Sequencing Legal DNA

NLP for Law and Political Economy

5. Neural Nets and Word Embeddings

Machine Learning Produces Representations of the Data

- Text classifiers produce $\hat{\boldsymbol{y}}_i = f(\boldsymbol{x}_i; \hat{\theta})$, a vector of predicted probabilities across classes for each document i.

Machine Learning Produces Representations of the Data

- Text classifiers produce $\hat{\boldsymbol{y}}_i = f(\boldsymbol{x}_i; \hat{\theta})$, a vector of predicted probabilities across classes for each document i.
- This vector is a compressed representation of the outcome-predictive text features $\boldsymbol{x}_i$

Machine Learning Produces Representations of the Data

- Text classifiers produce $\hat{\boldsymbol{y}}_i = f(\boldsymbol{x}_i; \hat{\theta})$, a vector of predicted probabilities across classes for each document i.
- This vector is a compressed representation of the outcome-predictive text features $\boldsymbol{x}_i$
  - $\boldsymbol{x}_i$ is itself a compressed representation of the unprocessed document $\mathcal{D}_i$.

Machine Learning Produces Representations of the Data

- Text classifiers produce $\hat{\boldsymbol{y}}_i = f(\boldsymbol{x}_i; \hat{\theta})$, a vector of predicted probabilities across classes for each document i.
- This vector is a compressed representation of the outcome-predictive text features $\boldsymbol{x}_i$
  - $\boldsymbol{x}_i$ is itself a compressed representation of the unprocessed document $\mathcal{D}_i$.
- Correspondingly: the parameters $\hat{\theta}$ can also be understood as a compressed (or "learned") representation:
  - it contains information about the training corpus, the text features, and the outcomes.

Information in $\hat{\theta}$

- Say we train a multinomial logistic regression on a bag-of-words representation of the documents.
- Let θ be the learned matrix of parameters relating words to outcomes:
  - It contains n_y columns $= n_x$-vectors representing the outcome classes.

Information in $\hat{\theta}$

- Say we train a multinomial logistic regression on a bag-of-words representation of the documents.
- Let θ be the learned matrix of parameters relating words to outcomes:
  - It contains n_y columns $=$ n_x-vectors representing the outcome classes.
  - It contains n_x rows $=$ n_y-vectors representing each word in the vocabulary.

Information in $\hat{\theta}$

- Say we train a multinomial logistic regression on a bag-of-words representation of the documents.
- Let θ be the learned matrix of parameters relating words to outcomes:
  - It contains n_y columns $=$ n_x-vectors representing the outcome classes.
  - It contains n_x rows $=$ n_y-vectors representing each word in the vocabulary.
- How to use this?
  - could cluster column vectors to understand which outcomes are similar/related.
  - could cluster row vectors to understand which features are similar/related.

Preview of Word Embeddings

- Let's say $\boldsymbol{x}_i$ is a bag-of-words representation for document i with length n_i. We can write

$$\boldsymbol{x}_i = \frac{1}{n_i} \sum_{l=1}^{n_i} \boldsymbol{x}_i^{[l]}$$

  - l indexes words in the the document
  - each vector $\boldsymbol{x}_i^{[l]}$ is an n_x-dimensional one-hot vector – all entries are zero except the single entry corresponding to the word at l, which is 1.

Preview of Word Embeddings

- Let's say $\boldsymbol{x}_i$ is a bag-of-words representation for document i with length n_i. We can write

$$\boldsymbol{x}_i = \frac{1}{n_i}\sum_{l=1}^{n_i} \boldsymbol{x}_i^{[l]}$$

  - l indexes words in the the document
  - each vector $\boldsymbol{x}_i^{[l]}$ is an n_x-dimensional one-hot vector – all entries are zero except the single entry corresponding to the word at l, which is 1.

- Now let $\theta^{[l]}$ be the row of θ corresponding to the word w_l. We can write

$$\hat{\boldsymbol{y}}_i = \frac{1}{n_i}\sum_{l=1}^{n_i} \theta^{[l]}$$

  the sum of the n_y-dimensional word representations (the row vectors from above).

  - this is called the "continuous bag of words (CBOW)" representation.
  - θ is a word embedding matrix.

Outline

Intro to Neural Nets

Practicalities

Autoencoders

Embedding Layers

Word Embeddings

Kozlowski, Evans, and Taddy (2019)

“Neural Networks”

- “Neural”:
  - nothing like brains

"Neural Networks"

- "Neural":
  - nothing like brains
- "Networks":
  - nothing to do with "networks" as normally understood – in particular, nothing to do with network theory in math or social science.

Recent History

- NNs frequently outperform other ML techniques on large and complex problems.

Recent History

- NNs frequently outperform other ML techniques on large and complex problems.
- Increase in computing power makes them computationally tractable, graphical processing units (GPUs, designed for video games) give you over 100x performance gain over CPUs.

Recent History

- NNs frequently outperform other ML techniques on large and complex problems.
- Increase in computing power makes them computationally tractable, graphical processing units (GPUs, designed for video games) give you over 100x performance gain over CPUs.
- Training algorithms have improved – small tweaks have made a huge impact.

