Word2Vec Research Paper Implementation

Abstract

We propose two novel model architectures for computing continuous vector representations of words from very large data sets. The quality of these representations is measured in a word similarity task, and the results are compared to the previously best performing techniques based on different types of neural networks. We observe large improvements in accuracy at much lower computational cost, i.e. it takes less than a day to learn high quality word vectors from a 1.6 billion words data set. Furthermore, we show that these vectors provide state-of-the-art performance on our test set for measuring syntactic and semantic word similarities.

Many current NLP systems and techniques treat words as atomic units - there is no notion of similarity between words, as these are represented as indices in a vocabulary. This choice has several good reasons - simplicity, robustness and the observation that simple models trained on huge amounts of data outperform complex systems trained on less data. An example is the popular N-gram model used for statistical language modeling - today, it is possible to train N-grams on virtually all available data (trillions of words [3]).

However, the simple techniques are at their limits in many tasks. For example, the amount of relevant in-domain data for automatic speech recognition is limited - the performance is usually dominated by the size of high quality transcribed speech data (often just millions of words). In machine translation, the existing corpora for many languages contain only a few billions of words or less. Thus, there are situations where simple scaling up of the basic techniques will not result in any significant progress, and we have to focus on more advanced techniques.

With progress of machine learning techniques in recent years, it has become possible to train more complex models on much larger data set, and they typically outperform the simple models. Probably the most successful concept is to use distributed representations of words [10]. For example, neural network based language models significantly outperform N-gram models [1, 27, 17].

We use recently proposed techniques for measuring the quality of the resulting vector representations, with the expectation that not only will similar words tend to be close to each other, but that words can have **multiple degrees of similarity** [20]. This has been observed earlier in the context of inflectional languages - for example, nouns can have multiple word endings, and if we search for similar words in a subspace of the original vector space, it is possible to find words that have similar endings [13, 14].

Somewhat surprisingly, it was found that similarity of word representations goes beyond simple syntactic regularities. Using a word offset technique where simple algebraic operations are performed on the word vectors, it was shown for example that *vector("King") - vector("Man") + vector("Woman")* results in a vector that is closest to the vector representation of the word *Queen* [20].

For all the following models, the training complexity is proportional to

$$O = E \times T \times Q, \tag{1}$$

where E is number of the training epochs, T is the number of the words in the training set and Q is defined further for each model architecture. Common choice is $E = 3 - 50$ and T up to one billion. All models are trained using stochastic gradient descent and backpropagation [26].

2.1 Feedforward Neural Net Language Model (NNLM)

The probabilistic feedforward neural network language model has been proposed in [1]. It consists of input, projection, hidden and output layers. At the input layer, N previous words are encoded using 1-of-V coding, where V is size of the vocabulary. The input layer is then projected to a projection layer P that has dimensionality $N \times D$, using a shared projection matrix. As only N inputs are active at any given time, composition of the projection layer is a relatively cheap operation.

The NNLM architecture becomes complex for computation between the projection and the hidden layer, as values in the projection layer are dense. For a common choice of $N = 10$, the size of the projection layer (P) might be 500 to 2000, while the hidden layer size H is typically 500 to 1000 units. Moreover, the hidden layer is used to compute probability distribution over all the words in the vocabulary, resulting in an output layer with dimensionality V. Thus, the computational complexity per each training example is

$$Q = N \times D + N \times D \times H + H \times V, \tag{2}$$

where the dominating term is $H \times V$. However, several practical solutions were proposed for avoiding it; either using hierarchical versions of the softmax [25, 23, 18], or avoiding normalized models completely by using models that are not normalized during training [4, 9]. With binary tree representations of the vocabulary, the number of output units that need to be evaluated can go down to around $log_2(V)$. Thus, most of the complexity is caused by the term $N \times D \times H$.

