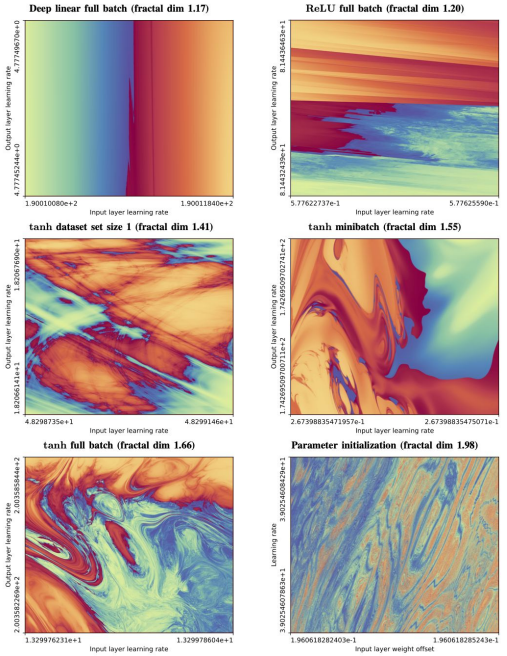
Natural Language Processing & Vector Embeddings

Agenda - Schedule

1. Kahoot
2. Pre-class Review
3. NLP
4. Break
5. Vector Embeddings

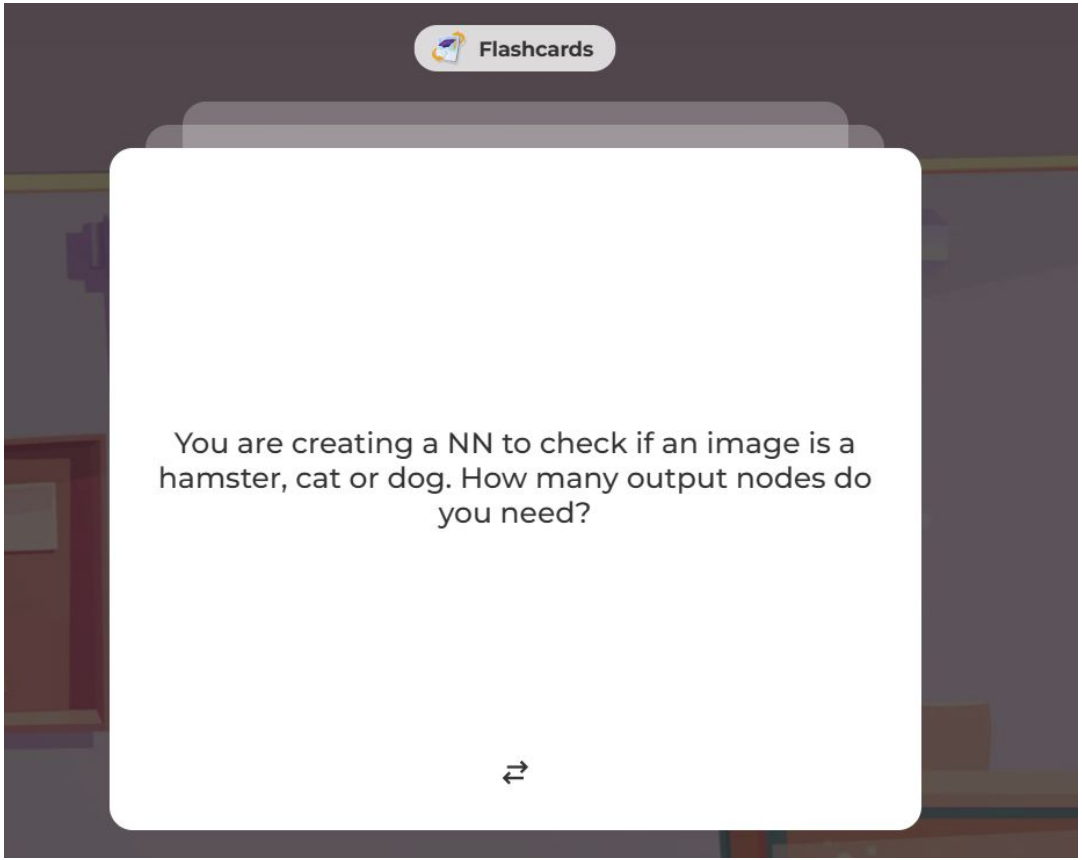


"The boundary of neural network trainability is fractal" (i.e. neural networks are incomprehensibly complex)

Agenda - Goals

- ...

Kahoot



Let’s begin todays Kahoot

Pre-Class Review

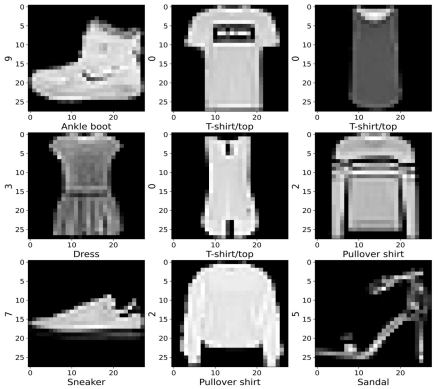
Pre-Class Review

Let’s go through pre-class content.

- A visual introduction to vector embeddings.
- LLM Embeddings Explained: A Visual and Intuitive Guide
- Tensorflow - word2vec
- OpenAI - Vector Embeddings
- Transformer Architecture Visually Explained

Natural Language Processing

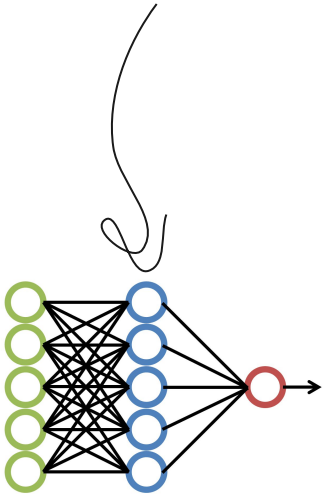
Natural Language Processing



Before we continue our exploration of LLMs, let's take a second to **pause** and consider what happened in our previous code-alongs.

We were given a series of **images** which we then fed into a neural network. This algorithm then classified our images based on the features that it learned.

However this abstracts away a lot. In which **format** was our data in before we fed it into the network? Was our data expressed as "just an image" before training started?



Natural Language Processing

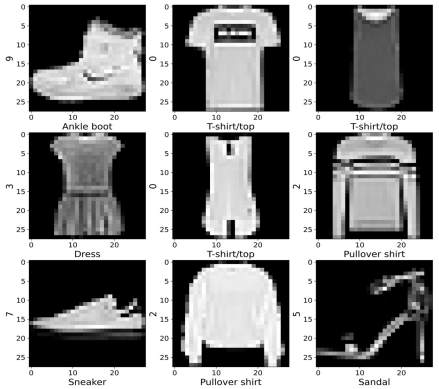
Not quite!

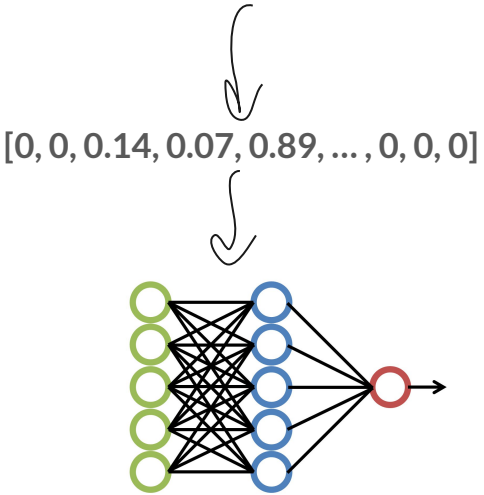
As we saw, our data was a matrix of **numerics** which expressed the **intensity of the pixel**. Additionally we…

- **flattened** our image,
- **normalized** our pixel value

which allowed the neural network to easily learn the features in our dataset. This expresses a fundamental reality of machine learning:

Every sample must be transformed into a number before we begin ML (or AI) training.



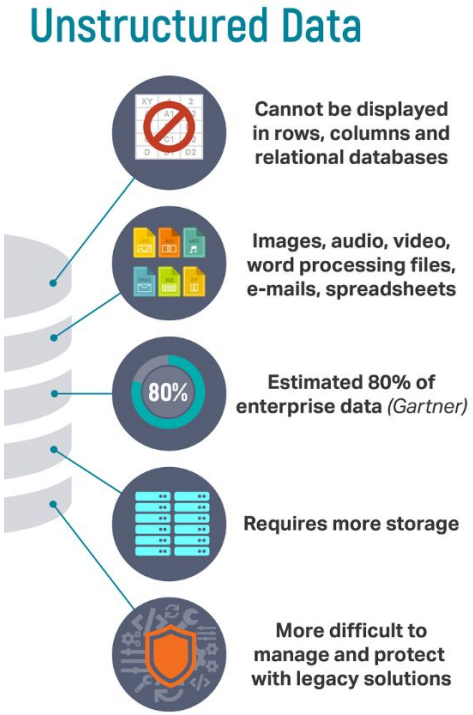


Natural Language Processing

While this sounds like an easy enough proposition to internalize, consider **all the data** that exists out in the world.

As we established already roughly 80% of all enterprise data is unstructured.