Recent History

- NNs frequently outperform other ML techniques on large and complex problems.
- Increase in computing power makes them computationally tractable, graphical processing units (GPUs, designed for video games) give you over 100x performance gain over CPUs.
- Training algorithms have improved – small tweaks have made a huge impact.
- Some theoretical limitations of NNs have turned out to be benign in practice – for example, they work well on non-convex functions.

Will it last?

- Three key principles of deep learning will persist:
  - **Simplicity**
    - feature engineering is obsolete
    - complex, brittle, engineering-heavy pipelines replaced with simple, end-to-end trainable models, composed of 5-6 tensor operations.

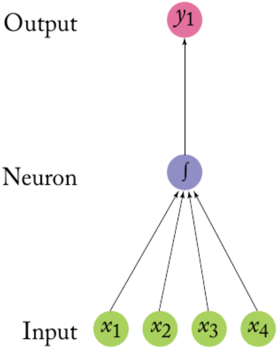
Will it last?

- Three key principles of deep learning will persist:
  - **Simplicity**
    - feature engineering is obsolete
    - complex, brittle, engineering-heavy pipelines replaced with simple, end-to-end trainable models, composed of 5-6 tensor operations.
  - **Scalability**
    - amenable to parallelization on GPUs or TPUs (tensor processing units)
    - trained on batches of data, so can be scaled to datasets of arbitrary size.

Will it last?

- Three key principles of deep learning will persist:
  - **Simplicity**
    - feature engineering is obsolete
    - complex, brittle, engineering-heavy pipelines replaced with simple, end-to-end trainable models, composed of 5-6 tensor operations.
  - **Scalability**
    - amenable to parallelization on GPUs or TPUs (tensor processing units)
    - trained on batches of data, so can be scaled to datasets of arbitrary size.
  - **Versatility and reusability**
    - can be trained on additional data without restarting from scratch, therefore amenable for continuous online learning.
    - deep-learning models are repurposable and thus reusable

A "Neuron"



- A neuron multiplies each input by its weight, sums them, applies a non-linear function to the result, and passes the output.
  - e.g., the $\int$ shape indicates a sigmoid transformation.

In Notation

- The simplest neural network is called a perceptron:

$$\text{MLP0}(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{\omega}$$

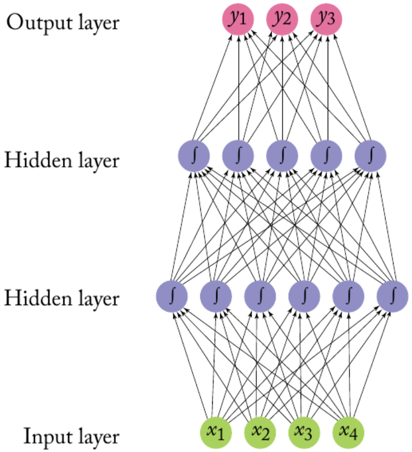
$$\boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega} \in \mathbb{R}^{n_x \times n_y}$$

  here, $\boldsymbol{\omega}$ is the matrix of weights in the layer.
- In more standard notation, there would be an additional constant (or "bias") term:

$$\text{MLP0}(\boldsymbol{x}) = \alpha + \boldsymbol{x} \cdot \boldsymbol{\omega}$$

  - We leave it out by assuming that $\boldsymbol{x}$ is de-meaned or has an extra column of ones.

A Feed-Forward Neural Network



- A feed-forward network is simply a stack of linear models, separated by non-linear functions.

Multi-Layer Perceptron

- An multi-layer perceptron (MLP) with one hidden layer is

$$\text{MLP1}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{x} \cdot \omega_1) \cdot \boldsymbol{\omega}_2$$

$$\boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega}_1 \in \mathbb{R}^{n_x \times n_1}, \boldsymbol{\omega}_2 \in \mathbb{R}^{n_1 \times n_y},$$

  - n_1 = dimensionality in first (and only) hidden layer
  - $\boldsymbol{\omega}_1$ = set of learnable weights for the first linear transformation of the inputs.
  - $\boldsymbol{g}(\cdot)$ = an element-wise non-linear function (an "activation function")
  - $\boldsymbol{\omega}_2$ = weights on the second linear transformation leading to the output.

Multi-Layer Perceptron

- An multi-layer perceptron (MLP) with one hidden layer is

$$\text{MLP1}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{x} \cdot \boldsymbol{\omega}_1) \cdot \boldsymbol{\omega}_2$$

$$\boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega}_1 \in \mathbb{R}^{n_x \times n_1}, \boldsymbol{\omega}_2 \in \mathbb{R}^{n_1 \times n_y},$$

  - n_1 = dimensionality in first (and only) hidden layer
  - $\boldsymbol{\omega}_1$ = set of learnable weights for the first linear transformation of the inputs.
  - $\boldsymbol{g}(\cdot)$ = an element-wise non-linear function (an "activation function")
  - $\boldsymbol{\omega}_2$ = weights on the second linear transformation leading to the output.
- MLP1 can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^n$, and any mapping from one finite discrete space to another finite discrete space (Hornik et al 1989, Cybenko 1989).
  - But MLP1 would have to be exponentially large in some cases (Telgarsky 2016) .

