T.C. GALATASARAY ÜNİVERSİTESİ

MÜHENDİSLİK VE TEKNOLOJİ FAKÜLTESİ

BİLGİSAYAR MÜHENDİSLİĞİNDE ÖZEL KONULAR PROJESİ RAPORU

Kaan OFLAZ / 19401858

Sena ATAKAN / 19401853

Bilgisayar Mühendisliği

Dr. Öğr. Üyesi Uzay Çetin

HAZİRAN 2023

PART 1 - word2vec

In the scope of this task, we are going to be explaining the integration of a new word with its respective embedding layer, to a simple word2vec architecture model, and then, the application of negative sampling in order to reduce the computation costs for a larger model. Firstly the concepts hidden behind the word2vec model will be explained briefly in the following paragraphs.

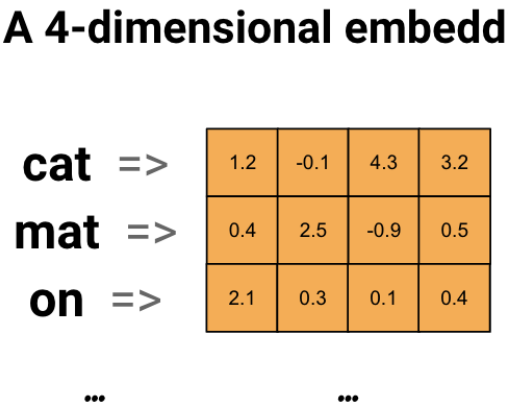
WORD2VEC ARCHITECTURE

Word2vec is not a singular algorithm, rather, it is a family of model architectures and optimizations that can be used to learn word embeddings from large datasets. Embeddings learned through word2vec have proven to be successful on a variety of downstream natural language processing tasks. The Word2vec family consists of two main models: Continuous Bag-of-Words (CBOW) and Skip-gram.

Word embbedings

Machine learning models take vectors (arrays of numbers) as input. When working with text, the first thing you must do is come up with a strategy to convert strings to numbers (or to "vectorize" the text) before feeding it to the model.

Word embeddings give us a way to use an efficient, dense representation in which similar words have a similar encoding. Importantly, you do not have to specify this encoding by hand. An embedding is a dense vector of floating point values (the length of the vector is a parameter you specify). Instead of specifying the values for the embedding manually, they are trainable parameters (weights learned by the model during training, in the same way a model learns weights for a dense layer). It is common to see word embeddings that are 8-dimensional (for small datasets), up to 1024-dimensions when working with large datasets. A higher dimensional embedding can capture fine-grained relationships between words, but takes more data to learn.



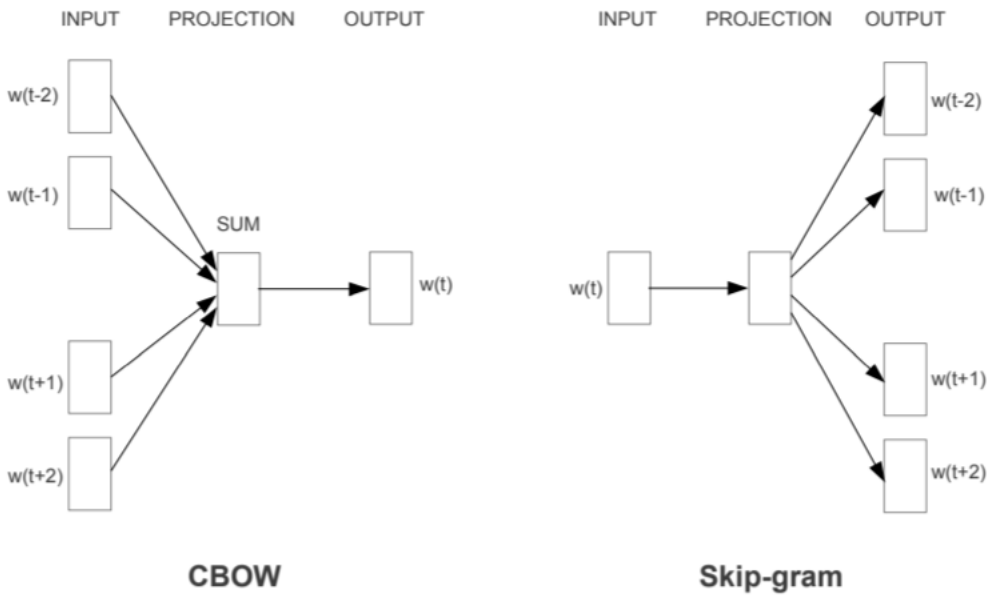
Above is a diagram for a word embedding. Each word is represented as a 4-dimensional vector of floating point values. Another way to think of an embedding is as "lookup table". After these weights have been learned, you can encode each word by looking up the dense vector it corresponds to in the table.