In our models, we use hierarchical softmax where the vocabulary is represented as a Huffman binary tree. This follows previous observations that the frequency of words works well for obtaining classes in neural net language models [16]. Huffman trees assign short binary codes to frequent words, and this further reduces the number of output units that need to be evaluated: while balanced binary tree would require $log_2(V)$ outputs to be evaluated, the Huffman tree based hierarchical softmax requires only about $log_2(Unigram\_perplexity(V))$. For example when the vocabulary size is one million words, this results in about two times speedup in evaluation. While this is not crucial speedup for neural network LMs as the computational bottleneck is in the $N \times D \times H$ term, we will later propose architectures that do not have hidden layers and thus depend heavily on the efficiency of the softmax normalization.

Breaking Down the Explanation in Simple Terms

This passage is discussing the **computational complexity** of training **Neural Network Language Models (NNLMs)** and how to improve their efficiency. Let's break it down step by step with an easy-to-understand example.

---

1. What is an NNLM (Neural Network Language Model)?

A **Neural Network Language Model (NNLM)** is a model that predicts the next word in a sentence based on the previous words. It consists of several layers:

- **Input Layer:** Represents previous words using one-hot encoding.
- **Projection Layer:** Maps input words into a smaller vector space.
- **Hidden Layer:** A dense neural layer that processes information.
- **Output Layer:** Computes the probability of each word in the vocabulary being the next word.

↓

2. Computational Complexity in NNLMs

The main problem with NNLMs is that they are **computationally expensive** to train, especially due to the **output layer**.

🔹 **Formula for Complexity:**

$$Q = N \times D + N \times D \times H + H \times V$$

Where:

- N = Number of previous words used for prediction (e.g., 10 words).
- D = Dimensionality of word vectors (e.g., 500–2000).
- H = Number of hidden layer neurons (e.g., 500–1000).
- V = Vocabulary size (e.g., 1 million words).

🚨 **Biggest problem:** The term $H \times V$ is the slowest part because calculating probabilities for every word in a huge vocabulary is computationally expensive.

3. Why is Computing the Output Layer So Slow?

The **output layer** predicts the next word by computing probabilities for **every word in the vocabulary**. For example:

- If you have **1 million words** in the vocabulary, the model must calculate **1 million probabilities** for each prediction.
- This takes a lot of time, making training very slow.

---

4. How to Speed it Up?

To make training faster, the paper suggests using **Hierarchical Softmax with Huffman Trees**.

🔹 **How does a Huffman Tree help?**

- Instead of predicting a word directly, we represent words as binary trees.
- **Frequent words** get **shorter binary codes**, meaning they are evaluated with fewer computations.
- **Rare words** get **longer binary codes**, meaning they take more time but occur less frequently.
- This reduces the number of calculations needed.

🔹 **Example of Speedup:**

- A **normal softmax** requires **$\log_2(V)$** computations (for $V = 1,000,000$, that's ~20 steps).
- A **Huffman tree-based softmax** requires about **$\log_2$(unigram perplexity(V))**, which is **faster** (reducing computation by ~2x).

5. What is the Main Bottleneck?

Even though hierarchical softmax speeds up training, the biggest computational challenge remains the $N \times D \times H$ term.

- This happens because the **hidden layer** is dense, meaning every word must pass through a large number of computations before reaching the output layer.

💡 **Solution:** Later in the paper, the authors suggest models that remove the hidden layer to make training even faster.

Key Takeaways

1. **NNLMs are slow** because they compute probabilities for all words in a large vocabulary.
2. The biggest bottleneck is **computing the output layer** (probabilities for all words).
3. **Hierarchical Softmax with Huffman Trees** speeds up training by reducing the number of computations needed.
4. The main remaining issue is the **hidden layer's complexity** (which is later optimized in other architectures).

2.2 Recurrent Neural Net Language Model (RNNLM)

Recurrent neural network based language model has been proposed to overcome certain limitations of the feedforward NNLM, such as the need to specify the context length (the order of the model N), and because theoretically RNNs can efficiently represent more complex patterns than the shallow neural networks [15, 2]. The RNN model does not have a projection layer; only input, hidden and output layer. What is special for this type of model is the recurrent matrix that connects hidden layer to itself, using time-delayed connections. This allows the recurrent model to form some kind of short term memory, as information from the past can be represented by the hidden layer state that gets updated based on the current input and the state of the hidden layer in the previous time step.