Other than images, **what are other forms of unstructured data** that we might encounter when **performing data analysis on the job?**

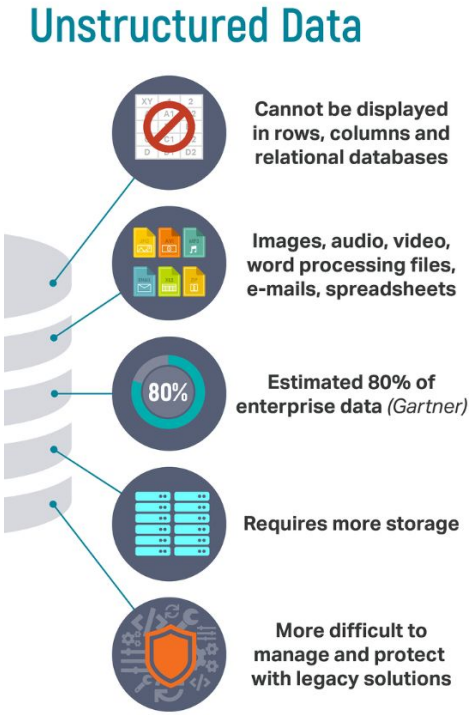


Natural Language Processing

We could apply data science techniques to…

- **Video** data: *We have video data of our basketball team moving the ball up the court, how do we find optimal strategies?*
- **Audio** data: *We have the audio recording of a lecture, how can we measure student engagement by parsing instructor-student word ratios?*
- **Text** data: *We have 5 thousand customer reviews of one product, how do we extract general feedback?*

For the remainder of this fellowship, we will focus our efforts on **solving the problem of analyzing text data**.



Natural Language Processing

…which if you continue to consider, is an innately difficult task! How would you take the following comment:

> ***Absolutely abysmal order. I asked for green and they gave me blue. I'll never buy ugly christmas sweaters from here ever again.***

And "teach" a computer to figure out the **meaning** of this person's review. **Any ideas?**

Natural Language Processing Techniques

As it turns out, computer scientists have been attempting to solve this problem for a quite a long time. Let's review the foundational techniques of analyzing a "corpus" of text.

- ***Bag of Words*** (Harris 1954)
- ***TF-IDF*** (Jones 1972)

These techniques are dated by this point, **but it doesn't mean these techniques are useless.** Let's see what their purpose is in the world of document analysis.

Bag of Words

The Bag of Words (BOW) is a **naive technique** which simply counts the frequency of words in a piece of text, and **compares frequencies to gauge similarity of text**.

The steps to this include:

1. ***Tokenize your text into words (or n-grams)***
2. ***Eliminate stop-words and punctuation***
3. ***Calculate frequencies of words***

Let's walk through an example of this

My boyfriend loves this product! **Review: 5**

This product was a waste of money. Hate it. **Review: 1**

Love it. Boyfriend hates it but what does he know? **Review: 4**

Let's take three example text files and see how the BOW technique analyzes them.

We will use **3 example reviews from Amazon and their subsequent reviews (out of 5 stars)**.

What's the first step to the BOW technique?

My boyfriend loves this product! → *["My", "boyfriend", "loves", "this", "product", "!"]*

This product was a waste of money. Hate it. → *["This", "product", "was", "a", "waste", "of", "money, ".", "Hate" "it", "."]*

Love it. Boyfriend hates it but what does he know? → *["Love", "it", ".", "Boyfriend", "hates", "it", "but", "what", "does", "he", "know", "?"]*

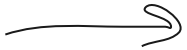
(1)

We "tokenize" our data. The process of tokenization in natural language processing involves breaking down a body of text **into smaller components**.

For example, in BOW we tokenize our sentences to **get individual words**. This also has the effect of extracting **punctuation** and **stop words** such as "the" or "this."

Do you think that these **stop words reveal any insight in our textual analysis?**

[“My”, “boyfriend”, “loves”, “this”, “product”, “!”]



[“My”, “boyfriend”, “loves”, “product”]

[“This”, “product”, “was”, “a”, “waste”, “of”, “money, “.”, “Hate” “it”, “.”]

[“product”, “waste”, “money, “Hate”]

[“Love”, “it”, “.”, “Boyfriend”, “hates”, “it”, “but”, “what”, “does”, “he”, “know”, “?”]

[“Love”, “Boyfriend”, “hates”, “what”, “does”, “he”, “know”]

(1) **(2)**

Most likely not (*although the answer could be yes*!) For that reason, we remove these words in the next step so that we only focus in on the most-important tokens.

[“My”, “boyfriend”, “loves”, “this”, “product”, “!”] → *[“My”, “boyfriend”, “loves”, “product”]*

[“This”, “product”, “was”, “a”, “waste”, “of”, “money, “.”, “Hate” “it”, “.”] → *[“product”, “waste”, “money, “Hate”]*

[“Love”, “it”, “.”, “Boyfriend”, “hates”, “it”, “but”, “what”, “does”, “he”, “know”, “?”] → *[“Love”, “Boyfriend”, “hates”, “what”, “does”, “he”, “know”]*

(1) **(2)**

For example, if we used a **2-gram (bigram)** in this BOW model, the algorithm will search for **terms that are often paired together in our dataset.**

“My boyfriend”

If we specify a larger “n”, we can search for larger phrases (3-gram)

“***Waste of money***”

Note that this will result in **sparse-vectors** (a vector with a lot of 0’s) when our documents are large enough. This will result in computationally **expensive operations for minimal data.**

Furthermore, notice that some terms make more sense **when paired with previous terms** (ex: “my boyfriend”, “waste of money”). To satisfy both needs, we often incorporate “***n-grams***” which pair together at most “***n***” number of words. **For now, we will skip this.**

[“My”, “boyfriend”, “loves”, “this”, “product”, “!”]

[“My”, “boyfriend”, “loves”, “product”]

“my”: 1
“boyfriend”: 1
“love”: 1
“product”: 1

Review: 5

[“This”, “product”, “was”, “a”, “waste”, “of”, “money, “.”, “Hate” “it”, “.”]

[“product”, “waste”, “money, “Hate”]

“product”: 1
“waste”: 1
“money”: 1
“hate”: 1

Review: 1

[“Love”, “it”, “.”, “Boyfriend”, “hates”, “it”, “but”, “what”, “does”, “he”, “know”, “?”]

[“Love”, “Boyfriend”, “hates”, “what”, “does”, “he”, “know”]

“love”: 1
“boyfriend”: 1
“hate”: 1
“what”: 1
“does”: 1
“he”: 1
“know”: 1

Review: 4

(1) (2) (3)

Lastly, we calculate the frequencies (or ratios) of each term in our reviews. To account for differences in punctuation, we ensure that we only consider the lowercase version of our words.

Also, notice that words such as “loves” and “love” will be considered different tokens. To account for this, we also typically take off all suffixes and prefixes to get the “**root token.**”

["My", "boyfriend", "loves", "this", "product", "!"]

["My", "boyfriend", "loves", "product"]

"my": 1
"boyfriend": 1
"love": 1
"product": 1

Review: 5

["This", "product", "was", "a", "waste", "of", "money, ".", "Hate" "it", "."]

["product", "waste", "money, "Hate"]

"product": 1
"waste": 1
"money": 1
"hate": 1

Review: 1

["Love", "it", ".", "Boyfriend", "hates", "it", "but", "what", "does", "he", "know", "?"]

["Love", "Boyfriend", "hates", "what", "does", "he", "know"]

"love": 1
"boyfriend": 1
"hate": 1
"what": 1
"does": 1
"he": 1
"know": 1

Review: 4

(1) (2) (3)

After calculating these "bag of word" frequencies, we can apply supervised learning techniques to figure out which terms get a "low" or "high" review.

However, you've all seen this ML algo before. What is the name of one possible technique that uses this bag of words???

[“My”, “boyfriend”, “loves”, “this”, “product”, “!”]

[“My”, “boyfriend”, “loves”, “product”]

“my”: 1
“boyfriend”: 1
“love”: 1
“product”: 1

Review: 5

[“This”, “product”, “was”, “a”, “waste”, “of”, “money, “.”, “Hate” “it”, “.”]

[“product”, “waste”, “money, “Hate”]

“product”: 1
“waste”: 1
“money”: 1
“hate”: 1

Review: 1

[“Love”, “it”, “.”, “Boyfriend”, “hates”, “it”, “but”, “what”, “does”, “he”, “know”, “?”]

[“Love”, “Boyfriend”, “hates”, “what”, “does”, “he”, “know”]

“love”: 1
“boyfriend”: 1
“hate”: 1
“what”: 1
“does”: 1
“he”: 1
“know”: 1

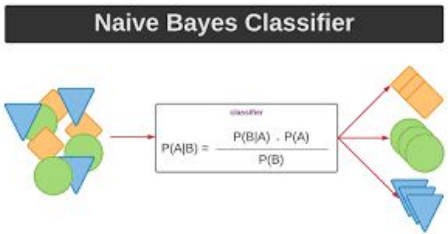
Review: 4

(1)

(2)

(3)

This is **essentially the start to Naive Bayes!**



Term Frequency-Inverse Document Frequency (TF-IDF)

While BOW is a trivial technique to interpret text, we can iterate upon this implementation via the **Term Frequency-Inverse Document Frequency** algorithm.

The steps to this include:

1. *BOW steps up to step 2*
2. *Calculate the total number of times word t appears in document d, divided by all the terms in document d (term frequency)*
3. *Calculate the total number of documents divided by the number of documents containing term t (inverse document frequency)*
4. *Multiply these two values together to calculate the tf-idf.*

[“My”, “boyfriend”, “loves”, “this”, “product”, “!”]

[“My”, “boyfriend”, “loves”, “product”]

[“This”, “product”, “was”, “a”, “waste”, “of”, “money, “.”, “Hate” “it”, “.”]

