National University of Computer and Emerging Sciences

Natural Language Processing

“Assignment 4 & 5”

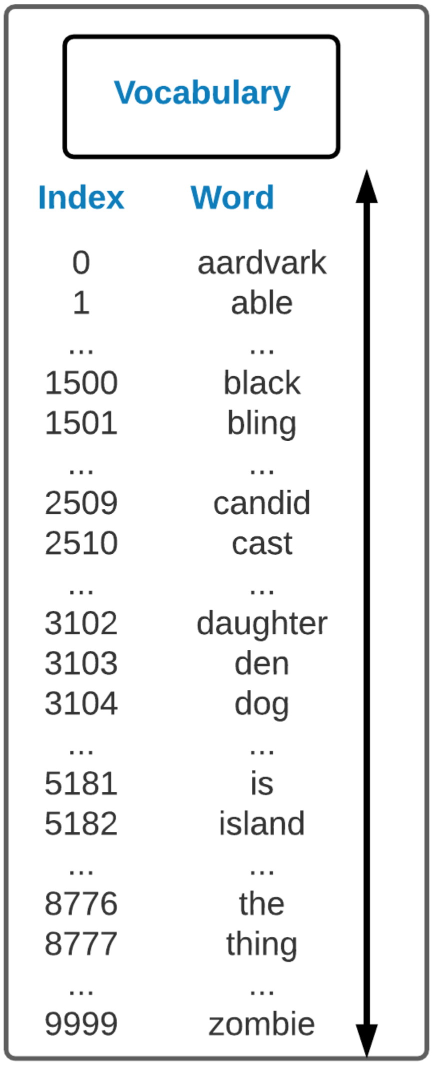
STUDENTS NAME:	Usman Ali
ROLL NUMBERS:	16i-0164
DEGREE PROGRAM:	BS(CS)
SEMESTER:	Spring 2021
SUBJECT NAME:	NLP
DATE OF SUBMISSION:	8th June, 2021
SUBMITTED TO:	Sir Muhammad Bin Arif
SECTION:	B

Contents

1. Task 1

1.1. The Skip-gram Algorithm

One of the best ways to analyze words for machine learning is to convert the data into a vector format. This is called word embeddings. Given the vocabulary we can represent each word numerically. Let’s assume we have a vocabulary of 10,000 words.



Vocabulary

Index	Word
0	aardvark
1	able
...	...
1500	black
1501	bling
...	...
2509	candid
2510	cast
...	...
3102	daughter
3103	den
3104	dog
...	...
5181	is
5182	island
...	...
8776	the
8777	thing
...	...
9999	zombie

Figure 1.1: Word embeddings

This process allows us to represent a word as a vector of numbers. This is one hot vector encoding. Meaning that each word’s representation will mostly be 0’s.

There will be only ‘1’ entry in the position corresponding to the word’s index in the vocabulary.

The vector for aardvark will be *[1,0,0, … , 0]*. Which implies that there will be 9999 zeros after the first ‘1’ for aardvark. Similarly, The one hot vector encoding for ‘zombie’ in this example, will be *[0, … , 0, 0, 0 ,1]*.

This is used to generate word representations. Let’s understand the end goal before delving too deep into how its done. Our goal is to get the computer to understand the likeliness of using two words together in context, and finding out what other similar words exist in the corpus. Word representation represents the word in vector space so that if the word vectors are close to one another means that those words are related to one other. In the given image, we can see various words which are mostly seen with a female are clustered on the left side while the words associated with males are clustered at the right side. So, if we pass the word like earrings the computer will relate it to the female gender which is logically correct.