The complexity per training example of the RNN model is

$$Q = H \times H + H \times V, \tag{3}$$

where the word representations D have the same dimensionality as the hidden layer H. Again, the term $H \times V$ can be efficiently reduced to $H \times log_2(V)$ by using hierarchical softmax. Most of the complexity then comes from $H \times H$.

Understanding Recurrent Neural Network Language Model (RNNLM) in Simple Terms

This passage describes how **Recurrent Neural Networks (RNNs)** are used for language modeling and how they improve upon traditional **Feedforward Neural Networks (FNNs)**. Let's break it down step by step.

---

1. What's the Problem with Traditional NNLMs?

Traditional **Neural Network Language Models (NNLMs)**, such as **Feedforward NNLMs**, have two major issues:

1. **Fixed Context Length**:
   - In NNLMs, we need to specify a fixed number of previous words (**N**) to predict the next word.
   - This means the model can only "see" a limited number of past words, even though language depends on long-term context.
2. **Shallow Structure**:
   - Feedforward networks do not have any "memory" of past words beyond the **N** words in the input.
   - They process words in a **static** way, without keeping track of what was said before.

---

2. How Does RNNLM Fix These Issues?

RNNLM (Recurrent Neural Network Language Model) improves upon NNLMs by introducing a special mechanism:

- **"Memory" with a Recurrent Hidden Layer**
- Instead of just processing a fixed number of previous words, RNNs maintain a **hidden state** that carries information from past words.
- The hidden state is **updated at every time step** based on the **current input** and the **previous hidden state**.

💡 **Key Difference:**

- In a **feedforward network**, each input is treated independently.
- In an **RNN**, the hidden layer carries information from previous steps, allowing it to "remember" past words.

---

3. How Does RNN "Remember" Information?

🔹 The **Hidden Layer Connects to Itself!**

- RNNs have a **special matrix** (called the **recurrent matrix**) that allows the hidden layer to **pass its previous state to itself** in the next step.
- This means the model can use information from previous time steps to improve its predictions.

4. Computational Complexity of RNNLM

The training complexity of an RNNLM is given by:

$$Q = H \times H + H \times V$$

Where:

- **H** = Number of neurons in the hidden layer.
- **V** = Vocabulary size (number of words in the dictionary).

🚀 **What does this mean?**

1. **H × H:** The biggest computation comes from the hidden layer updating itself. This is expensive because it happens at every time step.
2. **H × V:** The model also has to compute probabilities for every word in the vocabulary at the output layer.

🔹 **Optimization:**

- Instead of computing probabilities for all **V** words, we can use **Hierarchical Softmax**, which reduces this term from **H × V** to **H × $\log_2$(V)** (making it much faster).
- This speeds up training, but most of the computational burden still comes from **H × H**.

2.3 Parallel Training of Neural Networks

To train models on huge data sets, we have implemented several models on top of a large-scale distributed framework called DistBelief [6], including the feedforward NNLM and the new models proposed in this paper. The framework allows us to run multiple replicas of the same model in parallel, and each replica synchronizes its gradient updates through a centralized server that keeps all the parameters. For this parallel training, we use mini-batch asynchronous gradient descent with an adaptive learning rate procedure called Adagrad [7]. Under this framework, it is common to use one hundred or more model replicas, each using many CPU cores at different machines in a data center.

3 New Log-linear Models

In this section, we propose two new model architectures for learning distributed representations of words that try to minimize computational complexity. The main observation from the previous section was that most of the complexity is caused by the non-linear hidden layer in the model. While this is what makes neural networks so attractive, we decided to explore simpler models that might not be able to represent the data as precisely as neural networks, but can possibly be trained on much more data efficiently.

