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

(!) TODO: This is just some text here that i must later change in order to complete this presentation. This is some text that has overlap with the introduction, summarizes the project, and does not contain referencs. Oh, also remember it can be read independently of the entire report itself.

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 and between human cognition and AI-based systems.

It is known that concept mapping enhances human cognition. Especially in domain-specific areas of knowledge,

better interlinking would be achieved if concepts would be extracted using surrounding context, accounting for polysemy and key phrases. "You shall know a word by the company it keeps" (Firth, 1957).

In my project I seek to understand models that create good language representations using lexical and semantic structure, at the *entity and phrase level*.

Previous count-based models like GloVe and Word2Vec motivated recent models like models like Transformer, ELMo, BERT, Transformer-XL, XLNet, and ERNIE 2.0 to move beyond simple co-occurrence counts to extract meaning. ERNIE 2.0 instead "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). For instance, ERNIE 1.0 can associate entire entity names with other terms in a given sentence, while on the same data, BERT lacks this ability.

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 the PyTorch deep learning library, I aim to illustrate key model architecture while applying the model to machine translation (MT).

Statement of Authorship:

This report is planned and written entirely by me, and I cite authors of each model, where applicable. I learned from and adapted the PyTorch Code for the machine translation (MT)task using GitHub.

2 Word Embeddings

Word embeddings, also called latent vector representations, which are fixed-length vector representations of words, 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).

2.1 Usage of Word Embeddings in Natural Language Processing

An important idea in linguistics is that words used in similar ways have similar meanings (Firth, 1957). A distributional view of word meaning arises when accounting for the full distribution of contexts in a corpus where the word is found. For instance, words that tend to occur in the same neighboring context can be clustered to signify they have similar meaning. 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. With the onset of ever-larger text collections on the web, these programs have progressed from count-based statistics to more advanced methods. There are many insights into the power of word embeddings; similar words being close together allows generalization from one sentence to a class of similar sentences. For instance "the wall is blue" to "the ceiling is red" (Smith, 2019, p. 4). Put succinctly, "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).

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; for neural networks, this means one neuron is active at a time. The n-gram model is considered a local representation due to is usage of 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. Embedding methods analyze text data, learning distributed semantic representations of the vocabulary to capture its co-occurrence statistics. These learned representations are then useful for reasoning about word usage and meaning (Melamud et al. 2016, p. 1).

Word vectors can be also calculated from sentences, phrases, or characters to create sentence embedding, phrase embedding, or character embedding, respectively. Character embeddings can be used to explain language morphology. For example, the following variants of the word "would" in social media would have similar character embeddings because they are spelled similarly: "would", "wud", "wld", "wuld", "wouldd", "woud", and so on (Smith, 2019, p. 5). Tokenization is a key step in segmenting text to create word embeddings, as the difference between BERT and Transformer-XL will show.

2.2.1 Analogical Reasoning Property of Word Embeddings

Word embeddings can also represent analogies that have been encoded in the difference vectors between words. For example, gender differences can be represented by a constant difference vector, enabling mathematical operations between vectors based on ***vector space semantics*** (Colah, 2014). The famous analogy "man is to woman as king is to queen" can thus be expressed using learned word vectors as follows: $vector(man) - vector(woman) = vector(king) - vector(queen)$ (Smith, 2019). In the NLP task of machine translation (MT), this property of learned word vectors would suggest the two languages being translated have a similar 'shape' and that 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 Overview For Word Embeddings

A word embedding $W : [Words] \rightarrow R^n$ is a parametrized function mapping words in a language to an n-dimensional numeric vector. An example is shown in fig. 1:

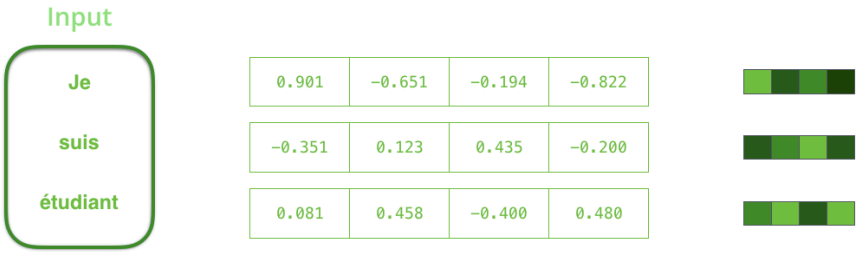


Figure 1: Example Word Embeddings. From *Visualizing Neural Machine Translation Mechanics of Seq2Seq Models with Attention*, by Jay Alammar, 2018. http://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/. Copyright 2018 by Jay Alammar.

According to 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) note that a word embedding uses vector representations to parameterize the conditional probabilities of words in a surrounding context. In other words, 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 given text data by maximizing the conditional probabilities of observed text (Rudolph et al. 2017).

2.4 Static Embeddings vs. Contextual Embeddings

2.4.1 What is Polysemy?

Polysemy means that a word can have multiple, distinct meanings. The **distributional hypothesis** in NLP states that meaning depends on context, and words occurring in the 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 in a static way: this means each word has a single word vector representation regardless of its context (Ethayarajh, 2019). Skip-Gram Model (Mikolov et al., 2013a) and GloVe (Pennington et al., 2014) are well-known algorithms for producing these "context-independent representations," as Peters et al. (2018) calls them, due to the fact that their word embedding matrix, inputted to a neural network representation, is trained to use co-occurring information in text, rather than using dynamic computation offered by language models (Batista, 2018). Although still able to capture latent syntactic and semantic meaning by training over large corpora, static embeddings by definition create a single vector representation per word, so all senses of a polysemous word are collapsed within a single representation (Ethayarajh, 2019). This can significantly reduce model performance. 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 the Annual Review of Linguistics, Lenci (2018) states that "distributional semantics is a usage-based model of meaning, based on the assumption that the statistical distribution of linguistic items in context plays a key role in characterizing their semantic behavior".

Rudolph et al. (2016) states that "each data point i has a *context* c, which is a set of indices of other data points." Context is also a modeling choice; in different domains, context can differ. In language, 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).

A **contextual word embedding (CWE)** is usually obtained using a bidirectional language model (biLM) to capture contextual information using forward and backward history to incorporate surrounding phrases of the word (Antonio, 2019). Typically, a model uses an encoder to process input sequence and squeeze the information into a fixed-length context vector. While word vectors are "lookup tables", contextual embeddings include type information and neural network parameters to "contextualize" a word (Smith, 2019).

Recent efforts to capture polysemy for word embeddings cast aside the idea of using a fixed word sense inventory. This allows contextual embeddings to "not only create one vector representation for each [word] type in the vocabulary" but to also create separate vectors for each token in a surrounding context. Indeed, experiments show that contextual embeddings can capture word senses successfully (Wiedemann et al., 2019). Wiedemann concludes that this allows for a more realistic model of natural language; contextual embeddings have proven their superiority over static embeddings for many NLP tasks such as text classification (Zampieri et al., 2019) and sequence tagging (Akbig et al., 2018). Although contextualization models such as the Transformer, BERT, ELMo, Transformer-XL, XLNet, and ERNIE 2.0 differ widely, modeling "sentence or context-level semantics together with word-level semantics proved to be a powerful innovation" in the NLP world (Wiedemann et al., 2019).

3 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 work by computing the conditional probability of a word w_t given a context, such as its previous $n-1$ words, where the probability is: $P(w_t|w_{t-1},...,w_{t-n+1})$ (Ruder, 2016). Chromiak adds that the probability chain rule is the main tool used to find the joint probability of a word sequence. For events A and B, the probability chain rule states:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

which lends the following formula for a set of T word tokens $w_1,...,w_T$ from a sentence S:

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

Typically, the **Markov Assumption**, which states that the probability of a word depends only on its previous word, is used to reduce context history and thus intake of model data. Thus the joint probability is estimated using the n previous words of the current word w_t:

$$P\left(w_1,...,w_T\right) \approx \prod_{t=1}^{T} P\left(w_t \mid w_{t-1},...,w_{t-n+1}\right)$$

There are several kinds of language models.

3.1 N-gram Language Model

An n-gram is a sequence of n words. The n-gram language model is one of the simplest models that 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):

$$P\left(w_t \mid w_{t-1},...,w_{t-n+1}\right) = \frac{count\left(w_{t-n+1},...,w_{t-1},w_t\right)}{count\left(w_{t-n+1},...,w_{t-1}\right)}$$

3.2 Neural Network Language Model

3.2.1 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).

3.2.2 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, which is expressed as a linear formula: $W \cdot x + b$. By applying a nonlinear function $f(\cdot)$ to this equation and by incorporating many hidden layers and by stacking neurons together, a neural network can model any function. A neuron is shown in fig. 2.

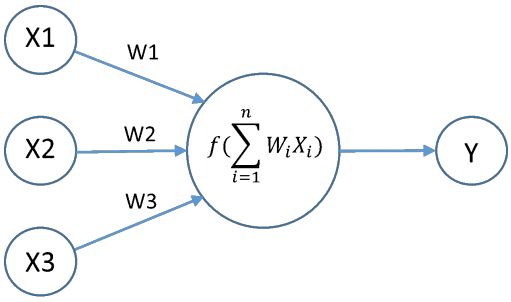


Figure 2: Neuron. From *Applying Unsupervised Machine Learning to Sequence Labeling*, by Jacob, 2019. https://medium.com/mosaix/deep-text-representation-for-sequence-labeling-2f2e605ed9d. Copyright n.d. by n.d.

Many NLP applications using neural networks feed in word tokens that is transformed based on its context words, resulting in an updated version of the word embedding (Smith, 2019). Embeddings are fed as parameters or weights into a neural network which *optimizes* them to best fit the text by minimizing a continuous loss function using gradient-descent based algorithms. Every neural network representation consists of three components (Ruder, 2016):

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.

$$P\left(w_t \mid w_{t-1}, \ldots, w_{t-n+1}\right) = \frac{\exp\left(h^T \cdot v'_{w_t}\right)}{\sum_{w_i \in V} \exp\left(h^T \cdot v'_{w_i}\right)}$$

where V = vocabulary of a corpus, h = output vector of the hidden layer, and v'_w = the output embedding of word w.

Many neural networks are **feed-forward neural network (FNN)**s, in which input is fed in the forward direction only.

3.2.3 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. The model generalizes to unseen data successfully

because unseen word sequences get a high probability if containing words that are similar to words that were already seen.

Advantageously, this model can capture longer contexts better than the n-gram, which is limited to short contexts. As a result of continuous word vector representations, the learned probability function's parameters increase linearly not exponentially with the vocabulary size, and they increase linearly with vector dimension, thus resolving the curse of dimensionality (Bengio et al., 2003).

For more information, APPENDIX: Training Neural Networks mathematically explains how neural networks learn parameters.

3.3 Bidirectional Language Model

3.3.1 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 $(t_1, t_2, ..., t_N)$, 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, $(t_1, ..., t_{k-1})$:

$$P(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} P(t_k \mid t_1, t_2, ..., t_{k-1})$$

3.3.2 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:

$$P(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} P(t_k \mid t_{k+1}, t_{k+2}, ..., t_N)$$

3.3.3 Combining Forward and Backward

A **bidirectional language model (biLM)** such as in fig. 3 combines the forward and backward language models and uses maximum likelihood estimation to *jointly* maximize the log likelihood of the forward and backward directions:

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

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

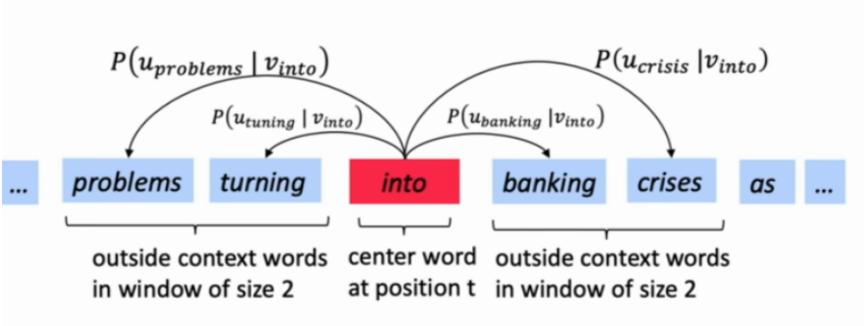


Figure 3: Example Bidirectional Language Model. From *Word2Vec Overview With Vectors*, by CS224n: Natural Language Processing with Deep Learning (Stanford), 2018. https://sangminwoo.github.io/2019-08-28-cs224n-lec1/. Copyright n.d. by n.d.

Akbik et al. (2018) use the hidden states of a bidirectional recurrent neural network to create contextualized word representations.

For example, 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 the polysemy of 'bank.' But a bidirectional language model represents "bank" using both previous and next context to ameliorate this problem.

4 Word2Vec

4.1 Word Embedding Representations: Count-Based vs. Context-Based

Word embeddings can be learned using two kinds of contextual vector space models: the **count-based** or **context-based vector space models**.

Count-based vector space models are unsupervised learning algorithms based on matrix factorization of a global word co-occurrence matrix. The main assumption is words in similar contexts share related semantic meanings. Examples include PCA and neural probabilistic language models. Another term for this type is **distributed semantic model** (DSM) (Weng, 2017).

Context-based vector space models are supervised algorithms that use a local context to predict target words. These are predictive models which take dense word vectors as parameters and update word representations during training.

In 2014, Baroni et al. showed that predictive approaches outperformed count models significantly and consistently.

Although **Word2Vec** and GloVe are predictive and context-based vector space models, they still rely on co-occurrence counts.

Word2Vec is an unsupervised learning algorithm for obtaining word vector representations using a two-layer neural network. Its input is the text corpus and output is the set of feature vectors, as one vector per word. Existing word representations already capture linguistic patterns, allowing algebraic operations to be done on the word vectors in their semantic vector space; for example, $vector("King") - vector("Man") + vector("Woman")$ outputs a vector close to the word vector for "Queen". But Mikolov et al. (2013b) created the Skip-Gram Model and Continuous Bag of Words Model (CBOW) as part of **Word2Vec** to learn word vector representations more efficiently from large data. Previous architectures used fewer words and smaller vector dimensionality, thus reducing the quality of the learned representations.

4.1.1 One-Hot Encodings

A key concept for what follows is **one-hot vector encoding**. This is the simplest type of word embedding where each cell in the vector corresponds to a distinct vocabulary word, thus its dimension equals the vocabulary size. A 1 is placed in the cell marking the position of the word in the vocabulary, and a 0 is placed in all other cells. However, this can result in high-dimensionality vector representations for large vocabularies, causing increased computational costs, and similarity between categories cannot be represented.

4.2 Basic Structure of Skip-Gram and CBOW

Both the Skip-Gram Model and Continuous Bag of Words Model (CBOW) are neural network language models with one hidden layer.

The input vector $\vec{x} = (x_1, ..., x_V)$ and output vector $\vec{y} = (y_1, ..., y_V)$ are both **one-hot encodings**, and the hidden layer of the neural network is a word embedding with dimension N. (Thus even though the vocabulary size is V, the goal is to learn embeddings with size N). For a specific time step t, the model predicts one output word w_{t+j} whose vector representation is $\vec{y}$, given one input word w_t, whose vector representation is $\vec{x}$.

For **Skip-Gram**, w_{t+j} is the predicted context word and w_t is the input target word, but for **CBOW** w_{t+j} is the predicted target word and w_t is the input context word.

Vectors v_w and v'_w are two representations of word w. Vector v_w comes from the rows of the *input layer → hidden layer weight matrix* W, and vector v'_w comes from the rows of the *hidden layer → output layer weight matrix* W'. We call v_w the **input vector** and v'_w is the **output vector** of the word w.

4.2.1 Skip-Gram Model

The Skip-Gram model predicts context words given a single target word. It uses a fixed sliding window c, or size of the training context, to capture context along a sentence. A target word thus has left and right context words within that sliding window. This target center word is represented as a one-hot encoding that is input to a neural network which updates the vector with values near 1 in cells corresponding to predicted context words (Weng, 2017).

Consider the following sentence from Weng (2017): "The man who passes the sentence should swing the sword." Using a context window size of $c = 5$ 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").