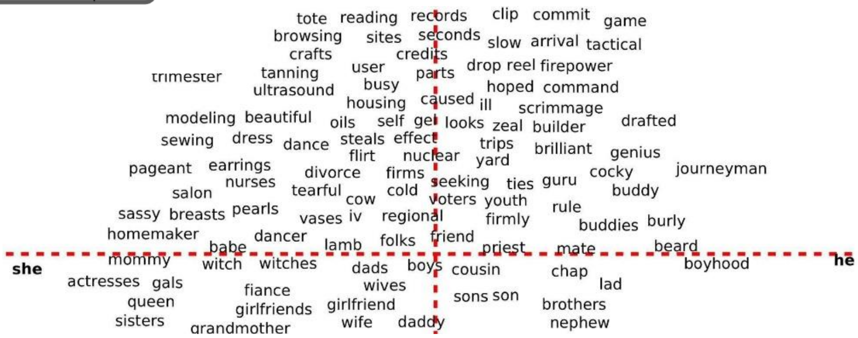


Figure 1.2: Word representations

Let’s focus on the one hot vector embeddings to understand how they help us classify in the following example. Ideally, we would want similar words like “toxin” and “venom” to have somewhat similar features. But with one hot vectors, “toxin” is as similar to “venom” as literally any other word, which isn’t great

1.1.1. The vocabulary size

With this approach, as you increase your vocabulary by n, your feature size vectors also increase by length n. One hot vector dimensionality is the same as number of words

1.1.2. The computational

Each word’s embedding/feature vector is mostly zeros, and many machine learning models won’t work well with very high dimensional and sparse features.

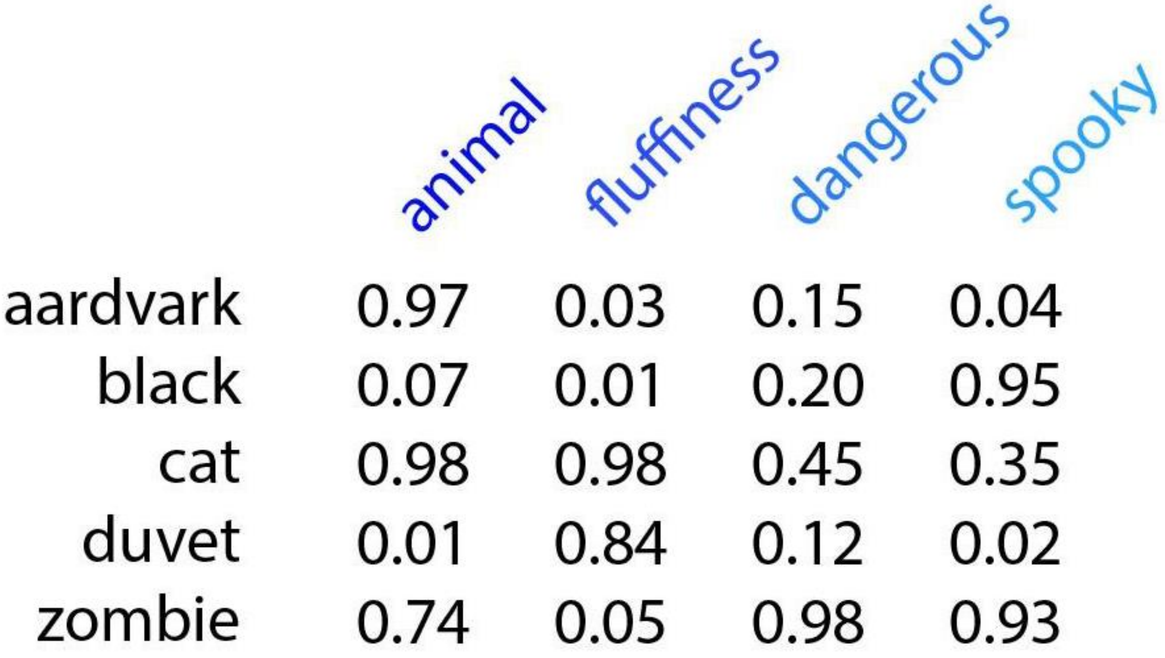
1.1.3. The Generalization

The core problem that embeddings solve is Generalization. If we assume that words like “toxin” and “venom” are indeed similar, we want some way to pass that information on to the model.

Increasing the number of dimensions reduces the generalization issue. As more context is given better suited similarities can be formed.

Increasing the window size helps with increasing accuracy but a large window size also reduces accuracy as different contexts are mixed up.