[“product”, “waste”, “money, “Hate”]

[“Love”, “it”, “.”, “Boyfriend”, “hates”, “it”, “but”, “what”, “does”, “he”, “know”, “?”]

[“Love”, “Boyfriend”, “hates”, “what”, “does”, “he”, “know”]

(1) **(2)**

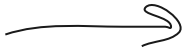
D1 TF(boyfriend)
= count of boyfriend/total words
= ¼

D1 IDF(boyfriend)
= log(count of docs/docs containing word “boyfriend”)
= log(3/2)
= 0.405

*TF-IDF(boyfriend) = 0.25 * 0.405*
= 0.101

Let’s utilize the previous example to calculate the tf-idf of **each term** in these documents. Note that one “document” is considered a separate piece of text.

["My", "boyfriend", "loves", "this", "product", "!"]



["My", "boyfriend", "loves", "product"]

*TF-IDF(boyfriend) = 0.25 * 0.405*
= 0.101

["This", "product", "was", "a", "waste", "of", "money, ".", "Hate" "it", "."]

["product", "waste", "money, "Hate"]

L2 Normalization
*= **SUM** of TF-IDF value*
= 0.4139

["Love", "it", ".", "Boyfriend", "hates", "it", "but", "what", "does", "he", "know", "?"]

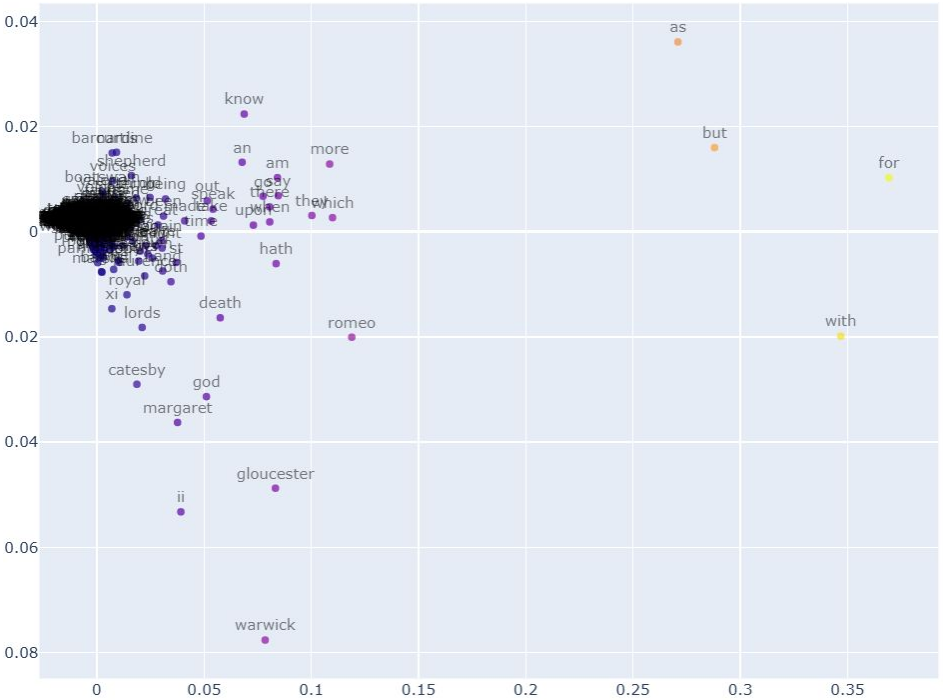
["Love", "Boyfriend", "hates", "what", "does", "he", "know"]

Normed TF-IDF
*= **0.101 / 0.4139***
= 0.244

(1) **(2)**

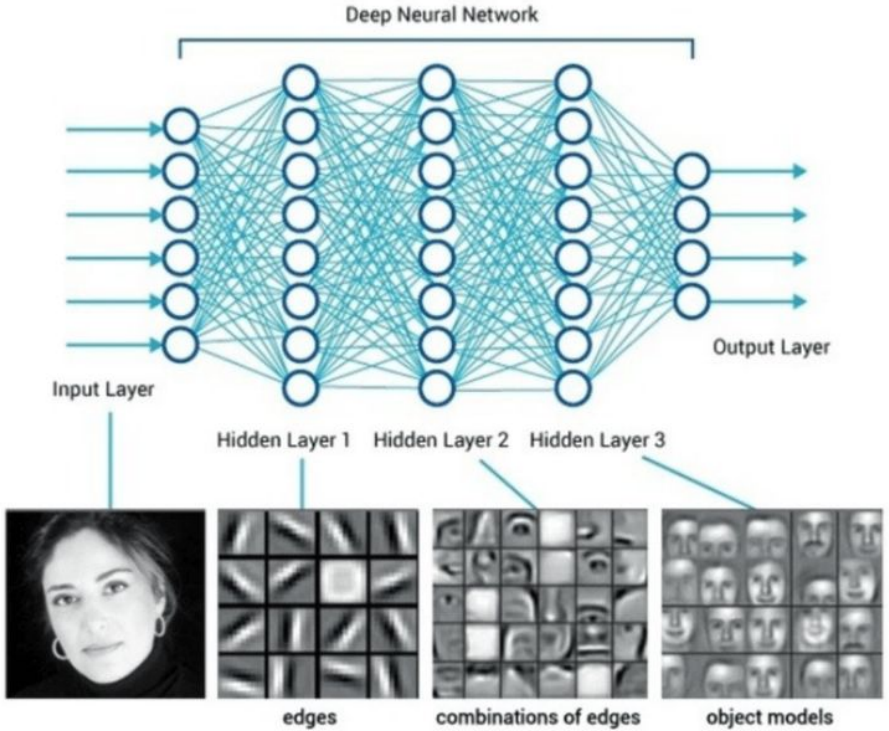
Lastly, we also often **normalize** our TF-IDF value. This ensures that we are operating on the same scale for each TF-IDF value, which subsequently allows for "fair" comparison on values.

Vector Embeddings

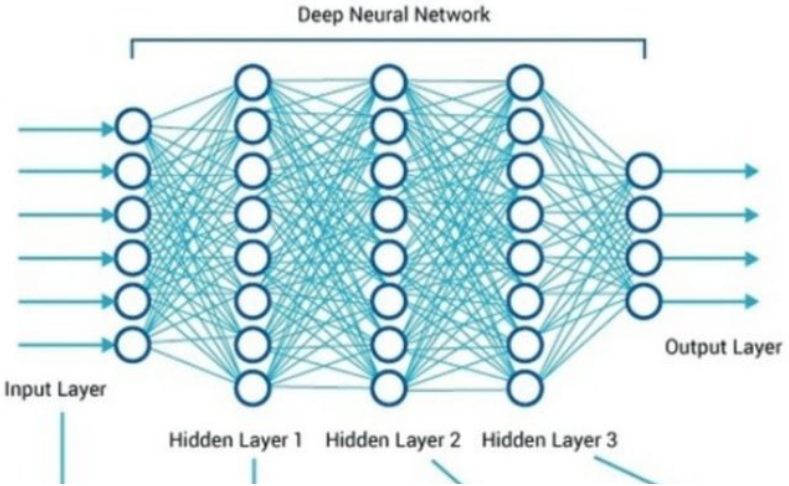


TF-IDF example showing word embeddings plotted in 2D space after dimensionality reduction

While **BOW** & **TF-IDF** are good starts & give us a quick (and **cheap**) analysis of text, these techniques are not commonly used for advanced data science applications.
Note how most words cluster around one **centroid** after applying TF-IDF.



Recall how our **neural networks** picked up the distinct features of a face (or a shoe) when creating **hidden layers** that recalculate the weights of our previous layer. **Could we somehow pick up the "distinct features" of a sentence by simply feeding in text data instead?**



Absolutely abysmal order. I asked for green and they gave me blue. I'll never buy ugly christmas sweaters from here ever again.

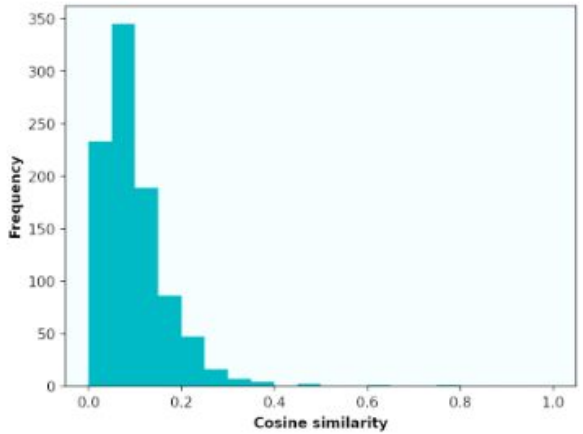
[0.53, 0.34, 0.14, … , 0.385]

As it turns out, **absolutely**! By feeding in text data and predicting for the **next word** (or the missing word), we can calculate something called a **vector embedding** which picks up on features of a word **without any hard-coded algorithm**.

word2vec similarity

For the word2vec model, here are the closest words to "dog", and the similarity distribution across all 1000 words:

word	similarity
dog	1.000000
cat	0.760946
animal	0.643801
horse	0.482581

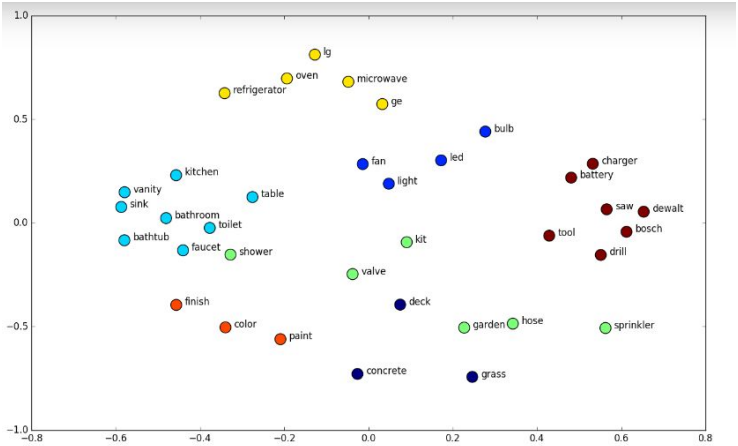


Of course, **there is no such a thing as magic in computer science**. You should always feel like you could implement a bottom-up implementation of any algorithm you work with. However, the results of these techniques “feel” like magic. Notice how training a neural network on text data embeds an “understanding” that the word dog is similar to the word “cat”, “animal”, and “horse.”