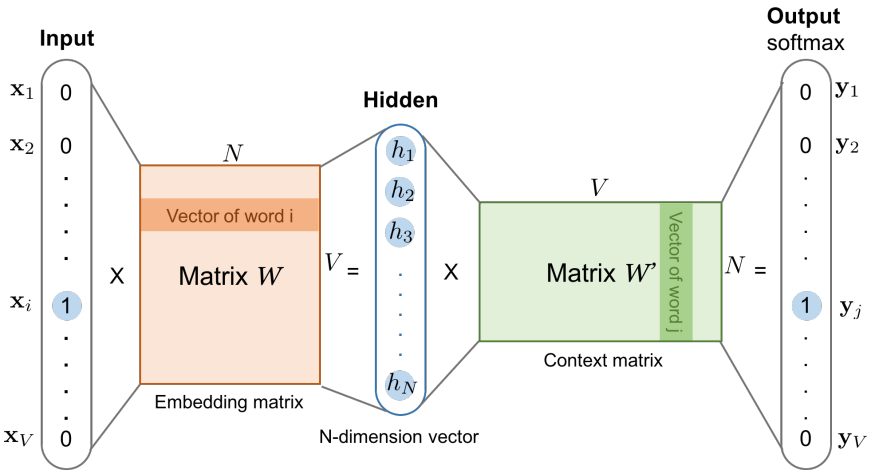


Figure 4: Skip-Gram Model; simplified version, with one input target word and one output context word. From *Learning Word Embeddings*, by Lilian Weng, 2017. https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html. Copyright n.d. by n.d.

4.2.2 Forward and Backward Pass for Skip-Gram

According to the Skip-Gram illustrated in fig. 4, the procedure for learning word vectors is:

1. the input word w_i and output word w_j are encoded as one-hot vectors, $\vec{x}$ and $\vec{y}$ respectively. (For Skip-Gram, $\vec{x}$ is the target vector and $\vec{y}$ is the context vector).
2. A randomly initialized word embedding matrix W with size $V \times N$ at the input → hidden layer is multiplied with $\vec{x}$ to give the N-dimensional embedding for target word w_i. This embedding resides in the i-th row of W and is considered as the hidden layer of the model.
3. Next, the hidden layer is multiplied by weight matrix W' with size $N \times V$ to produce the one-hot encoded output vector, $\vec{y}$. **NOTE:** the output context matrix W' encodes words as context and is distinct from the embedding matrix W.
4. The result of the above multiplication is sent through the softmax layer to create a probability distribution over the words. The above steps constitute the forward pass.

5. Errors are obtained by subtracting the output vector with the target vector.
6. The error vector is backward-propagated through the neural network to update the weight matrix. The procedure continues until errors are small enough.

4.2.3 Loss Function for Skip-Gram

The loss function is a key step for backward propagating errors through the neural network. Mikolov et al. (2013a) defines the loss function to be able to find word representations to predict the output word w_{t+j} given an input word w_t. Formally, given the training word sequence $w_1, w_2, ..., w_T$ with T training samples, the Skip-Gram maximizes the average log probability

$$J_\theta = \frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log\left(P\left(w_{t+j} \mid w_t\right)\right)$$

where c is the training size context window.

From Peters et al. (2013), the above probability in the loss function is:

$$P\left(w_{t+j} \mid w_t\right) = \frac{\exp\left((v'_{w_{t+j}})^T v_{w_t}\right)}{\sum_{w=1}^{V}\exp\left((v'_w)^T \cdot v_{w_t}\right)}$$

where w_t is the inputted target word, w_{t+j} is the predicted context word, and V is the number of vocabulary words.

4.2.4 Continuous Bag of Words Model (CBOW)

The continuous bag of words model (CBOW) is opposite of the Skip-Gram since it predicts the *target* word based on a *context* word. In the general case, CBOW receives a window of n context words around the target word w_t at each time step t and tries to predict the target word. The CBOW in fig. 5 is called "continuous" bag-of-words since it uses continuous distributed representations of the context words whose order is not important (Mikolov et al., 2013b).

4.2.5 Forward and Backward Pass for CBOW

The forward pass for CBOW is similar to Skip-Gram's. The key difference is due to having multiple context words: the CBOW averages the context word vectors while multiplying the input vector $\vec{x}$ and input $\rightarrow$ hidden layer matrix W. Rong (2016) describes the hidden layer calculation as:

$$\begin{aligned}\vec{h} &= \tfrac{1}{c}W\cdot\left(\overrightarrow{x_1}+\overrightarrow{x_2}+...+\overrightarrow{x_c}\right)\\ &= \tfrac{1}{c}\cdot\left(\overrightarrow{v_{w_1}}+\overrightarrow{v_{w_2}}+...+\overrightarrow{v_{w_c}}\right)\end{aligned}$$

where c is the number of context words, $w_1, ..., w_c$ 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 it better suited for small datasets.

4.2.6 Loss Function for CBOW

On the output layer, the CBOW outputs c multinomial distributions, for each of the c context words. The training object is to maximize the conditional probability of observing the true output word given several context words, while accounting for their weights.

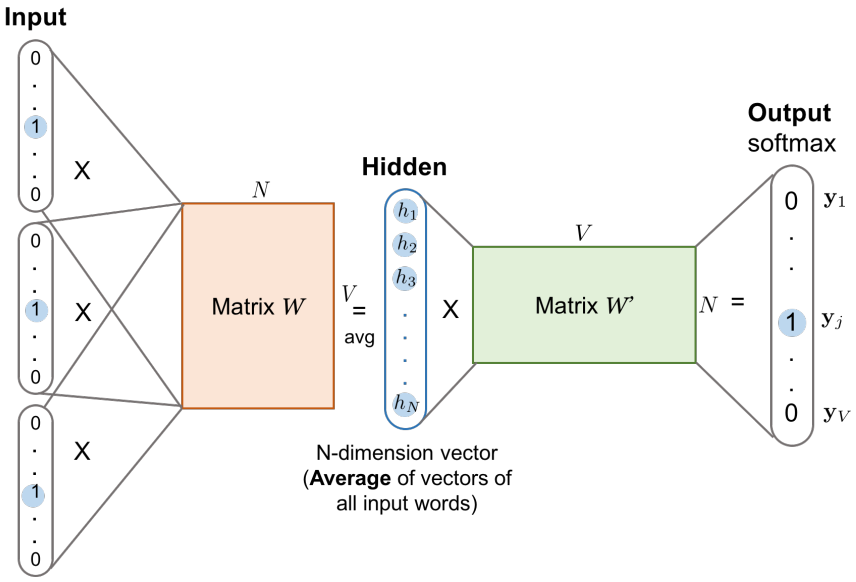


Figure 5: CBOW Model with several one-hot encoded context words at the input layer and one target word at the output layer. From *Learning Word Embeddings*, by Lilian Weng, 2017. https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html. Copyright n.d. by n.d.

From Ruder (2016), the loss function is:

$$J_\theta = \frac{1}{T}\sum_{t=1}^{T} \log\left(P\left(w_t \mid w_{t-n}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+n}\right)\right)$$

where n is number of context words and $w_{t-n}, \ldots, w_{t+n}$ are context words around the target word w_t.

From Rong (2016), the above probability in the loss function, for the one context word case, is:

$$P\left(w_t \mid w_I\right) = \frac{\exp\left((v'_{w_O})^T v_{w_I}\right)}{\sum_{w=1}^{V} \exp\left((v'_w)^T \cdot v_{w_I}\right)}$$

where w_I is the input context word, w_O is the predicted target word, and V is the number of vocabulary words.

4.3 Phrase-Learning in Word2Vec Using Skip-Gram

Mikolov et al. (2013a) state that a problem with previous word vectors is their lack of phrase representation - the phrases "Canada" and "Air" could not be recognized as part of a larger concept and thus 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 their own unique tokens in the training data. In contrast, a bigram like "this is" should remain unchanged (Mikolov et al., 2013a, p. 5).

To solve this issue, the Skip-Gram was updated using a data-driven scoring technique. Phrases are formed based on unigram and bigram counts:

$$S_{phrase} = \frac{C(w_i w_j) - \delta}{C(w_i)C(w_j)}$$

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 formation of infrequent words and phrases. High values of S_{phrase} means the phrase is most likely a phrase rather the simple concatenation of two words.

Regarding model accuracy, Mikolov et al. (2013a) found that the phrase Skip-Gram model with hierarchical softmax and subsampling performed significantly better on a large data set than the original Skip-Gram without the phrase feature.

It was found that the phrase Skip-Gram exhibits a linear structure **additive compositionality** of word vectors, which allows words to be combined meaningfully by adding their word vectors. For instance, composing word vectors *vector*("French") + *vector*("actress") results in the phrase *vector*("Juliette Binoche").

This **additive compositionality** feature of word vectors stems from the loss function's formulation. Since learned context vectors can represent the overall distribution of context words in which the target word appears, and since the vectors 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 functions 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*("Russian") + *vector*("river") results in the phrase *vector*("Volga River") or at least a vector close to it (Mikolov et al., 2013a).

5 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 only local contextual information (Pennington et al., 2014). The GloVe authors show that it is the *ratio* of co-occurrence probabilities of two words rather than their actual probabilities that contains meaning and they carve out this information as vector offsets.

5.1 Problem with Word2Vec: Secret in the Loss Function

A major deficiency in Word2Vec is that it only accounts for local contexts and ignores global count information. Kurita (2018) exemplifies that the words "the" and "cat" might be used together frequently but Word2Vec doesn't know if this is because "the" is a common word or because "the" and "cat" are actually correlated.

Pennington et al. (2014) say that Word2Vec implicitly optimizes over a co-occurrence matrix while streaming over input sentences. The key point is that Word2Vec optimizes the log likelihood of seeing words in the same context windows together, resulting in the below alternative way of expressing Word2Vec's loss function:

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

where $X_i = \sum_k X_{ik}$ 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 a softmax: $Q_{ij} = \text{softmax}\left(w_i \cdot w_j\right)$. This shows that the loss of Word2Vec is just another form for cross entropy between the predicted and actual word distributions found in the context of word i. However, the authors of GloVe say that cross entropy models long-tailed distributions poorly. Additionally, the cross-entropy here is weighted with factor X_i which represents equal-streaming over data, so a word appearing n times contributes to the loss n times.

However, there is no inherent justification for streaming across all words equally. In fact, GloVe computes differences between unnormalized probabilities, contrary to Word2Vec (Kurita, 2018a).

5.2 Motivation for GloVe

Previous models using global counts, such as Latent Semantic Analysis (LSA) produced word embeddings that lacked the interesting vector analogical property of word vectors produced by Word2Vec. Thus, they failed to capture *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*.

5.3 Describing GloVe

5.3.1 Notation in GloVe

Let X be the matrix of word co-occurrence counts; let X_{ij} be the ij-th entry X that counts how many times any word appears in the context of word i, and let $P_{ij} = p_{\text{co}}\left(w_j \mid w_i\right) = \frac{X_{ij}}{X_i}$ be the probability that word j appears in the context of word i (Pennington et al., 2014; Weng, 2017).

5.3.2 Meaning Extraction Using Co-Occurrence Counts

GloVe uses a co-occurrence matrix that describes how words co-occur within a fixed sliding window, relying on the assumption that counts and co-occurrences can reveal word meaning. Words are said to **co-occur** when they appear together within this fixed window (Kurita, 2018a). Then, GloVe takes this matrix as input, rather than the entire corpus. Thus, sentence boundaries no longer matter since GloVe accounts for corpus-wide co-occurrence, rather than relying on sentence-level co-occurrences.

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

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

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

To illustrate how GloVe uses these counts, consider two words $w_i =$ "ice" and $w_j =$ "steam".

- **Case 1:** For words $\tilde{w}_k$ related to "ice" but not "steam" such as $\tilde{w}_k =$ "solid", we expect the co-occurrence probability $p_{\text{co}}\left(\tilde{w}_k \mid w_i\right)$ to be much larger than $p_{\text{co}}\left(\tilde{w}_k \mid w_j\right)$, making the ratio $\frac{p_{\text{co}}\left(\tilde{w}_k \mid w_i\right)}{p_{\text{co}}\left(\tilde{w}_k \mid w_j\right)}$ very large.
- **Case 2:** Conversely, for words related to "steam" but not "ice" such as $\tilde{w}_k =$ "gas", the co-occurrence ratio $\frac{p_{\text{co}}\left(\tilde{w}_k \mid w_i\right)}{p_{\text{co}}\left(\tilde{w}_k \mid w_j\right)}$ should be small.
- **Case 3:** On the other hand, if the third word is taken to be $\tilde{w}_k =$ "water" which is related to both, or $\tilde{w}_k =$ "fashion" which is unrelated to either "ice" or "steam", then the co-occurrence probability ratio $\frac{p_{\text{co}}\left(\tilde{w}_k \mid w_i\right)}{p_{\text{co}}\left(\tilde{w}_k \mid w_j\right)}$ is expected to be close to one.

GloVe's insight is that word meanings are captured by ratios of co-occurrence probabilities rather than the probabilities themselves. The model between the relation of the two words w_i and w_j regarding the third context word $\tilde{w}_k$ is:

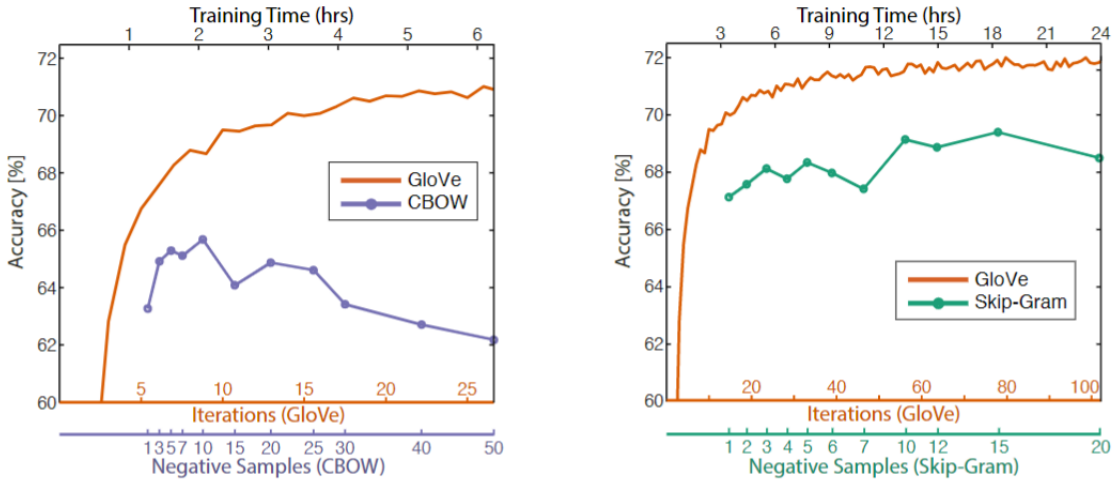
$$F\left(w_i, w_j, \tilde{w}_k\right) = \frac{p_{\text{co}}\left(\tilde{w}_k \mid w_i\right)}{p_{\text{co}}\left(\tilde{w}_k \mid w_j\right)}$$

GloVe chooses the function F to be a function taking the linear difference $w_i - w_j$ between the two words to be consistent with the goal of learning meaningful word vectors using a linear vector space. Also, they pass a dot product as input to be consistent with the linear structure of the vector space:

$$F((w_i - w_j)^T \tilde{w}_k) = \frac{p_{co}(\tilde{w}_k \mid w_i)}{p_{co}(\tilde{w}_k \mid w_j)}$$

5.3.3 Comparing Performance of Word2Vec and GloVe

fig. 6 from Pennington et al. (2014) shows that GloVe's learned word embeddings had higher prediction accuracy over both those of Skip-Gram Model and CBOW (using negative sampling) on tasks like word analogy and named entity recognition (NER).



(a) GloVe vs CBOW (b) GloVe vs Skip-Gram

Figure 6: Overall accuracy on word analogy task as a function of training time, which is governed by the number of iterations for GloVe and by the number of negative samples for CBOW (a) and Skip-Gram Model (b). Pennington et al. (2014) train 300-dimensional vectors on the same 6B token corpus from Wikipedia and use a symmetric context window of size 10. From *GloVe: Global Vectors for Word Representation*, by Pennington et al., 2014. https://nlp.stanford.edu/pubs/glove.pdf. Copyright 2014 by Pennington et al.