Two hidden layers

- Adding a second hidden layer gives

$$\text{MLP2}(\boldsymbol{x}) = \boldsymbol{g}_2(\boldsymbol{g}_1(\boldsymbol{x} \cdot \boldsymbol{\omega}_1) \cdot \boldsymbol{\omega}_2) \cdot \boldsymbol{\omega}_3$$

$$\boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega}_1 \in \mathbb{R}^{n_x \times n_1}, \boldsymbol{\omega}_2 \in \mathbb{R}^{n_1 \times n_2}, \boldsymbol{\omega}_3 \in \mathbb{R}^{n_2 \times n_y}$$

  - n_2= number of neurons in second hidden layer.

Two hidden layers

- Adding a second hidden layer gives

$$\text{MLP2}(\boldsymbol{x}) = \boldsymbol{g}_2(\boldsymbol{g}_1(\boldsymbol{x} \cdot \boldsymbol{\omega}_1) \cdot \boldsymbol{\omega}_2) \cdot \boldsymbol{\omega}_3$$

$$\boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega}_1 \in \mathbb{R}^{n_x \times n_1}, \boldsymbol{\omega}_2 \in \mathbb{R}^{n_1 \times n_2}, \boldsymbol{\omega}_3 \in \mathbb{R}^{n_2 \times n_y}$$

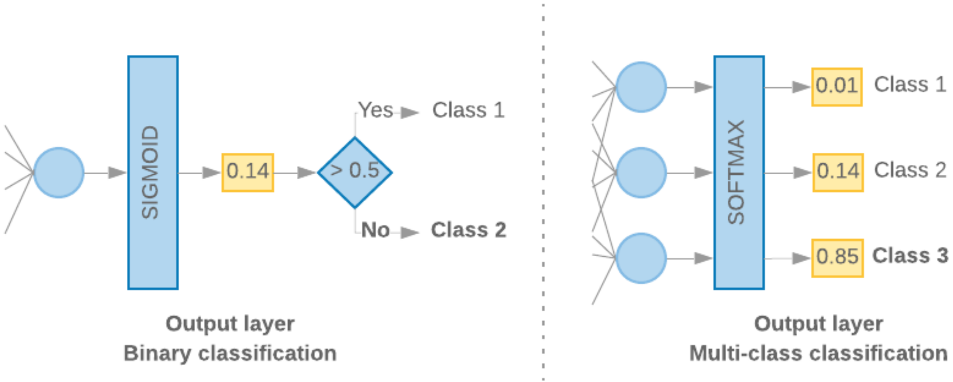
  - n_2= number of neurons in second hidden layer.
- MLP2 can be written in the following decomposed notation:

$$\text{MLP2}(\boldsymbol{x}) =$$

$$\begin{aligned} \boldsymbol{h}_1 &= \boldsymbol{g}_1(\boldsymbol{x} \cdot \omega_1) \\ \boldsymbol{h}_2 &= \boldsymbol{g}_2(\boldsymbol{h}_1 \cdot \omega_2) \\ \boldsymbol{y} &= \boldsymbol{h}_2 \cdot \omega_3 \end{aligned}$$

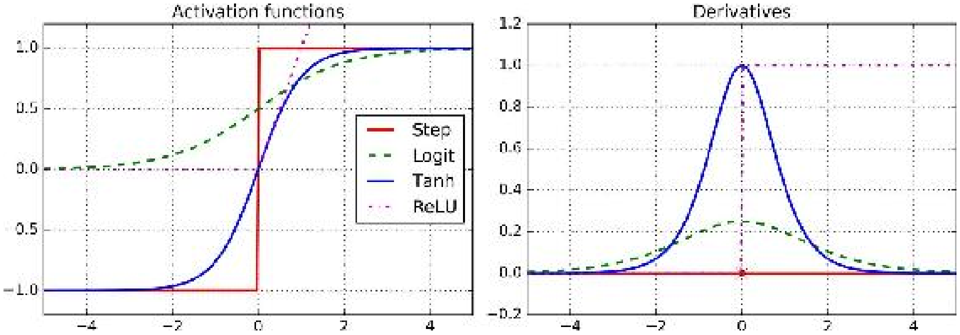
where $\boldsymbol{h}_l$ give hidden layers.