Continuous Bag-of-Words model vs Skip-gram model

In CBOW, the goal is to predict a target word given its surrounding context words. The model takes the context words as input and tries to predict the target word. The context words are represented as one-hot vectors, and the model learns to map them to the target word's embedding vector.

In Skip-gram, the objective is reversed: given a target word, the model aims to predict the context words that are likely to appear around it. The target word's embedding vector is used to generate the context word predictions. The model learns to adjust the target word's embedding to improve its ability to predict the surrounding words.

Both CBOW and Skip-gram architectures utilize a shallow neural network with a hidden layer, and the weights of the network are learned through a training process called stochastic gradient descent. The learned word embeddings capture semantic and syntactic relationships between words, enabling them to capture similarities and differences.

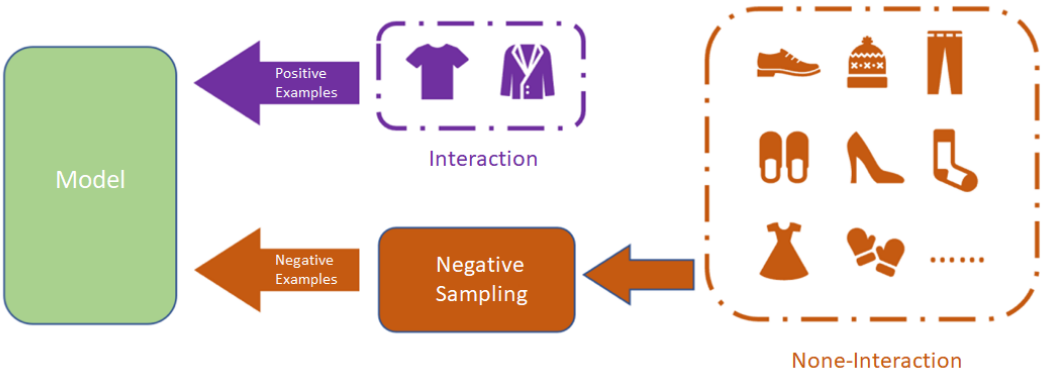


As illustrated, while the Bag of Continuous Words Model takes context words as input and tries to predict the target word, for the Skip-gram Model the goal is reversed: given a target word, the model aims to predict possible context words within its scope.

SKIP-GRAM AND NEGATIVE SAMPLING

Negative sampling is a technique used when training models such as language models. Its main goal is to provide meaningful learning signals while reducing computational complexity to make the training process more efficient.

When Negative Sampling is running, given a positive context word and target word, the model assumes that this pair is a positive example. Negative sampling involves randomly selecting a few words as negative examples. The number of negative examples is determined by a hyperparameter. The model learns to predict positive examples as "positive" (1) and negative examples as "negative" (0). During training, the model updates its parameters to minimize the loss between the predicted probabilities and the actual labels.



In traditional language modeling tasks, the aim is to estimate the probability of the target word based on its context. For this, the model needs to compute the softmax probability distribution considering the entire vocabulary. For large vocabularies, this calculation can be costly.

The skip-gram model is trained on skip-grams, which are n-grams that allow tokens to be skipped (see the diagram below for an example). The context of a word can be represented through a set of skip-gram pairs of (target_word, context_word) where context_word appears in the neighboring context of target_word.

Consider the following sentence of eight words:

The wide road shimmered in the hot sun.

The context words for each of the 8 words of this sentence are defined by a window size. The window size determines the span of words on either side of a target_word that can be considered a context word. Below is a table of skip-grams for target words based on different window sizes.

Window Size	Text	Skip-grams
2	[The **wide** road shimmered] in the hot sun.	wide, the wide, road wide, shimmered
	The [wide road **shimmered** in the] hot sun.	shimmered, wide shimmered, road shimmered, in shimmered, the
	The wide road shimmered in [the hot **sun**].	sun, the sun, hot
3	[The **wide** road shimmered in] the hot sun.	wide, the wide, road wide, shimmered wide, in
	[The wide road **shimmered** in the hot] sun.	shimmered, the shimmered, wide shimmered, road shimmered, in shimmered, the shimmered, hot
	The wide road shimmered [in the hot **sun**].	sun, in sun, the sun, hot

The training objective of the skip-gram model is to maximize the probability of predicting context words given the target word. For a sequence of words w1, w2, ... wT, the objective can be written as the average log probability.

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t)$$

Where c is the size of the training context. The basic skip-gram formulation defines this probability using the softmax function.

$$p(w_O|w_I)=\frac{\exp\left({v'_{w_O}}^{\top}v_{w_I}\right)}{\sum_{w=1}^{W}\exp\left({v'_w}^{\top}v_{w_I}\right)}$$

Where v and v' are target and context vector representations of words and W is vocabulary size.