6 Sequence To Sequence Model

A **sequence-to-sequence (Seq-to-Seq) model** is often used in natural language processing for machine translation (MT). It takes a sequence of items such as words and outputs another sequence of items. It uses an **Encoder** that processes the inputs, squashes this information into a *fixed-length* **context vector**, also known as a **sentence embedding** or **thought vector**. The Seq-to-Seq model sends this representation of the source sentence to a **Decoder** that outputs a target sentence one word at a time, using the context vector (Alammar, 2018a). Commonly, the Encoder and Decoder are RNNs such as LSTMs or GRUs.

6.1 Describing Seq-to-Seq Model

The key feature in the Seq-to-Seq model different from a recurrent neural network (RNN) is the **context vector**, or the final hidden state of the Encoder. When the Encoder processes the input sequence $\vec{x} = \{x_1, ..., x_{T_x}\}$ of individual word vectors x_t in the input sentence X, the information is squashed into a *fixed-length* context vector. Formally, a gated recurrent unit (GRU) as the Encoder would output a hidden state given a previous hidden state and the current input:

$$h_t = \text{EncoderGRU}(x_t, h_{t-1})$$

where the context vector named z equals 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:

$$s_t = \text{DecoderGRU}\left(y_t, s_{t-1}, z\right)$$

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.

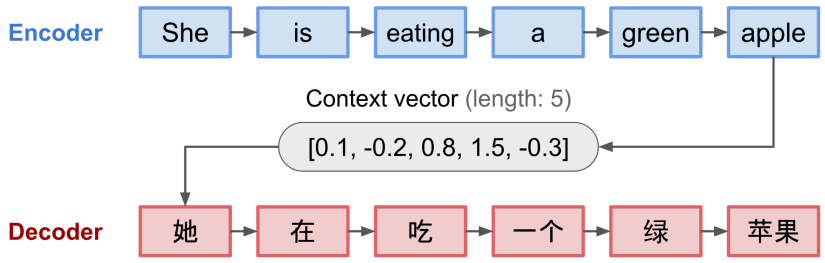


Figure 7: Encoder-Decoder with Context Vector in a Seq-to-Seq model, translating the sentence "She is eating a green apple" to Chinese. From *Attention? Attention!*, by Weng, 2018. https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html. Copyright 2018 by Weng.

6.2 Problem with Seq-to-Seq Models

Compressing the inputs into such a **fixed-length** vector leads to a **long-term dependency problem** since only the last hidden state of the Encoder is used. Thus, the Seq-to-Seq model becomes *incapable of memory*, similar to RNNs.

6.3 The Attention Mechanism

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. This means the attention mechanism uses all the hidden states generated by the Encoder as inputs for the decoding process. For each Decoder output, the attention mechanism "selectively picks out specific elements from the [input] sequence to produce the [Decoder] output" (Loye, 2019). This essentially creates links between the context vector and entire source input. A general illustration is in fig. 8.

6.4 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.

6.4.1 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** align is used to calculate **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 are weights

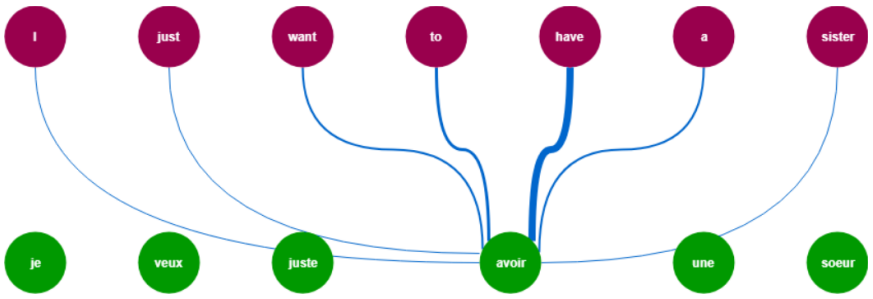


Figure 8: Attention Mechanism: How words are considered for contextual evidence. From *Intuitive Understanding of Seq2seq model and Attention Mechanism in Deep Learning*, by Medium, 2019. https://medium.com/analytics-vidhya/intuitive-understanding-of-seq2seq-model-attention-mechanism-in-deep-learning-1c1c24aace1e. Copyright n.d by n.d.

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).

$$e_{ti} = \text{align}\left(s_{t-1}, h_i\right)$$

2. Next, the energy score e_{ti} is passed through a feed-forward neural network and softmax to calculate the attention scores, α_{ti}:

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

Transforming the energy scores via softmax ensures the attention vector $\vec{\alpha} = \left\{\alpha_{ti}\right\}$ has values normalized between 0 and 1.

6.4.2 Forward Pass of Decoder

Once the attention scores have been calculated, the Decoder can calculate the context vector, c_t, which is a sum of Encoder hidden states h_i weighted by attention scores $\vec{\alpha} = \left\{\alpha_{ti}\right\}$ (Ta-Chun, 2018):

$$c_t = \sum_{i=1}^{T_x} \alpha_{ti} \cdot h_i$$

Intuitively, the context vector is an **expectation**. The attention score α_{ti} is the probability that target word y_t is aligned to (or translated from) an input word x_j. Then the t-th context vector c_t is the expected hidden state over all hidden states h_i with probabilities α_{ti}, where these corresponding energies e_{ti}) quantify the importance of Encoder hidden state h_i with respect to the previous Decoder hidden state s_{t-1} in deciding the next Decoder state s_t for generating target word y_t.

Through this attention mechanism in the Decoder, we can relieve the Encoder of the burden of encoding all source sentence information into a fixed-length vector, allowing information to spread through the hidden state sequence $\vec{h} = \left\{h_1, ..., h_T\right\}$ and later be retrieved by the Decoder (Trevett, 2020).

6.4.3 Forward Pass of Seq-to-Seq Model

Finally, the Seq-to-Seq model can use the Encoder, Decoder and attention in conjunction. Its forward pass is (Trevett, 2020):

1. Create an output tensor to hold predicted words $\hat{y}$
2. Pass the source sequence $\vec{x} = \{x_1, ..., x_T\}$ into the Encoder to receive the contexts $\vec{z}$ alongside the hidden states $\vec{h} = \{h_1, ..., h_T\}$.
3. Set equal the initial Decoder hidden state s_0 and last Encoder hidden state h_T.
4. Decode within a loop: insert the target token y_t and previous hidden state s_t and all Encoder outputs $\vec{h}$ into the Decoder to get a prediction $\hat{y}_{t+1}$ and new hidden state s_t.

As an application of the Seq-to-Seq model with attention to neural machine translation (NMT), PyTorch code with explanations can be found in APPENDIX: Application To Machine Translation in NLP (Using PyTorch and Seq-To-Seq Model With Attention)

7 Transformer

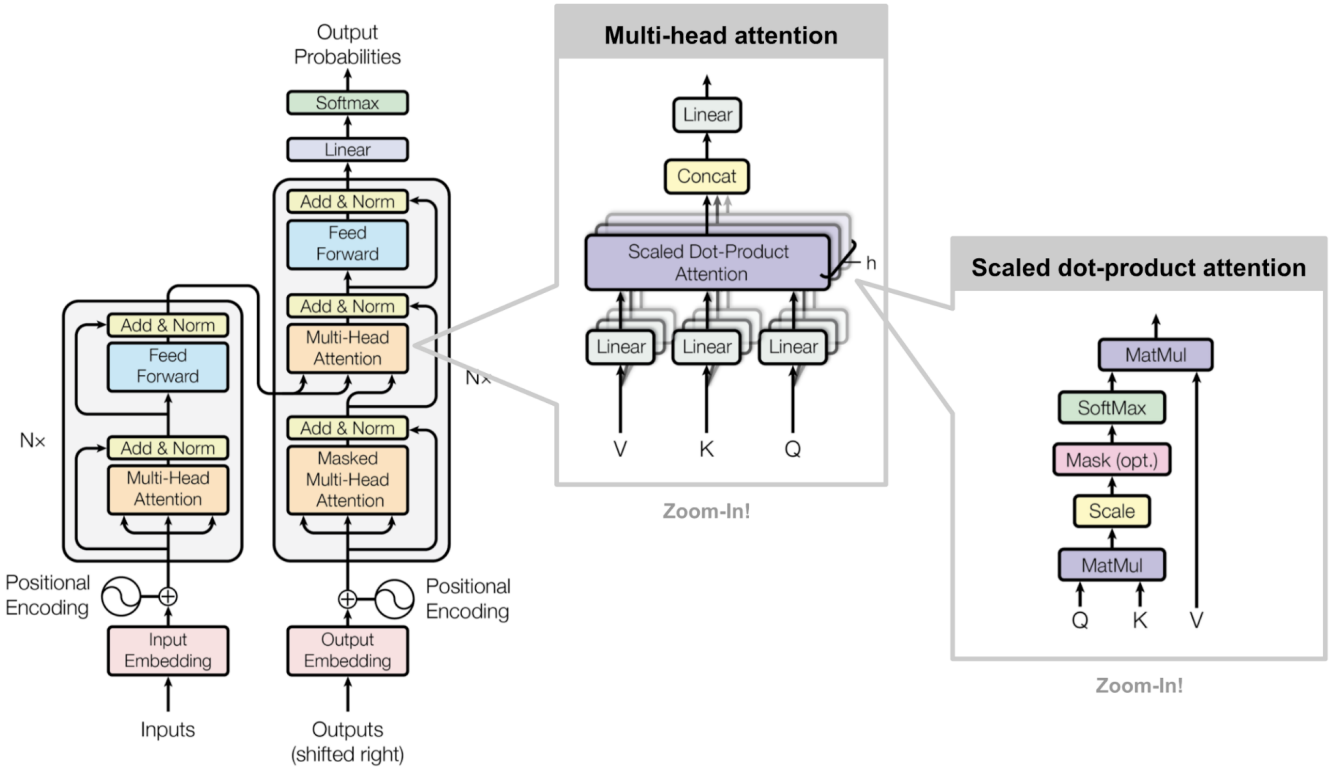


Figure 9: Transformer model architecture. The gray boxes hold Encoder and Decoder layers, respectively, which are each repeated $N = 6$ times. From *Attention? Attention*, by Weng, 2018. https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html. Copyright 2018 by Weng.

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 using recurrent neural networks (RNNs) combined with the **attention mechanism**, the Transformer sequence-to-sequence model uses only a **self attention mechanism** to attend to different word tokens in an input sentence and thus generate a sequence of contextual embeddings. It is illustrated in fig. 9.

7.1 Self-Attention

7.1.1 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 **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 to create 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" when encoding this in the model (Trevett, 2020).

7.1.2 Query, Key, Value

Formally, "an ***attention function*** can be described as mapping a query and a set of key-value pairs to an output, where the **query, keys, values**, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key" (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.
- 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 the previous sentence, "The animal didn't cross the road because it was too tired," Q query refers to the word "it"; V contains vectors for words other than "it", and K contains vectors for each word, including "it".

The final embedding of the word or **output** is a weighted sum of **value** vectors and softmax probabilities of the dot product between query and key vectors:

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

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 sentence, let the input be the matrix $X = \{\overrightarrow{x_1}, \overrightarrow{x_2}, ..., \overrightarrow{x_n}\}$, where vector $\overrightarrow{x_i}$ corresponds to word $\overrightarrow{w_i}$, and there are n words. Then the input word vectors are:

$$\begin{aligned}
\overrightarrow{x_1} &= \text{"The"} \\
\overrightarrow{x_2} &= \text{"animal"} \\
\overrightarrow{x_3} &= \text{"didn't"} \\
\overrightarrow{x_4} &= \text{"cross"} \\
\overrightarrow{x_5} &= \text{"the"} \\
\overrightarrow{x_6} &= \text{"road"} \\
\overrightarrow{x_7} &= \text{because"} \\
\overrightarrow{x_8} &= \text{"it"} \\
\overrightarrow{x_9} &= \text{"was"} \\
\overrightarrow{x_{10}} &= \text{"too"} \\
\overrightarrow{x_{11}} &= \text{"tired"} \\
\overrightarrow{x_{12}} &= \text{"."}
\end{aligned}$$

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.

7.1.3 Self-Attention: Vector Calculation

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

1. **Create Query, Key, Value Vectors**: The first step is to create query, key, value vectors from each of the Encoder's input word embeddings $\overrightarrow{w_i}$ corresponding each word $\overrightarrow{x_i}$ by multiplying the embedding by appropriate rows in the three matrices obtained during training.

2. **Calculate a Score**: The scores 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** vector with the **key** vector of the respective word being scored. Thus for word $\overrightarrow{w_i}$, the scores are:

$$\text{scores}_{w_i} = \left\{\overrightarrow{q_i} \cdot \overrightarrow{k_1}, \overrightarrow{q_i} \cdot \overrightarrow{k_2}, ..., \overrightarrow{q_i} \cdot \overrightarrow{k_n}\right\}$$

3. **Scale The Score**: The scores are scaled using d_k, which is 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:

$$\text{scores}_{\overrightarrow{w_i}} = \left\{\frac{\overrightarrow{q_i} \cdot \overrightarrow{k_1}}{\sqrt{d_k}}, \frac{\overrightarrow{q_i} \cdot \overrightarrow{k_2}}{\sqrt{d_k}}, ..., \frac{\overrightarrow{q_i} \cdot \overrightarrow{k_n}}{\sqrt{d_k}}\right\}$$

4. **Apply Softmax**: The softmax function normalizes the scores into probabilities.

$$\text{scores}_{\overrightarrow{w_i}} = softmax\left(\left\{\frac{\overrightarrow{q_i} \cdot \overrightarrow{k_1}}{\sqrt{d_k}}, \frac{\overrightarrow{q_i} \cdot \overrightarrow{k_2}}{\sqrt{d_k}}, ..., \frac{\overrightarrow{q_i} \cdot \overrightarrow{k_n}}{\sqrt{d_k}}\right\}\right)$$

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.

$$\text{weights}_{\overrightarrow{w_i}} = \text{scores}_{\overrightarrow{w_i}} * (\overrightarrow{v_1}, ..., \overrightarrow{v_n})$$

6. **Compute Output Vector**: The weight vector's cells are summed to produce the **output vector** of the self-attention layer for word embedding $\overrightarrow{w_i}$:

$$\overrightarrow{output_{w_i}} = softmax\left(\frac{\overrightarrow{q_i} \cdot \overrightarrow{k_1}}{\sqrt{d_k}}\right) \cdot \overrightarrow{v_1} + softmax\left(\frac{\overrightarrow{q_i} \cdot \overrightarrow{k_1}}{\sqrt{d_k}}\right) \cdot \overrightarrow{v_2} + \ldots + softmax\left(\frac{\overrightarrow{q_i} \cdot \overrightarrow{k_1}}{\sqrt{d_k}}\right) \cdot \overrightarrow{v_n}$$

7.2 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:

$$Q = X \cdot W^Q$$
$$K = X \cdot W^K$$
$$V = X \cdot W^V$$

2. **Calculate Self Attention**: Steps 2 through 6 of the vector calculation for self attention can be condensed into a single matrix step where $Q = \left\{\overrightarrow{q_1}, \overrightarrow{q_2}, \ldots, \overrightarrow{q_n}\right\}, K = \left\{\overrightarrow{k_1}, \overrightarrow{k_2}, \ldots, \overrightarrow{k_n}\right\}, V = \left\{\overrightarrow{v_1}, \overrightarrow{v_2}, \ldots, \overrightarrow{v_n}\right\}$:

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

7.3 Multi-Head Attention

7.3.1 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." A single attention head cannot do this because of averaging (Vaswani et al., 2017).

While a single attention function has d_{model}-dimensional keys, values and queries, a multi-head attention function "linearly projects the queries, keys and values H (number of attention heads) times with different, learned linear projections to d_k, d_k, and d_v dimensions, respectively." 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).

