Andriy Burkov

THE HUNDRED-PAGE MACHINE LEARNING BOOK

"All models are wrong, but some are useful."
— George Box

The book is distributed on the "read first, buy later" principle.

10 Other Forms of Learning

10.1 Metric Learning

I mentioned that the most frequently used metrics of similarity (or dissimilarity) between two feature vectors are **Euclidean distance** and **cosine similarity**. Such choices of metric seem logical but arbitrary, just like the choice of the squared error in linear regression (or the form of linear regression itself). The fact that one metric can work better than another depending on the dataset is an indicator that none of them are perfect.

You can *create* a metric that would work better for your dataset. It's then possible to integrate your metric into any learning algorithm that needs a metric, like k-means or kNN. How can you know, without trying all possibilities, which equation would be a good metric? As you could already guess, a metric can be learned from data.

Remember the Euclidean distance between two feature vectors $\mathbf{x}$ and $\mathbf{x}'$:

$$d(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\| \stackrel{\text{def}}{=} \sqrt{(\mathbf{x} - \mathbf{x}')^2} = \sqrt{(\mathbf{x} - \mathbf{x}')(\mathbf{x} - \mathbf{x}')}.$$

We can slightly modify this metric to make it parametrizable and then learn these parameters from data. Consider the following modification:

$$d_{\mathbf{A}}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_{\mathbf{A}} \stackrel{\text{def}}{=} \sqrt{(\mathbf{x} - \mathbf{x}')^\top \mathbf{A} (\mathbf{x} - \mathbf{x}')},$$

where $\mathbf{A}$ is a $D \times D$ matrix. Let's say $D = 3$. If we let $\mathbf{A}$ be the identity matrix,

$$\mathbf{A} \stackrel{\text{def}}{=} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

then $d_{\mathbf{A}}$ becomes the Euclidean distance. If we have a general diagonal matrix, like this:

$$\mathbf{A} \stackrel{\text{def}}{=} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

then different dimensions have different importance in the metric. (In the above example, the second dimension is the most important in the metric calculation.) More generally, to be called a metric a function of two variables has to satisfy three conditions:

1.	$d(\mathbf{x}, \mathbf{x}') \geq 0$	nonnegativity,
2.	$d(\mathbf{x}, \mathbf{x}') \leq d(\mathbf{x}, \mathbf{z}) + d(\mathbf{z}, \mathbf{x}')$	triangle inequality,
3.	$d(\mathbf{x}, \mathbf{x}') = d(\mathbf{x}', \mathbf{x})$	symmetry.

To satisfy the first two conditions, the matrix $\mathbf{A}$ has to be *positive semidefinite*. You can see a positive semidefinite matrix as the generalization of the notion of a nonnegative real number to matrices. Any positive semidefinite matrix $\mathbf{M}$ satisfies:

$$\mathbf{z}^\top \mathbf{M}\mathbf{z} \geq 0,$$

for any vector $\mathbf{z}$ having the same dimensionality as the number of rows and columns in $\mathbf{M}$.

The above property follows from the definition of a positive semidefinite matrix. The proof that the second condition is satisfied when the matrix $\mathbf{A}$ is positive semidefinite can be found on the book's companion website.

To satisfy the third condition, we can simply take $(d(\mathbf{x}, \mathbf{x}') + d(\mathbf{x}', \mathbf{x}))/2$.

Let's say we have an unannotated set $\mathcal{X} = \{\mathbf{x}_i\}_{i=1}^N$. To build the training data for our metric learning problem, we manually create two sets. The first set $\mathcal{S}$ is such that a pair of examples $(\mathbf{x}_i, \mathbf{x}_k)$ belongs to set $\mathcal{S}$ if $\mathbf{x}_i$ and $\mathbf{x}_k$ are similar (from our subjective perspective). The second set $\mathcal{D}$ is such that a pair of examples $(\mathbf{x}_i, \mathbf{x}_k)$ belongs to set $\mathcal{D}$ if $\mathbf{x}_i$ and $\mathbf{x}_k$ are dissimilar.