Word Embeddings - Data Representations

In summary, a word embedding is a **natural language processing technique** that takes a “corpus” (body) of text and translates it into **numerical representation in order to extract statistical text analysis out of it!**

This, in effect, allows a computer to “understand” the **meaning of a sentence.**

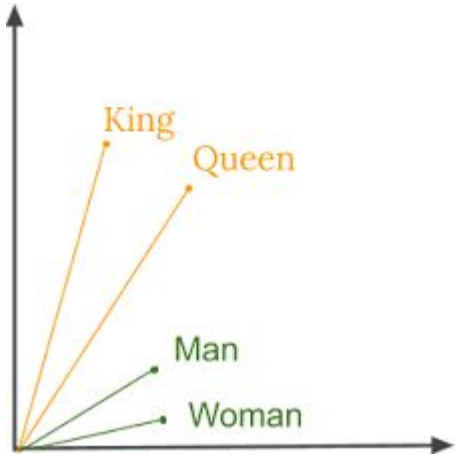


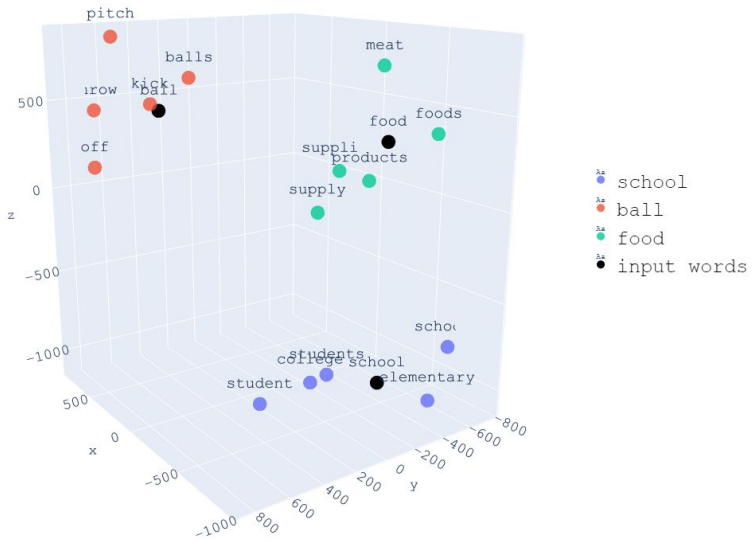
Word Embeddings - Word2Vec

Like we established, the BOW model makes no use of the proximity of words to one another (i.e. their dependence).

So instead, let's take a look at the **word2vec model** which utilizes **a two layer neural network to figure out the similarity of words in a corpus of text.**

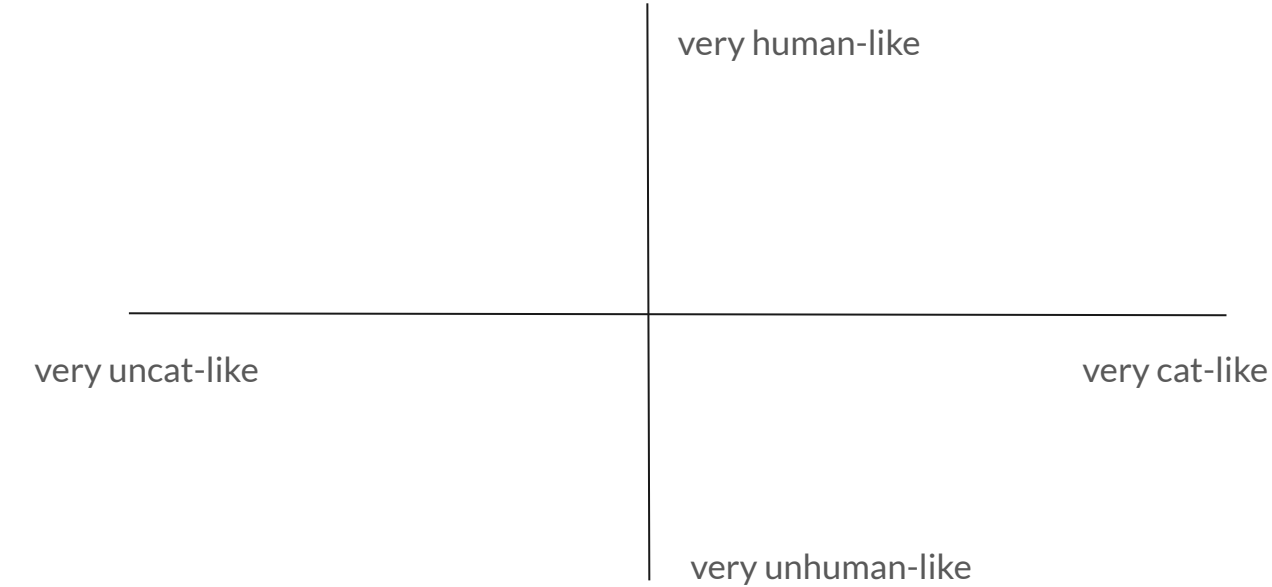
The name of this technique should give you a hint as to how it works "word to vector"





Before we dive deeper into the implementation, let's get an understanding of the output of **word2vec**.

We are aiming to generate what is known as the **semantic feature space** (the *magic place* where **abstract ideas** become **concrete representations)**



For example, let's say someone approaches you and says:

"My cat is like a person. She loves to watch TV!"

Let's forget **cultural and linguistic** lens, and simply understand why we consider this to be a "*silly*", "*humorous*", or just simply "*nonsensical*" using the **semantic space.**

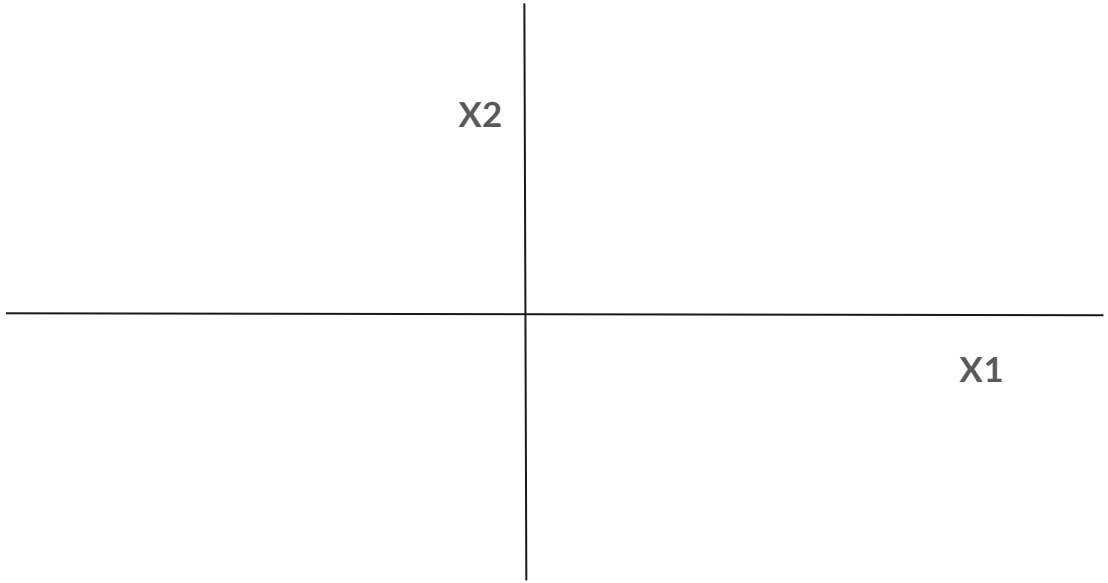
We have “cat-like” traits on the x-axis, and “human-like” traits on the y-axis. Let’s measure various activities on this scale:

- eats liver pate
- chases birds
- watches tv
- pays taxes

Where would you place these traits in the semantic feature space?

By observing wheres these data-samples exist in the **semantic feature space** we can formally observe that the statements that are more human-like exist in a cluster separate from the cat-like cluster.