The new architectures directly follow those proposed in our earlier work [13, 14], where it was found that neural network language model can be successfully trained in two steps: first, continuous word vectors are learned using simple model, and then the N-gram NNLM is trained on top of these distributed representations of words. While there has been later substantial amount of work that focuses on learning word vectors, we consider the approach proposed in [13] to be the simplest one. Note that related models have been proposed also much earlier [26, 8].

The term **"simple model"** in this context refers to models that do **not** involve a non-linear hidden layer, unlike traditional neural network language models (NNLMs). Instead, these models use **log-linear architectures** to learn **continuous word vectors** efficiently. These simpler models include:

1. Skip-Gram Model (Word2Vec)

- Instead of predicting the **next word** given the previous words (as in NNLM), the **Skip-Gram** model tries to predict the **context words** given a target word.
- Uses a **single linear transformation** (no hidden layer), making it computationally efficient.
- Example: Given the word **"dog"**, predict its surrounding words like **"barks"**, **"pet"**, or **"loyal"**.

2. Continuous Bag-of-Words (CBOW)

- Predicts the **target word** given a **set of surrounding words**.
- Example: Given the words **"The __ is barking"**, predict **"dog"**.
- Similar to Skip-Gram but works in the opposite direction.

Why are these called "Simple Models"?

Unlike **NNLMs** that require a non-linear hidden layer (which increases computation time), these models:

- ✅ **Use only linear transformations** (faster).
- ✅ **Do not require backpropagation through a deep network**.
- ✅ **Scale well to large datasets** (like billions of words).

So, Do These Models Involve NN?

- 🔹 **Yes, but only a very basic one!** They use a **single-layer neural network** (log-linear model) with no hidden layers.
- 🔹 Instead of learning complex patterns via multiple layers, they **learn word embeddings** (continuous word vectors) efficiently by predicting word co-occurrences.

🚀 **Bottom Line:**

The **Skip-Gram and CBOW models** in **Word2Vec** are examples of **simple models** used to learn **continuous word vectors**. They drop the hidden layer for speed and efficiency, making them more scalable than traditional NNLMs.

3.1 Continuous Bag-of-Words Model

The first proposed architecture is similar to the feedforward NNLM, where the non-linear hidden layer is removed and the projection layer is shared for all words (not just the projection matrix); thus, all words get projected into the same position (their vectors are averaged). We call this architecture a bag-of-words model as the order of words in the history does not influence the projection. Furthermore, we also use words from the future; we have obtained the best performance on the task introduced in the next section by building a log-linear classifier with four future and four history words at the input, where the training criterion is to correctly classify the current (middle) word. Training complexity is then

$$Q = N \times D + D \times log_2(V). \tag{4}$$

We denote this model further as CBOW, as unlike standard bag-of-words model, it uses continuous distributed representation of the context. The model architecture is shown at Figure 1. Note that the weight matrix between the input and the projection layer is shared for all word positions in the same way as in the NNLM.

3.2 Continuous Skip-gram Model

The second architecture is similar to CBOW, but instead of predicting the current word based on the context, it tries to maximize classification of a word based on another word in the same sentence. More precisely, we use each current word as an input to a log-linear classifier with continuous projection layer, and predict words within a certain range before and after the current word. We found that increasing the range improves quality of the resulting word vectors, but it also increases the computational complexity. Since the more distant words are usually less related to the current word than those close to it, we give less weight to the distant words by sampling less from those words in our training examples.

The training complexity of this architecture is proportional to

$$Q = C \times (D + D \times log_2(V)), \tag{5}$$

where C is the maximum distance of the words. Thus, if we choose $C = 5$, for each training word we will select randomly a number R in range $< 1; C >$, and then use R words from history and

R words from the future of the current word as correct labels. This will require us to do $R \times 2$ word classifications, with the current word as input, and each of the $R + R$ words as output. In the following experiments, we use $C = 10$.