To train the matrix of parameters $\mathbf{A}$ from the data, we want to find a positive semidefinite matrix $\mathbf{A}$ that solves the following optimization problem:

$$\min_{\mathbf{A}} \sum_{(\mathbf{x}_i, \mathbf{x}_k) \in \mathcal{S}} \|\mathbf{x} - \mathbf{x}'\|^2_{\mathbf{A}} \text{ such that } \sum_{(\mathbf{x}_i, \mathbf{x}_k) \in \mathcal{D}} \|\mathbf{x} - \mathbf{x}'\|_{\mathbf{A}} \geq c,$$

where c is a positive constant (can be any number).

The solution to this optimization problem is found by gradient descent with a modification that ensures that the found matrix $\mathbf{A}$ is positive semidefinite. We leave the description of the algorithm out of the scope of this book for further reading.

I should point out that **one-shot learning** with **siamese networks** and **triplet loss** can be seen as metric learning problem: the pairs of pictures of the same person belong to the set $\mathcal{S}$, while pairs of random pictures belong to $\mathcal{D}$.

There are many other ways to learn a metric, including non-linear and kernel-based. However, the one presented in this book, as well as the adaptation of one-shot learning should suffice for most practical applications.

10.2 Learning to Rank

Learning to rank is a supervised learning problem. Among others, one frequent problem solved using learning to rank is the optimization of search results returned by a search engine for a query. In search result ranking optimization, a labeled example $\mathcal{X}_i$ in the training set of size N is a ranked collection of documents of size r_i (labels are ranks of documents). A feature vector represents each document in the collection. The goal of the learning is to find a ranking function f which outputs values that can be used to rank documents. For each training example, an ideal function f would output values that induce the same ranking of documents as given by the labels.

Each example $\mathcal{X}_i$, $i = 1, \ldots, N$, is a collection of feature vectors with labels: $\mathcal{X}_i = \{(\mathbf{x}_{i,j}, y_{i,j})\}_{j=1}^{r_i}$. Features in a feature vector $\mathbf{x}_{i,j}$ represent the document $j = 1, \ldots, r_i$. For example, $x_{i,j}^{(1)}$ could represent how recent is the document, $x_{i,j}^{(2)}$ would reflect whether the words of the query can be found in the document title, $x_{i,j}^{(3)}$ could represent the size of the document, and so on. The label $y_{i,j}$ could be the rank $(1, 2, \ldots, r_i)$ or a score. For example, the lower the score, the higher the document should be ranked.

There are three approaches to solve that problem: **pointwise**, **pairwise**, and **listwise**.

The pointwise approach transforms each training example into multiple examples: one example per document. The learning problem becomes a standard supervised learning problem, either regression or logistic regression. In each example $(\mathbf{x}, y)$ of the pointwise learning problem, $\mathbf{x}$ is the feature vector of some document, and y is the original score (if $y_{i,j}$ is a score) or a synthetic score obtained from the ranking (the higher the rank, the lower is the synthetic score). Any supervised learning algorithm can be used in this case. The solution is usually far from perfect. Principally, this is because each document is considered in isolation, while the original ranking (given by the labels $y_{i,j}$ of the original training set) could optimize the positions of the whole set of documents. For example, if we have already given a high rank to a Wikipedia page in some collection of documents, we would prefer not giving a high rank to another Wikipedia page for the same query.

In the pairwise approach, we also consider documents in isolation, however, in this case, a pair of documents is considered at once. Given a pair of documents $(\mathbf{x}_i, \mathbf{x}_k)$ we build a model f, which, given $(\mathbf{x}_i, \mathbf{x}_k)$ as input, outputs a value close to 1, if $\mathbf{x}_i$ should be higher than $\mathbf{x}_k$ in the ranking; otherwise, f outputs a value close to 0. At the test time, the final ranking for an unlabeled example $\mathcal{X}$ is obtained by aggregating the predictions for all pairs of documents in $\mathcal{X}$. The pairwise approach works better than pointwise, but still far from perfect.