For the example sentence, adding more attention heads enables the Transformer to focus on different words while encoding the meaning of word "it." As "it" is encoded, one attention head may focus most on "the animal" while another focuses on "tired", so the model's representation of "it" incorporates some representation of all the words in the sentence (Trevett, 2020).

7.3.2 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 $1 \leq h \leq H$. The rows of input matrix X correspond to a word in the input

sentence, as before. For each attention head, X is multiplied by trained parameter matrices to produce the separate query, key, value matrices:

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

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

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:

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

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. Thus, this step concatenates the H matrices and multiplies the result by an additional weights matrix W^O to return to expected dimensions.

$$MultiHeadAttention(Q, K, V) = Concat(head_1, ..., head_H) \cdot W^O$$

where $head_i = Attention(Q \cdot W_h^Q, K \cdot W_h^K, V \cdot W_h^V)$, where the attention function is simply $Attention(Q, K, V$ $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) \cdot V$. The parameter matrices have the following dimensions: $W^O \in \mathbb{R}^{H \cdot d_v \times d_{model}}$, $W_h^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_h^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_h^V \in \mathbb{R}^{d_{model} \times d_v}$

7.4 Positional Encodings

7.4.1 Motivation for Positional Encodings

Since the Transformer contains no recurrence mechanism it does not yet account for *order in the sequence sentence*. Vaswani et al. (2017) found that injecting information about relative or absolute position of tokens in the sequence in the form of **positional encodings** helps resolve this issue. Otherwise, the sentences "I like dogs more than cats" and "I like cats more than dogs" would encode the same meaning (Raviraja, 2019).

7.4.2 Describing Positional Encodings

A **positional encoding** follows a specific, learned pattern to identify word position or the distance between words in the sequence (Alammar, 2018b). The Transformer adds the positional encoding vector to each input embedding in both Encoder and Decoder stacks.

According to Vaswani et al. (2017), positional encodings use sinusoidal waves to allow the Transformer to more easily attend to relative positions since for any fixed offset k, the positional encoding $PosEnc_{pos+k}$ can be represented as a linear function of $PosEnc_{pos}$.

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

where pos = a position, i = a dimension.

7.5 Position-wise Feed Forward Layer

A **positionwise feed-forward layer** is a kind of feed-forward neural network (FNN) that is also "position-wise" because the FFN is applied to each position separately and identically. The FFN contains two linear transformations with a *ReLU* or *max* activation function in between them:

$$FFN(x) = ReLU(0, xW_1 + b_1)W_2 + b_2$$

7.6 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 flow through a network directly rather than being transformed by nonlinear activations (Raviraja, 2019):

$$LayerNorm(x + Sublayer(x))$$

where $Sublayer(x)$ 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.

7.7 Masked Multi-Head Attention

The **masked self attention** is an attention mechanism that forms a sub-layer in the Decoder stack. It uses **masking** to prevent positions from attending to subsequent positions. Rather, while decoding a word embedding $\overrightarrow{w_i}$, the Decoder is not aware of words $\overrightarrow{w_{>i}}$ past position i. It can only use words $\overrightarrow{w_{\leq i}}$. This masking is done to render invisible the words $\overrightarrow{w_{>i}}$ so that no superfluous predictions can be made (Ta-Chun, 2018).

7.8 Encoder-Decoder Attention

This attention layer differs from attention layers found in either Encoder or Decoder. It is like multi-head self attention but the difference is that the **encoder-decoder attention layer** creates the query matrix Q from the layer below it (a Decoder self attention layer) and uses the key K and value V matrices from the Encoder stack's output (Alammar, 2018b).

7.9 Encoder

The Encoder is a **bidirectional recurrent network (RNN)** consisting of a forward RNN and backward RNN. The forward RNN reads the input sequence $\vec{x} = \{x_1, ..., x_{T_x}\}$ from left to right to produce a sequence of forward hidden states $\{\overrightarrow{h_1}, ..., \overrightarrow{h_{T_x}}\}$. The backward RNN reads the sequence in reverse order, so taking x_{T_x} to x_1 and returns a sequence of backward hidden states $\{\overleftarrow{h_1}, ..., \overleftarrow{h_{T_x}}\}$. Then, for each word x_t an annotation is obtained by concatenating the corresponding forward hidden state vector $\vec{h}_t$ with the backward one $\overleftarrow{h}_t$, such that $h_t = \{\vec{h}_t^T ; \overleftarrow{h}_t^T\}^T, t = 1, ..., T_x$. (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 contextual information by using previous and latter words (Bahdanau et al., 2016). These annotations are later used in the decoder to compute the context vector.

The Encoder is composed of N identical **Encoder layers**, which together are named the **Encoder stack**. A single **Encoder layer** is composed of two sub-layers:

1. **Multi-Head Attention (layer)**
2. **Position-wise Feed Forward Layer (layer)**

A residual connection layer surrounds each of these sub-layers, and is followed by **layer normalization** (Trevett, 2020).

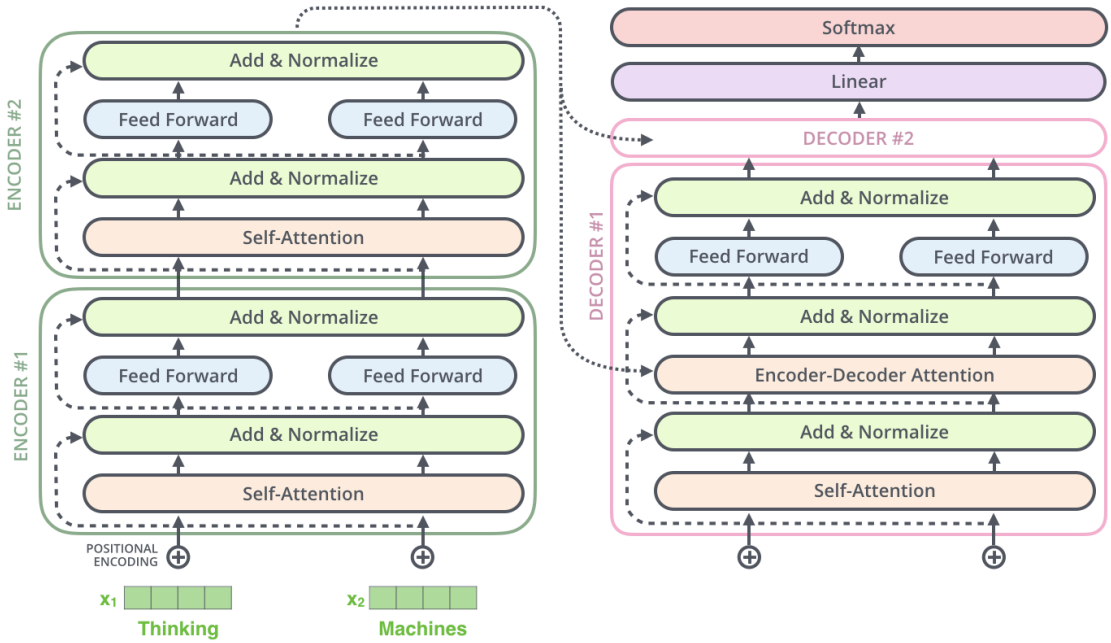


Figure 10: The layers inside Encoder and Decoder. From *The Illustrated Transformer*, by Alammar, 2018. https://jalammar.github.io/illustrated-transformer/. Copyright 2018 by Alammar.

7.10 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 $c_t = \sum_{i=1}^{n} a_{ti} \cdot h_i$ 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 consist of several sub-layers:

1. **Masked Multi-Head Attention (layer)**
2. **Encoder-Decoder Attention (layer)**
3. **Position-wise Feed Forward Layer (layer)** .

Like in the Encoder, in between each of these layers is a residual connection followed by layer normalization. The Encoder and Decoder stack are shown in fig. 10.

7.11 Final Linear and Softmax Layer

The Decoder stack outputs a vector of floats which is converted to a word using a Linear layer, followed by a Softmax layer in the Transformer neural network (Alammar, 2018b).

- **Linear Layer** is a simple, fully-connected neural network that projects the Decoder's output vector in a larger-dimension "logits vector" in which each cell holds a score corresponding to each unique vocabulary word.
- **Softmax Layer** then converts the Linear Layer's scores into probabilities via the 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.

7.12 Transformer Workflow

Alammar (2018b) describes the procedure governing the Transformer's moving parts as follows:

1. The Encoder processes the input sentence in the given language, adding the positional encoding to input embeddings.
2. The output of the top Encoder layer is then transformed into a set of attention vectors K and V.
3. The Decoder uses K and V in its encoder-decoder attention layer to help the Decoder focus on appropriate places in the input sequence. Subsequent outputs are fed to the bottom Decoder, allowing Decoders to accumulate results. Also, the Decoder includes positional encodings to its inputs.
4. The previous steps are repeated until a special symbol is reached, indicated the Decoder has finished generating output.
5. The Decoder's numeric output vector is passed through the final linear and softmax layer to find a predicted, translated word in the target language.

8 ELMo

8.1 Combing Back To Polysemy

As explained in The Problem With Context-Free, Static Embeddings, the term **polysemy** is the correspondence of one word to distinct meanings, and traditional embeddings fail to capture these senses, thus performing poorly on language modeling NLP tasks. However, contextual embeddings (CWE) prove superior since they create one vector representation for each word type in the vocabulary and also create distinct word vectors per token given a context (Wiedemann et al., 2019).

For example, if instead models used only word and character embedding, the homonyms "book" (text) and "book" (reservation) would be assigned the *same vector representation even though these are different words*. This vector representation may of course be created using context, as per the Word2Vec or GloVe, but these meanings are still *collapsed into a single representation*.

8.2 Motivation for ELMo

In contrast to traditional word embeddings, **deep contextualized word embeddings** from Peters et al. (2018) capture word semantics to account for the context-dependent and polysemous nature of words.

These word vectors are "learned functions of the internal states of a bidirectional language model (biLM)" that is pretrained on a large corpus. Due to this, the resulting word embeddings are coined **ELMo (Embeddings from Language Models)**.

ELMo representations are *deep* since they are derived from all internal layers of the biLM. ELMo learns a "linear combination of vectors stacked above each input word for each end task," improving performance of models that use only the top LSTM layer.

Higher-level LSTM layers capture contextual meaning, useful for supervised word sense disambiguation (WSD), and lower layers capture syntax information, useful for part of speech tagging (POS). Coupled with the fact that mixing these signals allows the learned embeddings to select the types of semi-supervision most needed for each end task, ELMo embeddings are richer than traditional word vectors (Peters et al., 2018).

8.3 Describing ELMo

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

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

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 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 (Peters et al., 2018):

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

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.

The biLM in ELMo is task-agnostic, and ELMo combines the task-independent representations. Although the hidden states are fixed, this ELMo formulation is flexible enough to be combined in downstream models for various tasks (Kurita, 2018b). ELMo can be applied to specific tasks by concatenating the ELMo word embeddings with context-free word embeddings, such as those from GloVe, Peters et al. (2018) say the concatenated input vector $\left\{ \mathbf{x}_k \; ; \; \mathbf{ELMo}_k^{task} \right\}$ can be fed into a task-specific RNN for processing.

8.4 ELMo Experimental Results

- **question answering (QA):** Peters et al. (2018) added ELMo embeddings to a baseline biLM model with attention from Seo et al. (2017) and found this improved the test set F_1 score of the biLM model from 81.1% to 85.8%, a state-of-the-art result.
- **semantic role labeling (SRL):** When adding ELMo representations to an 8-layer deep biLSTM that interwove forward and backward directions, the single test set F_1 score increased by 3.2%, which is a 1.2% improvement over the previous best model.
- **coreference resolution (CR):** Using Lee et al. (2017)'s model that uses attention to compute span vectors transformed by softmax ranking to find coreference chains, adding ELMo embeddings improved the average F_1 score by 3.2%, improving over the previous best model by nearly two percentage points.
- **named entity recognition (NER):** The baseline model had pretrained, character embeddings from a convolutional neural network (CNN), as well as two biLSTM layers and a conditional random field (CRF) loss function formulation (Lafferty et al., 2001). Adding ELMo embeddings 92.22% in F_1 score by letting the task model learn an average of all biLM layers, rather than only using the top layer.
- **sentiment analysis (SA):** A classification model using contextual embeddings from CoVe (McCann et al., 2018) was used as the baseline model, with CoVe embeddings replaced by ELMo embeddings. This resulted in a full percentage increase in absolute accuracy over state of the art.

8.4.1 ELMo's Key Feature

Because ELMo improves task performance over word embeddings, this must mean its biLM's contextual representation outputs are encoding task-specific information that traditional word vectors do not otherwise capture. In other words, the biLM must be sifting meaning of words using their context.

	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.

The word "play" is highly polysemous since it has many different meanings; table 1 displays words nearest to "play" found using GloVe word embeddings and a biLM model. The 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 ability to learn context using part of speech tagging (POS) and word sense disambiguation (WSD).

9 BERT

9.1 Problem with ELMo

Simple word embedding models, unlike LSTMs, 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.*** This deficiency does not let the model perform as well as it should since information from the entire sentence is not being used ***simultaneously***, regardless of position.

9.2 Motivation for BERT

Instead of using ELMo's "shallow" combination of "independently-trained" biLMs, "BERT is designed to pretrain deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers" (Devlin et al., 2019).

Another motivation for BERT stems from previous limitations. The **OpenAI GPT** model uses a left-to-right construction, where each token only attends to past tokens in the self attention layers of the Transformer. This approach performs poorly on *sentence-level* and *token-level* tasks, like question answering (QA), where **bidirectional context** is required.

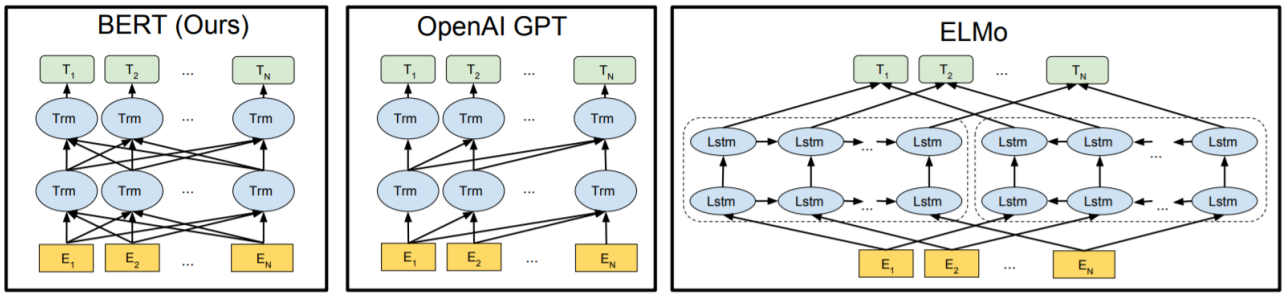


Figure 11: Comparing pre-training models: BERT uses a bidirectional Transformer. OpenAI GPT uses a forward Transformer. ELMo combines independently-trained forward and backward LSTMs. Among the three, only BERT embeddings are jointly conditioned on forward and backward context in all layers. Alongside architectural differences, BERT and OpenAI GPT are fine-tuning approaches, while ELMo is a feature-based approach (Devlin et al., 2019). *NOTE:* $\boldsymbol{E}_n$ *= the* $\boldsymbol{n}$*-th token in the input sequence, and* $\boldsymbol{T}_n$ *= 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.

BERT (Bidirectional Encoder Representations from Transformers) has progressed over past models since, in conjunction with its biLM, self attention, and Transformer (more powerful than LSTMs), BERT uses a masked language model (MLM) to avoid unidirectionality, resulting in vast performance gains over previous models (Wiedemann et al., 2019).

9.3 Describing BERT

9.3.1 Input Embedding in BERT

The input embedding in BERT is created by summing three kinds of other 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. Consider: 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, rather than discrete sentences. The term "sequence" for BERT refers to the input tokens, which can be parts of sentences or several sentences packed together. Contrary to BERT, Transformer-XL's **segment embeddings** respect sentence boundaries.

3. **Positional embeddings:** to account for word ordering.

9.3.2 BERT's Framework

There are two steps in BERT's framework: **pre-training**, in which BERT is trained 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 BERT using the Masked Language Model (MLM) task and Next Sentence Prediction (NSP) task allows BERT to learn **bidirectional context** and *sentence-level* information, rather than just predict subsequent tokens given some context, as was the norm for simple language models.