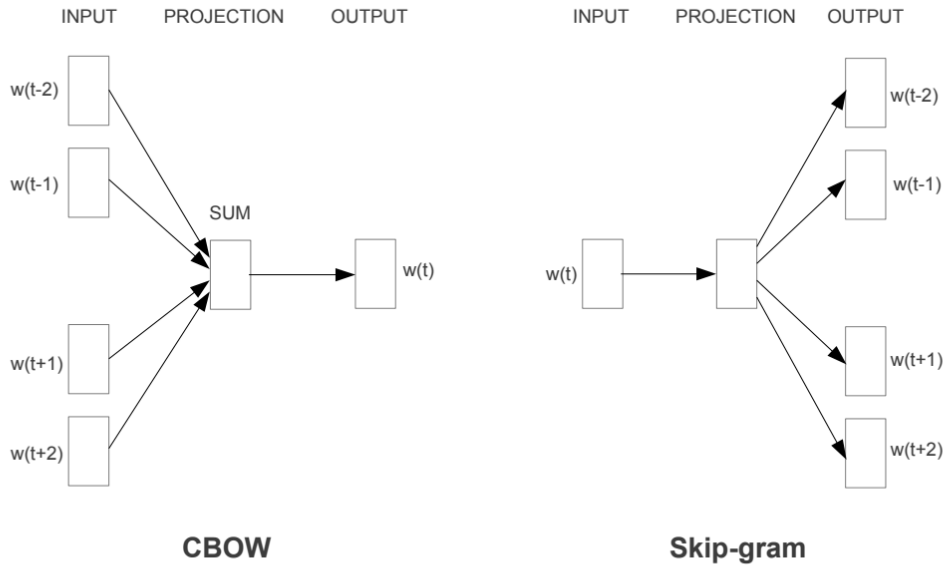

Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

Table 1: *Examples of five types of semantic and nine types of syntactic questions in the Semantic-Syntactic Word Relationship test set.*

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter
Adjective to adverb	apparent	apparently	rapid	rapidly
Opposite	possibly	impossibly	ethical	unethical
Comparative	great	greater	tough	tougher
Superlative	easy	easiest	lucky	luckiest
Present Participle	think	thinking	read	reading
Nationality adjective	Switzerland	Swiss	Cambodia	Cambodian
Past tense	walking	walked	swimming	swam
Plural nouns	mouse	mice	dollar	dollars
Plural verbs	work	works	speak	speaks

Table 2: *Accuracy on subset of the Semantic-Syntactic Word Relationship test set, using word vectors from the CBOW architecture with limited vocabulary. Only questions containing words from the most frequent 30k words are used.*

Dimensionality / Training words	24M	49M	98M	196M	391M	783M
50	13.4	15.7	18.6	19.1	22.5	23.2
100	19.4	23.1	27.8	28.7	33.4	32.2
300	23.2	29.2	35.3	38.6	43.7	45.9
600	24.0	30.1	36.5	40.8	46.6	50.4

Table 3: *Comparison of architectures using models trained on the same data, with 640-dimensional word vectors. The accuracies are reported on our Semantic-Syntactic Word Relationship test set, and on the syntactic relationship test set of [20]*

Model Architecture	Semantic-Syntactic Word Relationship test set		MSR Word Relatedness Test Set [20]
	Semantic Accuracy [%]	Syntactic Accuracy [%]	
RNNLM	9	36	35
NNLM	23	53	47
CBOW	24	64	61
Skip-gram	55	59	56

(such as 50 - 100). Given Equation 4, increasing amount of training data twice results in about the same increase of computational complexity as increasing vector size twice.

For the experiments reported in Tables 2 and 4, we used three training epochs with stochastic gradient descent and backpropagation. We chose starting learning rate 0.025 and decreased it linearly, so that it approaches zero at the end of the last training epoch.

4.3 Comparison of Model Architectures

First we compare different model architectures for deriving the word vectors using the same training data and using the same dimensionality of 640 of the word vectors. In the further experiments, we use full set of questions in the new Semantic-Syntactic Word Relationship test set, i.e. unrestricted to the 30k vocabulary. We also include results on a test set introduced in [20] that focuses on syntactic similarity between words[3].