Computing the denominator of this formulation involves performing a full softmax over the entire vocabulary words, which are often large (105-107) terms.

The noise contrastive estimation (NCE) loss function is an efficient approximation for a full softmax. With an objective to learn word embeddings instead of modeling the word distribution, the NCE loss can be simplified to use negative sampling.

The simplified negative sampling objective for a target word is to distinguish the context word from *k* negative samples drawn from noise distribution Pn(w) of words. More precisely, an efficient approximation of full softmax over the vocabulary is, for a skip-gram pair, to pose the loss for a target word as a classification problem between the context word and *k* negative samples.

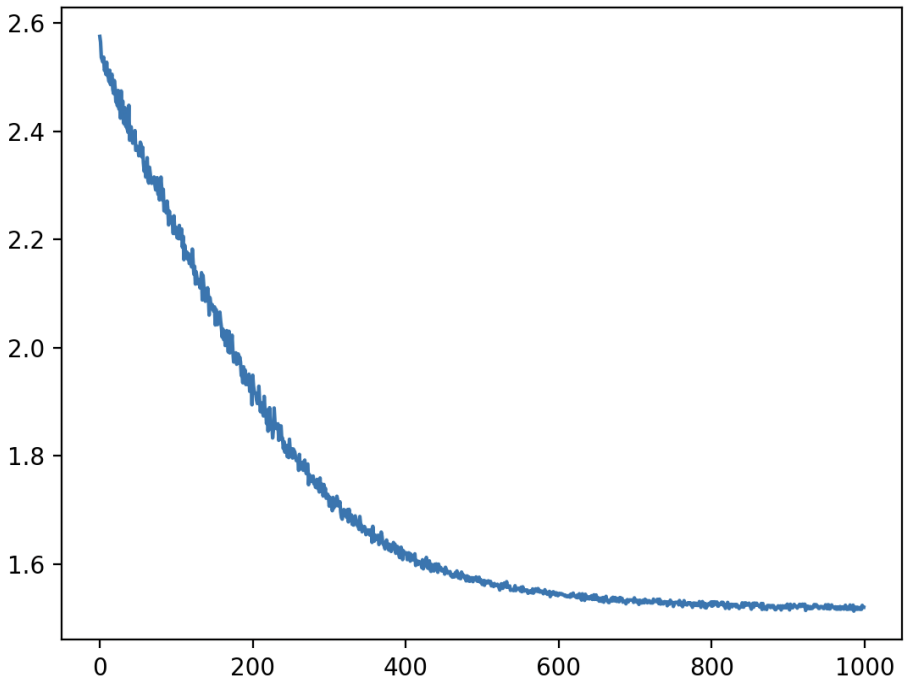
A negative sample is defined as a (target_word, context_word) pair such that the context_word does not appear in the window_size neighborhood of the target_word. For the example sentence, these are a few potential negative samples (when window_size is 2).

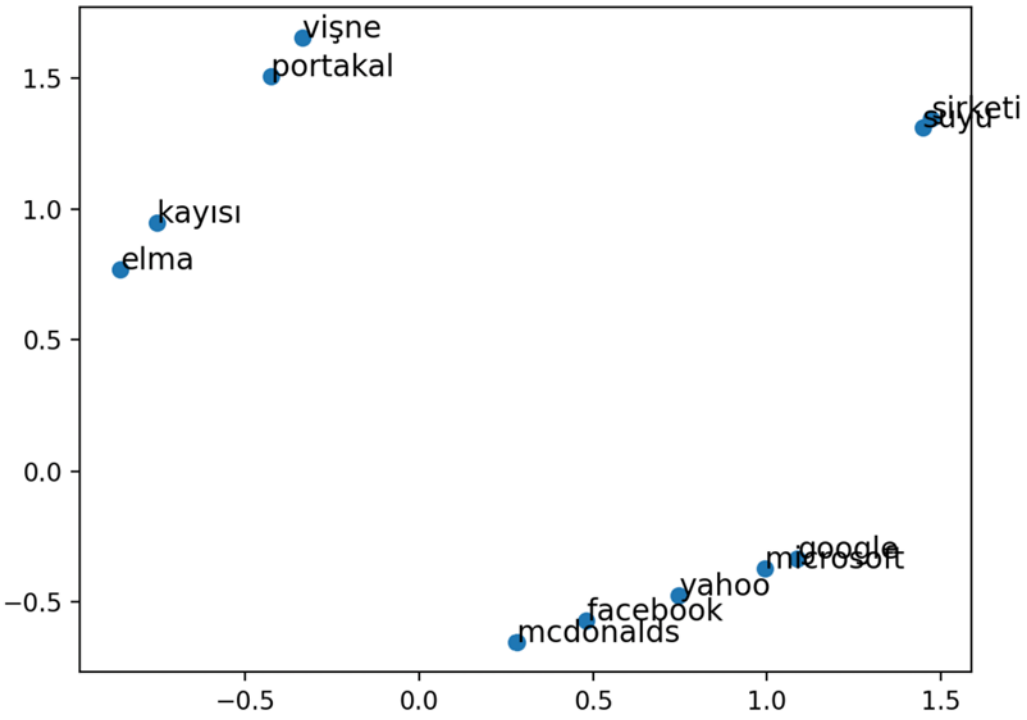
To sum up, negative sampling aims at maximizing the similarity of the words in the same context and minimizing it when they occur in different contexts. However, instead of doing the minimization for all the words in the dictionary except for the context words, it randomly selects a handful of words ($2 < k < 20$) depending on the training size and uses them to optimize the objective. We choose a larger k for smaller datasets and vice versa.