Constructing the Last Layer



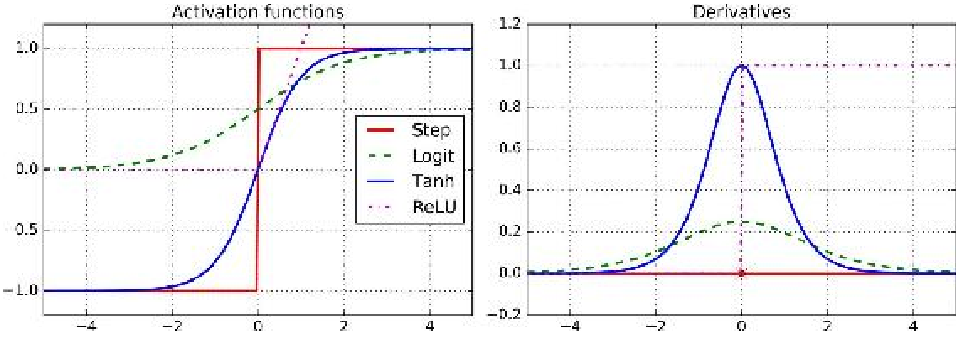
- MLPs will output a probability distribution across output classes.
  - can also output a real number, which would make a regression model.

What to pick for $g(\,)$



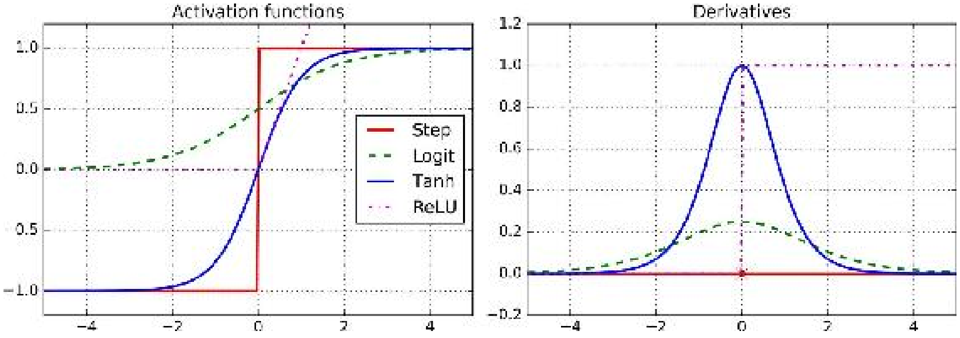
- logistic function: $\text{logit}(z) = \frac{1}{1+\exp(-z)}$

What to pick for $g(\ )$



- logistic function: $\text{logit}(z) = \frac{1}{1+\exp(-z)}$
- hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$
  - ranges between -1 and 1 (rather than between 0 and 1, as the case with the logistic)
  - centered on zero, can speed up convergence

What to pick for $g(\ )$



- logistic function: $\text{logit}(z) = \frac{1}{1+\exp(-z)}$
- hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$
  - ranges between -1 and 1 (rather than between 0 and 1, as the case with the logistic)
  - centered on zero, can speed up convergence
- ReLU (rectified linear unit) function: $\max\{0, z\}$,
  - deceptively simple, fast to compute, and very effective in practice
  - gradient does not saturate to zero for large values (but is flat below zero)

Outline

Intro to Neural Nets

Practicalities

Autoencoders

Embedding Layers

Word Embeddings

Kozlowski, Evans, and Taddy (2019)

MLP baseline for Text Classification

Google Developers Advice

1. Calculate the number of samples/number of words per sample ratio.
2. If this ratio is less than 1500, tokenize the text as n-grams and use a simple multi-layer perceptron (MLP) model to classify them.

- In the case of N-grams models, Google testers found that MLPs tended to out-perform logistic regression and gradient boosting machines.

Keras Basics

- See the Geron book and sample notebooks for Keras examples.
- "Dense" layer is the DNN baseline – means that all neurons are connected.

Keras Basics

- See the Geron book and sample notebooks for Keras examples.
- "Dense" layer is the DNN baseline – means that all neurons are connected.
- Output layer:
  - for regression, do not use an activation function
  - for binary classification, use `activation='sigmoid'`
  - for multi-class classification, use `activation='softmax'`

Loss function and metrics

- Loss function:
  - for regression, use `mean_squared_error`
  - for binary classification, use `binary_crossentropy`
  - for multi-class classification, use `sparse_categorical_crossentropy`

Loss function and metrics

- Loss function:
  - for regression, use `mean_squared_error`
  - for binary classification, use `binary_crossentropy`
  - for multi-class classification, use `sparse_categorical_crossentropy`
- Metrics:
  - for classification, can use accuracy and F_1
  - for regression, use R^2

Tuning NN Hyperparameters

- Number of hidden layers:
  - having a single hidden layer will generally give decent results.
    - more layers with fewer neurons can recover hierarchical relations and complex functions
    - for text classification, try one or two hidden layers as a baseline.