The training data consists of several LDC corpora and is described in detail in [18] (320M words, 82K vocabulary). We used these data to provide a comparison to a previously trained recurrent neural network language model that took about 8 weeks to train on a single CPU. We trained a feed-forward NNLM with the same number of 640 hidden units using the DistBelief parallel training [6], using a history of 8 previous words (thus, the NNLM has more parameters than the RNNLM, as the projection layer has size 640×8).

In Table 3, it can be seen that the word vectors from the RNN (as used in [20]) perform well mostly on the syntactic questions. The NNLM vectors perform significantly better than the RNN - this is not surprising, as the word vectors in the RNNLM are directly connected to a non-linear hidden layer. The CBOW architecture works better than the NNLM on the syntactic tasks, and about the same on the semantic one. Finally, the Skip-gram architecture works slightly worse on the syntactic task than the CBOW model (but still better than the NNLM), and much better on the semantic part of the test than all the other models.

Next, we evaluated our models trained using one CPU only and compared the results against publicly available word vectors. The comparison is given in Table 4. The CBOW model was trained on subset

[3] We thank Geoff Zweig for providing us the test set.

Table 4: *Comparison of publicly available word vectors on the Semantic-Syntactic Word Relationship test set, and word vectors from our models. Full vocabularies are used.*

Model	Vector Dimensionality	Training words	Accuracy [%]		
			Semantic	Syntactic	Total
Collobert-Weston NNLM	50	660M	9.3	12.3	11.0
Turian NNLM	50	37M	1.4	2.6	2.1
Turian NNLM	200	37M	1.4	2.2	1.8
Mnih NNLM	50	37M	1.8	9.1	5.8
Mnih NNLM	100	37M	3.3	13.2	8.8
Mikolov RNNLM	80	320M	4.9	18.4	12.7
Mikolov RNNLM	640	320M	8.6	36.5	24.6
Huang NNLM	50	990M	13.3	11.6	12.3
Our NNLM	20	6B	12.9	26.4	20.3
Our NNLM	50	6B	27.9	55.8	43.2
Our NNLM	100	6B	34.2	**64.5**	50.8
CBOW	300	783M	15.5	53.1	36.1
Skip-gram	300	783M	**50.0**	55.9	**53.3**

Table 5: *Comparison of models trained for three epochs on the same data and models trained for one epoch. Accuracy is reported on the full Semantic-Syntactic data set.*

Model	Vector Dimensionality	Training words	Accuracy [%]			Training time [days]
			Semantic	Syntactic	Total	
3 epoch CBOW	300	783M	15.5	53.1	36.1	1
3 epoch Skip-gram	300	783M	50.0	55.9	53.3	3
1 epoch CBOW	300	783M	13.8	49.9	33.6	0.3
1 epoch CBOW	300	1.6B	16.1	52.6	36.1	0.6
1 epoch CBOW	600	783M	15.4	53.3	36.2	0.7
1 epoch Skip-gram	300	783M	45.6	52.2	49.2	1
1 epoch Skip-gram	300	1.6B	52.2	55.1	53.8	2
1 epoch Skip-gram	600	783M	56.7	54.5	55.5	2.5

of the Google News data in about a day, while training time for the Skip-gram model was about three days.

For experiments reported further, we used just one training epoch (again, we decrease the learning rate linearly so that it approaches zero at the end of training). Training a model on twice as much data using one epoch gives comparable or better results than iterating over the same data for three epochs, as is shown in Table 5, and provides additional small speedup.