TASK 1 : adding word2vec embedding without training the model

The model designed for this task implements a simple Word2Vec model and shows how adding a new word can be done without the need to retrain the Word2Vec model. The code initially creates a corpus and extracts unique words from this corpus. It then creates a dataset using the word pairs obtained from the corpus. The Word2Vec model is described as a neural network with an embedding layer and an expansion layer. The model is trained on a dataset generated from the corpus and during the training process, the AdamW optimizer is used to optimize the parameters of the model and minimize the cross-entropy loss.

We can see below the value of the mean squared error through the learning process of the embedding layers. In fact through the 1000 iterations, the loss is constantly decreasing starting from 2.6 to 0.7.

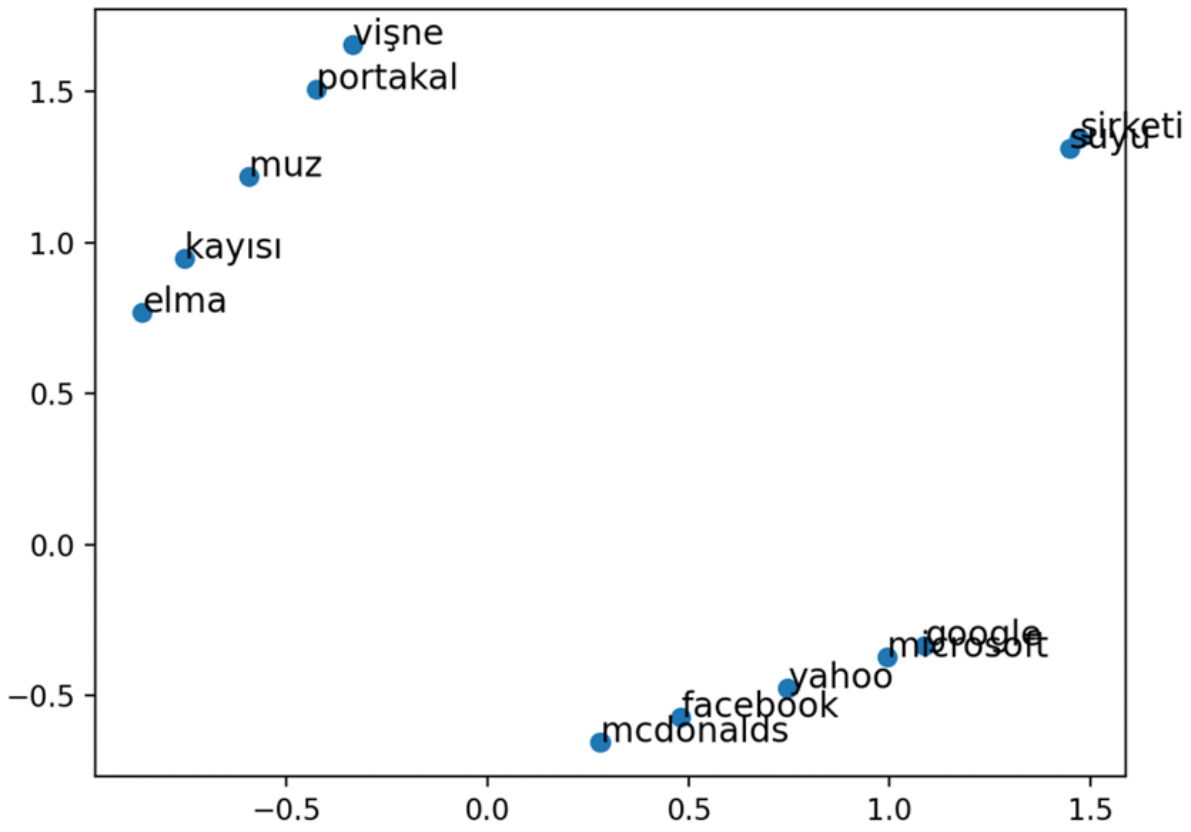




After learning process, the embedding which were random at the start, have taken a really interesting form which we can visualize below. Here is above the graphical representation of the 2 dimensional word embeddings of the model, after having performed the training step.