Using this we can generate the following example to completely understand why one hot vectors work. Let’s look at the result and then focus on how we achieved it. Clearly, we can see the relation of the words was correctly established.



	animal	fluffiness	dangerous	spooky
aardvark	0.97	0.03	0.15	0.04
black	0.07	0.01	0.20	0.95
cat	0.98	0.98	0.45	0.35
duvet	0.01	0.84	0.12	0.02
zombie	0.74	0.05	0.98	0.93

Figure 1.3: Similarity

The easiest way to go about this is to just look at embeddings as a fully connected layer. We will call this layer as embedding layer and the weights as embedding weights.

Now, instead of doing the matrix multiplication between the inputs and hidden layer we directly grab the values from embedding weight matrix. We can do this because the multiplication of one hot vector with weight matrix returns the row of the matrix corresponding to the index of ‘1’ input unit

$$\underset{\text{One-hot vector}}{\begin{bmatrix}0 & 0 & 0 & 1 & 0\end{bmatrix}} \times \underset{\text{Embedding Weight Matrix}}{\begin{bmatrix}8 & 2 & 1 & 9\\ 6 & 5 & 4 & 0\\ 7 & 1 & 6 & 2\\ 1 & 3 & 5 & 8\\ 0 & 4 & 9 & 1\end{bmatrix}} = \underset{\text{Hidden layer output}}{\begin{bmatrix}1 & 3 & 5 & 8\end{bmatrix}}$$

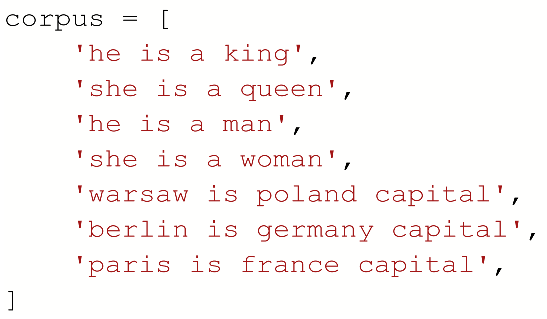
Figure 1.4: Embedding Weight Matrix

In conclusion, we can say that

1.1.4. The embedding layer is just a hidden layer
1.1.5. The lookup table is just an embedding weight matrix
1.1.6. The lookup is just a shortcut for matrix multiplication
1.1.7. The lookup table is trained just like any weight matrix

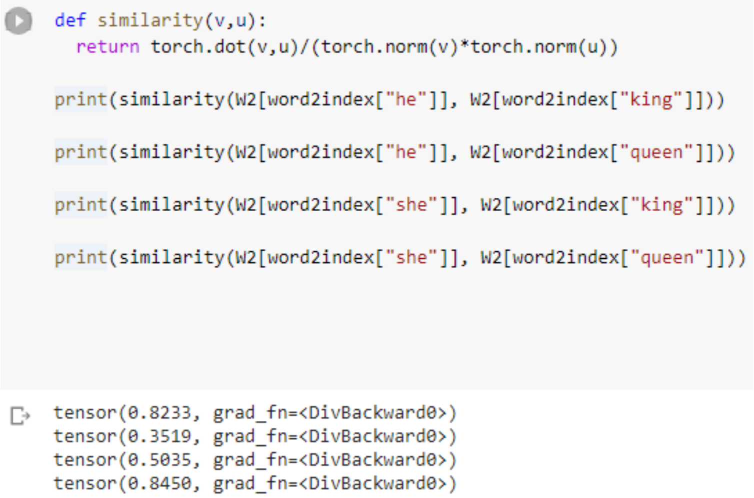
To demonstrate it on a personalized example.

Let us assume the following corpus.



```
corpus = [
    'he is a king',
    'she is a queen',
    'he is a man',
    'she is a woman',
    'warsaw is poland capital',
    'berlin is germany capital',
    'paris is france capital',
]
```

Figure 1.5: Corpus



```
def similarity(v,u):
  return torch.dot(v,u)/(torch.norm(v)*torch.norm(u))

print(similarity(W2[word2index["he"]], W2[word2index["king"]]))

print(similarity(W2[word2index["he"]], W2[word2index["queen"]]))

print(similarity(W2[word2index["she"]], W2[word2index["king"]]))

print(similarity(W2[word2index["she"]], W2[word2index["queen"]]))
```

```
tensor(0.8233, grad_fn=<DivBackward0>)
tensor(0.3519, grad_fn=<DivBackward0>)
tensor(0.5035, grad_fn=<DivBackward0>)
tensor(0.8450, grad_fn=<DivBackward0>)
```

Figure 1.6: Similarity example from corpus

Visualizing this in a table form clearly shows that by introducing pairs even on a very small corpus we can establish the relation of which is more likely to occur with its given pair. This is essentially how the algorithm works.