4.4 Large Scale Parallel Training of Models

As mentioned earlier, we have implemented various models in a distributed framework called DistBelief. Below we report the results of several models trained on the Google News 6B data set, with mini-batch asynchronous gradient descent and the adaptive learning rate procedure called Adagrad [7]. We used 50 to 100 model replicas during the training. The number of CPU cores is an

Table 6: *Comparison of models trained using the DistBelief distributed framework. Note that training of NNLM with 1000-dimensional vectors would take too long to complete.*

Model	Vector Dimensionality	Training words	Accuracy [%]			Training time [days x CPU cores]
			Semantic	Syntactic	Total	
NNLM	100	6B	34.2	64.5	50.8	14 x 180
CBOW	1000	6B	57.3	68.9	63.7	2 x 140
Skip-gram	1000	6B	66.1	65.1	65.6	2.5 x 125

Table 7: *Comparison and combination of models on the Microsoft Sentence Completion Challenge.*

Architecture	Accuracy [%]
4-gram [32]	39
Average LSA similarity [32]	49
Log-bilinear model [24]	54.8
RNNLMs [19]	55.4
Skip-gram	48.0
Skip-gram + RNNLMs	**58.9**

estimate since the data center machines are shared with other production tasks, and the usage can fluctuate quite a bit. Note that due to the overhead of the distributed framework, the CPU usage of the CBOW model and the Skip-gram model are much closer to each other than their single-machine implementations. The result are reported in Table 6.

5 Examples of the Learned Relationships

Table 8 shows words that follow various relationships. We follow the approach described above: the relationship is defined by subtracting two word vectors, and the result is added to another word. Thus for example, *Paris* - *France* + *Italy* = *Rome*. As it can be seen, accuracy is quite good, although there is clearly a lot of room for further improvements (note that using our accuracy metric that

Table 8: *Examples of the word pair relationships, using the best word vectors from Table 4 (Skip-gram model trained on 783M words with 300 dimensionality).*

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

assumes exact match, the results in Table 8 would score only about 60%). We believe that word vectors trained on even larger data sets with larger dimensionality will perform significantly better, and will enable the development of new innovative applications. Another way to improve accuracy is to provide more than one example of the relationship. By using ten examples instead of one to form the relationship vector (we average the individual vectors together), we have observed improvement of accuracy of our best models by about 10% absolutely on the semantic-syntactic test.

It is also possible to apply the vector operations to solve different tasks. For example, we have observed good accuracy for selecting out-of-the-list words, by computing average vector for a list of words, and finding the most distant word vector. This is a popular type of problems in certain human intelligence tests. Clearly, there is still a lot of discoveries to be made using these techniques.

1. Intuition Behind CBOW and Skip-gram

Both CBOW and Skip-gram are based on a **shallow neural network** with a single hidden layer. They learn word representations by predicting words in a local context.

CBOW (Continuous Bag of Words)

- **Task**: Given a set of context words (surrounding words), predict the **target word** in the center.
- **Example**: If the sentence is *"The cat sat on the mat"*, and we take a window size of 2:
  - Context: `["The", "sat", "on", "the"]`
  - Target: `"cat"`
- **Intuition**: CBOW works like a **denoising autoencoder**, where multiple context words are compressed into a single representation to predict the missing target word.
- **Advantage**: Since CBOW aggregates multiple words into one representation, it is more stable and trains faster.

Skip-gram

- **Task**: Given a **target word**, predict the context words surrounding it.
- **Example**: Given the target word `"cat"`, predict `["The", "sat", "on", "the"]`.
- **Intuition**: Skip-gram is a **self-supervised classification task** where the model learns which words are likely to appear in the same context.
- **Advantage**: Since Skip-gram treats each word-context pair separately, it learns better representations, especially for rare words.

2. Why Are CBOW and Skip-gram Better Than NNLM and RNNLM?

Traditional NNLMs (Neural Network Language Models) and RNNLMs (Recurrent Neural Network Language Models) have significant drawbacks:

A. Traditional NNLM (Bengio et al., 2003)

- NNLMs use a **feedforward neural network** to predict the next word, given a fixed-size window of previous words.
- They require a **softmax layer** over a very large vocabulary, which makes training slow.
- Each word is represented as a **one-hot vector**, making them inefficient in capturing word relationships.