As we can see the fruit vectors have been stacked up at the top left corner of the graph while the companies have been at the bottom right one.

To add a new word to this model, the code provides a solution. First, it calculates the average of the available word vectors for a given category (e.g. fruits). In this example, the category contains words like "apple", "orange", "cherry" and "apricot". The average vector is used to create the vector representation for the new word "banana". The code adds the new word by expanding the vocabulary without the need to retrain the new word and its vector into the Word2Vec model.



This approach allows new words to be added to the Word2Vec model without retraining and thus provides a computationally efficient solution. By leveraging existing word vectors and relations, the code works efficiently in adding the new word to the Word2Vec model and incorporating the semantic information of the new word.

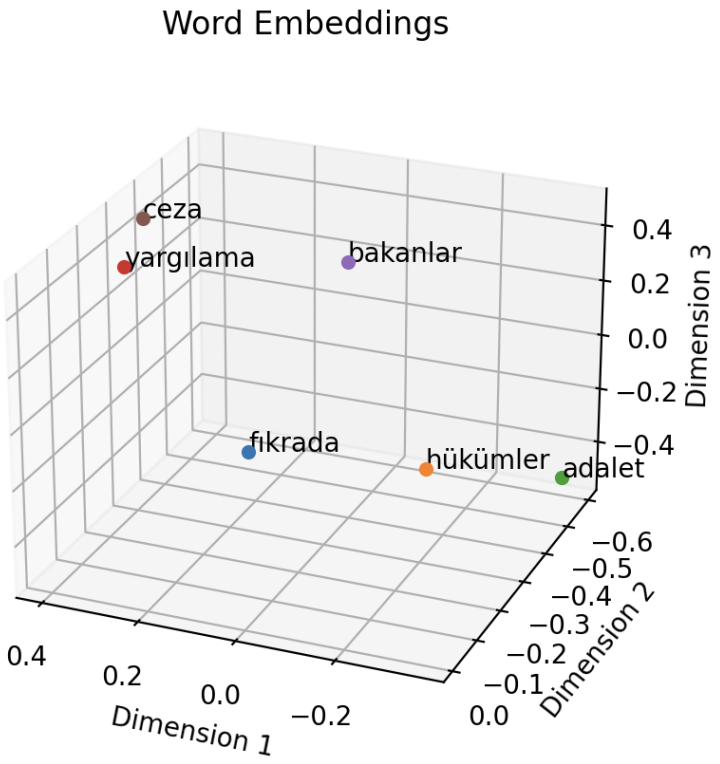
Once the training process is complete, the word vectors of the model are taken as wordvecs and converted into an array. The word vectors are visualized using the plotit function.

To add a new word, the existing word vectors are averaged, and this vector is assigned to the new word. Finally, the new word and its vector are added to the vocabulary and the weights of the expansion layer of the Word2Vec model.

TASK 2: negative sampling

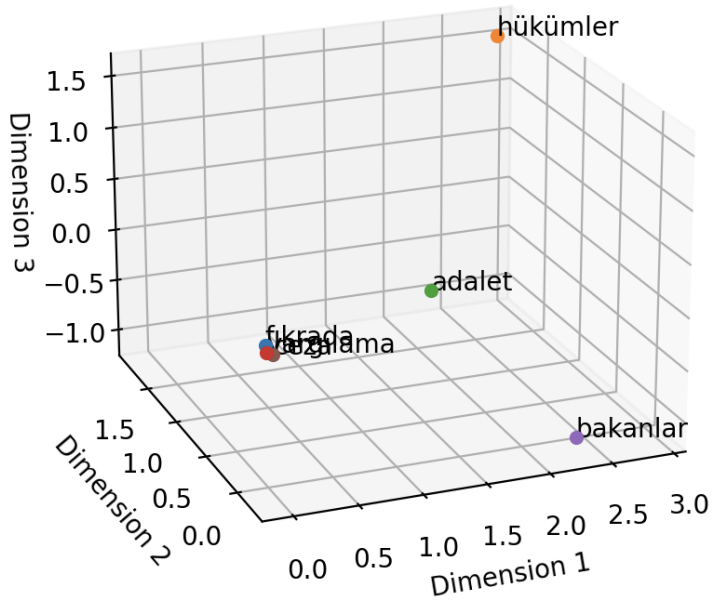
For a little model like the 11 words word2vec model of the first task, performing a full softmax though all the possible words which could potentially fit to the context isn't that expensive, since the vocabulary of the model is quite small. But when it comes to applying the same algorithm on a much bigger model, things start to be much more complicated to deal with.