	King	**Queen**
She	.5035	.8450
He	.8233	.3519

Table 1.1: Probabilities of the relation

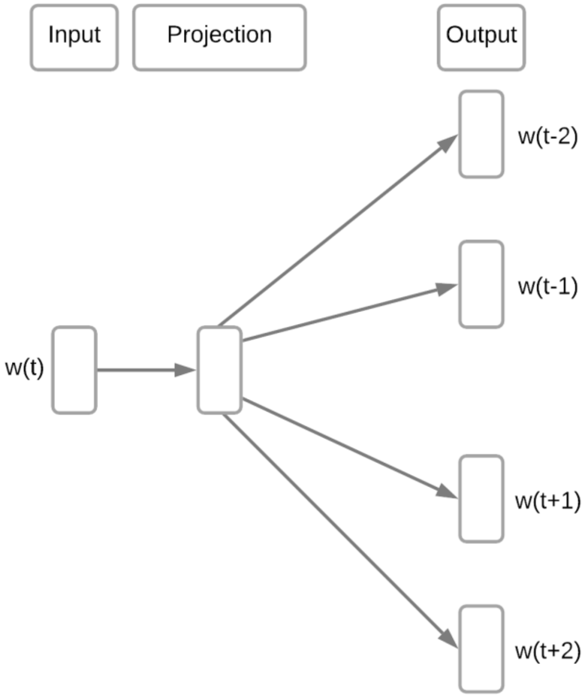


Figure 1.7: Skip-gram working

We input one word to predict a target context using the similarity between them. As we can see w(t) is the target word or input given. There is one hidden layer which performs the dot product between the weight matrix and the input vector w(t). No activation function is used in the hidden layer. Now the result of the dot product at the hidden layer is passed to the output layer. Output layer computes the dot product between the output vector of the hidden layer and the weight matrix of the output layer. Then we apply the 'Softmax' activation function to compute the probability of words appearing to be in the context of w(t) at given context location.

1.2. Negative Sampling

Training a neural network means adjusting and tuning all neuron wights so that it predicts the training sample more accurately. When training the network on the word pair (“he”, “king”), recall that the “label” or “correct output” of the network is a one-hot vector. That is, for the output neuron corresponding to “king” to output a 1, and for all of the other output neurons to output a 0.

With negative sampling, we are instead going to randomly select just a small number of “negative” words to update the weights for. In this context, a “negative” word is one for which we want the network to output a 0 for. We will also still update the weights for our “positive” word (which is the word “king” in our example).

For instance, lets suppose our entire training corpus as a list of words, and by randomly choosing 3 negative samples from the list. In this case, the probability for picking the word “chip” would be equal to the number of times “chip” appears in the corpus, divided the total number of words in the corpus. This is expressed by the following equation:

$$P(wi) = \frac{f(wi)}{\sum_{j=0}^{n}(f(wj))}$$

1.3. Implementation Using Skip-gram With Negative Sampling and Cosine Similarity

```
pip install d2l==0.16.5
```

Installing the dataset and its packages

```
import torch
from torch import nn
from d2l import torch as d2l
```

The d2l dataset typically offers a large of word embeddings which works a lot in our benefit. This dataset is not labelled so we can show that the power of unsupervised learning.

```
batch_size, max_window_size, num_noise_words = 512, 5, 5
data_iter, vocab = d2l.load_data_ptb(batch_size, max_window_size, num_noise_words)
```

Setting the max window size and max negative sampling words.

```
class SigmoidBinaryCrossEntropyLoss(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, inputs, target, mask=None):
        out = nn.functional.binary_cross_entropy_with_logits(
            inputs, target, weight=mask, reduction="none")
        return out.mean(dim=1)

loss = SigmoidBinaryCrossEntropyLoss()
```