✅ **How Word2Vec Improves:**

- Instead of predicting the next word **sequentially**, Word2Vec learns embeddings **in parallel**.
- Uses **distributed representations** instead of one-hot encoding.
- Avoids the expensive **softmax over vocabulary**, leading to faster training.

B. RNNLM (Mikolov et al., 2010)

- Uses recurrent neural networks (RNNs) to model word sequences, which capture long-term dependencies.
- However, RNNs suffer from **vanishing gradient problems**, making it difficult to train deep networks.
- Training is computationally expensive because each word prediction depends on the previous steps.

✅ **How Word2Vec Improves:**

- Word2Vec **does not require sequential computation**, so training is much faster.
- Instead of relying on explicit sequence modeling like RNNs, it **learns word relationships purely from co-occurrence** in text.
- Can be trained using **negative sampling or hierarchical softmax**, making it scalable to large corpora.

3. Thinking About Such Solutions

Word2Vec was designed with efficiency and scalability in mind. Here's how to think about it:

- **Word meaning emerges from context**: Instead of modeling full sequences, we learn embeddings based on word co-occurrences.

- **Trade-off between speed and accuracy**: CBOW is faster (since it predicts one word per context), while Skip-gram is more accurate (since it learns from multiple word-context pairs).
- **Inspired by real-world learning**: Similar to how humans infer meaning from surrounding words, these models learn word relationships from context.

My Notes:

- What is the problem?

Main goal of the research paper: Improve the existing techniques for estimating the word vectors

- Key ideas in your own words.

Note from paper : we can increase the accuracy of word estimation by increasing dimensions as well as training words, also providing more example of similar words (10 examples) can increase the accuracy by 10%.

- What is the proposed solution?

Solution are two shallow neural network model architectures which make estimating the word vectors computationally efficient ast the same time having a greater accuracy then previous approaches

CBOW = Learn from many words to predict one word (faster, good for frequent words).

Skip-gram = Learn from one word to predict many words (better for rare words, more accurate embeddings).

- Why is it better than previous methods?

Previous models like RNNLM and NNLM were computational expensive and at the same time they represented words in one hot encoding way which is inefficient when it comes to capturing meanings for a word

- Questions or unclear points for further research. - NA

Mathematical Intuition Behind CBOW and Skip-gram

Both models learn **word embeddings** by maximizing the probability of a target word given its context (CBOW) or the probability of context words given a target word (Skip-gram).

1. CBOW Model (Continuous Bag of Words)

- **Objective:** Given a set of context words, predict the center (target) word.
- **Mathematical Formulation:**

$$P(w_t|w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}) = \frac{e^{v_{w_t}^T \cdot h}}{\sum_{w \in V} e^{v_w^T \cdot h}}$$

where:

- v_{w_t} is the embedding of the target word.
- $h = \frac{1}{N} \sum_{i=1}^{N} v_{w_i}$ is the average of context word embeddings.
- V is the vocabulary size.

2. Skip-gram Model

- **Objective:** Given a target word, predict its surrounding context words.
- **Mathematical Formulation:**

$$P(w_c|w_t) = \frac{e^{v_{w_c}^T \cdot v_{w_t}}}{\sum_{w \in V} e^{v_w^T \cdot v_{w_t}}}$$

  - w_c is a context word.
  - w_t is the target word.
  - v_{w_t} and v_{w_c} are embeddings of target and context words.

Loss Function: Negative Log-Likelihood

Both models use the **Negative Log-Likelihood (NLL)**:

$$\mathcal{L} = -\sum \log P(w_t|\text{context})$$

This encourages the model to assign **high probabilities** to correct predictions.

Why Not Use Simple Cross-Entropy?

The denominator in softmax (**sum over all words in the vocabulary**) makes it expensive for large vocabularies. Instead, we use:

1. **Negative Sampling** – Approximate softmax by only updating a few negative samples.
2. **Hierarchical Softmax** – Replace softmax with a binary tree structure.