In fact, with a vocabulary size which exceeds a certain amount, the computation cost starts to be much higher, which forces us to find some ways to optimize the performance. For this reason, we applied the Negative Sampling method using a larger set of words. In order to choose a text with a long vocabulary size, we used the text "Türk Anayasası" in our study. While visualizing the results, we tried different epoch and dimension values and observed the accuracy values and the resulting models.



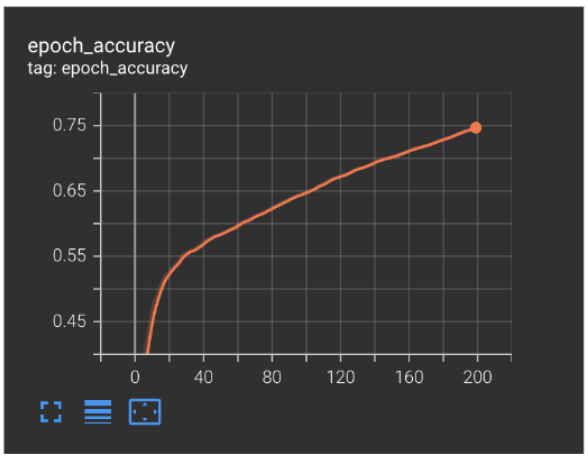
In the model in this image above, the number of dimensions is 80, the number of epochs is 200 and the accuracy value is very close to 100% with 0.9986.

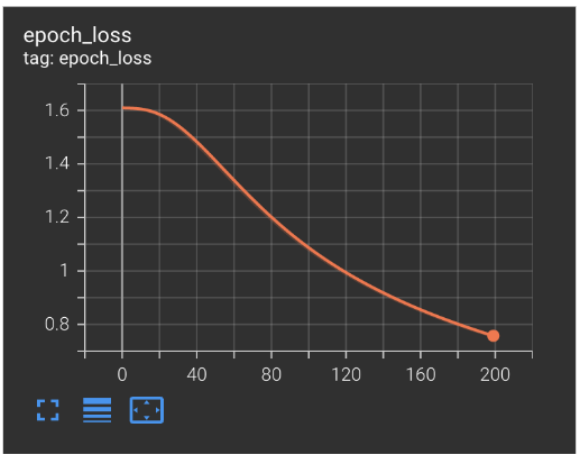
Word Embeddings



The image above shows the word embedding model with dimension number 3 and epoch number 200. Accuracy value is 0.7498. "fıkrada", "yargılama" and "ceza" are almost in the same position, while "adalet" is in the middle of the words.

The following two graphs show the changes in epoch accuracy and epoch loss values of this model during training.



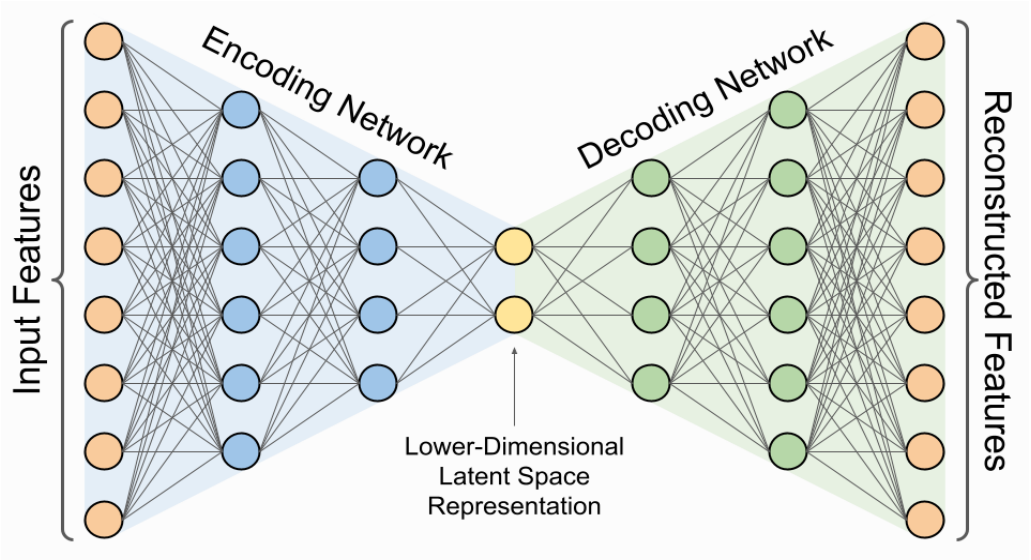


PART 2 - auto-encoder

TASK 4: pre-trained models and auto-encoders

The fast developpement of NLP and deep-learning allowed the built of models becoming much more accurate, complex, but also way bigger in size. For example the pre-trained model of Abdullah Köksal, being a relatively small one, exceeds 650 mb. With those models growing in size, the concept of auto encoders earned a lot in popularity.