The state of the art rank learning algorithms, such as **LambdaMART**, implement the listwise approach. In the listwise approach, we try to optimize the model directly on some metric that reflects the quality of ranking. There are various metrics for assessing search engine result ranking, including precision and recall. One popular metric that combines both precision and recall is called **mean average precision** (MAP).

To define MAP, let us ask judges (Google call those people *rankers*) to examine a collection of search results for a query and assign relevancy labels to each search result. Labels could be binary (1 for "relevant" and 0 for "irrelevant") or on some scale, say from 1 to 5: the higher the value, the more relevant the document is to the search query. Let our judges build such relevancy labeling for a collection of 100 queries. Now, let us test our ranking model on this collection. The **precision** of our model for some query is given by:

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|},$$

where the notation $|\cdot|$ means "the number of." The **average precision** metric, AveP, is defined for a ranked collection of documents returned by a search engine for a query q as,

$$\text{AveP}(q) = \frac{\sum_{k=1}^{n}(P(k) \cdot \text{rel}(k))}{|\{\text{relevant documents}\}|},$$

where n is the number of retrieved documents, $P(k)$ denotes the precision computed for the top k search results returned by our ranking model for the query, rel(k) is an indicator function equaling 1 if the item at rank k is a relevant document (according to judges) and zero otherwise. Finally, the MAP for a collection of search queries of size Q is given by,

$$\text{MAP} = \frac{\sum_{q=1}^{Q} \text{AveP(q)}}{Q}.$$

Now we get back to LambdaMART. This algorithm implements a listwise approach, and it uses gradient boosting to train the ranking function $h(\mathbf{x})$. Then the binary model $f(\mathbf{x}_i, \mathbf{x}_k)$ that predicts whether the document $\mathbf{x}_i$ should have a higher rank than the document $\mathbf{x}_k$ (for the same search query) is given by a sigmoid with a hyperparameter α,

$$f(\mathbf{x}_i, \mathbf{x}_k) \stackrel{\text{def}}{=} \frac{1}{1 + \exp((h(\mathbf{x_i}) - h(\mathbf{x_k}))\alpha}.$$

Again, as with many models that predict probability, the cost function is cross-entropy computed using the model f. In our gradient boosting, we combine multiple regression trees to build the function h by trying to minimize the cost. Remember that in gradient boosting we add a tree to the model to reduce the error that the current model makes on the training data. For the classification problem, we computed the derivative of the cost function to replace real labels of training examples with these derivatives. LambdaMART works similarly, with one exception. It replaces the real gradient with a combination of the gradient and another factor that depends on the metric, such as MAP. This factor modifies the original gradient by increasing or decreasing it so that the metric value is improved.

That is a very bright idea and not many supervised learning algorithms can boast that they optimize a metric directly. Optimizing a metric is what we really want, but what we do in a typical supervised learning algorithm is we optimize the cost instead of the metric (we do that because metrics are usually not differentiable). Usually, in supervised learning, as soon as we have found a model that optimizes the cost function, we try to tweak hyperparameters to improve the value of the metric. LambdaMART optimizes the metric directly.

The remaining question is how do we build the ranked list of results based on the predictions of the model f which predicts whether its first input has to be ranked higher than the second input. It's generally a computationally hard problem, and there are multiple implementations of rankers capable of transforming pairwise comparisons into a ranking list.

The most straightforward approach is to use an existing sorting algorithm. Sorting algorithms sort a collection of numbers in increasing or decreasing order. (The simplest sorting algorithm is called *bubble sort*. It's usually taught in engineering schools.) Typically, sorting algorithms iteratively compare a pair of numbers in the collection and change their positions in the list based on the result of that comparison. If we plug our function f into a sorting algorithm to execute this comparison, the sorting algorithm will sort documents and not numbers.

10.3 Learning to Recommend

Learning to recommend is an approach to build recommender systems. Usually, we have a user who consumes content. We have the history of consumption and want to suggest this user new content that they would like. It could be a movie on Netflix or a book on Amazon.

Traditionally, two approaches were used to give recommendations: **content-based filtering** and **collaborative filtering**.