Tuning NN Hyperparameters

- Number of hidden layers:
  - having a single hidden layer will generally give decent results.
    - more layers with fewer neurons can recover hierarchical relations and complex functions
    - for text classification, try one or two hidden layers as a baseline.
- Number of neurons:
  - a common practice is to set neuron counts like a funnel, with fewer and fewer neurons at each level
  - or just pick 128 neurons per layer
  - overall, better to have too many neurons, and use regularization

Tuning NN Hyperparameters

- Number of hidden layers:
  - having a single hidden layer will generally give decent results.
    - more layers with fewer neurons can recover hierarchical relations and complex functions
    - for text classification, try one or two hidden layers as a baseline.
- Number of neurons:
  - a common practice is to set neuron counts like a funnel, with fewer and fewer neurons at each level
  - or just pick 128 neurons per layer
  - overall, better to have too many neurons, and use regularization
- Activation functions:
  - use ReLU in hidden layers

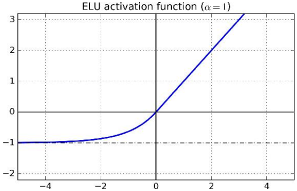
Xavier and He Initialization

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

- Connection weights should be initialized randomly according to a uniform distribution or normal distribution, as indicated in the table (see Geron Chapter 11).

Other Activation Functions

- Leaky ReLU
$$\max(\alpha z, z)$$
where α is set to a small number, such as .01, or learned in training.
- Exponential linear unit
$$\mathrm{ELU}(z) = \begin{cases} \alpha(\exp(z) - 1) & z < 0 \\ z & z \geq 0 \end{cases}$$



- Until recently, ELU has had the best performance so far, but it is slower than ReLU.

Batch normalization

- Another trick to speed up training:
  - in between layers, zero-center and normalize the inputs to variance one.
  - normally done before a non-linear activation function

Regularization for Sparse Models

- As with linear models, neural network parameters can be regularized with an L1 and/or L2 penalty to push weak neurons to zero and produce a sparse model.
- But usually its better/simpler to use dropout.

Dropout

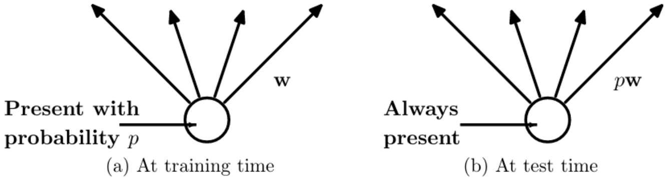


Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

Source: Srivastava et al, JMLR 2014

An elegant regularization technique:

- at every training step, every neuron has some probability (typically $p = 0.5$) of being temporarily dropped out, so that it will be ignored at this step.
- at test time, neurons dont get dropped any more but coefficients are down-weighted by p.

Dropout

- Approximately equivalent to averaging the output of N models (where N is the number of neurons).

Dropout

- Approximately equivalent to averaging the output of N models (where N is the number of neurons).
- Neurons trained with dropout:
  - cannot co-adapt with neighboring neurons and must be independently useful.
  - cannot rely excessively on just a few input neurons; they have to pay attention to all input neurons.
  - makes the model less sensitive to slight changes in the inputs.

Optimizers

- Choice of optimization algorithm is the topic of active research, which has shown that it can have a big impact on model performance.
  - Until recently, a good starting choice would be Adam (adaptive moment estimation), which is fast and usually works well. For robustness, can also try SGD.
  - A recent paper says that AdaBound dominates Adam or SGD.

Early stopping

- A popular/efficient regularization method is to continually evaluate your model at regular intervals, and then to stop training when the test-set accuracy starts to decrease.

Early stopping

- A popular/efficient regularization method is to continually evaluate your model at regular intervals, and then to stop training when the test-set accuracy starts to decrease.
- Split data into three sets: training, validation, and test.
  - every few epochs, check accuracy in validation set.
  - if it has gone down since last check, stop and use the model at the previous checkpoint.

Batch Training with Large Data

- If data sets don't fit in memory, can load the data in batches from disk.
- can also continuously update a saved model.

Outline

Intro to Neural Nets

Practicalities

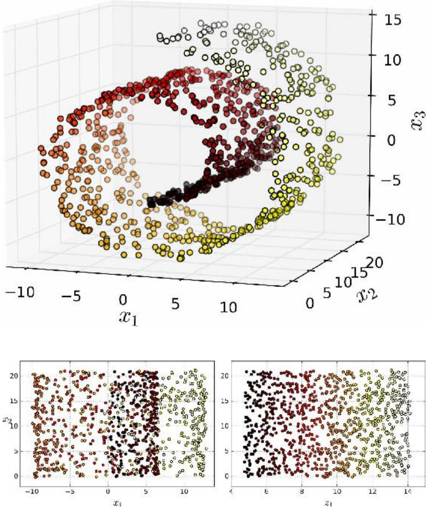
Autoencoders

Embedding Layers

Word Embeddings

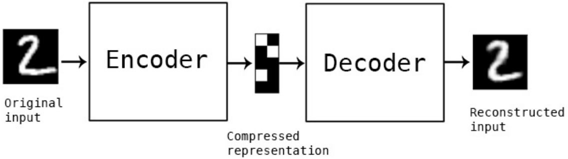
Kozlowski, Evans, and Taddy (2019)

Remember the Swiss roll?