Autoencoders are a class of Neural Network architectures that learn an encoding function, which maps an input to a compressed latent space representation, and a decoding function, which maps from the latent space back into the original space. Ideally, these functions are pure inverses of each other - passing data through the encoder and then passing the result through the decoder perfectly reconstructs the original data in what is called lossless compression.



The use of auto encoders in the scope of this task was to reduce the overall size of the word2vec model of Abdullah Köksal of 680 MB, shrink it in order to get a reduced size model of approximately 60MB, still performing well.

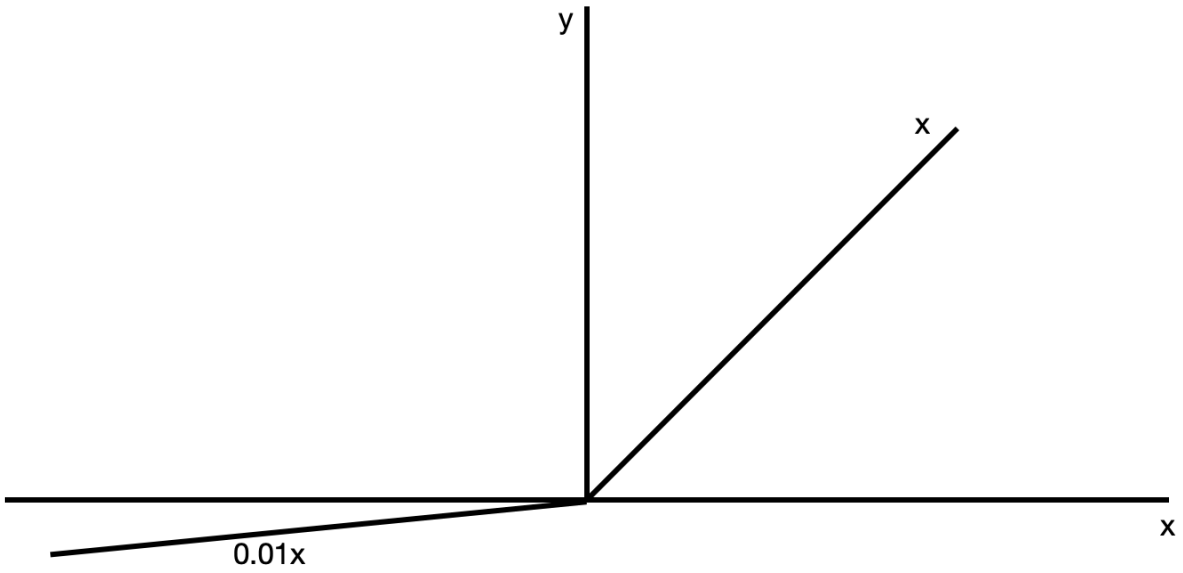
This is an overview of the autoencoder architecture used bellow:

```
class AutoEncoders(Model):

    def __init__(self, output_units):
        super().__init__()
        activ_func = "LeakyReLU"

        self.encoder = Sequential(
            [
                Dense(320, activation=activ_func),
                Dense(160, activation=activ_func),
                Dense(80, activation=activ_func),
                Dense(40, activation=activ_func),
            ]
        )

        self.decoder = Sequential(
            [
                Dense(40, activation=activ_func),
                Dense(80, activation=activ_func),
                Dense(160, activation=activ_func),
                Dense(320, activation=activ_func),
                Dense(output_units, activation="linear")
            ]
        )
```

```
Model: "auto_encoders"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential (Sequential)     (None, 40)                195800

 sequential_1 (Sequential)   (None, 400)               197800

=================================================================
Total params: 393,600
Trainable params: 393,600
Non-trainable params: 0
_________________________________________________________________
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 320)               128320

 dense_1 (Dense)             (None, 160)               51360

 dense_2 (Dense)             (None, 80)                12880

 dense_3 (Dense)             (None, 40)                3240

=================================================================
Total params: 195,800
Trainable params: 195,800
Non-trainable params: 0
_________________________________________________________________
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 40)                1640

 dense_5 (Dense)             (None, 80)                3280

 dense_6 (Dense)             (None, 160)               12960

 dense_7 (Dense)             (None, 320)               51520

 dense_8 (Dense)             (None, 400)               128400

=================================================================
Total params: 197,800
Trainable params: 197,800
```