9.3.3 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 of the input tokens with the aim of predicting the original vocabulary id of the masked word using only its 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).

An issue with masking is that the model only predicts the `[MASK]` token when it is present in the input, while the intention was for the model to predict the correct tokens regardless of which tokens were present in input (Kurita, 2019a). This hampers performance since BERT would learn a contextual meaning of the mask token, causing it to learn slower since only 15% of tokens are masked. To resolve this, 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. For example, if the sentence was "The cow jumped over the moon," and if the token to be masked was "moon", then 80% of the time, the replacement would be "The cow jumped over the `[MASK]`"; 10% of the time a random token would be used "The cow jumped over the **seaweed**"; and the remaining times the sentence would remain the same.

9.3.4 Next Sentence Prediction (NSP)

Ordinary language models perform badly for tasks like question answering (QA) and natural language inference (NLI) which require modeling sentence relationships, therefore BERT is pre-trained on a **next sentence prediction (NSP)** task for finding whether one sentence is the next sentence of the other.

Training BERT using NSP is as follows: (1) sample sentence pairs (`A`, `B`) so that half the time `B` follows `A` (labeled as `IsNext`) and half the other time, `B` does not follow `A` (labeled as `NotNext`), then (2) BERT processes both sentences and uses a binary classifier to decide of `B` is the next sentence after `A` (Weng, 2019).

9.4 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.

9.5 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.

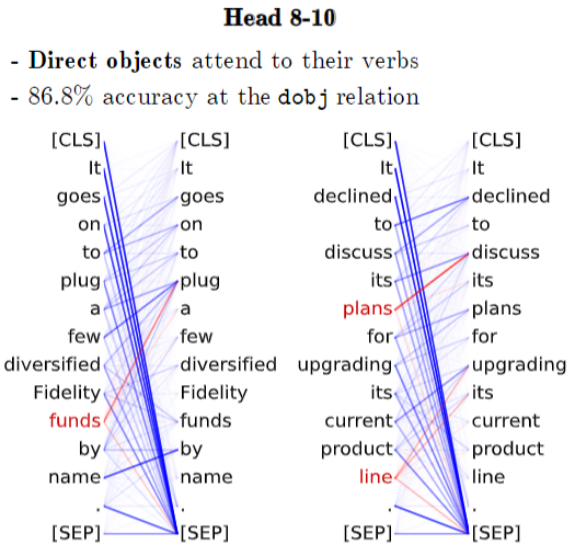


Figure 12: BERT attention heads capture syntax. In heads 8-10, direct objects are found to attend to their verbs. Line darkness indicates attention strength. Red indicates attention to/from red words, to highlight certain attentional behaviors. From *What Does BERT Look At? An Analysis of BERT's Attention*, by Clark et al., 2019. https://arxiv.org/abs/1906.04341. Copyright 2019 by Clark et al.

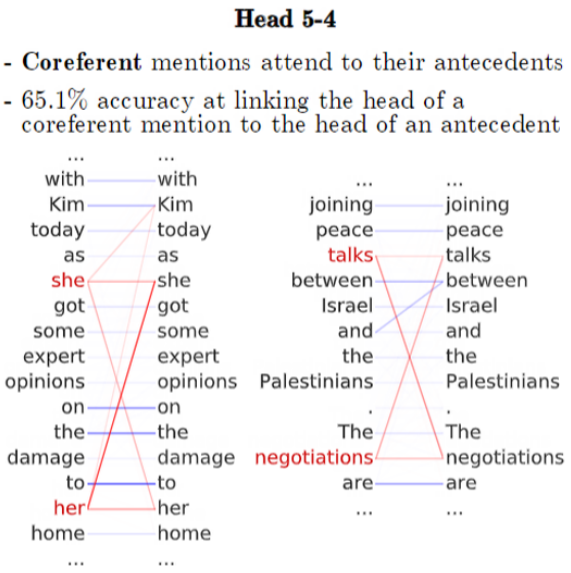


Figure 13: BERT attention heads capture syntax. In heads 5-4, BERT does some coreference resolution (CR) with high accuracy at linking the head of a coreferent mention to the head of an antecedent. From *What Does BERT Look At? An Analysis of BERT's Attention*, by Clark et al., 2019. https://arxiv.org/abs/1906.04341. Copyright 2019 by Clark et al.

9.5.1 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, it was found that certain attention 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. 12 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. 13 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.

9.5.2 BERT's Limitation In Segment-Level Representation

Instead of using separator tokens to gather segment-level information, BERT actually views separator tokens [SEP] as "no-op" or stub operations for attention heads when the head is not needed for a current task.

Authors hypothesized as to why so many of BERT's attention heads focus on separator tokens, a feature visible in fig. 14. But there are several investigations that prove this may not be the case:

1. If BERT were indeed trying to gather segment-level information, then attention heads processing [SEP] should attend broadly over the entire segment to create the segment representations. But fig. 12 shows that in heads 8-10 direct objects attend to their verbs, while all other words attend to the [SEP] token.

2. Gradient measures were used to study how much the attention to a token would change BERT's outputs.

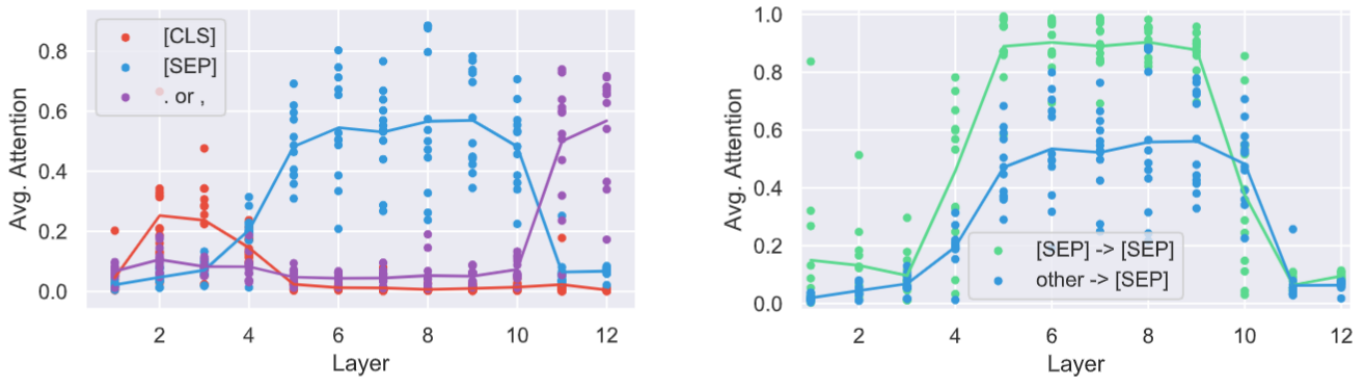


Figure 14: BERT's attention heads in layers 6-10 spend more than half the average attention to separator tokens; deep heads attend to punctuation, while middle heads attend to [SEP], and early heads attend to [CLS]. From *What Does BERT Look At? An Analysis of BERT's Attention*, by Clark et al., 2019. https://arxiv.org/abs/1906.04341. Copyright 2019 by Clark et al.

While attention to [SEP] increases from layer 5 onwards (visible in fig. 14), the gradients for attention to [SEP] decrease substantially (visible in fig. 15). This means attending to [SEP] does not significantly change BERT's outputs.

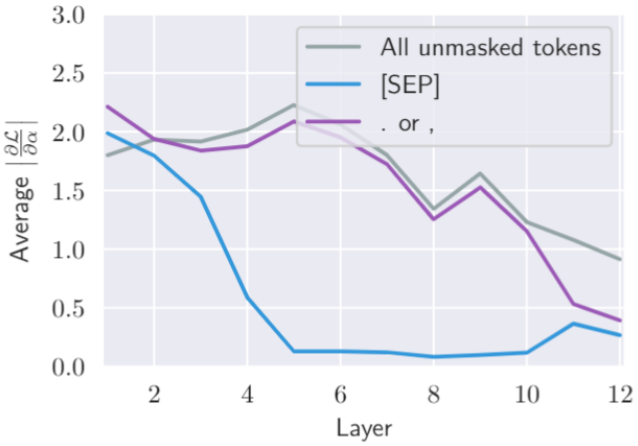


Figure 15: Gradient-based estimates for attention to separator and punctuation tokens. Authors "compute the magnitude of the gradient of the loss from the Masked Language Model (MLM) task with respect to each attention weight." From *What Does BERT Look At? An Analysis of BERT's Attention*, by Clark et al., 2019. https://arxiv.org/abs/1906.04341. Copyright 2019 by Clark et al.

The above two investigations suggest BERT's attention heads only attend to [SEP] when they have no other job, not to gather segment-level information. However, Transformer-XL by design improves over BERT in this regard.

9.5.3 BERT's Contribution To Polysemy

Using a kNN classification, Wiedemann et al. (2019) compared the contextual embeddings (CWE)s of ELMo and BERT on the word sense disambiguation (WSD) task and found that BERT places polysemic words into distinct regions according to their senses, while ELMo cannot. Specifically, in ELMo embeddings, major word senses are less strongly separated than in the BERT embedding space, where some senses are sharply clustered.

In table 3 Wiedemann et al. (2019) also made inferences about the semantic features of BERT's embeddings by studying how BERT matched word senses from the test set to a given polysemic word from a training set.

BERT did well when the data had *vocabulary overlap* in context, such as 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*, such as in example (3)'s input "little earthy bank" and nearest neighbor "huge bank [of snow]".

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.

	Example sentence	Nearest neighbor
	SE-3 (train)	SE-3 (test)
(1)	President Aquino, admitting that the death of Ferdinand Marcos had sparked a wave of sympathy for the late dictator, urged Filipinos to stop weeping for the man who had laughed all the way to the **bank**[A Bank Building].	They must have been filled in at the **bank**[A Bank Building] either by Mr Hatton himself or else by the cashier who was attending to him.
(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.
(10)	Americans it seems have followed Malcolm Forbes's hot-air lead and taken to **balloon**[To Ride in a Hot-Air Balloon] in a heady way.	Just like the **balloon**[Large Tough Nonrigid Bag] would go up and you could sit all day and wish it would spring a leak or blow to hell up and burn and nothing like that would happen.
(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.

However, *vocabulary and semantic overlap in conjunction* led BERT to make false predictions. Example (5) in table 3 shows BERT predicted "land bank" as in a *supply or stock* while the correct sense of "land bank" was *financial institution*. In example (10), the correct word sense of "balloon" is a verb while BERT predicted a noun sense, so it did not even get the word class correct. Even harder for BERT was to distinguish verb senses correctly; example (12) shows the correct sense label of the polysemic word "watch" was *to look attentively* while BERT's predicted sense was *to follow with the eyes or the mind; observe*.

Despite BERT's limitations, the authors conclude still that BERT captures polysemy more successfully than ELMo, inspiring future work in using word sense disambiguation (WSD) to compare BERT to models like XLNet.

10 Transformer-XL

10.1 Problem With Transformer

10.1.1 Fixed-Length Segments

Transformers are capable of learning longer term dependencies but instead perform poorer than they should because their inputs are dominated by **fixed-length context**. Limited context-dependency means that the largest

dependency distance between characters is limited to the input length, which does not allow the Transformer to remember a word that appeared several sentences ago (Dai et al., 2019).

A *vanilla* (ordinary) Transformer model adapted by Al-Rfou et al. (2018) uses fixed-length context since the model is trained within its segment embeddings. This effectively blocks information from flowing across the segment embeddings during both the forward and backward pass. This result, visible in fig. 16, causes the Transformer to forget context from previous segments.

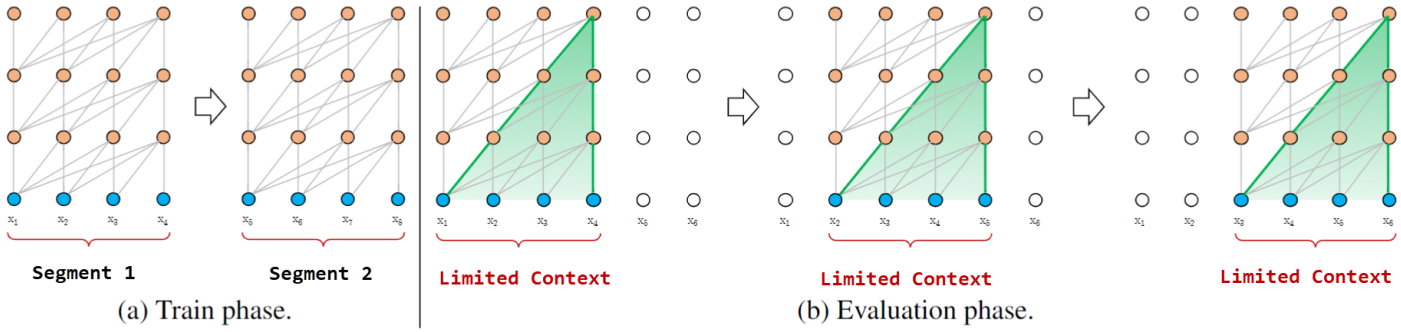


Figure 16: Vanilla Transformer with segment embedding length = 4. Training the model in fixed-length segments while disregarding natural sentence boundaries results in the *context fragmentation problem*: during each evaluation step, the Transformer consumes a segment embedding and makes a prediction at the last position. Then at the next step, the segment is shifted right by one position only, and the new segment must be processed from scratch, so there is no context dependency for 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.

10.1.2 Context Fragmentation Problem

The Transformer's use of **fixed-length context** and lack of regard for the natural semantic boundaries of sentences causes it to lose contextual information and perform more poorly than it could. This is called the **context fragmentation problem**.

10.2 Motivation for Transformer-XL

According to Dai et al. (2019), the **Transformer-XL** (extra long) uses a new architecture that enables learning longer dependencies without "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 sentences, paragraphs, and even longer texts like documents. It is composed of a segment-level recurrence mechanism and relative positional encoding method to resolve **context fragmentation** and enable representation of longer-spanning dependencies.

10.3 Describing Transformer-XL

10.3.1 Segment-Level Recurrence Mechanism

To resolve **context fragmentation**, the Transformer-XL employs a **segment-level recurrence mechanism** to capture long-term dependencies using information from previous segments. While intaking the first segment of tokens like the vanilla Transformer, this new recurrence mechanism also stores hidden layer outputs. This is so that when a segment is being processed, each hidden layer receives two inputs: (1) the previous hidden layer outputs of the *current segment* (as the vanilla model, shown as gray arrows in fig. 17), and (2) the previous hidden layer outputs of the *previous segment* (green arrows in fig. 17). These features build long-term memory (Horev, 2019).

Although gradient updates or training still occurs within a segment, the extended context feature allows historical information to be fully used, avoiding context fragmentation.

(a) Training phase. (b) Evaluation phase.

Figure 17: 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 other part "I bought some cookies" with the cached states into the model (Kurita, 2019b).

Formally, the recurrence mechanism is described as follows. Let $\mathbf{s}_\tau = \{x_{\tau_1}, x_{\tau_2}, ..., x_{\tau_L}\}$ and $\mathbf{s}_{\tau+1} = \{x_{\tau+1_1}, x_{\tau+1_2},$ 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 the n-th layer's hidden state for next segment $\mathbf{s}_{\tau+1}$ is calculated as:

$$\tilde{\mathbf{h}}_{\tau+1}^{n-1} = \left\{ StopGradient\left(\mathbf{h}_\tau^{n-1} \circ \mathbf{h}_{\tau+1}^{n-1}\right) \right\}$$

$$\mathbf{q}_{\tau+1}^n, \quad \mathbf{k}_{\tau+1}^n, \quad \mathbf{v}_{\tau+1}^n = \mathbf{h}_{\tau+1}^{n-1} \cdot \mathbf{W}_q^T, \quad \tilde{\mathbf{h}}_{\tau+1}^{n-1} \cdot \mathbf{W}_k^T, \quad \tilde{\mathbf{h}}_{\tau+1}^{n-1} \cdot \mathbf{W}_v^T$$

$$\mathbf{h}_{\tau+1}^n = TransformerLayer\left(\mathbf{q}_{\tau+1}^n, \quad \mathbf{k}_{\tau+1}^n \quad \mathbf{v}_{\tau+1}^n\right)$$

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. 17.