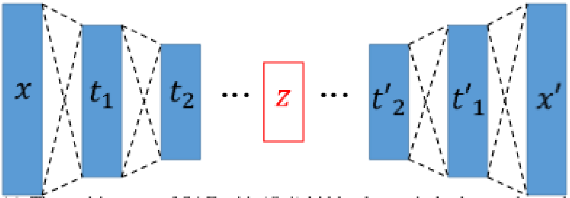


- The dimension reduction process matters: projecting down to two dimensions directly (left panel) might not isolate the variation we are interested in (as done in the right panel, which unrolls the Swiss Roll)

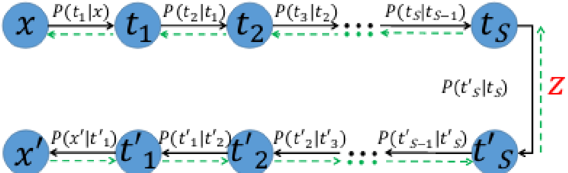
Autoencoders: Domain-specific dimension reduction



- "Autoencoder" refers to a class of deep neural network that performs domain-specific dimension reduction.
  - They learn efficient encodings of the data, which can then be decoded back to a (minimally) lossy representation of the original data.
  - Can also randomly generate new data that looks like the training data.



(a) The architecture of SAE with (S-1) hidden layers in both encoder and decoder.



(b) The graph representation of SAE with (S-1) hidden layers in both encoder and decoder.

- Autoencoders work by stacking layers that gradually decrease in dimensionality to create the compressed representation (Z), and then gradually increase in dimensionality to try to reconstruct the input.
  - the autoencoder is implicitly solving the problem of maximizing entropy in the bottleneck layer.

Autoencoding for data visualization

- For 2D visualization, t-SNE is probably the best algorithm
  - but quite slow, and typically requires relatively low-dimensional data.

Autoencoding for data visualization

- For 2D visualization, t-SNE is probably the best algorithm
  - but quite slow, and typically requires relatively low-dimensional data.
- Decent baseline:
  - use an autoencoder to compress your data to relatively low dimension (e.g. 32 dimensions)
  - then use t-SNE for mapping the compressed data to a 2D plane.

Outline

Intro to Neural Nets

Practicalities

Autoencoders

Embedding Layers

Word Embeddings

Kozlowski, Evans, and Taddy (2019)

What is an Embedding?

- A (relatively) low-dimensional vector representation of a categorical variable.
  - spatial location of the vector encodes predictive information about the category.

What is an Embedding?

- A (relatively) low-dimensional vector representation of a categorical variable.
  - spatial location of the vector encodes predictive information about the category.
- e.g., trying to predict how employment responds to economic growth with data from U.S. states:
  - instead of including a fifty-dimensional categorical variable, include two-dimensional latitude and longitude

What is an Embedding?

- A (relatively) low-dimensional vector representation of a categorical variable.
  - spatial location of the vector encodes predictive information about the category.
- e.g., trying to predict how employment responds to economic growth with data from U.S. states:
  - instead of including a fifty-dimensional categorical variable, include two-dimensional latitude and longitude
  - or initialize each state to a random two-dimensional vector, and let the model decide where to move the states to improve prediction on your task (e.g.).

An embedding layer is just matrix multiplication

- An embedding layer can be represented as

$$\underbrace{x}_{n_E \times 1} = \underbrace{\Omega}_{n_E \times n_w} \cdot \underbrace{w}_{n_w \times 1}$$

  - w, a categorical variable (e.g., representing a word)
    - one-hot vector with a single item equaling one.
    - The input to the embedding layer.

An embedding layer is just matrix multiplication

- An embedding layer can be represented as

$$\underbrace{x}_{n_E \times 1} = \underbrace{\Omega}_{n_E \times n_w} \cdot \underbrace{w}_{n_w \times 1}$$

  - w, a categorical variable (e.g., representing a word)
    - one-hot vector with a single item equaling one.
    - The input to the embedding layer.
  - x, a dense representation of the variable.
    - The output of the embedding layer.