Setting the loss function

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = torch.bmm(v, u.permute(0, 2, 1))
    return pred
```

The probabilistic skip-gram function

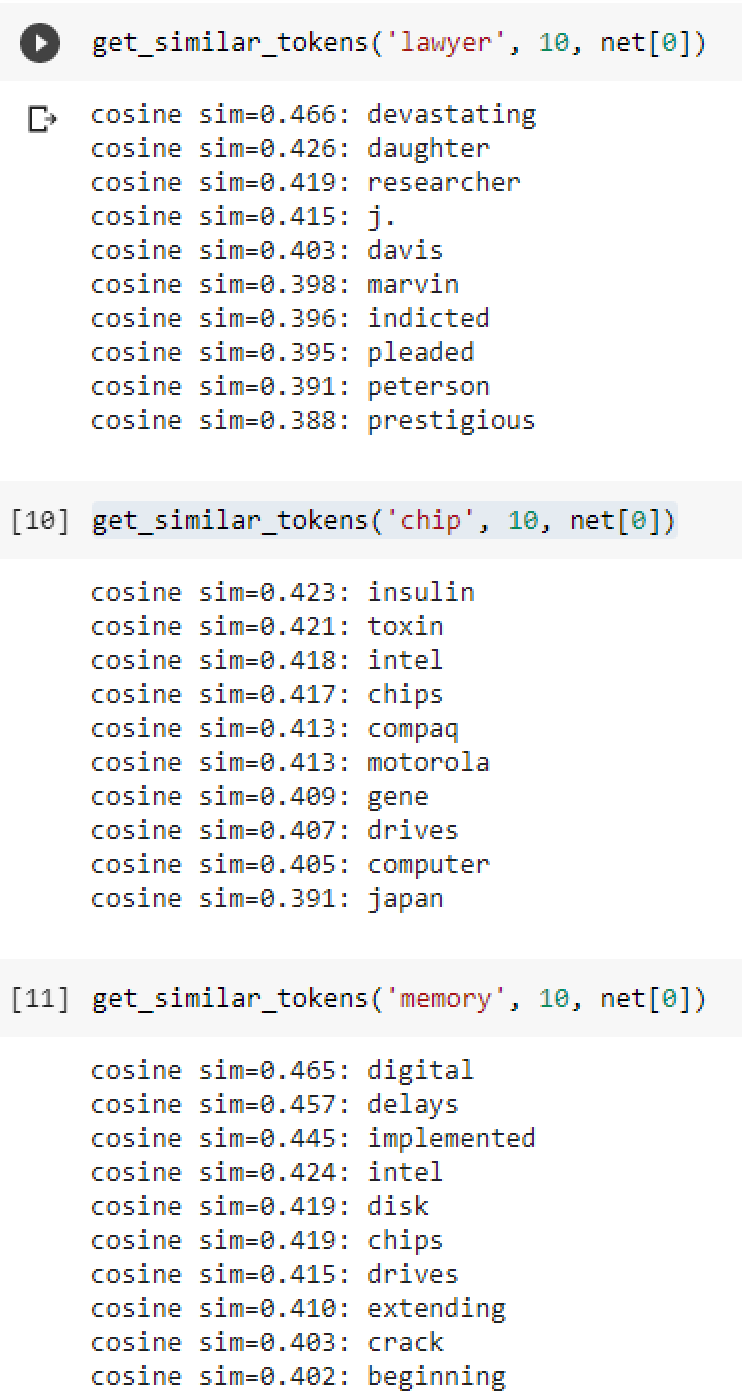
```
def train(net, data_iter, lr, num_epochs, device=d2l.try_gpu()):
    def init_weights(m):
        if type(m) == nn.Embedding:
            nn.init.xavier_uniform_(m.weight)

    net.apply(init_weights)
    net = net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    animator = d2l.Animator(xlabel='epoch',
 ylabel='loss', xlim=[1, num_epochs])
    metric = d2l.Accumulator(2)
    for epoch in range(num_epochs):
        timer, num_batches = d2l.Timer(), len(data_iter)
        for i, batch in enumerate(data_iter):
            optimizer.zero_grad()
            center, context_negative, mask, label = [
                data.to(device) for data in batch]

            pred = skip_gram(center, context_negative, net[0], net[1])
            l = (loss(pred.reshape(label.shape).float(),
label.float(), mask) / mask.sum(axis=1) * mask.shape[1])
            l.sum().backward()
            optimizer.step()
            metric.add(l.sum(), l.numel())
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f},
 'f'{metric[1] / timer.stop():.1f} tokens/sec on {str(device)}')
```

This function trains the neural network.