Content-based filtering consists of learning what users like based on the description of the content they consume. For example, if the user of a news site often reads news articles on science and technology, then we would suggest to this user more documents on science and technology. More generally, we could create one training set *per user* and add news articles to this dataset as a feature vector $\mathbf{x}$ and whether the user recently read this news article as a label y. Then we build the model of each user and can regularly examine each new piece of content to determine whether a specific user would read it or not.

The content-based approach has many limitations. For example, the user can be trapped in the so-called filter bubble: the system will always suggest to that user the information that looks very similar to what user already consumed. That could result in complete isolation of the user from information that disagrees with their viewpoints or expands them. On a more practical side, the users might just stop following recommendations, which is undesirable.

Collaborative filtering has a significant advantage over content-based filtering: the recommendations to one user are computed based on what other users consume or rate. For instance, if two users gave high ratings to the same ten movies, then it's more likely that user 1 will appreciate new movies recommended based on the tastes of the user 2 and vice versa. The drawback of this approach is that the content of the recommended items is ignored.

In collaborative filtering, the information on user preferences is organized in a matrix. Each row corresponds to a user, and each column corresponds to a piece of content that user rated or consumed. Usually, this matrix is huge and extremely sparse, which means that most of its cells aren't filled (or filled with a zero). The reason for such a sparsity is that most users consume or rate just a tiny fraction of available content items. It's is very hard to make meaningful recommendations based on such sparse data.

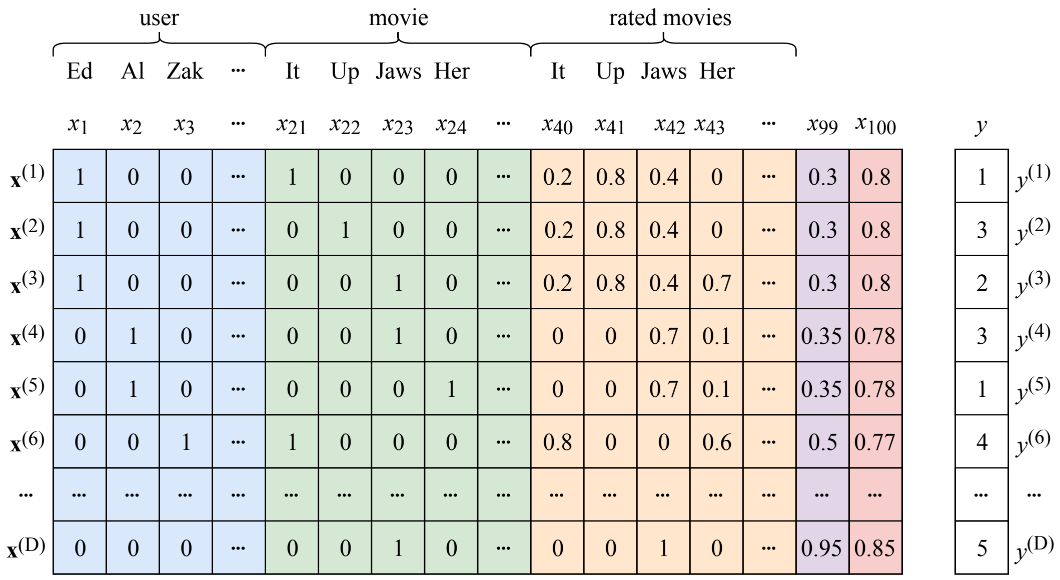
Most real-world recommender systems use a hybrid approach: they combine recommendations obtained by the content-based and collaborative filtering models.

I already mentioned that content-based recommender model could be built using a classification or regression model that predicts whether a user will like the content based on the content's features. Examples of features could include the words in books or news articles the user liked, the price, the recency of the content, the identity of the content author and so on.

Two effective recommender system learning algorithms are **factorization machines** (FM) and **denoising autoencoders** (DAE).