An embedding layer is just matrix multiplication

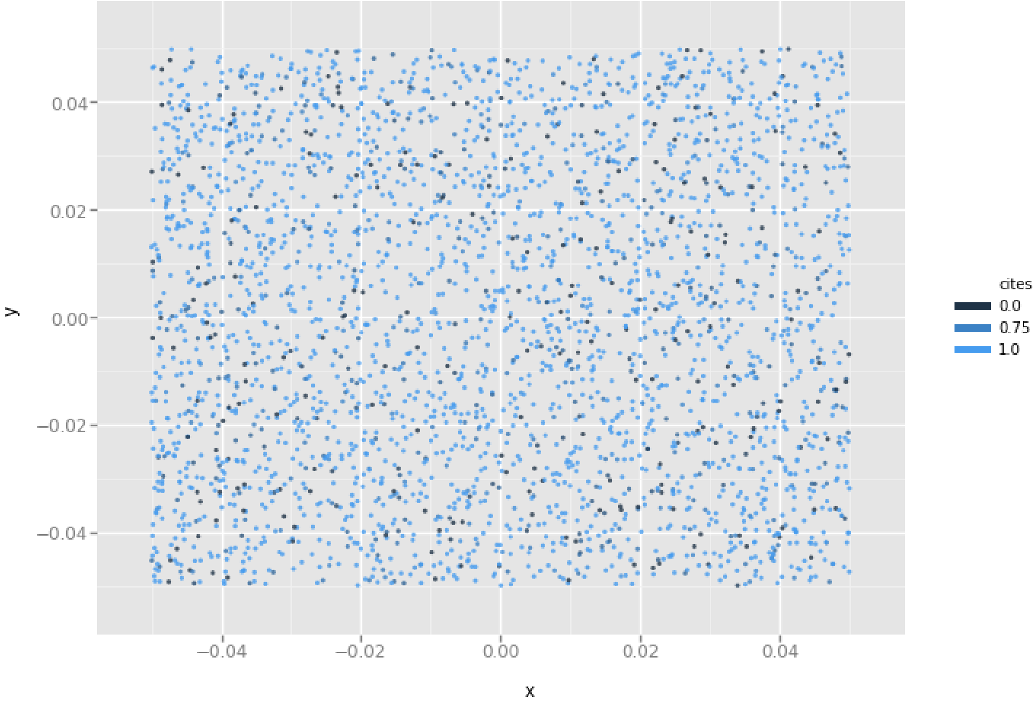
- An embedding layer can be represented as

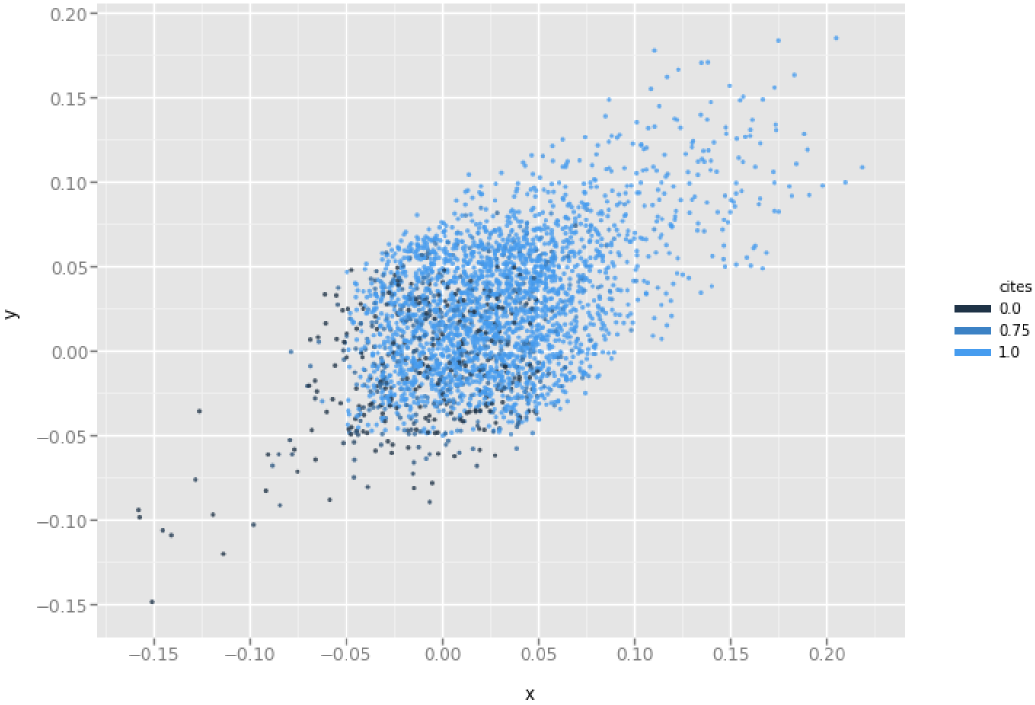
$$\underbrace{x}_{n_E \times 1} = \underbrace{\Omega}_{n_E \times n_w} \cdot \underbrace{w}_{n_w \times 1}$$

  - w, a categorical variable (e.g., representing a word)
    - one-hot vector with a single item equaling one.
    - The input to the embedding layer.
  - x, a dense representation of the variable.
    - The output of the embedding layer.
  - An embedding matrix Ω, learnable by the DNN

