Unit 6
Embedding and Representation Learning

Prepared By: Arjun Singh Saud

Asst. Prof. CDCSIT

Learning Lower-Dimensional Representations

- An embedding is a mapping of a discrete (categorical) variable to a vector of continuous numbers.
- In the context of neural networks, embeddings are low-dimensional, learned continuous vector representations of discrete variables.
- Neural network embeddings are useful because they can reduce the dimensionality of categorical variables and meaningfully represent categories in the transformed space.

Learning Lower-Dimensional Representations

- Neural network embeddings have 3 primary purposes:
  1. Finding nearest neighbors in the embedding space. These can be used to make recommendations based on user interests or cluster categories.
  2. As input to a machine learning model for a supervised task.
  3. For visualization of concepts and relations between categories.

Learning Lower-Dimensional Representations

- For example, using neural network embeddings, we can take all 37,000 book articles on Wikipedia and represent each one using only 50 numbers in a vector.
- Moreover, because embeddings are learned, books that are more similar in the context of our learning problem are closer to one another in the embedding space.
- Neural network embeddings overcome the two limitations of a common method for representing categorical variables: one-hot encoding.

Learning Lower-Dimensional Representations

- The operation of one-hot encoding categorical variables is actually a simple embedding where each category is mapped to a different vector.
- This process takes discrete entities and maps each observation to a vector of 0s and a single 1 signaling the specific category. The one-hot encoding technique has two main drawbacks.
- For high-cardinality variables the dimensionality of the transformed vector becomes unmanageable.

Learning Lower-Dimensional Representations

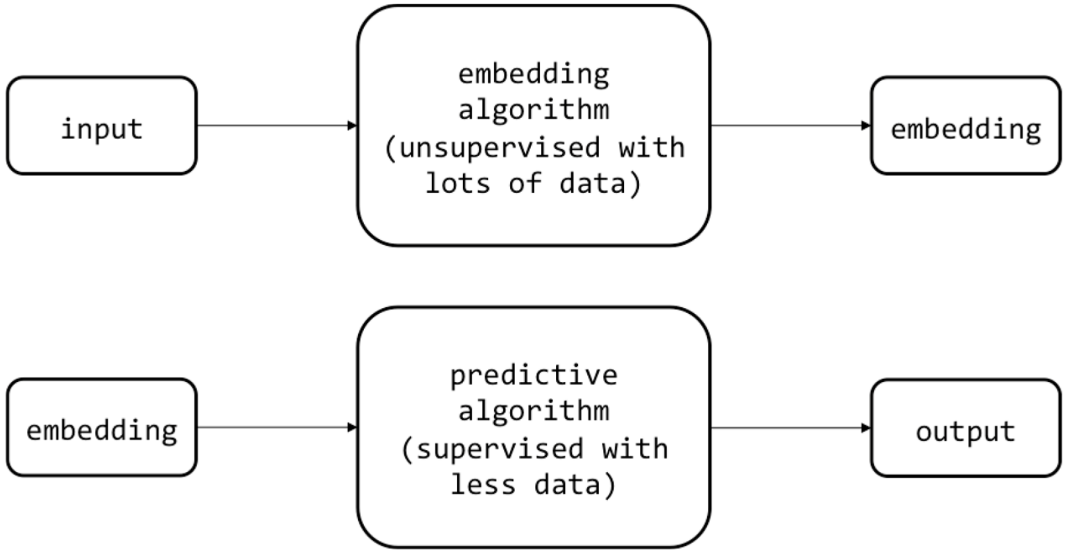
- For example, if we have 37,000 books on Wikipedia, then representing these requires a 37,000-dimensional vector for each book, which makes training any machine learning model on this representation infeasible.
- The mapping is completely uninformed. Similar categories are not placed closer to each other in embedding space.
- For example, if we measure similarity between vectors using the cosine distance, then after one-hot encoding, the similarity is 0 for every comparison between entities.

Learning Lower-Dimensional Representations

- We can greatly improve embeddings by learning them using a neural network on a supervised task. The embeddings form the parameters (weights) of the network which are adjusted to minimize loss on the task.
- The resulting embedded vectors are representations of categories where similar categories relative to the task are closer to one another.

- For example, if we have a vocabulary of 50,000 words used in a collection of movie reviews, we could learn 100-dimensional embeddings for each word using an embedding neural network trained to predict the sentimentality of the reviews.

Learning Lower-Dimensional Representations


input
embedding algorithm (unsupervised with lots of data)
embedding
embedding
predictive algorithm (supervised with less data)
output

- Commonly used Embedding models are PCA, SVD, Word2Vec, Bidirectional Encoder Representations of Transformers (BERT) etc.

Principle Component Analysis.