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.

10.3.2 Relative Positional Encoding

Introducing the segment-level recurrence mechanism presented a problem: how can positional word order be kept coherent when reusing hidden states? The vanilla Transformer kept word order straight using positional encodings. But since the standard Transformer's positional encodings are based on *absolute distance* between tokens, applying these directly to the Transformer-XL caused consecutive word embedding sequences to be associated with the same positional encoding, so the model could not distinguish the difference in positions of consecutive input tokens. This means that tokens from different segments had the same positional encodings, even though their position and importance could differ. This confused the model.

To remedy this and make their recurrence scheme work, the authors created a **relative positional encoding** scheme that works in conjunction with each attention score of each layer, rather than encoding position only before the first layer, and which is based on *relative* distance between tokens, not *absolute* position (Horev, 2019). Formally, the authors created a relative positional encodings matrix whose i-th row indicates a relative distance

of i between two positions, and injected this dynamically into the attention module. This lets the query vector now distinguish between two tokens from 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 get distance between tokens instead of the *absolute position* of a single 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.

10.4 Experimental Results of Transformer-XL

10.4.1 Ablation Study for Segment-Level Recurrence and Relative Positional Encodings

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.

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** 23.16 23.35	**640** 450 300

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 table 4 shows that both segment-level recurrence and the 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 does not use the new recurrence and 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.

11 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 (AR) language modeling** and **autoencoding (AE) language modeling** have been the most powerful pretraining objectives.

11.1 autoregressive language model (AR)

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

11.2 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 must guess the correct tokens. Since AE modeling does not estimate densities, it can learn bidirectional contexts.

11.3 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 using a *forward* autoregressive decomposition:

$$max_\theta \left(log\, P_\theta(\mathbf{x}) \right) = \sum_{t=1}^{T} log\, P_\theta\left(x_t \mid \mathbf{x}_{<t}\right)$$

Meanwhile, autoencoding language model (AE)s like BERT takes an input sequence $\mathbf{x}$, and corrupts it $\hat{\mathbf{x}}$ by masking some tokens. Let $\bar{\mathbf{x}}$ denote only masked tokens. Then the autoencoding's objective is to recreate the masked tokens $\bar{\mathbf{x}}$ from the corrupted input $\hat{\mathbf{x}}$:

$$max_\theta \left(log\, P_\theta(\bar{\mathbf{x}} \mid \hat{\mathbf{x}}) \right) \approx \sum_{t=1}^{T} m_t\, log\, P_\theta\left(x_t \mid \hat{\mathbf{x}}\right)$$

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 the last equation indicates that BERT factorizes the 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 long-range dependencies are the norm.
2. **Data Corruption**: Artificial symbols like masking tokens used in BERT's pretraining Masked Language Model (MLM) task do not appear in real data during fine-tuning, causing a discrepancy between pretraining and fine-tuning.

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:

"I went to *New York* and saw the *Empire State building*," or

"I went to *San Francisco* and saw the *Golden Gate bridge*."

But BERT might incorrectly predict something like: "I went to *San Francisco* and saw the *Empire State building*."

Since BERT predicts masked tokens simultaneously, it fails to learn their interlocking dependencies, which weakens the "learning signal." This is a major point against BERT, since even simple language models can learn at least unidirectional word dependencies (Kurita, 2019b).

11.4 Motivation for XLNet

Confronted with BERT's limitations, Yang et al. (2020) conceived **XLNet** which seeks to keep the benefits of *both* autoencoding and autoregressive language modeling while avoiding their issues:

1. Firstly, XLNet's use of an autoregressive language model (AR) objective function lets the probability $P_\theta(\mathbf{x})$ be factored using the probability product rule, which holds *universally* without having to default to BERT's false *independence assumption*.
2. Secondly, XLNet uses a **permutation language model** which can capture bidirectional context, coupled with a two-stream attention mechanism to create target-aware predictions.

Since XLNet uses an autoregressive language model (AR), it can predict tokens sequentially and autoregressively (albeit not necessarily left-to-right) while BERT is stuck predicting masked words simultaneously. Also, XLNet improves over Transformer-XL when incorporating its segment-level recurrence mechanism and relative positional encodings to learn longer-spanning dependencies.

11.5 Describing XLNet

11.5.1 Permutation Language Model

Can a model be trained to use **bidirectional context** while avoiding masking and its resulting problem of independent predictions?

To this scope, Yang et al. (2020) built XLNet with a **permutation language model**. Like language models, a **permutation language model** predicts unidirectionally, but instead of predicting in order, the model predicts tokens in a random order.

Formally, for an input sequence $\mathbf{x} = \{x_1, ..., x_T\}$, a permutation model uses that fact that there are $T!$ orders to factorize autoregressively, so assuming model parameters were shared across all these factorization orders then it would be expected that the model could accumulate forward *and* backward information (Yang et al., 2020). In other words, A permutation language model is forced to accumulate bidirectional context by finding dependencies between all possible input combinations. For example:

"I like cats more than dogs."

A traditional language model would predict the individual words sequentially, using previous tokens as context. 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", etc (Kurita, 2019b).

NOTE: the permutation language model only permutes factorization order; it does not change word order in the input sequence, only the order in which words are predicted. XLNet still builds "target-aware" predictions: input tokens can be fed in arbitrary order into XLNet using a masking feature in the Transformer's attention to temporarily cover certain tokens. Coupled with the use of positional embeddings and their associated original sequences, XLNet will still receive the tokens in correct order (Kurita, 2019b). This improves upon previous versions of permutation models which used an "implicit position awareness" (Yang et al., 2020).

11.5.2 Need for Target-Aware Representations

Trying to merge permutation language model and Transformer made XLNet's target predictions blind 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, the model masks the token's embedding but *unfortunately, also* its positional encoding. Thus it remains blind about the position of the target it should be predicting. 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, the problem is as follows: let $\mathcal{Z}_T$ = all possible permutations of the T-length sequence $\left[1, 2, ..., T\right]$; 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 $\mathbf{z}_{<t}$.

The authors began by parameterizing the next-token predictive distribution using the softmax:

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

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 which target position z_t it must predict. As a result, the model predicts the same distribution regardless of target position.

To remedy this, the authors replaced the hidden state h_θ in eq. (1) by $g_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}}, z_t\right)$ which also takes in the target position z_t in the permutation sequence.

11.5.3 Two-Stream Self-Attention

When formulating, g_θ the problem with the Transformer architecture produced a contradiction: (1) to predict the 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 attention mechanism** which uses two sets of hidden states to create an overall hidden state:

Content Stream: $h_\theta\left(\mathbf{x}_{\mathbf{z}_{\leq t}}\right)$ which 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 $m = 1, ..., M$:

$$h_{z_t}^{(m)} = \texttt{Attention}\left(\mathbf{Q} = h_{z_t}^{(m-1)}, \ \mathbf{KV} = \mathbf{h}_{\mathbf{z}_{\leq t}}^{(m-1)}; \ \theta\right) \tag{2}$$

Query Stream: $g_\theta\left(\mathbf{x}_{\mathbf{z}_{<t}}, z_t\right)$ to encode the *context* $\mathbf{x}_{\mathbf{z}\leq t}$ with the *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 seeing x_{z_t}) is computed:

$$g_{z_t}^{(m)} = \texttt{Attention}\left(\mathbf{Q} = g_{z_t}^{(m-1)},\ \mathbf{KV} = \mathbf{h}_{\mathbf{z}_{<t}}^{(m-1)};\ \theta\right) \tag{3}$$

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" and content stream information, so the model can predict "like" (Kurita, 2019b).

11.5.4 Relative Segment Encodings

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

11.6 Experimental Results of XLNet

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: $\log P(\texttt{New York} \mid \texttt{is a city})$. 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:

$$\begin{aligned} \mathcal{J}_{\text{BERT}} &= \log P\left(\texttt{New} \mid \texttt{is a city}\right) + \log P\left(\texttt{York} \mid \texttt{is a city}\right) \\ \mathcal{J}_{\text{XLNet}} &= \log P\left(\texttt{New} \mid \texttt{is a city}\right) + \log P\left(\texttt{York} \mid \texttt{New}, \texttt{is a city}\right) \end{aligned} \tag{4}$$

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

For reading comprehension datasets like SQuAD and RACE where longer-context representation is required, XLNet outperformed BERT as measured by their GLUE scores in table 5.

SQuAD2.0	EM	F1	SQuAD1.1	EM	F1
Dev set results (single model)					
BERT [10]	78.98	81.77	BERT† [10]	84.1	90.9
RoBERTa [21]	86.5	89.4	RoBERTa [21]	88.9	94.6
XLNet	**87.9**	**90.6**	XLNet	**89.7**	**95.1**
Test set results on leaderboard (single model, as of Dec 14, 2019)					
BERT [10]	80.005	83.061	BERT [10]	85.083	91.835
RoBERTa [21]	86.820	89.795	BERT* [10]	87.433	93.294
XLNet	**87.926**	**90.689**	XLNet	**89.898‡**	**95.080‡**

Table 5: Comparing GLUE scores of BERT and BERT-based model called RoBERTa with XLNet. From *Table 3 in XLNet: Generalized Autoregressive Pretraining for Language Understanding*, by Dai et al., 2019. https://arxiv.org/pdf/1906.08237.pdf. Copyright 2019 by Dai et al.

Dai et al. (2019) used an ablation study to determine which component of XLNet is better than BERT: the permutation language model, the Transformer-XL backbone, or some other details used in implementation, such

as span-based prediction and next sentence prediction. The table 6 compares BERT, Transformer-XL, and XL-Net variations. Rows 1-4 show that Transformer-XL and the permutation language model contribute to XLNet's success since its scores across different datasets are higher than for BERT. Row 5 shows removing the memory caching reduces performance for the longer-context containing RACE data set.

#	Model	RACE	SQuAD2.0 F1	SQuAD2.0 EM	MNLI m/mm	SST-2
1	BERT-Base	64.3	76.30	73.66	84.34/84.65	92.78
2	DAE + Transformer-XL	65.03	79.56	76.80	84.88/84.45	92.60
3	XLNet-Base ($K = 7$)	66.05	**81.33**	**78.46**	**85.84/85.43**	92.66
4	XLNet-Base ($K = 6$)	66.66	80.98	78.18	85.63/85.12	**93.35**
5	- memory	65.55	80.15	77.27	85.32/85.05	92.78
6	- span-based pred	65.95	80.61	77.91	85.49/85.02	93.12
7	- bidirectional data	66.34	80.65	77.87	85.31/84.99	92.66
8	+ next-sent pred	**66.76**	79.83	76.94	85.32/85.09	92.89

Table 6: Ablation study for XLNet's permutation language model on RACE, SQuAD, MNLI, SST-2 datasets. From *Table 6 in XLNet: Generalized Autoregressive Pretraining for Language Understanding*, by Dai et al., 2019. https://arxiv.org/pdf/1906.08237.pdf. Copyright 2019 by Dai et al.

XLNet's integration of an autoregressive language model (AR) with Transformer-XL and a two-stream attention mechanism results in clear improvements over masked language models like BERT, which are prone to false assumptions and data corruption.

12 ERNIE 1.0

12.1 Motivation for ERNIE 1.0

Previous models like Word2Vec, GloVe, and BERT create word vector representations only through 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 can is limited to predicting the token "K." but utterly fails at recognizing the whole entity *J. K. Rowling*. A model could use simple co-occurrence counts to predict the missing entity *Harry Potter* even without using long contexts, but it would not be making use of the relationship between the novel name and its writer.

Current NLP models analyzing simple word co-occurrence may miss additional information like sentence order and nearness and named entities.

This is where **ERNIE** steps in. **ERNIE (Enhanced Representation through Knowledge Integration)** can extrapolate the relationship between the *Harry Potter* entity and *J. K. Rowling* entity using implicit knowledge of words and entities, and uses this relationship to predict that Harry Potter is a series written by J. K. Rowling (Sun et al., 2019a).

ERNIE leverages a Transformer encoder with self-attention alongside novel knowledge integration techniques like entity-level masking and phrase-level masking so that prior knowledge contained in conceptual units like phrases and entities can contribute to learning longer semantic dependencies for better model generalization and adaptability (Sun et al., 2019a).

12.1.1 phrase-level masking

A phrase is a "small group of words of characters together acting as a conceptual unit" (Sun et al., 2019a). ERNIE uses lexical and chunking methods to determine phrase boundaries in sentences. In phrase-level masking,

ERNIE randomly selects phrases from the sentences and masks them (as in fig. 18), so that it can train by predicting the subpieces of the phrase. This way, phrase information can be built into ERNIE's learned word embeddings.

Figure 18: ERNIE 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.

12.1.2 entity-level masking

Sun et al. (2019a) say that name entities contain "persons, locations, organizations, products" which can be 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 shown in fig. 18.

12.2 Experimental Results of ERNIE 1.0

Task	Metrics	Bert		ERNIE	
		dev	test	dev	test
XNLI	accuracy	78.1	77.2	79.9 (+1.8)	78.4 (+1.2)
LCQMC	accuracy	88.8	87.0	89.7 (+0.9)	87.4 (+0.4)
MSRA-NER	F1	94.0	92.6	95.0 (+1.0)	93.8 (+1.2)
ChnSentiCorp	accuracy	94.6	94.3	95.2 (+0.6)	95.4 (+1.1)
nlpcc-dbqa	mrr	94.7	94.6	95.0 (+0.3)	95.1 (+0.5)
	F1	80.7	80.8	82.3 (+1.6)	82.7 (+1.9)

Table 7: Comparing ERNIE and BERT on five major nlp tasks in Chinese. From *Table 1 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.

As in table 7, ERNIE 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).

Sun et al. assert this is because of ERNIE's knowledge integration masking strategies, and this is supported in table 8; adding phrase masking to basic word-level masking improved ERNIE's performance almost a full percent, and adding entity-level masking to this combination resulted in still higher gains when sampling more data from larger texts.

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 8: 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.

Additionally, the authors tested ERNIE's knowledge learning ability using fill-in-the-blanks on named entities in paragraphs. In case 1 from table 9, 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 can learn contextual patterns to predict the correct named entity type but fails to fill the slot with the actual correct entity, while ERNIE fills the slots with the correct entity. In cases 3,4,6 BERT fills the slots with characters related to the sentences but not with the semantic concept, while ERNIE again predicts the correct entities.

Case	Text	Predicted by ERNIE	Predicted by 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
2	"The Reform Movement of 1898, also known as the Hundred-Day Reform, was a bourgeois reform carried out by the reformists such as ___ and Qichao Liang through Emperor Guangxu."	Youwei Kang	Schichang Sun	Youwei Kang
3	"Hyperglycemia is caused by defective ___ secretion or impaired biological function, or both. Long-term hyperglycemia in diabetes leads to chronic damage and dysfunction of various tissues, generally eyes, kidneys, heart, blood vessels and nerves."	Insulin	(Not a word in Chinese)	Insulin
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 9: Comparing ERNIE to BERT on Cloze Chinese Task. From *Figure 4 in ERNIE: Enhanced Representation Through Knowledge Integration*, by Sun et al., 2019. . Copyright 2019 by Sun et al.

However in case 4, ERNIE predicts the wrong city name, though it still understands the semantic type. It is evident that ERNIE's contextual knowledge understanding is far superior to BERT's predictions (Sun et al., 2019a).

13 ERNIE 2.0

13.1 Motivation for ERNIE 2.0

Models such as Word2Vec, GloVe, and BERT extract meaning using co-occurrences. Even XLNet's permutation language model relies on co-occurrences.

However, ERNIE 2.0 instead broadens the vision to more than just simple co-occurrence counts. Using a continual multi-task learning framework (fig. 19) to remember previously learned knowledge, ERNIE 2.0 can capture "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., 2019b).