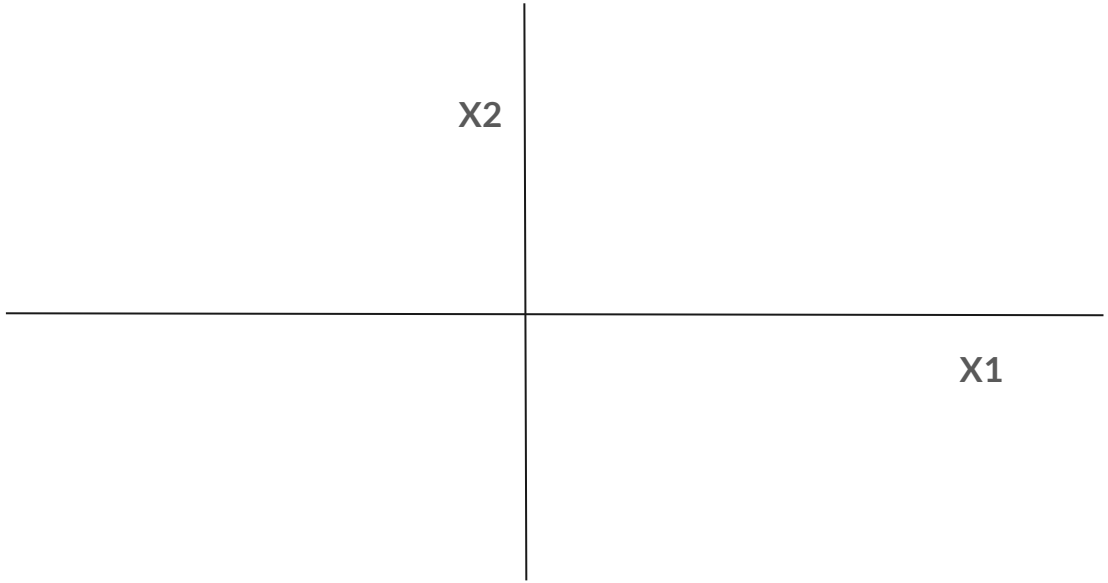
We could then determine that the statement “cat likes watching tv” is “funny” because there’s some sort of trait-switch that is occuring.



In more concrete terms, we use the **word2vec** model to transform tokens into a vector of numerals which maps meaning to our **semantic feature space.**

This results in some very interesting and clever properties…

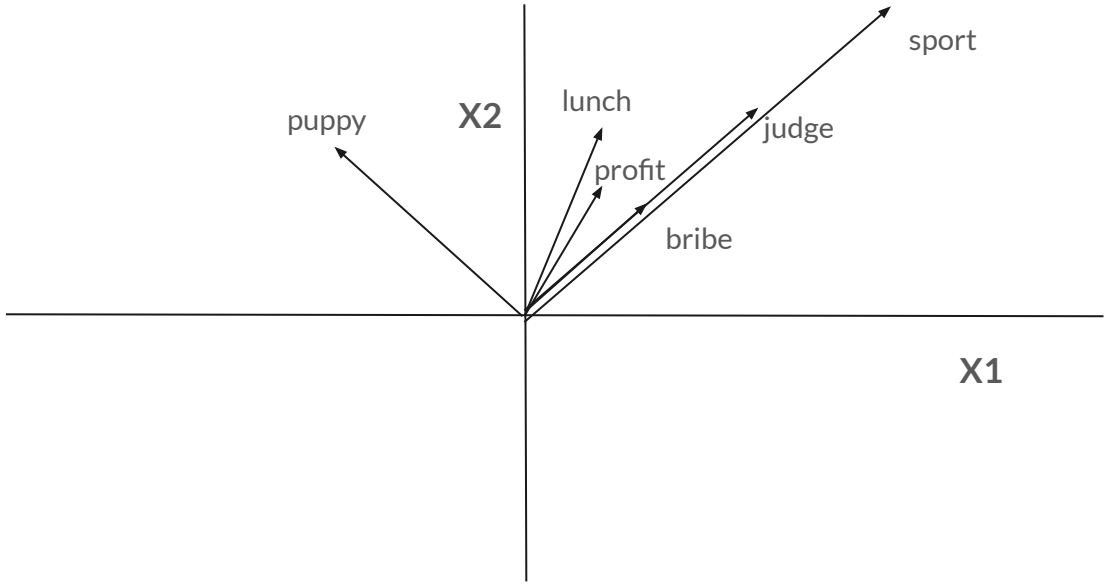
bribe → *[1, 2]*
judge → *[2, 4]*
lunch → *[1, 4]*
profit → *[1, 3]*
sports → *[3, 5]*
puppy → *[-1, 4]*



For example, let's say that we analyze a text document that contains emails from a **financial scandal** (*where a company conducted financial fraud and occasionally used code-words*)

After training our word2vec model we might get the listed vectors for the following words

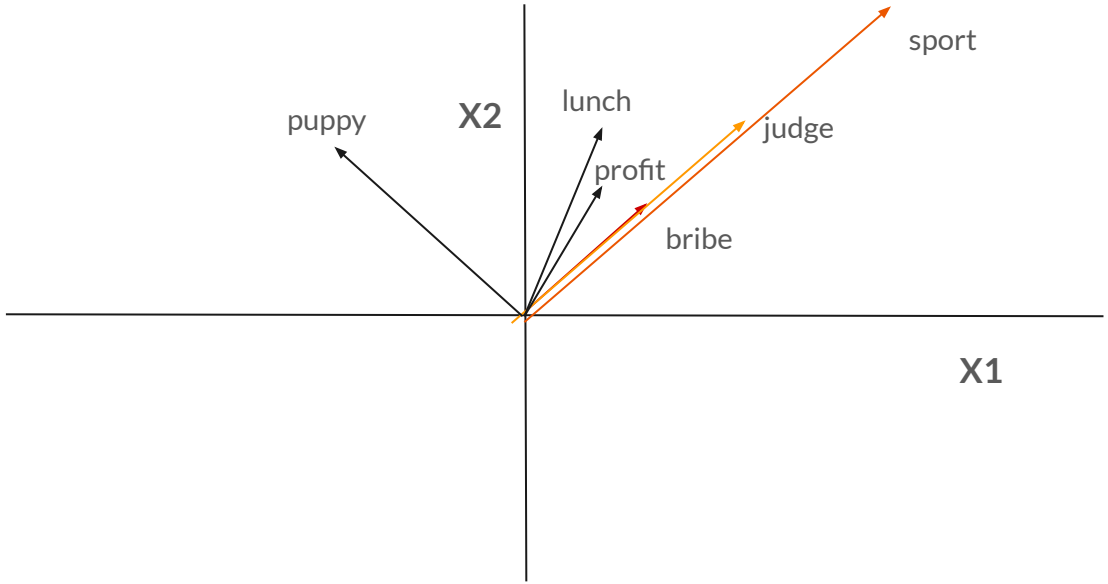
bribe → *[1, 2]*
judge → *[2, 4]*
lunch → *[1, 4]*
profit → *[1, 3]*
sports → *[3, 5]*
puppy → *[-1, 4]*



We can plot this in our semantic feature space to observe the following vectors.

Keep in mind that code-words were used to mask nefarious action. What do you notice about the direction of "**bribe**", "**judge**" and "**sport**"?

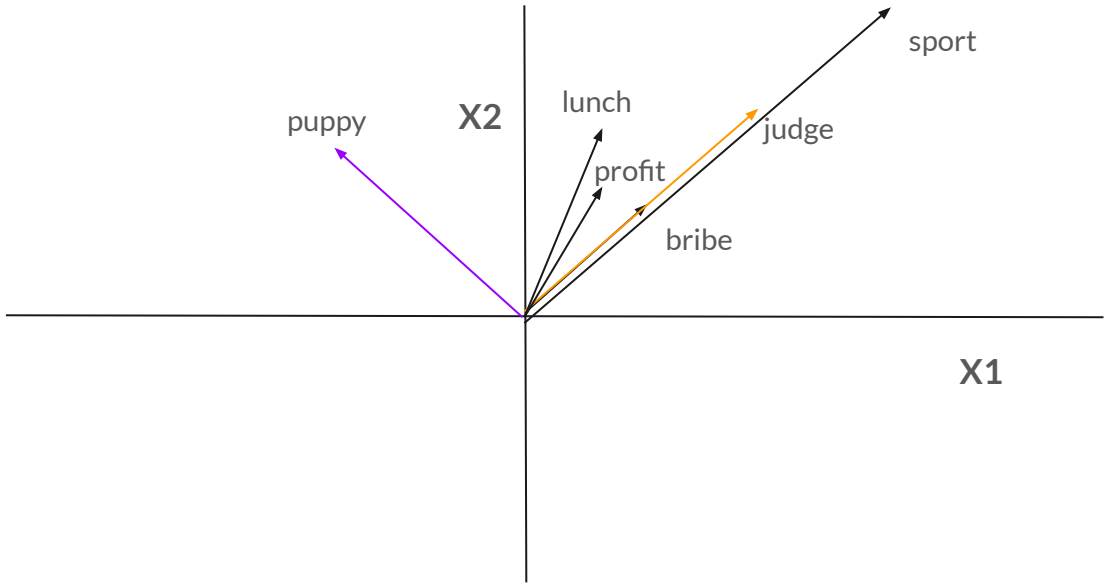
bribe → *[1, 2]*
judge → *[2, 4]*
lunch → *[1, 4]*
profit → *[1, 3]*
sports → *[3, 5]*
puppy → *[-1, 4]*



They all follow the same direction, ***aka they have cosine similarity of 1 indicating*** they are all used in similar contexts! This might indicate that they mean the same thing.

On the other hand, what do you notice about the direction of "**puppy**" and "**judge**"?