- See From AML slide

Autoencoders

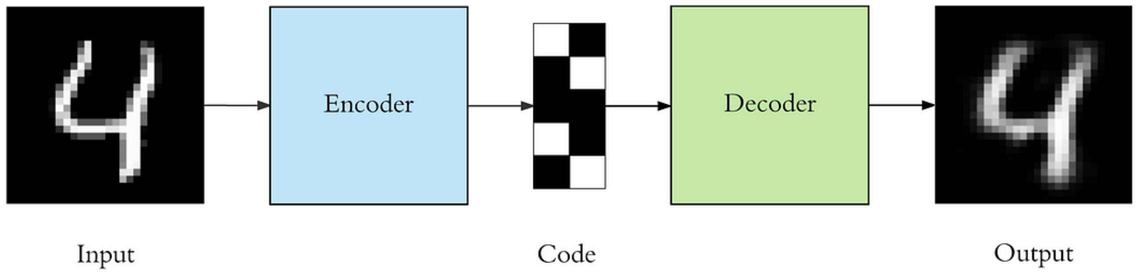
- Autoencoders are an unsupervised learning technique in which we leverage neural networks for the task of representation learning.
- Autoencoders are a specific type of feedforward neural networks where the input is the same as the output.

$$f(x) = x$$

- They compress the input into a lower-dimensional code and then reconstruct the output from this representation.
- The code is a compact summary or compression of the input, also called the latent-space representation.

Autoencoders

- An Autoencoder consists of 3 components: encoder, code and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code.



- Autoencoders are mainly a dimensionality reduction (or compression) algorithm with a couple of important properties:

Autoencoders

- **Data-specific:** Autoencoders are only able to meaningfully compress data similar to what they have been trained on. Since they learn features specific for the given training data, they are different than a standard data compression algorithm like gzip. So we can’t expect an Autoencoder trained on handwritten digits to compress landscape photos.
- **Lossy:** The output of the Autoencoder will not be exactly the same as the input, it will be a close but degraded representation. If you want lossless compression they are not the way to go.

Autoencoders

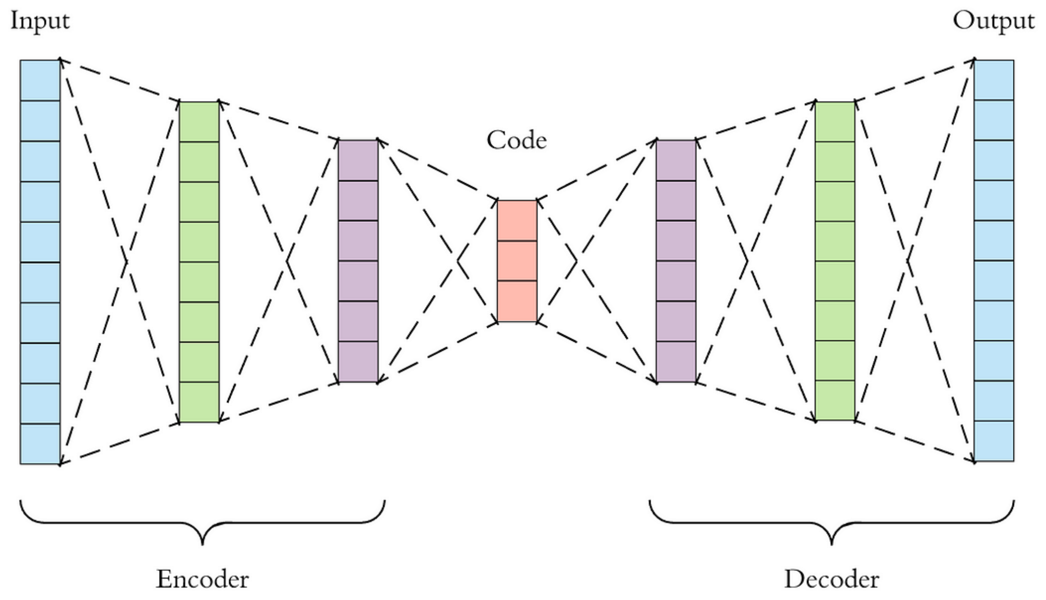
- **Unsupervised:** To train an Autoencoder we don’t need to do anything fancy, just throw the raw input data at it. Autoencoders are considered an unsupervised learning technique since they don’t need explicit labels to train on. But to be more precise they are self-supervised because they generate their own labels from the training data.

Autoencoder Architecture

- Autoencoder architecture is designed by imposing a bottleneck network which forces a compressed knowledge representation of the original input.
- An Autoencoder consists of three layers: *Encoder, Code, Decoder.*
- The Encoder layer compresses the input image into a latent space representation. It encodes the input image as a compressed representation in a reduced dimension. The compressed image is a distorted version of the original image.
- The Code layer represents the compressed input fed to the decoder layer.

Autoencoder Architecture

- The decoder layer decodes the encoded image back to the original dimension. The decoded image is reconstructed from latent space representation, which is lossy reconstruction of the original image and hence is only near match of the original image.



Autoencoder Architecture

- Both the encoder and decoder are fully-connected feedforward neural networks, essentially the ANNs.
- ode is a single layer of an ANN with the dimensionality of our choice. The number of nodes in the code layer (code size) is a hyperparameter that we set before training the Autoencoder.
- The decoder architecture is the mirror image of the encoder. This is not a requirement but it's typically the case. The only requirement is the dimensionality of the input and output needs to be the same.