10.3.1 Factorization Machines

Factorization machines is a relatively new kind of algorithm. It was explicitly designed for sparse datasets. Let's illustrate the problem.



	user				movie					rated movies								
	Ed	Al	Zak	···	It	Up	Jaws	Her		It	Up	Jaws	Her					
	x_1	x_2	x_3	···	x_{21}	x_{22}	x_{23}	x_{24}	···	x_{40}	x_{41}	x_{42}	x_{43}	···	x_{99}	x_{100}	y	
$\mathbf{x}^{(1)}$	1	0	0	···	1	0	0	0	···	0.2	0.8	0.4	0	···	0.3	0.8	1	$y^{(1)}$
$\mathbf{x}^{(2)}$	1	0	0	···	0	1	0	0	···	0.2	0.8	0.4	0	···	0.3	0.8	3	$y^{(2)}$
$\mathbf{x}^{(3)}$	1	0	0	···	0	0	1	0	···	0.2	0.8	0.4	0.7	···	0.3	0.8	2	$y^{(3)}$
$\mathbf{x}^{(4)}$	0	1	0	···	0	0	1	0	···	0	0	0.7	0.1	···	0.35	0.78	3	$y^{(4)}$
$\mathbf{x}^{(5)}$	0	1	0	···	0	0	0	1	···	0	0	0.7	0.1	···	0.35	0.78	1	$y^{(5)}$
$\mathbf{x}^{(6)}$	0	0	1	···	1	0	0	0	···	0.8	0	0	0.6	···	0.5	0.77	4	$y^{(6)}$
···	···	···	···	···	···	···	···	···	···	···	···	···	···	···	···	···	···	···
$\mathbf{x}^{(D)}$	0	0	0	···	0	0	1	0	···	0	0	1	0	···	0.95	0.85	5	$y^{(D)}$

Figure 1: Example for sparse feature vectors $\mathbf{x}$ and their respective labels y.

In Figure 1 you see an example of sparse feature vectors with labels. Each feature vector represents information about one specific user and one specific movie. Features in the blue section represent a user. Users are encoded as one-hot vectors. Features in the green section represent a movie. Movies are also encoded as one-hot vectors. Features in the yellow section represent scores the user in blue gave to each movie they rated. Feature x_{99} represents the ratio of movies with an Oscar among those the user has watched. Feature x_{100} represents the percentage of the movie watched by the user in blue before they scored the movie in green. The target y represents the score given by the user in blue to the movie in green.

In real recommender systems, the number of users can count in millions, so the matrix in Figure 1 would count hundreds of millions of rows. The number of features could be hundreds of thousands, depending on how rich is the choice of content and how creative you, as a data analyst, are in feature engineering. Features x_{99} and x_{100} were handcrafted during the feature engineering process, and I only show two features for the purposes of illustration.

Trying to fit a regression or classification model to such an extremely sparse dataset would result in poor generalization. Factorization machines approach this problem differently.

The factorization machine model is defined as follows:

$$f(\mathbf{x}) \stackrel{\text{def}}{=} b + \sum_{i=1}^{D} w_i x_i + \sum_{i=1}^{D} \sum_{j=i+1}^{D} (\mathbf{v}_i \mathbf{v}_j) x_i x_j,$$

where b and w_i, $i = 1, \ldots, D$, are scalar parameters similar to those used in linear regression. Vectors $\mathbf{v}_i$ are k-dimensional vectors of **factors**. k is a hyperparameter and is usually much smaller than D. The expression $\mathbf{v}_i \mathbf{v}_j$ is a dot-product of the i^{th} and j^{th} vectors of factors. As you can see, instead looking for one wide vector of parameters, which can reflect poorly interactions between features because of sparsity, we complete it by additional parameters that apply to pairwise interactions $x_i x_j$ between features. However, instead of having a parameter $w_{i,j}$ for each interaction, which would add an enormous[1] quantity of new parameters to the model, we factorize $w_{i,j}$ into $\mathbf{v}_i \mathbf{v}_j$ by adding only $Dk \ll D(D-1)$ parameters to the model[2].

Depending on the problem, the loss function could be squared error loss (for regression) or hinge loss. For classification with $y \in \{-1, +1\}$, with hinge loss or logistic loss the prediction is made as $y = \text{sign}(f(x))$. The logistic loss is defined as,

$$loss(f(\mathbf{x}), y) = \frac{1}{\ln 2} \ln(1 + e^{-yf(\mathbf{x})}).$$

Gradient descent can be used to optimize the average loss. In the example in Figure 1, the labels are in $\{1, 2, 3, 4, 5\}$, so it's a multiclass problem. We can use **one versus rest** strategy to convert this multiclass problem into five binary classification problems.