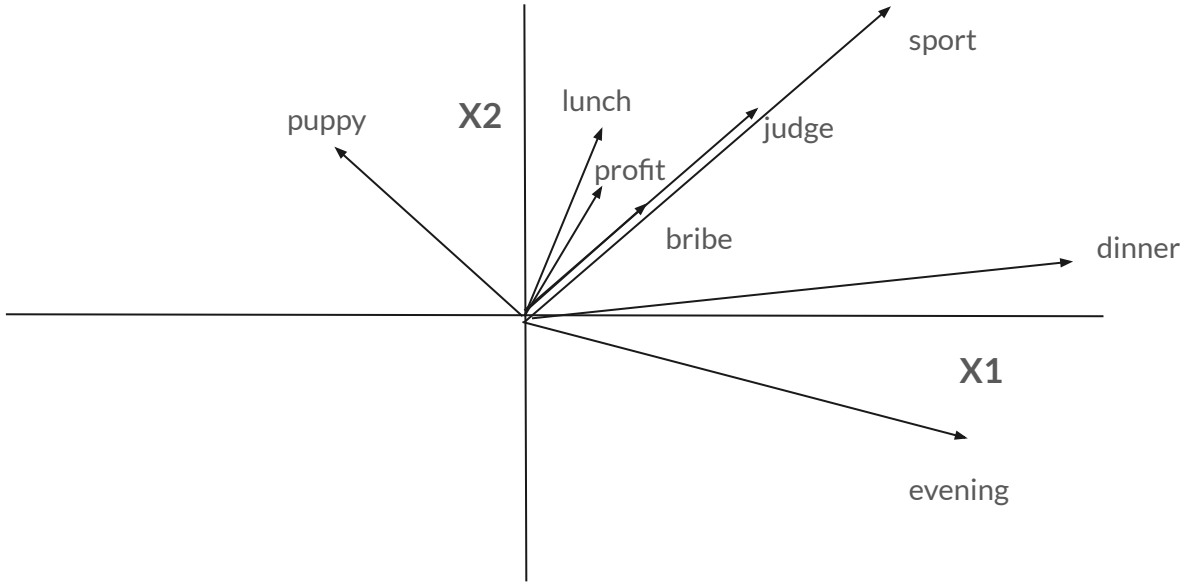
bribe → *[1, 2]*
judge → *[2, 4]*
lunch → *[1, 4]*
profit → *[1, 3]*
sports → *[3, 5]*
puppy → *[-1, 4]*



It seems that these two vectors are **orthogonal (90 degrees)** to each other, aka they have a cosine similarity of 0. In the world of vector math, orthogonality indicates "**opposite force.**"

This might indicate that the words "**puppy**" and "**judge**" don't have much in common in regards to the context they were used in.

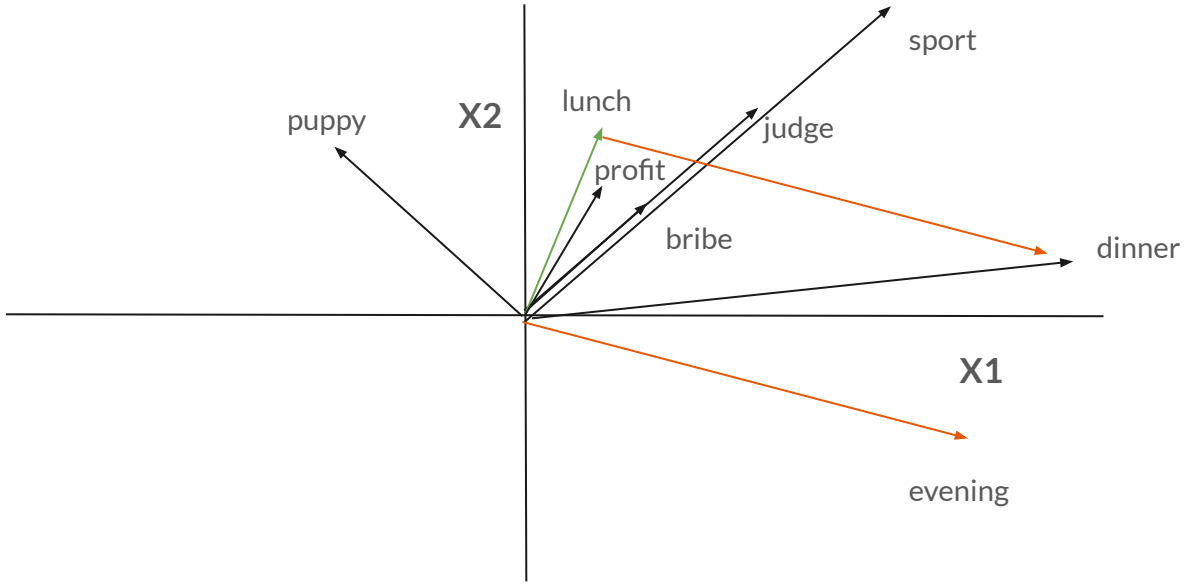
bribe → *[1, 2]*
judge → *[2, 4]*
lunch → *[1, 4]*
profit → *[1, 3]*
sports → *[3, 5]*
puppy → *[-1, 4]*
evening → *[3, -3]*
dinner → *[4, 1]*



Lastly, to demonstrate the power of word2vec transformations, let's say we have the additional data-samples "**evening**" and "**dinner**."

Try adding the vectors "lunch" + "evening" [1, 4] + [3, -3].
Which value do we get?

bribe → *[1, 2]*
judge → *[2, 4]*
lunch → *[1, 4]*
profit → *[1, 3]*
sports → *[3, 5]*
puppy → *[-1, 4]*
evening → *[3, -3]*
dinner → *[4, 1]*



[1, 4] + [3, -3] = [4, 1] (*lunch + evening = dinner*)

Even though this is a toy example, **this property actually exists in word2vec models**! Now that we have a formal idea of the semantic feature space, let's explore how its created.

Word Embeddings - Word2Vec

There are 2 ways to implement the word2vec model:

Skip-Gram:

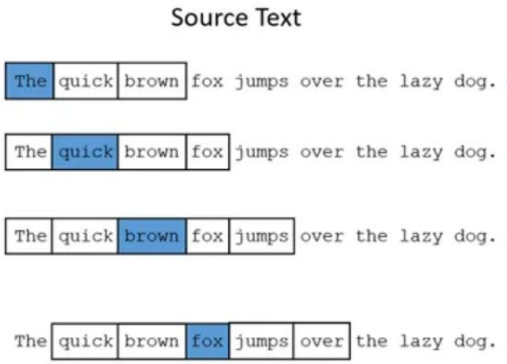
Predict **neighbors** using a specific word *w*

Continuous Bag of Words:

Predict specific word *w* using its **neighbors**

Both approaches entail looping over a body of text to train weights using a neural net.

We specify a hyperparameter called “window-size”(n) to control the number of neighbors we observe before and after our word..



Word Embeddings - Word2Vec

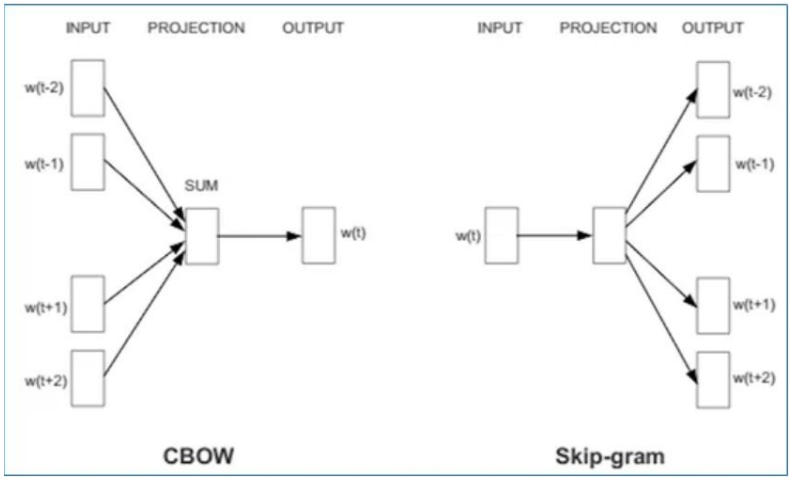
In other words…

Skip-Gram:

Use **context** to predict **target**

Continuous Bag of Words:

Use **target** to predict **context**



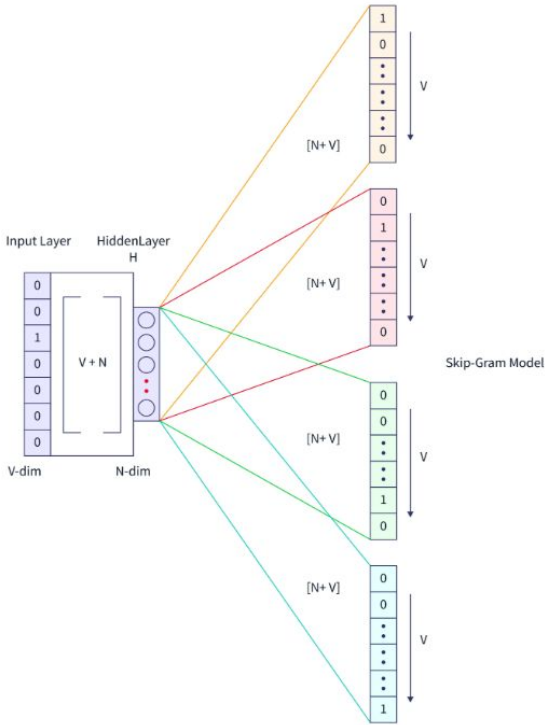
We will focus in on **skip-gram** as it is able to model infrequently used words.

Word Embeddings - Word2Vec

The skip-gram neural net architecture consists of an

- **input layer** that takes in **the target word**
- the **hidden layer which learns the weights of your context words**,
- and finally the **output layer** which predicts the **context word**

2 layers in total (recall that the input layer isn't really a layer).