```
def get_similar_tokens(query_token, k, embed):
    W = embed.weight.data
    x = W[vocab[query_token]]
    cos = torch.mv(
        W, x) / torch.sqrt(torch.sum(W * W, dim=1) * torch.sum(x * x) + 1e
-9)
    topk = torch.topk(cos, k=k + 1)[1].cpu().numpy().astype('int32')
    for i in topk[1:]:  # Remove the input words
        print(f'cosine sim={float(cos[i]):.3f}: {vocab.idx_to_token[i]}')
```

This is the function that calculates similarity as tokens and outputs them.

These are the 3 words with the top ten similar words.



```
get_similar_tokens('lawyer', 10, net[0])
```

```
cosine sim=0.466: devastating
cosine sim=0.426: daughter
cosine sim=0.419: researcher
cosine sim=0.415: j.
cosine sim=0.403: davis
cosine sim=0.398: marvin
cosine sim=0.396: indicted
cosine sim=0.395: pleaded
cosine sim=0.391: peterson
cosine sim=0.388: prestigious
```

```
[10] get_similar_tokens('chip', 10, net[0])
```

```
cosine sim=0.423: insulin
cosine sim=0.421: toxin
cosine sim=0.418: intel
cosine sim=0.417: chips
cosine sim=0.413: compaq
cosine sim=0.413: motorola
cosine sim=0.409: gene
cosine sim=0.407: drives
cosine sim=0.405: computer
cosine sim=0.391: japan
```