Autoencoder Architecture

- Both the encoder and decoder are fully-connected feedforward neural networks, essentially the ANNs.
- ode is a single layer of an ANN with the dimensionality of our choice. The number of nodes in the code layer (code size) is a hyperparameter that we set before training the Autoencoder.
- The decoder architecture is the mirror image of the encoder. This is not a requirement but it's typically the case. The only requirement is the dimensionality of the input and output needs to be the same.

Autoencoder Architecture

- While training Autoencoders we take an unlabeled dataset and frame it as a supervised learning problem tasked with outputting a reconstruction of the original input $\hat{x}$.
- This network can be trained by minimizing the reconstruction error $L(x, \hat{x})$, which measures the differences between our original input and the consequent reconstruction.
- We either use mean squared error (MSE) or binary cross-entropy. If the input values are in the range [0, 1] then we typically use cross-entropy, otherwise we use the mean squared error.

$$L(x, x) = \sqrt{\sum_{i=1}^{n} (x_i - \widehat{x_i})^2}$$

Denoising Autoencoder (DAE)

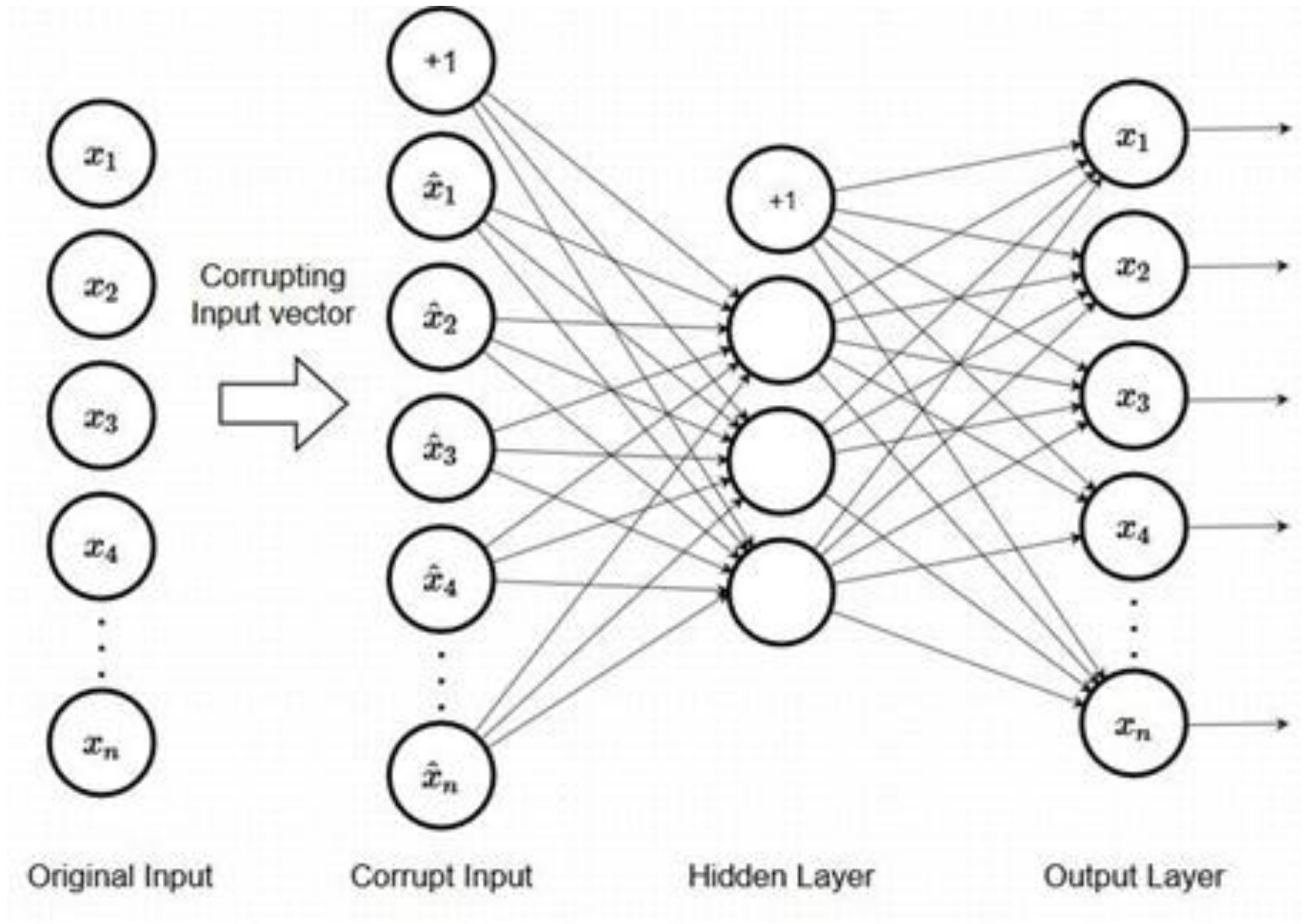
- Denoising Autoencoders simply corrupts input data using some probabilistic process before feeding it to the network.
- A simple probabilistic process for corrupting the data is following.

$$P(\tilde{x}_{ij} = 0|x_{ij})\text{=q}$$

$$P(\tilde{x}_{ij} = x_{ij}|x_{ij})\text{=1-q}$$

- DAE takes corrupted data as input and produces uncorrupted data as output. Thus, goal of DAE is not to learn exactly identity function.