Word Embeddings - Word2Vec (SkipGram)

Let's observe the steps to train a **skip-gram word2vec**

1. ***Tokenize your text into words (or n-grams)***
2. ***Eliminate stop-words and punctuation***
3. **Choose a window size of "n"**
4. **Loop through each window of your corpus**
    a. **Train your single hidden layer neural network on each (n) context words & target word**

The dog eats the bone

The cat eats fish.

The cat scratches its owner.

Let’s take three simple example text files and see how the word2vec technique analyzes them.

First, let’s tokenize this data and remove stop-words.

The dog eats the bone → *[“dog”, “eats”, “bone”]*

The cat eats fish. → *[“cat”, “eats”, “fish”]*

The cat scratches its owner. → *[“cat”, “scratches”, “owner”]*

(1)

(2)

Next, we must generate our “skip-grams” of size “n.” Keep in mind that the skip-gram looks at words that **come “n” positions after and before our target words.**

Since our dataset is limited, we’ll just use a **1-gram** (*since there is no sentence with context words greater than 2 positions away*)

The dog eats the bone → *[“dog”, “eats”, “bone”]*

The cat eats fish. → *[“cat”, “eats”, “fish”]*

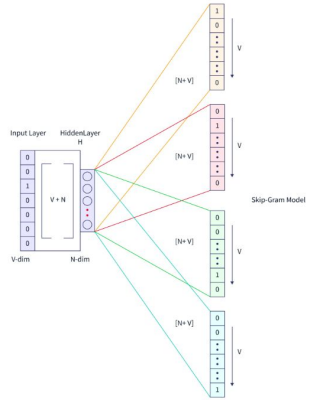
The cat scratches its owner. → *[“cat”, “scratches”, “owner”]*

(1)

(2)

(“dog”, “eats”)
(“dog”, “bone”)
(“cat”, “eats”)
(“cat”, “fish”)
(“cat”, “scratches”)
(“cat”, “owner”)
(...)

(3)



Keep in mind we must generate **all n-grams!** This means that we generate both **(“cat”, “eats”)** and **(“eats”, “cat”).**

Once we get this dataset, we set up our skip-gram architecture and loop through our skip-grams to learn our **hidden layer weights**.

["dog", "eats", "bone"]

("dog", "eats")

("dog", "bone")

["cat", "eats", "fish"]

("cat", "eats")

("cat", "fish")

["cat", "scratches", "owner"]

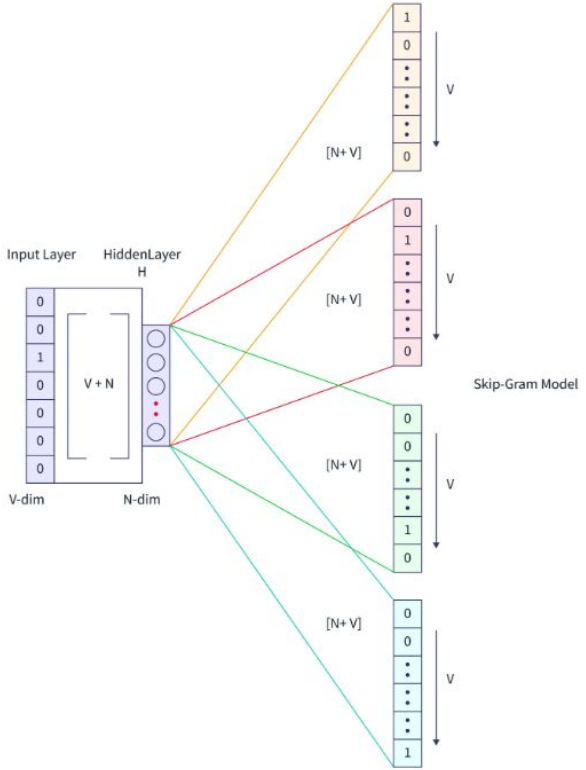
("cat", "scratches")

("cat", "owner")

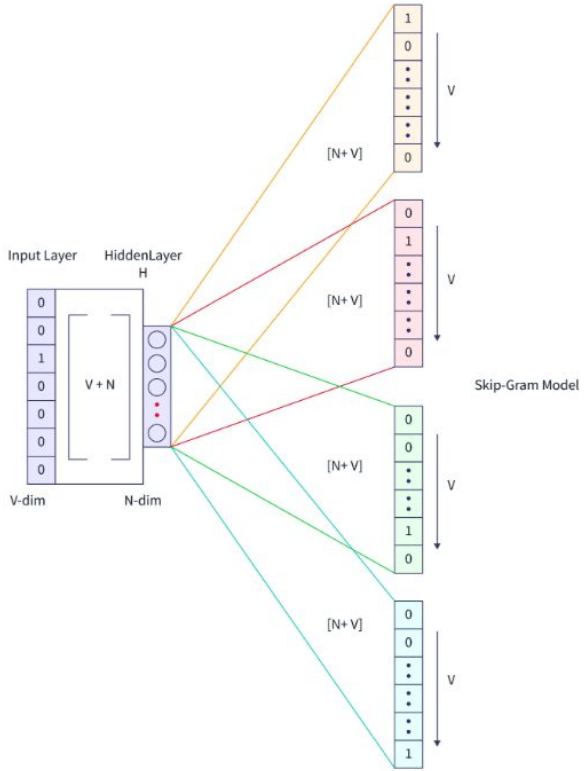
(...)

(2)

(3)



Note! We don't actually care about the **output** of the **word2vec neural network**, the **vector** of a specific target word is actually created by observing **specific weights that correspond to a specific word in the hidden layer.**



dog → *[0.04, 0.563]*
bone → *[0.0395,0.5578]*

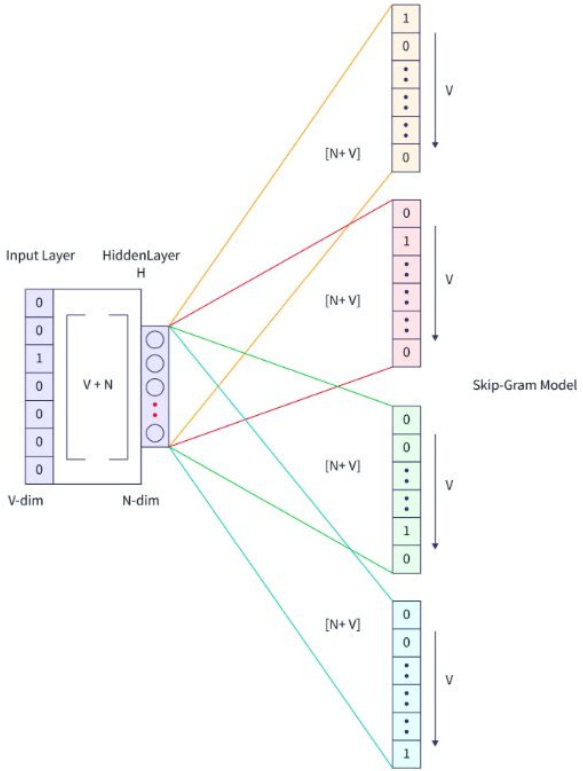
cat → *[-0.05, 0.632]*
fish → *[−0.0352,0.6269]*

Is this supervised or unsupervised learning?

Note: often times the vectors we create are extremely dimensional. **Which technique can we use to reduce these dimensions?**

For example, we take a look at the **weights of the word2vec hidden layer and notice** the following vectors after training.

These vectors represent some **learned expression of semantic and syntactic similarity**!
The neural network has learned some new features that express meaning! **Amazing.**



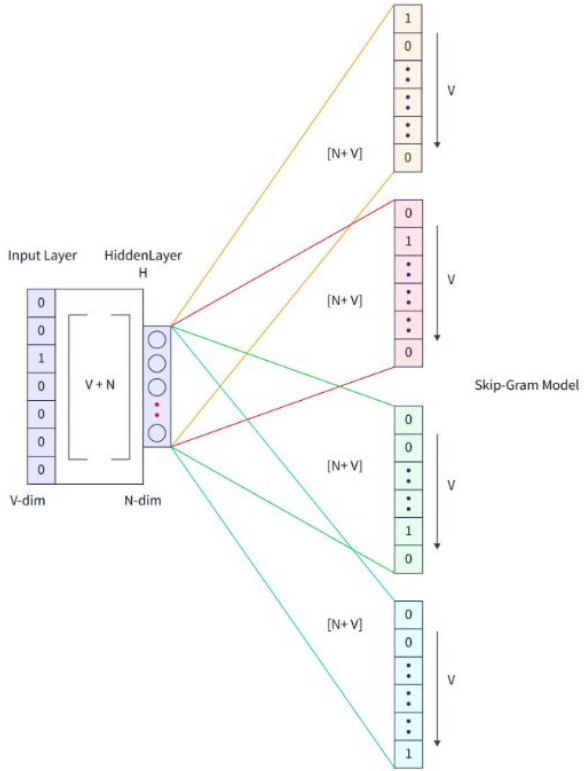
dog → *[0.04, 0.563]*
bone → *[0.0395,0.5578]*

cat → *[-0.05, 0.632]*
fish → *[−0.0352,0.6269]*

You might be wondering: "what do these numbers mean??"

It's **impossible to know this** at first glance, and most likely requires EDA to start to get an understanding.