```
[11] get_similar_tokens('memory', 10, net[0])
```

```
cosine sim=0.465: digital
cosine sim=0.457: delays
cosine sim=0.445: implemented
cosine sim=0.424: intel
cosine sim=0.419: disk
cosine sim=0.419: chips
cosine sim=0.415: drives
cosine sim=0.410: extending
cosine sim=0.403: crack
cosine sim=0.402: beginning
```

2. Task 2

2.1. Doc2Vec

In order to train a doc2vec model, the training documents need to be in the form of "TaggedDocument", which means each document receives a unique id, provided by the variable offset.

Furthermore, the function tokenization is done to transform the document from a string into a list of strings consisting of the document's words. It also allows for a selection of 'stopwords' as well as words exceeding a certain length to be removed.

A list of the 'm' unique words appearing in the set of documents has to be compiled. This develops into the vocabulary.

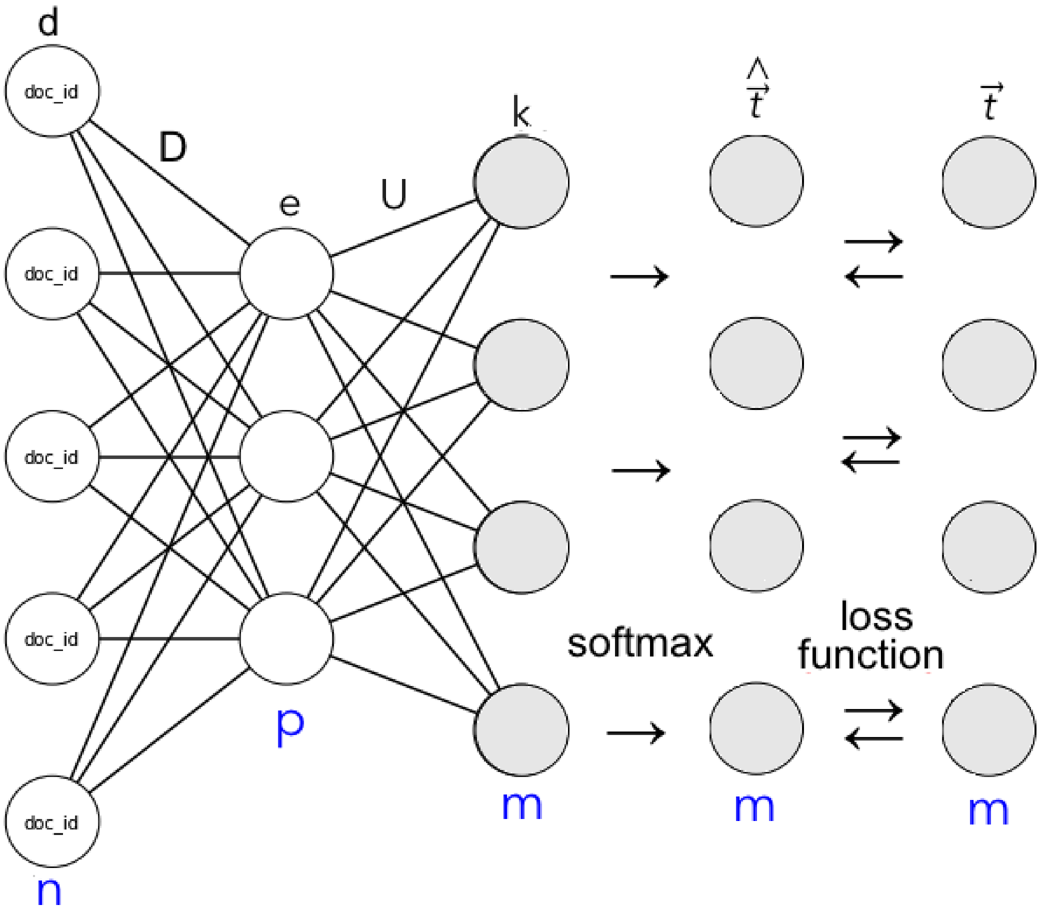


Figure 2.1:Working of Doc2Vec

2.1.1. Input Layer

The input vector 'd' is a one-hot encoded list of paragraph IDs of length 'n', with a node turning into 1 when the supplied ID matches the selected document.

2.1.2. Hidden layer

The hidden layer consists of a dense matrix ‘D’ that projects the document vector ‘d’ into the embedding space of dimension ‘p’, which we set. D thus has dimensions $p\ X\ n$ and is initialized randomly in the the beginning before any training has taken place.

After the training, ‘D’ will constitute a lookup matrix for the candidates, containing weights for the paragraph vector ‘e’ (representing embeddings). The latter will contain the estimated features of every document.

2.1.3. Output layer

In the output layer, the matrix ‘U’ projects the paragraph vector ‘e’ to the output activation vector ‘k’ which contains ‘m’ rows, representing the words in vocabulary. ‘U’ has dimension $m\ X\ p$ p and is initialized randomly similar to ‘D’

Using the softmax function and backpropagation normally. Adding the cross-entropy with Stochastic Gradient Decent we can ensure that the loss is gradually reduces with increasing iterations.

Doc2Vec also uses skip-gram algorithm. In Doc2vec an extra ‘paragraph id’ is added. The addition in the feature vector allows us to find the uniqueness of the document.

2.2. FastText

FastText uses a bag of n-grams representation along with word vectors to preserve some information about the surrounding words appearing near each word. This is the mixture of bag of continuous words and n-grams.

2.2.1. Subwords

Suppose that a word where was not in the training set. Previous to FastText, if where appears on the test set, then embedding models ignore unseen words and only use words in training set to embed sentences. This might seem reasonable, but we are missing quite information while doing so.

The key idea behind FastText is that subwords, character-level n-gram of each word, can be used to train word representation. The rationale is that similarly shaped words is more likely to have similar meanings (morphology). For example, where, who, when and why all have the 2-gram subword wh in common. This similarity in character composition has information that these words have similar semantics: the interogative.

Researchers carefully split words into subwords by adding special boundary symbols <, > which symbolize start of frame and end of frame respectively. Boundary symbols help the model to identify difference between similar but semantically different subwords. For instance, <where> and <her> both have her as their 3-gram subword. However nearby subwords of <her> (<he, er>) is quite different than those of <where> (whe, ere).

This helps to divide the unseen words and classify them.