Denoising Autoencoder (DAE)



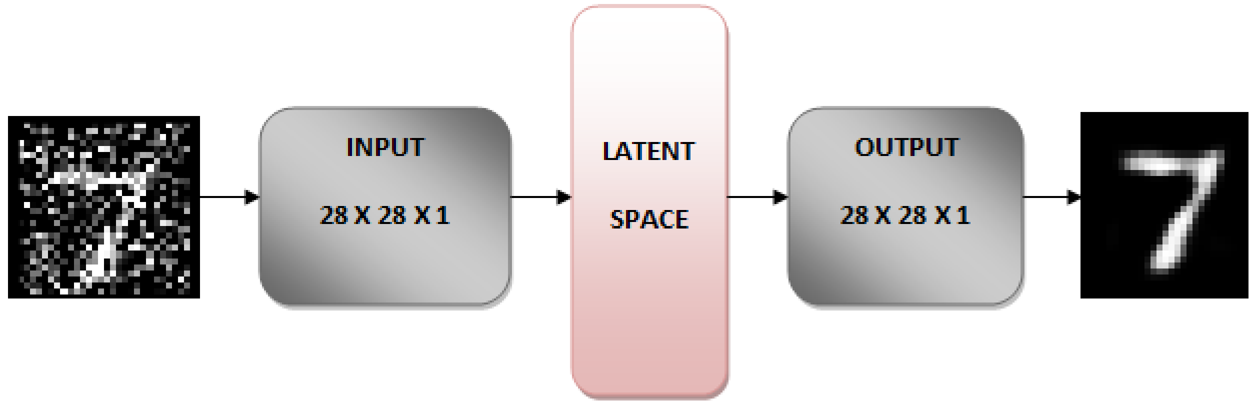
Denoising Autoencoder (DAE)

- DAE also learn lower dimensional representation of original data and not the corrupted data. This can be forced by using following loss function.

$$L(x,x) = \sqrt{\sum_{i=1}^{n}(x_i - \widehat{x_i})^2}$$

- The main goal of DAE is to learn generalizable encoding and decoding of the input data such that when there is variation in the test data compared to training data the model will be able to predict with accuracy.

Denoising Autoencoder (DAE)



Sparse Autoencoder (SAE)

- This is a type of Autoencoder that explicitly penalizes the use of hidden node connections. This regularizes the model, keeping it from overfitting the data.
- In SAE, we construct loss function by penalizing activations within a layer. For any given observation, we'll encourage our network to learn an encoding and decoding which only relies on activating a small number of neurons.
- Individual nodes of a trained model which activate are data-dependent, different inputs will result in activations of different nodes through the network.

Sparse Autoencoder (SAE)

- This is a type of Autoencoder that explicitly penalizes the use of hidden node connections. This regularizes the model, keeping it from overfitting the data.
- In SAE, we construct loss function by penalizing activations within a layer. For any given observation, we'll encourage our network to learn an encoding and decoding which only relies on activating a small number of neurons.
- Individual nodes of a trained model which activate are data-dependent, different inputs will result in activations of different nodes through the network.

Sparse Autoencoder (SAE)

- Am SAE will be forced to selectively activate regions of the network depending on the input data. As a result, we've limited the network's capacity to memorize the input data without limiting the networks capability to extract features from the data.
- There are two main ways by which we can impose this sparsity constraint: *L1 regularization and KL divergence*.

Sparse Autoencoder (SAE)

- **L1 Regularization:** We can add a term to our loss function that penalizes the absolute value of the vector of activations in layer h for observation i scaled by a tuning parameter λ.

$$\mathcal{L}(x, \hat{x}) + \lambda \sum_{i} \left| a_i^{(h)} \right|$$

- **KL Divergence:** In essence, KL-divergence is a measure of the difference between two probability distributions. We add KL divergence in loss function as below:

$$\mathcal{L}(x, \hat{x}) + \sum_{j} KL(\rho || \hat{\rho}_j)$$

Sparse Autoencoder (SAE)

- In the above equation ρ is Sparsity parameter which represent ideal probability distribution of Bernoulli random variable and $\hat{\rho}_j$ is average activation of a neuron j over a collection of samples. The activation $\hat{\rho}_j$ is given as below:

$$\hat{\rho}_j = \frac{1}{m} \sum_i \left[a_i^{(h)}(x) \right]$$

where the subscript j denotes the specific neuron in layer h, summing the activations for m training observations denoted individually as x.