[1]To be more precise we would add $D(D-1)$ parameters $w_{i,j}$.

[2]The notation $\ll$ means "much less than."

10.3.2 Denoising Autoencoders

From Chapter 7, you know what a denoising autoencoder is: it's a neural network that reconstructs its input from the bottleneck layer. The fact that the input is corrupted by noise while the output shouldn't be, makes denoising autoencoders an ideal tool to build a recommender model.

The idea is very straightforward: new movies a user could like are seen as if they were removed from the complete set of preferred movies by some corruption process. The goal of the denoising autoencoder is to reconstruct those removed items.

To prepare the training set for our denoising autoencoder, remove the blue and green features from the training set in Figure 1. Because now some examples become duplicates, keep only the unique ones.

At the training time, randomly replace some of the non-zero yellow features in the input feature vectors with zeros. Train the autoencoder to reconstruct the uncorrupted input.

At the prediction time, build a feature vector for the user. The feature vector will include uncorrupted yellow features as well as the handcrafted features like x_{99} and x_{100}. Use the trained DAE model to reconstruct the uncorrupted input. Recommend to the user movies that have the highest scores at the model's output.

Another effective collaborative-filtering model is an FFNN with two inputs and one output. Remember from Chapter 8 that neural networks are good at handling multiple simultaneous inputs. A training example here is a triplet $(\mathbf{u}, \mathbf{m}, r)$. The input vector $\mathbf{u}$ is a **one-hot encoding** of a user. The second input vector $\mathbf{m}$ is a one-hot encoding of a movie. The output layer could be either a sigmoid (in which case the label r is in $[0, 1]$) or ReLU, in which case r can be in some typical range, $[1, 5]$ for example.

10.4 Self-Supervised Learning: Word Embeddings

We have already discussed word embeddings in Chapter 7. Recall that **word embeddings** are feature vectors that represent words. They have the property that similar words have similar feature vectors. The question that you probably wanted to ask is where these word embeddings come from. The answer is (again): they are learned from data.

There are many algorithms to learn word embeddings. Here, we consider only one of them: **word2vec**, and only one version of word2vec called **skip-gram**, which works well in practice. Pretrained word2vec embeddings for many languages are available to download online.

In word embedding learning, our goal is to build a model which we can use to convert a one-hot encoding of a word into a word embedding. Let our dictionary contain 10,000 words.

The one-hot vector for each word is a 10,000-dimensional vector of all zeroes except for one dimension that contains a 1. Different words have a 1 in different dimensions.

Consider a sentence: "I almost finished reading the book on machine learning." Now, consider the same sentence from which we have removed one word, say "book." Our sentence becomes: "I almost finished reading the $\cdot$ on machine learning." Now let's only keep the three words before the $\cdot$ and three words after: "finished reading the $\cdot$ on machine learning." Looking at this seven-word window around the $\cdot$, if I ask you to guess what $\cdot$ stands for, you would probably say: "book," "article," or "paper." That's how the context words let you predict the word they surround. It's also how the machine can learn that words "book," "paper," and "article" have a similar meaning: because they share similar contexts in multiple texts.

It turns out that it works the other way around too: a word can predict the context that surrounds it. The piece "finished reading the $\cdot$ on machine learning" is called a skip-gram with window size 7 ($3 + 1 + 3$). By using the documents available on the Web, we can easily create hundreds of millions of skip-grams.

Let's denote a skip-gram like this: $[\mathbf{x}_{-3}, \mathbf{x}_{-2}, \mathbf{x}_{-1}, \mathbf{x}, \mathbf{x}_{+1}, \mathbf{x}_{+2}, \mathbf{x}_{+3}]$. In our sentence, $\mathbf{x}_{-3}$ is the one-hot vector for "finished," $\mathbf{x}_{-2}$ corresponds to "reading," $\mathbf{x}$ is the skipped word $(\cdot)$, $\mathbf{x}_{+1}$ is "on" and so on. A skip-gram with window size 5 will look like this: $[\mathbf{x}_{-2}, \mathbf{x}_{-1}, \mathbf{x}, \mathbf{x}_{+1}, \mathbf{x}_{+2}]$.