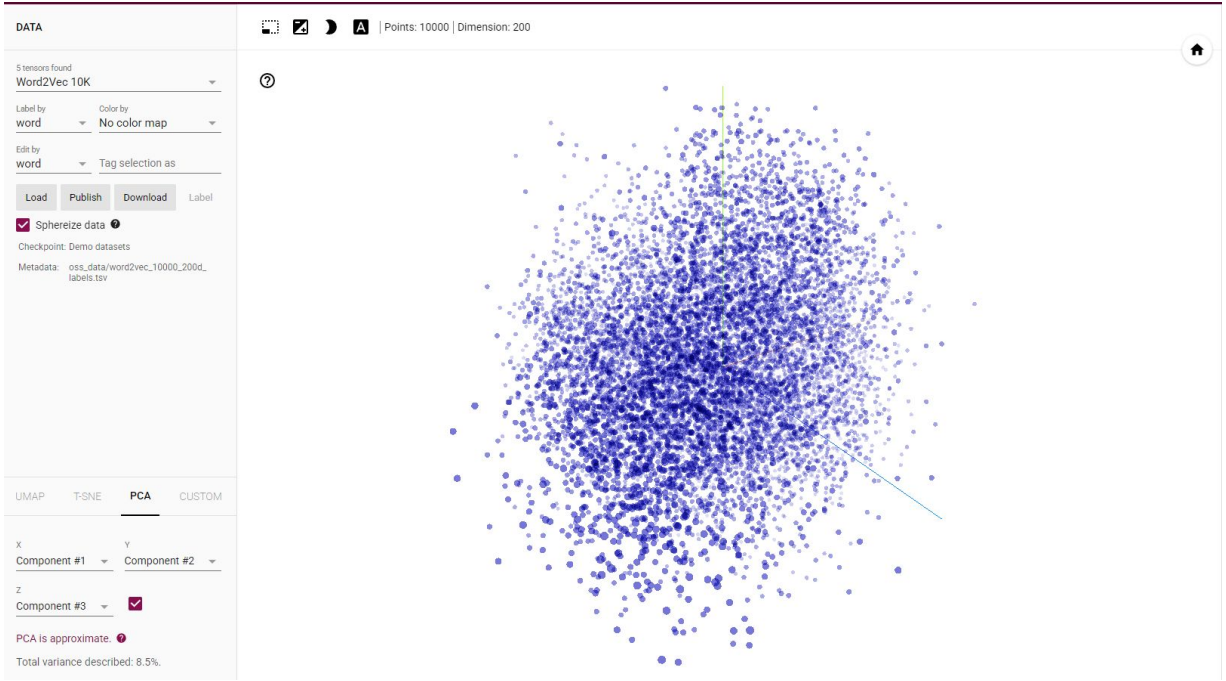
Once we get these numeric features, we can use these in a variety of supervised and unsupervised applications in order to discover more about the meaning of a sentence or document.



The king james bible fed into a word2vec model

```
{'god': ['true', 'effect', 'abound', 'waiteth', 'open'],
 'jesus': ['grievous', 'windows', 'glory', 'lamb', 'did'],
 'noah': ['teacheth', 'walls', 'residue', 'gathered', 'awake']
 'egypt': ['seir', 'beersheba', 'grievous', '108', 'synagogues
 'john': ['gilead', 'fatherless', 'breadth', 'dwell', 'forasmu
 'gospel': ['seventy', 'cause', 'boards', 'clothed', 'neverthe
 'moses': ['meet', 'reach', 'youth', 'died', 'scatter'],
```

We can use these vectors to notice similarities amongst words to form clusters!!!



To get an idea of the data that word2vec models output, check out the following link: https://projector.tensorflow.org/

Embedding Everything

The dog eats the bone → *["dog", "eats", "bone"]*

The cat eats fish. → *["cat", "eats", "fish"]*

The cat scratches its owner. → *["cat", "scratches", "owner"]*

(1)

(2)

("dog", "eats")

("dog", "bone")

("cat", "eats")

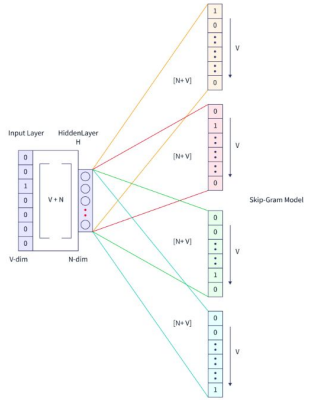
("cat", "fish")

("cat", "scratches")

("cat", "owner")

(...)

(3)



So far, we've seen word2vec's ability to learn the **vectors of english-language text.**

Do you think this could also apply to other languages with **different grammatical structure**???

собака ест кость

[“собака”, “ест”, “кость”]

(“собака”, “ест”)

(“собака”, “кость”)

кот ест рыбу

[“кот”, “ест”, “рыбу”]

(“кот”, “ест”)

(“кот”, “рыбу”)

кот царапает своего хозяина

[“кот”, “царапает”, “хозяина”]

(“кот”, “царапает

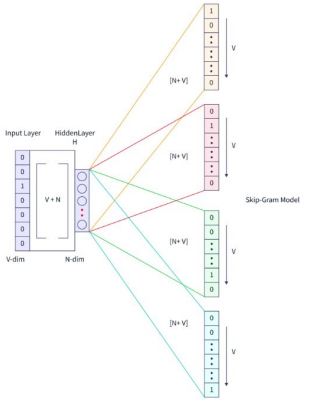
(“кот”, “хозяина”

(...)

(1)

(2)

(3)



Absolutely! We are simply taking note of the data that exists already in order to learn numerical representations.

Regardless of which language you use, **word2vec will always learn weights**!

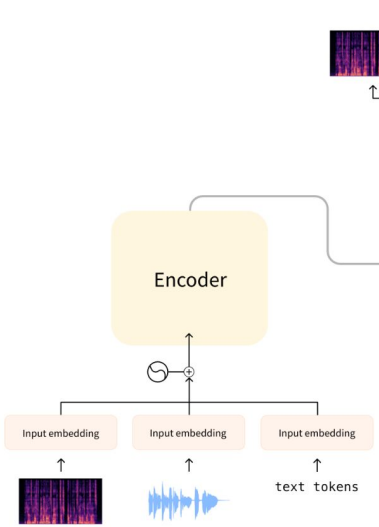
In fact, this isn’t just limited to text either…

Transformer Embeddings

After the explosion of interest (*and investment of capital*) in the transformer architecture around 2022, the transformer architecture has been used to generate **vector embeddings** for things like:

- *Image data*
- *Audio data*
- *Time series data*
- *EEG data (electrical brain activity)*

Which has led to some pretty **novel projects** such as attempting dolphin translation and "mind-reading" We will explore the application of this to music data in **Wednesday's lab.** **Note that this architecture is different than the simple word2vec model**.



[0.04. 0.28, ..., 0.03]

[0.08. 0.32, ..., 0.26]

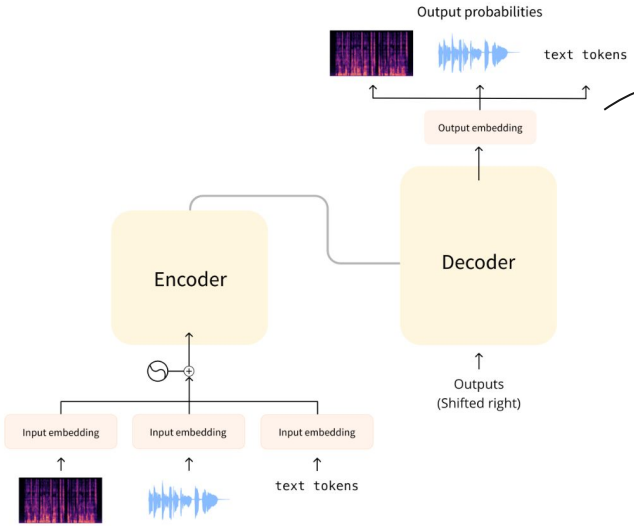
[0.92. 0.10, ..., 0.02]

As we will see tomorrow, the transformer allows us to take all types of data and generate **vector embeddings** that describe the "**essence**" of a sample of data which can then be transformed into **another** form of data…

[0.04. 0.28, ..., 0.03]

[0.08. 0.32, ..., 0.26]

[0.92. 0.10, ..., 0.02]



blend of gritty realism and hopeful idealism

hypnotic, house-driven anthem that contrasts an upbeat groove with a poignant, socially conscious message

haunting, psychedelic descent into existential despair and liberation

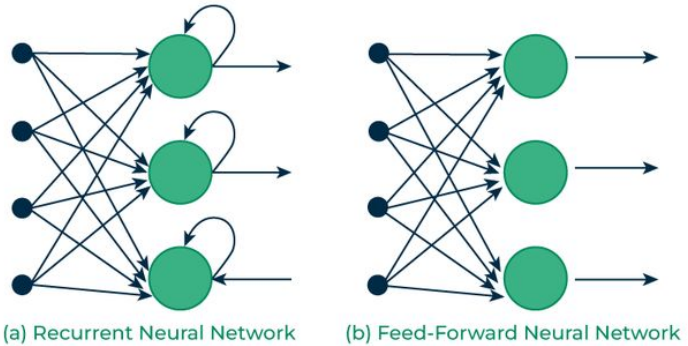
For example, the highly dimensional vectors of music data can then be decoded into **audio labels** which can then be subsequently used for further machine learning/data analysis.

End of Class Announcements

Tomorrow

The Transformer Architecture

- *What is an **attention** mechanism?*
- *How do we create **embeddings** from the Transformer*
- *How can we use Python packages to implement the **transformer**?*



How do we arrange perceptrons to create a transformer?