- We would like to (approximately) enforce the constraint: $\hat{\rho}_j = \rho$

Sparse Autoencoder (SAE)

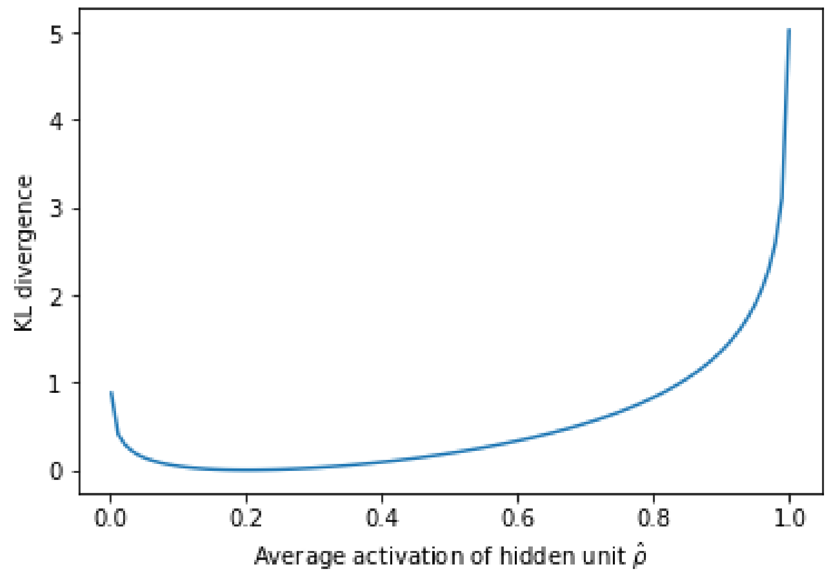
- ρ is a Sparsity parameter, typically a small value close to zero (say ρ = 0.005). In other words, we would like the average activation of each hidden neuron j to be close to 0.005 (say).
- KL-divergence is given as below:

$$\sum_{j} KL\left(\rho || \hat{\rho}_j\right) = \sum_{j=1}^{l^{(h)}} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$$

Here, l(h) is number of neurons in hidden layer h.

Sparse Autoencoder (SAE)

- This penalty function has the property that $KL(\rho||\hat{\rho}_j) = 0$ if $\hat{\rho}_j = \rho$, and otherwise it increases monotonically as $\hat{\rho}_j$diverges from ρ . For example, in the figure below, we have set $\rho = 0.2$, and plotted $KL(\rho||\hat{\rho}_j)$for a range of values of $\hat{\rho}_j$:



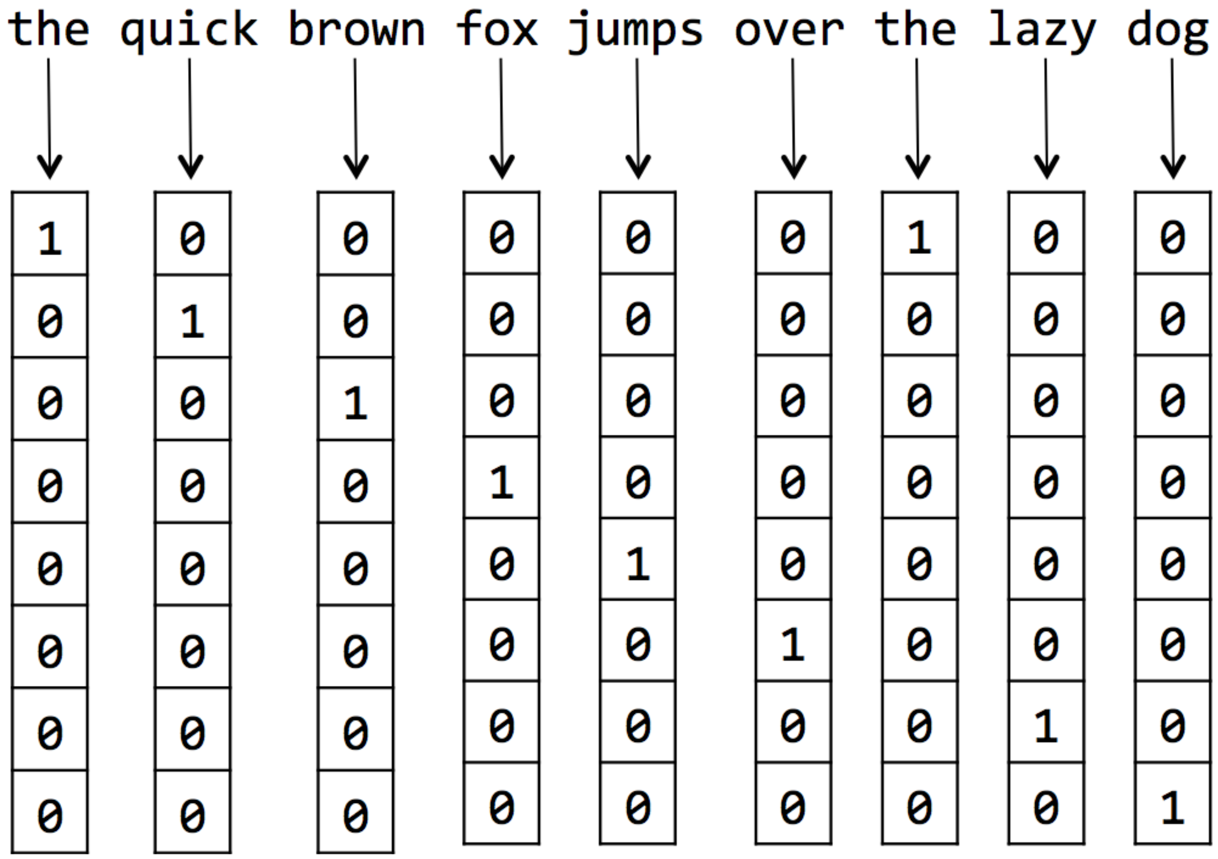
When Context is More Informative Than input Vector