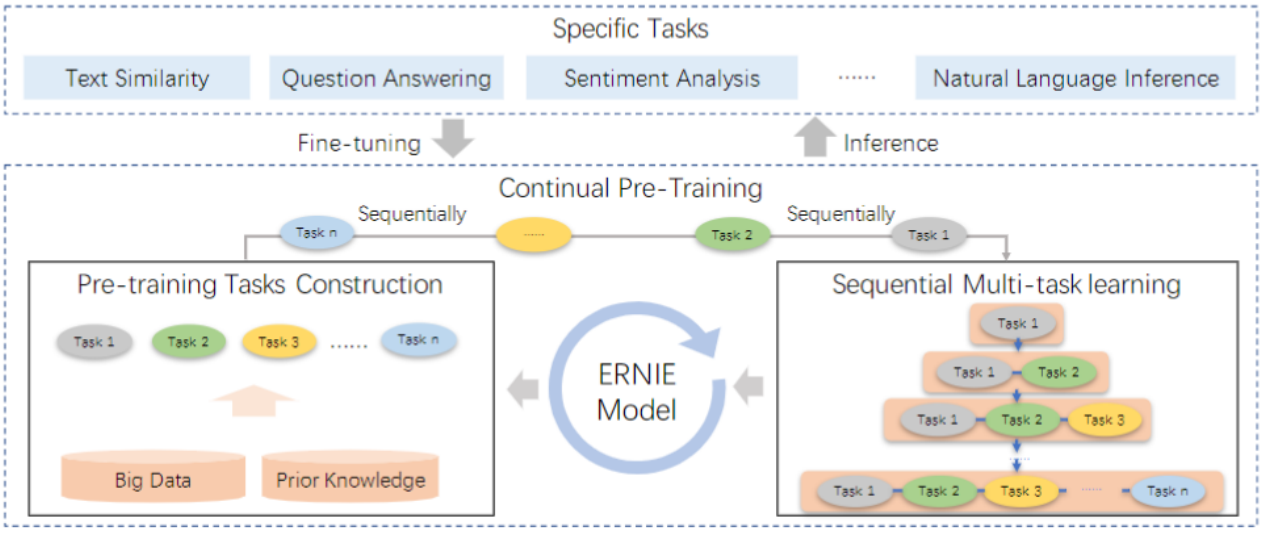


Figure 19: ERNIE 2.0's framework where embeddings are created by continual multi-task learning and then fine-tuned for specific nlp tasks. From *Figure 1 in ERNIE 2.0: A Continual Pre-Training Framework for Language Understanding*, by Sun et al., 2019. https://arxiv.org/pdf/1907.12412.pdf. Copyright 2019 by Sun et al.

13.2 Continual Multi-Task Learning

Traditional **continual learning** (Parisi et al. 2019; Chen and Liu, 2018) trains a model sequentially on multiple tasks to try to remember previously learned tasks while learning new ones. However, these procedures train the model with "only one task at each stage with the demerit that it may forget previously learned knowledge" (Sun et al., 2019b).

Multi-task learning means learning multiple tasks simultaneously. An Encoder is shared across task-specific architectures. In multitask model training, data is batched and allocated for specific task training. All tasks take turns learning on their mini-batched data then separately update the shared encoder based on the loss. However, this separate, individual task learning affects the weights in the shared encoder, causing "catastrophic forgetting" (Sequeira, 2019).

Inspired by human's ability to continuously accumulate information from past experience to develop new skills, **continual multi-task learning** combines **continual learning** and **multi-task learning**. Humans do not forget skating when learning how to ski; continual multi-task learning helps a model do likewise. It trains several tasks in parallel and the shared encoder is updated using the average of losses from all tasks. Specifically, for the first step, only Task 1 is learned and the encoder weights are updated from Task 1's loss. In step 2, the shared encoder is initialized with weights from the previous step and Task 1 and 2 are learned at the same time. Since the shared encoder has already learned from Task 1, only the loss from Task 2 will mostly updated the shared encoder, while retaining Task 1 information. Next, average loss is calculated to update the shared encoder. These steps continue until the model has trained on all tasks (Sequeira, 2019). ERNIE has two kinds of loss functions: the token-level loss and the sentence-level loss. In this way, ERNIE is similar to the Transformer-XL which respects sentence structure. During pre-training, ERNIE combines the sentence-level loss function with multiple token-level loss functions to update the model continually (Sun et al., 2019b).

13.3 Continual Pre-Training

ERNIE 2.0's pre-training component differs from previous pre-training ones since rather than using few pre-training objectives, it continually interweaves a large variety of word-aware, structure-aware, and semantic-aware pre-training tasks for capturing lexical, syntactic and semantic representations. The **continual pre-training** procedure has two steps: (1) continually create unsupervised pre-training tasks with large data and prior knowledge (like named entities, phrases, and discourse relations), and (2) use continual multi-task learning to update

ERNIE incrementally (Sun et al., 2019b). All the pre-training tasks are classification types and are described below.

13.3.1 Word-Aware Pre-Training Tasks

These tasks help ERNIE learn better representations at the word level. Closely following descriptions from Sun et al. (2019b):

- **Knowledge-Masking Task:** ERNIE masks words, phrases, and named entities then predicts them to learn dependency in local and global contexts.
- **Capitalization Prediction Task:** to classify whether a word is capitalizated or not. The reason for this task is that capitalized words usually specific kinds of semantic information compared to latter words in a sentence.
- **Token-Document Relation Prediction Task:** this task predicts if a token in a segment appears in other segments of the original document. The reason for this is that words appearing many times are common or relevant to the document's topic, and so can help the model capture the document's theme.

13.3.2 Structure-Aware Pre-Training Tasks

Structure-aware pre-training tasks help the model learn how sentences are related in a document.

- **Sentence Reordering Task:** a paragraph is broken into chunks, shuffled, and the model must reorder the shuffled segments into the original paragraph.
- **Sentence Distance Task:** the model classifies if "0" two sentences are adjacent in the same document, "1" two sentences are in the same document but not adjacent, and "2" two sentences are from two different documents.

13.3.3 Semantic-Aware Pre-Training Tasks

These tasks help ERNIE learn the semantics of sentences.

- **Discourse Relation Task:** to predict the semantic or rhetorical relation between two sentences. For example, "The pine tree crashed to the ground. *[because]* The strong mountain winds were merciless." The model must predict the discourse marker "because", showing it learns the contrast between the two sentences, and in doing so, their semantics.
- **Information Retrieval (IR) Relevance Task:** given a pair of user query and a document, the task is to classify if the document is strongly-relevant, weakly-relevant, or irrelevant or random to the user query, in terms of semantic information. This task helps ERNIE learn the semantics of the user query with respect to the document title.

13.4 Experimental Results of ERNIE 2.0

In table 10, ERNIE 2.0 outperforms BERT and even XLNet for most of the English tasks, as measured by GLUE benchmark (Sun et al., 2019b).

The models BERT, XLNet, and ERNIE 2.0 were compared on Chinese tasks: machine reading comprehension (MRC) (CMRC dataset), named entity recognition (NER) (MSRA-NER dataset), natural language inference (NLI) (XNLI dataset), sentiment analysis (SA) (ChnSenitCorp dataset), semantic textual similarity (STS) (LCQMC dataset), and question answering (QA) (NLPCC-DBQA dataset). table 10 shows that ERNIE 1.0 outperforms BERT in several tasks, but not all, while ERNIE 2.0 outperforms both previous models on all tasks (Sun et al., 2019b).

Task(Metrics)	*BASE model*		*LARGE model*				
	Test		Dev			Test	
	BERT	ERNIE 2.0	BERT	XLNet	ERNIE 2.0	BERT	ERNIE 2.0
CoLA (Matthew Corr.)	52.1	**55.2**	60.6	63.6	**65.4**	60.5	**63.5**
SST-2 (Accuracy)	93.5	**95.0**	93.2	95.6	**96.0**	94.9	**95.6**
MRPC (Accurary/F1)	84.8/88.9	**86.1/89.9**	88.0/-	89.2/-	**89.7/-**	85.4/89.3	**87.4/90.2**
STS-B (Pearson Corr./Spearman Corr.)	87.1/85.8	**87.6/86.5**	90.0/-	91.8/-	**92.3/-**	87.6/86.5	**91.2/90.6**
QQP (Accuracy/F1)	89.2/71.2	**89.8/73.2**	91.3/-	91.8/-	**92.5/-**	89.3/72.1	**90.1/73.8**
MNLI-m/mm (Accuracy)	84.6/83.4	**86.1/85.5**	86.6/-	**89.8/-**	89.1/-	86.7/85.9	**88.7/88.8**
QNLI (Accuracy)	90.5	**92.9**	92.3	93.9	**94.3**	92.7	**94.6**
RTE (Accuracy)	66.4	**74.8**	70.4	83.8	**85.2**	70.1	**80.2**
WNLI (Accuracy)	**65.1**	**65.1**	-	-	-	65.1	**67.8**
AX(Matthew Corr.)	34.2	**37.4**	-	-	-	39.6	**48.0**
Score	78.3	**80.6**	-	-	-	80.5	**83.6**

Table 10: Results on GLUE benchmark, where dev set contains medium of five runs and test set is scored by GLUE server. State of the art results are in bold. From *Table 5 in ERNIE 2.0: A Continual Pre-Training Framework for Language Understanding*, by Sun et al., 2019. https://arxiv.org/pdf/1907.12412.pdf. Copyright 2019 by Sun et al.

Task	Metrics	BERT_{BASE}		ERNIE 1.0_{BASE}		ERNIE 2.0_{BASE}		ERNIE 2.0_{LARGE}	
		Dev	Test	Dev	Test	Dev	Test	Dev	Test
CMRC 2018	EM/F1	66.3/85.9	-	65.1/85.1	-	69.1/88.6	-	**71.5/89.9**	-
DRCD	EM/F1	85.7/91.6	84.9/90.9	84.6/90.9	84.0/90.5	88.5/93.8	88.0/93.4	**89.7/94.7**	**89.0/94.2**
DuReader	EM/F1	59.5/73.1	-	57.9/72.1	-	61.3/74.9	-	**64.2/77.3**	-
MSRA-NER	F1	94.0	92.6	95.0	93.8	95.2	93.8	**96.3**	**95.0**
XNLI	Accuracy	78.1	77.2	79.9	78.4	81.2	79.7	**82.6**	**81.0**
ChnSentiCorp	Accuracy	94.6	94.3	95.2	95.4	95.7	95.5	**96.1**	**95.8**
LCQMC	Accuracy	88.8	87.0	89.7	87.4	**90.9**	**87.9**	**90.9**	**87.9**
BQ Corpus	Accuracy	85.9	84.8	86.1	84.8	86.4	85.0	**86.5**	**85.2**
NLPCC-DBQA	MRR/F1	94.7/80.7	94.6/80.8	95.0/82.3	95.1/82.7	95.7/84.7	95.7/85.3	**95.9/85.3**	**95.8/85.8**

Table 11: Results of Chinese nlp tasks. State of the art results are bolded. From *Table 6 in ERNIE 2.0: A Continual Pre-Training Framework for Language Understanding*, by Sun et al., 2019. https://arxiv.org/pdf/1907.12412.pdf. Copyright 2019 by Sun et al.

In table 12, we see how multi-task learning trains all tasks simultaneously, continual learning trains the tasks one by one, while ERNIE 2.0's continual multi-task learning can allocate different iterations to each task in the various stages of training. Continual multi-task learning scores higher on the MNLI, SST-2, and MRPC datasets than the other two pre-training methods, confirming the guess that separately, multi-task and continual learning are prone to forgetting and other inefficiencies (Sun et al., 2019b).

14 Discussion, Conclusion, and Future Work

- summarize what was written -> machine translation (MT) Did not expect this to be directly useful to my long term purpose of concept extraction (ml) but a good source of implementations -> .... summarize
- indicate future work directions -> bayes polysemy -> different representations: knowledge graph embeddings -> model fine-tuning, transfer learning, domain adaptation for easier use. -> more research into concept embeddings and key phrase extraction.

References

Pre-training method	Pre-training task	Training iterations (steps)				Fine-tuning result		
		Stage 1	Stage 2	Stage 3	Stage 4	MNLI	SST-2	MRPC
Continual Learning	Knowledge Masking	50k	-	-	-	77.3	86.4	82.5
	Capital Prediction	-	50k	-	-			
	Token-Document Relation	-	-	50k	-			
	Sentence Reordering	-	-	-	50k			
Multi-task Learning	Knowledge Masking	50k				78.7	87.5	83.0
	Capital Prediction	50k						
	Token-Document Relation	50k						
	Sentence Reordering	50k						
continual Multi-task Learning	Knowledge Masking	20k	10k	10k	10k	**79.0**	**87.8**	**84.0**
	Capital Prediction	-	30k	10k	10k			
	Token-Document Relation	-	-	40k	10k			
	Sentence Reordering	-	-	-	50k			

Table 12: Experimental results of the different continual pre-training methods, using knowledge-masking, capital prediction, token-document, and sentence reording pre-training tasks. From *Table 7 in ERNIE 2.0: A Continual Pre-Training Framework for Language Understanding*, by Sun et al., 2019. https://arxiv.org/pdf/1907.12412.pdf. Copyright 2019 by Sun et al.

A APPENDIX: Glossary of NLP Tasks

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

A.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).

A.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).

A.3 machine translation (MT)

Machine translation (MT) task translates an input text in a given language to a target language. There are many population neural translation models like the Sequence To Sequence Model and BERT that use the attention mechanism to account for contextual meaning across a sentence and not just translate word by word.

A.4 neural machine translation (NMT)

Neural machine translation (NMT) is machine translation (MT) applied in Neural Network Language Models.

A.5 question answering (QA)

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

A.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.

- **Example Input Sentence:** "Mary sold the book to John."
- **Example Output:** for the verb $\left[\text{"to sell"}\right]_{\text{predicate}}$; for the noun or argument $\left[\text{"Mary"}\right]_{\text{agent}}$; for the noun $\left[\text{"the book"}\right]_{\text{goods (theme)}}$; for the noun $\left[\text{"John"}\right]_{\text{recipient}}$.

SRL can give multiple labels depending on the usage of the syntactic constituent in the sentence.

A.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.

A.8 named entity disambiguation (NED)

Named entity disambiguation (NED) or **entity linking (EL)** links or maps mentions of an entity (such as persons, locations, companies) within a text to corresponding unique entities in a knowledge base (Shahbazi et al., 2019).

- **Example input text:** "Jordan as a member of the Tar Heels' national championship team."
- **Example output:** the language model should predict the named entity (person) "Michael Jordan", given this exists in the knowledge base, rather than the ambigious mention "Jordan".

A.9 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, advergb, adjective, ...).

From (Mohler, 2019):

- **Example Input Sentence:** "The tall man is going to quickly walk under the ladder."
- **Example Output:** $\left[\text{"man"}\right]_{\text{Noun}}$, $\left[\text{"walk"}\right]_{\text{Verb}}$, $\left[\text{"ladder"}\right]_{\text{Noun}}$, $\left[\text{"quickly"}\right]_{\text{Adverb}}$ and so on.

A.10 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*.

From (Mohler, 2019):

- **Example Input Sentence:** "The tall man is going to quickly walk under the ladder."
- **Example Output:** $\left[\text{"the tall man"}\right]_{\text{Noun Phrase}}$, $\left[\text{"is going to quickly walk"}\right]_{\text{Verb Phrase}}$, $\left[\text{"under the ladder"}\right]_{\text{Preopositional Phrase}}$.

A.11 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.

A.12 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).

A.13 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.

A.14 textual entailment (TE)

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

A.15 entailment directionality prediction

Given a pair of words, the **entailment directionality prediction** task must predict if the previous word entails the next one, or vice versa. It is known that entailment holds for the given word pair and only its directionality is being predicted (Brazinkas et al., 2018).

A.16 sentiment classification (SC)

See sentiment analysis (SA).

A.17 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.

This task may assign weighted sentiment values to the entities and themes within text. For instance, a hotel getting a review "astonishing scenery" would get a higher sentiment score than "banal lake view" because of the stronger adjective "astonishing."

A.18 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.

A.19 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.

A.20 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.

- **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."

- **Example output:** tag the coreferent phrases "he" and "the hairy ape" with "the monkey"; and also tag "it" with the same label as "banana."