The skip-gram model with window size 5 is schematically depicted in Figure 2. It is a fully-connected network, like the multilayer perceptron. The input word is the one denoted as $\cdot$ in the skip-gram. The neural network has to learn to predict the context words of the skip-gram given the central word.

You can see now why the learning of this kind is called **self-supervised**: the labeled examples get extracted from the unlabeled data such as text.

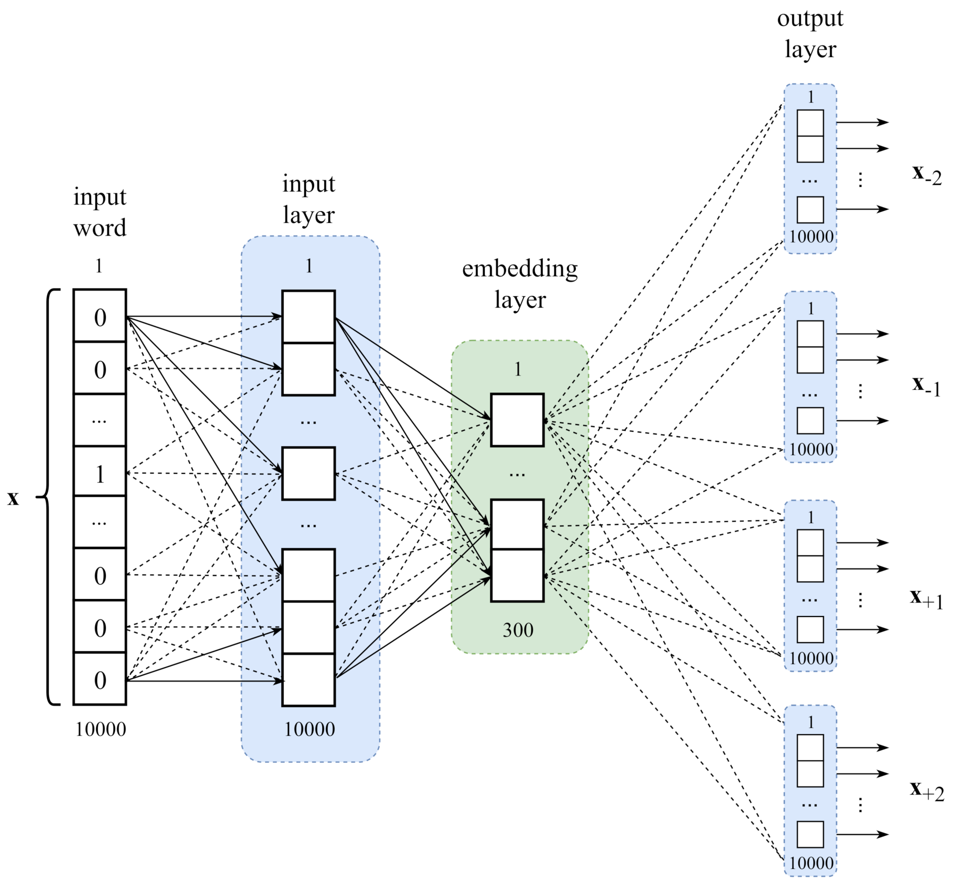


Figure 2: The skip-gram model with window size 5 and the embedding layer of 300 units.

The activation function used in the output layer is softmax. The cost function is the negative log-likelihood. The embedding for a word is obtained as the output of the embedding layer when the one-hot encoding of this word is given as the input to the model.

Because of the large number of parameters in the word2vec models, two techniques are used to make the computation more efficient: *hierarchical softmax* (an efficient way of computing softmax that consists in representing the outputs of softmax as leaves of a binary tree) and *negative sampling* (the idea is only to update a random sample of all outputs per iteration of gradient descent). I leave these for further reading.