- Context is defined as fixed window of words surrounding a target word.
- Context analysis involves breaking down sentences into n-grams to extract the meaning of a target word within a collection of unstructured text documents.
- One problem in NLP is to find a way of representing words in a document collection.

When Context is More Informative Than input Vector

- The naïve approach for representing words in a document collection is to use one-hot-encoding scheme.
- Suppose a document has a vocabulary V with |V| words. One-hot-encoding scheme represents words by using V-dimensional vectors where each unique word has unique index in this vector.
- To represent unique word w_i, we set the i^{th} component of the vector to be 1, and zero out all of the other components.

When Context is More Informative Than input Vector



the	quick	brown	fox	jumps	over	the	lazy	dog
1	0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1

When Context is More Informative Than input Vector

- Some of the problems associated with one-hot encoding scheme are listed below.
    - It can lead to increased dimensionality, as a separate column is created for each category in the variable. This can make the model more complex and slow to train.
    - It can lead to sparse data, as most observations will have a value of 0 in most of the one-hot encoded columns.
    - It can lead to overfitting, especially if there are many categories in the variable and the sample size is relatively small.
    - It does represent make similar words into similar vectors. This is problematic, because our models does not that the words “jump” and “leap” have very similar meanings.

When Context is More Informative Than input Vector

Brown fox jumps over the dog	Brown fox leaps over the dog
The boy jumps over the fence	The boy leaps over the fence

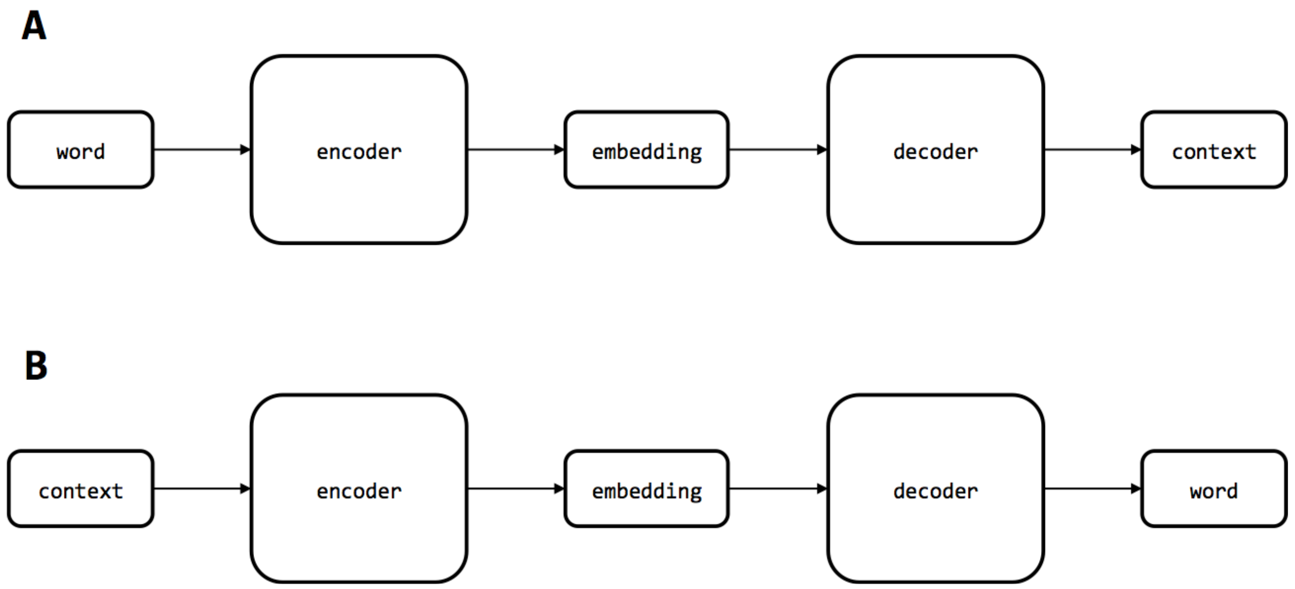
- Simply stating, one-hot-encoding scheme cannot represent context in NLP.
- To address this challenge, we need to find some way of discovering these relationships and encoding this information into a vector.

When Context is More Informative Than input Vector

- To address this challenge, we need to find some way of discovering these relationships and encoding this information into a vector.
- one way to discover relationships between words is by analyzing their surrounding context. For example, synonyms such as “jump” and “leap” both can be used interchangeably in their respective contexts.
- ee can use the same principles we used when building the Autoencoder to build a network that builds strong, distributed representations.