This task is used as a step towards more general tasks, and is not an end-user task.

A.21 sequence labeling (SL)

Sequence labeling (SL) is a general NLP task that assigns a label to every token in an input sequence, where tokens can be words or phrases. Two forms of sequence labeling include part of speech tagging (POS) and span labeling

A.22 span labeling

In the **span labeling** task, spans or groups of words are labeled. This can be used in search tasks to provide entities to spans of words for specifying a search query.

A.23 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.

A.24 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.

A.25 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).

A.26 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:

B APPENDIX: Training Neural Networks

The **backward propagation of errors** is a method used for training neural network to learn values for the parameters, and is used in conjunction with an optimization method, such as **gradient descent**. The backward propagation algorithm does a two-phase cycle consisting of error propagation across the graph followed by the parameter weight update.

Neural network training consists of three phases: 1) forward propagation, 2) error calculation, and 3) backward propagation, which itself consists of two phases. To describe this process, we use notation from Gibiansky (2014) and define the following:

- x_i^l is the input to the i-th in layer l, denoted u_i^l.
- u_i^l is the i-th **unit** or **node** in layer l, where u_i^0 with $l = 0$ means the i-th unit in the input layer.
- u^L is the last (output) layer L of the network.
- a_i^l is the output **activation** value of unit u_i^l, calculated from a nonlinearity function.
- a^L is the **activation** value in the last layer L.

A **fully-connected neural network** with L layers has three categories of layers:

1. the **input** or **embedding layer** with units u_i^0 whose values are determined by the input vectors.
2. the **hidden layers** with units u_i^l whose values are obtained from previous layers.
3. the **output layer** with units u_i^L whose values are computed from the last hidden layer.

The neural network trains to update its weight matrix $W = \left\{w_{ij}^l\right\}$, where w_{ij}^l is the weight value from unit u_i^l's output to another unit u_j^{l+1}. Whenever an output is computed from an input, a **nonlinearity function** $\sigma(\cdot)$ is applied to the input x and this is intuitively seen as "passing" the input through a layer.

B.1 Forward Propagation

Forward propagation or **forward pass** is the procedure used to propagate an input vector forward through the network layers. The original input is transformed over a series of nonlinear functions to get its activation values a_i^l, until the output layer is reached, where the activations are transformed via another nonlinearity to get the final activations a^L.

B.1.1 Forward Propagation Algorithm

The **forward propagation** algorithm is described as follows:

1. Compute the activation values a_i^l for layers with known inputs, using the activation nonlinearity function $\sigma(\cdot)$:

$$a_i^l = \sigma(x_i^l) + I_i^l$$

2. Compute the input values x_i^l for the next layer l from the activations a_i^l:

$$x_i^l = \sum_j w_{ji}^{l-1} a_j^{l-1}$$

3. Steps 1 and 2 are repeated until the output layer is reached, to get the final output values a^L at the last layer L.

B.2 Error Calculation

After **forward propagation**, errors $E(a^L)$ are computed by comparing the network's output with the target predictions, via a loss function, and the error value is calculated for each neuron in the output layer.

The derivative of the error $E(a^L)$ with respect to computed activations a_i^L is written $\frac{d}{da_i^L}\left(E(a^L)\right)$ and depends only on the activations. This will be used to optimize the weights to minimize the error in the **backward propagation algorithm**.

B.3 Backward Propagation

Backward propagation consists of a first phase when the error values are propagated backwards across the graph, starting from the output layer and moving back over the input layer, until each neuron is updated with the error value. Backpropagation uses these errors to *calculate the gradient of the loss function with respect to the weights in the network*. In the second phase, the gradient of the loss is passed as an argument to the optimization method so that the parameter weights can be adjusted, towards minimizing the loss function.

B.3.1 Derivation of Backward Propagation

In order to use an **optimization algorithm** to train the network, we must compute the error derivative $\frac{\partial E}{\partial w_{ij}^l}$ with respect to each weight value, w_{ij}^l, using the chain rule.

Also, since $x_i^l = \sum_j w_{ji}^{l-1} a_j^{l-1}$, the partial derivative with respect to any weight is equal to just the activation from the origin neuron: $\frac{\partial x_j^{l+1}}{\partial w_{ij}^l} = a_i^l$, resulting in:

$$\frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial x_j^{l+1}} \cdot \frac{\partial x_j^{l+1}}{\partial w_{ij}^l} = \frac{\partial E}{\partial x_j^{l+1}} \cdot a_i^l$$

Now, to decompose further the partial $\frac{\partial E}{\partial x_j^{l+1}}$, we can calculate the partial derivative of the error with respect to the input for the current layer l. Using the chain rule and $a_i^l = \sigma(x_i^l) + I_i^l$, we write:

$$\frac{\partial E}{\partial x_j^l} = \frac{\partial E}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial x_j^l} = \frac{\partial E}{\partial a_j^l} \cdot \frac{\partial}{\partial x_j^l}\left(\sigma(x_j^l) + I_j^l\right) = \frac{\partial E}{\partial a_j^l} \cdot \sigma'(x_j^l)$$

Lastly, to decompose the partial $\frac{\partial E}{\partial a_j^l}$, we consider two cases: if we are at the output layer, so $l = L$, or not. When $l = L$, then the partial $\frac{\partial E}{\partial a_j^l}$ is simply the derivative of the error function because the error is just a function of a_i^L and none of the other activations in the output layer:

$$\frac{\partial E}{\partial a_j^L} = \frac{\partial}{\partial a_j^L}\left(E(a^L)\right)$$

In the second case, for layers l other than the output layer L, we must use the chain rule to sum over all the contributions of the activation in all layers.

$$\frac{\partial E}{\partial a_j^l} = \sum \frac{\partial E}{\partial x_j^{l+1}} \cdot \frac{\partial x_j^{l+1}}{\partial a_i^l} = \sum \frac{\partial E}{\partial x_j^{l+1}} \cdot w_{ij}$$

This shows that $\frac{\partial E}{\partial a_j^l}$ equals the derivatives of the inputs to the next layer weighted by the importance of the activation a_i^l to each input. Intuitively, this means "the error at a particular node in layer l is a combination of errors at the next nodes (layer $l+1$), weighted by the size of the contribution of the node in layer l to each of those nodes in layer $l+1$" (Gibiansky, 2014).

B.3.2 Backward Propagation Algorithm

1. Calculate the errors at the output layer L, with respect to the activations a_i^L at the output layer:

$$\frac{\partial E}{\partial a_i^L} = \frac{d}{da_i^L}\left(E(a^L)\right)$$

2. Calculate the partial derivative of the error with respect to the neuron input $\frac{\partial E}{\partial x_j^l}$ (also denoted "deltas") at an arbitrary layer l:

$$\frac{\partial E}{\partial x_j^l} = \sigma'(x_j^l) \cdot \frac{\partial E}{\partial a_j^l}$$

3. Calculate errors at the previous layer (this is called backpropagating the errors):

$$\frac{\partial E}{\partial a_i^l} = \sum w_{ij}^l \cdot \frac{\partial E}{\partial x_j^{l+1}}$$

4. Repeat Steps 2 and 3 until the deltas are known for all layers excluding the input layer.

5. Complete the steps by calculating the gradient of the error with respect to the weights:

$$\frac{\partial E}{\partial w_{ij}^l} = a_i^l \cdot \frac{\partial E}{\partial x_j^{l+1}}$$

   Intuitively, this means to find derivatives with respect to weights in a given layer, we multiply activations for that layer and deltas for the next layer, so deltas for the input layer never need to be calculated.

C APPENDIX: Preliminary Building Blocks

C.1 Recurrent Neural Networks (RNN)

C.1.1 Motivation for RNNs

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).

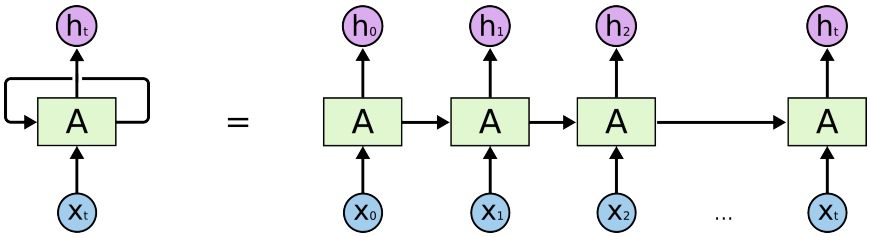


Figure 20: Unrolled view of Recurrent Neural Network with Hidden States h_i and inputs x_i. From *Understanding LSTMs*, by Colah., 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

C.1.2 Describing RNNs

An RNN is a unidirectional language model in that it uses infinite *left* context words to the left of the target word. It is a neural network consisting of a hidden state vector $\vec{h}$ and output vector $\vec{y}$ and takes a sequence (sentence) of input symbols (words) $\vec{x} = \{x_1, ..., x_T\}$, where each x_i is a word. At each time step 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 $P\left(x_t \mid x_{t-1}, ..., x_1\right)$. The probability of the entire sequence sentence $\vec{x}$ is the product of all the probabilities of the individual words, $P(\vec{x}) = \prod_{t=1}^{T} P\left(x_t \mid x_{t-1}, ..., x_1\right)$ (Cho, 2014).

Nguyen (2018b) describes the basic workflow of RNNs as follows:

1. First, words are transformed into numeric vectors, allowing the RNN to process the vector sequence, taking one vector at a time.
2. While processing the inputs in the above step, the RNN passes previous hidden state to the next step of the sequence. The hidden state serves as memory for the network by holding previous information.

   To calculate hidden state for a particular cell in the RNN, the input and previous hidden state are combined to form a vector, which is then passed through a "tanh" activation function so that its components are squashed betwen -1 and 1 to avoid large values. The output of this operation becomes the new hidden state.
3. This process is repeated until an output prediction word is generated.

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

C.2.1 Motivation for LSTM: Problem with RNNs

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, thus showing its lack of handling **long-term dependencies** (Colah, 2015).

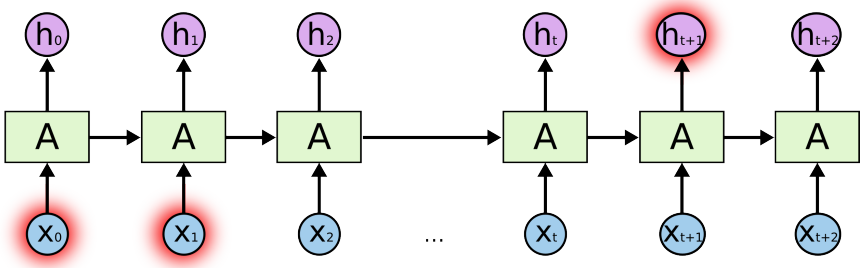


Figure 21: Long-Term Dependency Problem in RNNs (widening gap between inputs x_i and hidden states h_j. From *Understanding LSTMs*, by Colah, 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

The **short-term memory problem** occurs due to the **vanishing gradient problem**. During backward propagation of errors through the neural network (described in Appendix A), 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 (Nguyen, 2018b).

C.2.2 Describing LSTMs

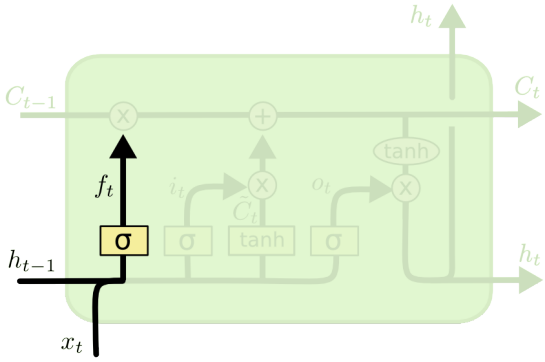
A long-short term memory network (LSTM) is a type of RNN that sequentially extracts information from each word in a sentence and embeds the information into a semantic vector.

Long-short term memory networks (LSTM) learn long-term information by design, contrary to RNNs. LSTMs use features such as **cell state** and **gates** to regulate information flow from earlier time steps to later time steps. The gates are separate neural networks that decide which information to add or remove from the cell state, thus explicitly letting the LSTM "remember" or "forget" information (Nguyen, 2018b). Simply, LSTMs differ from RNNs in their repeating module since the standard RNN contains a single tanh activation layer while the LSTM contains the four neural network layers (gates).

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**, which is shown in Figure 11 as the topmost line with the gates merging into it. For example, for a language predicting the next word based on previous ones, the cell state might include the gender of the present subject so that the correct pronouns are used. 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), the gates (neural network layers) regulating information are as follows:

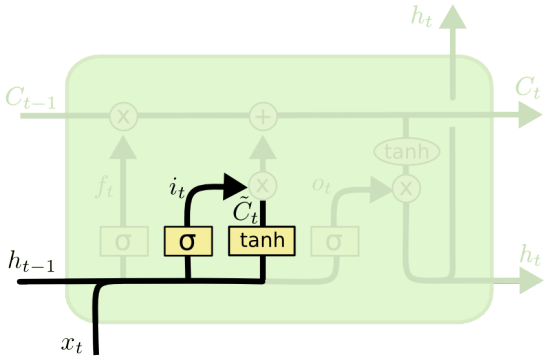


Forget Gate: the forget gate decides information to discard or keep. The previous hidden state h_{t-1} and current input x_t are passed through the sigmoid nonlinearity function. The forget gate outputs a number between 0 and 1 for each number in the cell state C_{t-1}; values closer to 0 indicate the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

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

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

Figure 22: Forget Gate Calculation. From *Understanding LSTMs*, by Colah, 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

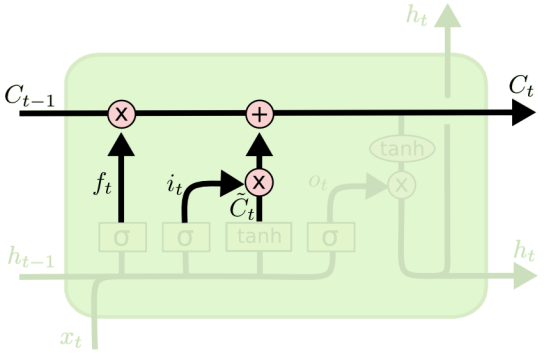


Input Gate: the input gate i_t updates the cell state C_t. Previous hidden state h_{t-1} and current input x_t are passed though a signmoid function normalize vector cells between 0 and 1. The input gate is later used with the cell state to decide how values are updated.

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

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



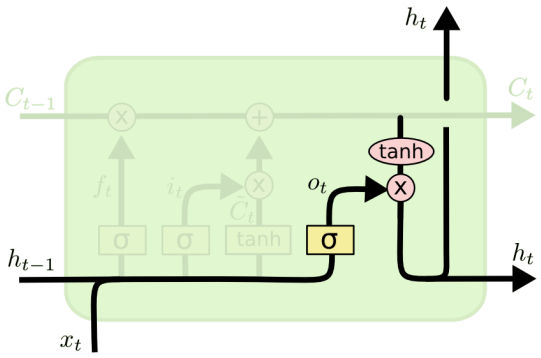
Cell State: Current cell state C_t takes h_{t-1} and x_t and normalizes them to be between -1 and 1 via a hyperbolic tangent nonlinearity:

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

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:

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

Figure 24: Cell State Calculation. From *Understanding LSTMs*, by Colah, 2015.https://colah.github.io/posts/2015-08-Understanding-LSTMs/.



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

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

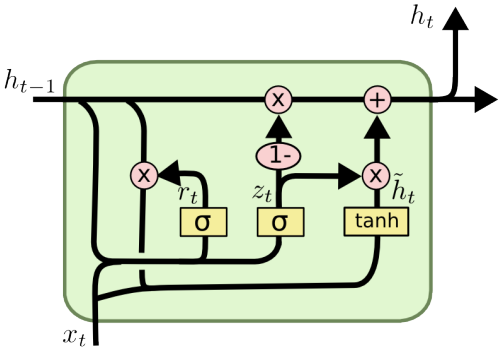
Figure 25: Output Gate Calculation. From *Understanding LSTMs*, by Colah, 2015.https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

C.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 26: 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).

D APPENDIX: Application To Machine Translation in NLP (Using PyTorch and Seq-To-Seq Model With Attention)