch-seq2seq. Retrieved from https://github.com/bentrevett/pytorch
45. Jangid, A. (2019, September 16). Intuitive Understanding of Seq2seq model and Attention Mechanism in Deep Learning. Retrieved from https://medium.com/analytics-vidhya/intuitive-understanding-of-seq2seq-model-a
46. Bahdanau et al. (2016, May 19). Neural Machine Translation by Jointly Learning to Align and Translate. Retrieved from https://arxiv.org/abs/1409.0473
47. Collobert, R et al. (2011, March 2). Natural Language Processing (almost) from Scratch. Retrieved from https://arxiv.org/pdf/1103.0398v1.pdf
48. Brazinkas, A et al. (2018, June 10). Embedding Words as Distributions with a Bayesian Skip-gram Model. Retrieved from https://arxiv.org/abs/1711.11027
49. Maheshwari. (2018, June 21). anantm95/Semantic-Textual-Similarity. Retrieved from https://github.com/anantm95/Semantic-Textual-Similarity
50. Weng, L. (2019, January 31). Generalized Language Models. Retrieved from https://lilianweng.github.io/lil-log/2019/01/31/generalized-language-models.html
51. Ruder, S. (2020, February 25). Natural language inference. Retrieved from http://nlpprogress.com/english/natural language_inference.html
52. Brownlee, J. (2019, September 16). A Gentle Introduction to Transfer Learning for Deep Learning. Retrieved from https://machinelearningmastery.com/transfer-learning-for-deep-learning/
53. Keyword Extraction. (2019, August 7). Retrieved from https://monkeylearn.com/keyword-extraction/