When Context is More Informative Than input Vector

- Two strategies are shown in Figure below.



When Context is More Informative Than input Vector

- One possible method (shown in A) passes the target through an encoder network to create an embedding. Then we have a decoder network take this embedding; but instead of trying to reconstruct the original input as we did with the Autoencoder, the decoder attempts to construct a word from the context.
- The second possible method (shown in B) does exactly the reverse: the encoder takes a word from the context as input, producing the target.

The Word2Vec Framework

- Word embedding is one of the most popular representation of document vocabulary. It is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc.
- Word embeddings are vector representations of words in document vocabulary.
- Word2Vec is one of the most popular technique to learn word embeddings using neural network.

The Word2Vec Framework

- Word2Vec learns word associations from a big text corpus such that the model is capable to detect similar terms or suggest extra words for a partial text after being trained.
- A straightforward mathematical function (cosine similarity) may be used with Word2Vec framework to determine the degree of semantic similarity between the words represented by those vectors.
- This is possible because the vectors are carefully selected to capture the semantic and syntactic characteristics of words.

The Word2Vec Framework

- Consider the following similar sentences: *Have a good day* and *Have a great day*. They hardly have different meaning.
- Document vocabulary for above two sentences is given as: V = {Have, a, good, great, day}.
- If we choose one-hot-encoding scheme to represent above vocabulary, the representation will be like below:

  Have = [1,0,0,0,0]; a=[0,1,0,0,0]; good=[0,0,1,0,0]; great=[0,0,0,1,0]; day=[0,0,0,0,1]

The Word2Vec Framework

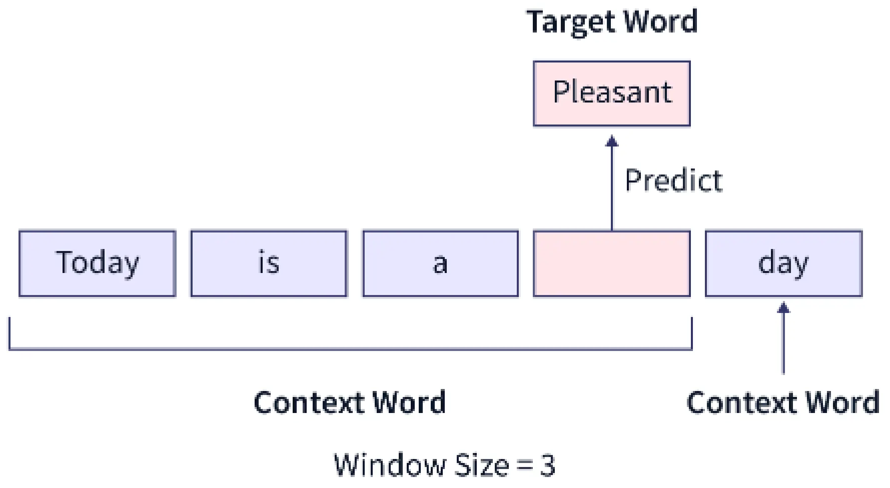
- We can think of a 5 dimensional space, where each word occupies one of the dimensions and has nothing to do with the rest.
- This means 'good' and 'great' are as different as 'day' and 'have', which is not true.
- Our objective is to have words with similar context occupy close spatial positions. Mathematically, the cosine of the angle between such vectors should be close to 1, i.e. angle close to 0.

The Word2Vec Framework

- Here comes the idea of generating distributed representations. Intuitively, we introduce some dependence of one word on the other words.
- Word2Vec is a method to construct such an embedding. It can be obtained using two methods: *Continuous Bag Of Words (CBOW) and Skip Gram*.

CBOW Model

- CBOW model predicts a target word based on the context of the surrounding words in a sentence or text.



- It is trained using a feedforward neural network where the input is a set of context words, and the output is the target word.

CBOW Model

- The model learns to adjust the weights of the neurons in the hidden layer to produce an output that is most likely to be the target word.
- The embeddings learned by the model can be used for a variety of NLP tasks such as *text classification, language translation, and sentiment analysis*.
- It's fast and efficient in learning the representation of context words and predict the target word in one go, which is also known as one pass learning.

CBOW Model

- Its method is widely used in learning word embeddings, where the goal is to learn a dense representation of words, where semantically similar words are close to each other in the embedding space.
- Consider the phrase, "*It is a pleasant day.*" The model transforms this sentence into word pairs (context word and target word).
- The user must configure the window size. The word pairings would appear like this if the context word's window were 2:

  ([it, a], is), ([is, pleasant], a) ([a, day], pleasant).