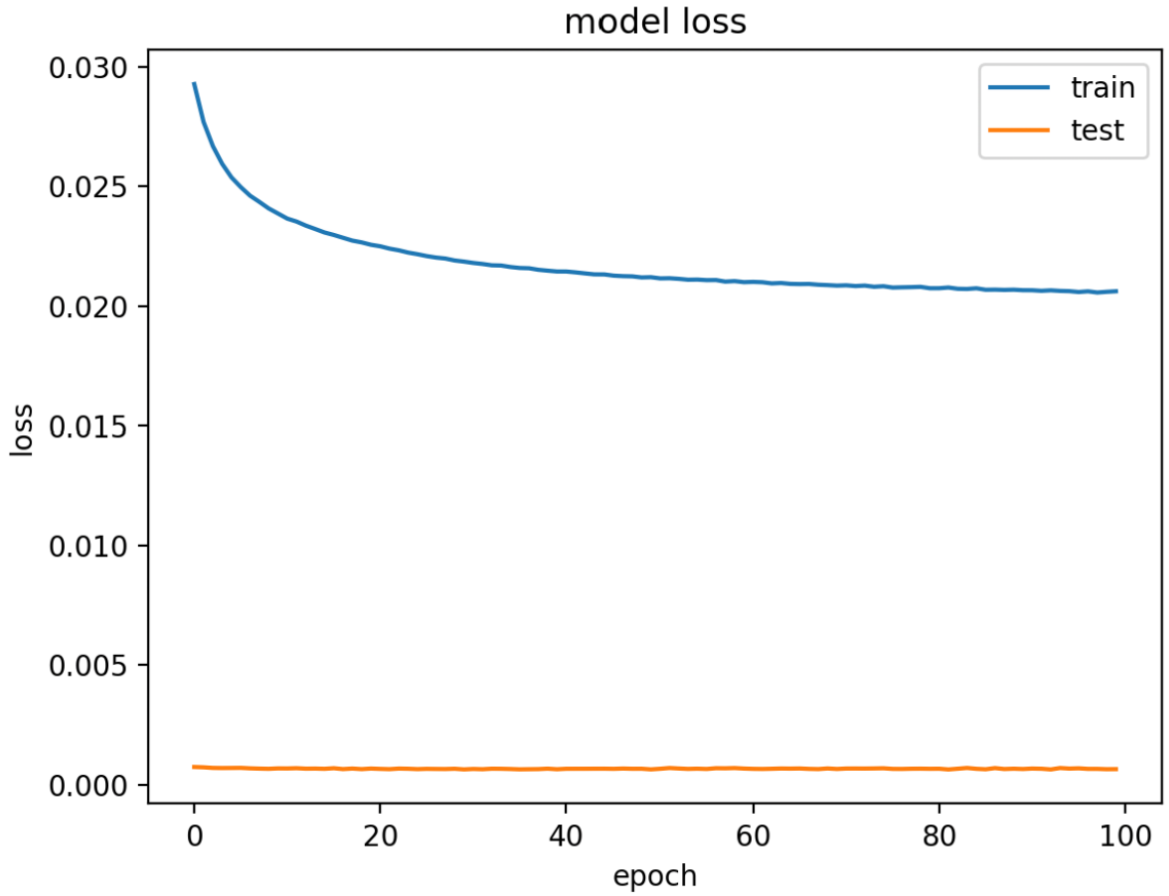
As we can see in the summary above, the auto encoder architecture is composed of an encoder part and a decoder part, which are symmetrical to each other.
When in comes to the activation function, we tested the sigmoid function for the hidden layers as well as the softmax activation for the output one, but the result we got with those were awful.

The ReLU activation wasn't doing well too. We figured out after testing that the property of ReLU to give a null signal when the input x were negative created to much sparsity, leading to in accurate encoding. Thus, we used leaky ReLU to overcome this problem:



We can see that for negative input values, the leaky ReLU activation function still gives negative outputs (different that 0), which allows the encoder side to keep track of data embeddings when the float values come to be negative.

With the use of this activation function, we ended up with quite impressive results:



Epoch 100/100
2256/2256 [==============================] - 11s 5ms/step - loss: 0.0206 - accuracy: 0.2660 - val_loss: 6.4804e-04 - val_accuracy: 0.2039

As we can see, the lower dimensional latent space of encoder architecture is equal to 40 dimensions. In other words, the model of abdullah köksal, using 400 dimensions embedding vectors, has been shrunk and compressed into an autoencoder model using 40 dimension embedding vectors. As a result, the model passed from a size of 680 MB to 60 MB, and still reaches a 25% accuracy, which is quite impressive for a more than 10 times smaller model.