CBOW Model

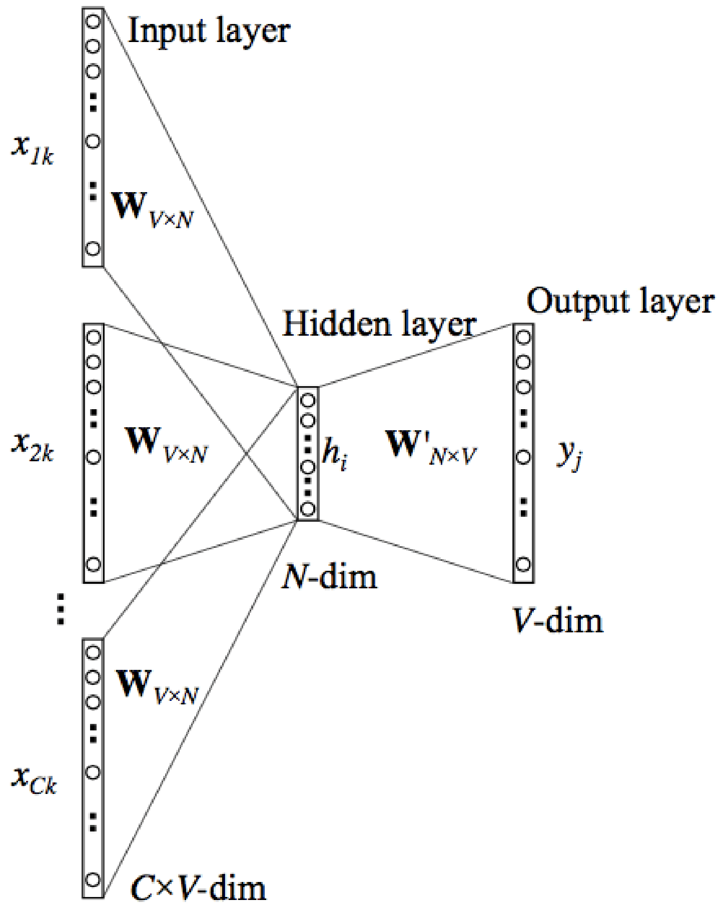
CBOW Architecture

- The architecture of the CBOW model is relatively simple, consisting of an input layer, a hidden layer, and an output layer.
- The input layer is used to represent the context words, the hidden layer is used to learn the word embeddings, and the output layer is used to predict the target word.

CBOW Architecture

- The architecture of the CBOW model is relatively simple, consisting of an input layer, a hidden layer, and an output layer.
- The input layer is used to represent the context words, the hidden layer is used to learn the word embeddings, and the output layer is used to predict the target word.

CBOW Architecture



CBOW Architecture

- The dimensions of the input vector will be 1xV, where V is the number of words in the vocabulary. This means input vector is one-hot representation of the word.
- The single hidden layer will have dimension VxN, where N is the size of the word embedding and is a hyper-parameter.
- The output from the hidden layer would be of the dimension 1xN, which we will feed into an softmax layer.
- The dimensions of the output layer will be 1xV, where each value in the vector will be the probability score of the target word at that position.

CBOW Architecture

- If we have 4 context words for a single target word, we will have four 1xV input vectors.
- Each will be multiplied with the VxN hidden layer returning 1xN vectors.
- All four 1xN vectors will be averaged element-wise to obtain the final activation which then will be fed into the softmax layer.

Skip-Gram Model

- The Skip-gram model architecture usually tries to achieve the reverse of what the CBOW model does.
- It tries to predict the source context words from a given target word.
- Consider the phrase, "*It is a pleasant day.*" If the window size is s, we have to predict [it, a] from the given *is*, [is, pleasant] from the given *'a'*, and [a, day] from the given *pleasant*.

Skip-Gram Model

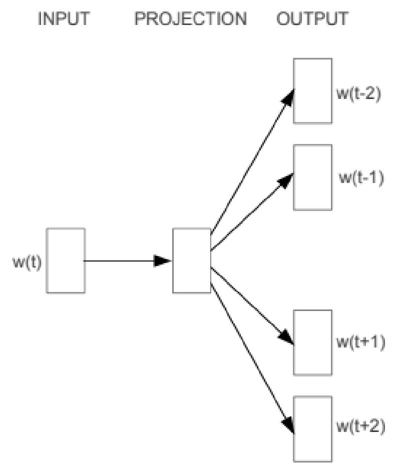
- Just like we discussed in the CBOW model, we need to model this Skip-gram architecture now as a deep learning classification model such that we take in the target word as our input and try to predict the context words.
- This becomes slightly complex since we have multiple words in our context.
- We simplify this further by breaking down each (target, context_words) pair into (target, context) pairs such that each context consists of only one word.

Skip-Gram Model

- Hence our dataset from earlier gets transformed into pairs like: (is, it),(is, a), (a, is), (a pleasant) so on.
- But how to supervise or train the model to know what is contextual and what is not?
- For this, we feed our skip-gram model pairs of (x, y) where x is our input and y is our label.
- We do this by using [(target, context), 1] pairs as positive input samples where target is our word of interest and context is a context word occurring near the target word and the positive label 1 indicates this is a contextually relevant pair.

Skip-Gram Model

- We also feed in [(target, random), 0] pairs as negative input samples where target is again our word of interest, random is just a randomly selected word from our vocabulary which has no context or association with our target word and the negative label 0 indicates this is a contextually irrelevant pair.



Skip-gram