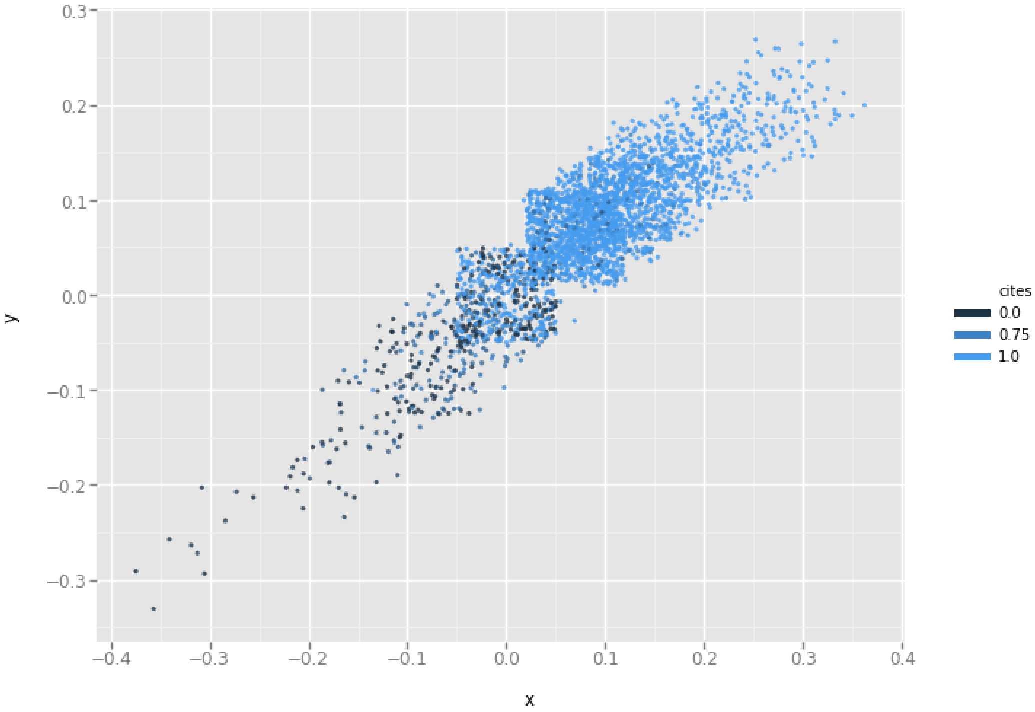
The Embedding Matrix Ω

- The model learns the weights of the embedding matrix in the same way that it would learn any model parameters.
- The rows of the matrix correspond to vectors for the n_w categories.
  - These are the "word vectors" that people talk about when they mention word embeddings or Word2Vec.


cites
0.0
0.75
1.0
y
x
0.04
0.02
0.00
−0.02
−0.04
−0.04
−0.02
0.00
0.02
0.04



0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
y
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20
x
cites
0.0
0.75
1.0



0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
y
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
x
cites
0.0
0.75
1.0

Embedding Layers versus Dense Layers

- An embedding layer is statistically equivalent to a fully-connected dense layer with sparse data set as input and linear activation.

Embedding Layers versus Dense Layers

- An embedding layer is statistically equivalent to a fully-connected dense layer with sparse data set as input and linear activation.
- Why use an embedding layer rather than a dense layer?
  - embedding layers are much faster for this purpose
  - batch updating with regularization and dropout do not work well on sparse data.

Outline

Intro to Neural Nets

Practicalities

Autoencoders

Embedding Layers

Word Embeddings

Kozlowski, Evans, and Taddy (2019)

Word Embeddings

- Embedding layer maps word indexes to dense vectors.

Word Embeddings

- Embedding layer maps word indexes to dense vectors.
- Documents are lists of word indexes $\{w_1, w_2, ..., w_{n_i}\}$.
  - equivalently, let w_i be a one-hot vector (dimensionality n_w = vocab size) where the associate word's index equals one .

Word Embeddings

- Embedding layer maps word indexes to dense vectors.
- Documents are lists of word indexes $\{w_1, w_2, ..., w_{n_i}\}$.
  - equivalently, let w_i be a one-hot vector (dimensionality n_w = vocab size) where the associate word's index equals one .
  - Normalize all documents to the same length L; shorter documents can be padded with a null token. This requirement can be relaxed with recurrent neural networks.

Word Embeddings

- Embedding layer maps word indexes to dense vectors.
- Documents are lists of word indexes $\{w_1, w_2, ..., w_{n_i}\}$.
  - equivalently, let w_i be a one-hot vector (dimensionality n_w = vocab size) where the associate word's index equals one .
  - Normalize all documents to the same length L; shorter documents can be padded with a null token. This requirement can be relaxed with recurrent neural networks.
- The embedding layer replaces the list of sparse one-hot vectors with a list of n_E-dimensional ($n_E << n_w$) dense vectors

$$\boldsymbol{X} = \left[\begin{array}{ccc} x_1 & \dots & x_L \end{array} \right]$$

where

$$\underbrace{x_i}_{n_E \times 1} = \underbrace{\Omega'}_{n_E \times n_w} \cdot \underbrace{w_i}_{n_w \times 1}$$

Word Embeddings

- Embedding layer maps word indexes to dense vectors.
- Documents are lists of word indexes $\{w_1, w_2, ..., w_{n_i}\}$.
  - equivalently, let w_i be a one-hot vector (dimensionality n_w = vocab size) where the associate word's index equals one .
  - Normalize all documents to the same length L; shorter documents can be padded with a null token. This requirement can be relaxed with recurrent neural networks.
- The embedding layer replaces the list of sparse one-hot vectors with a list of n_E-dimensional ($n_E << n_w$) dense vectors

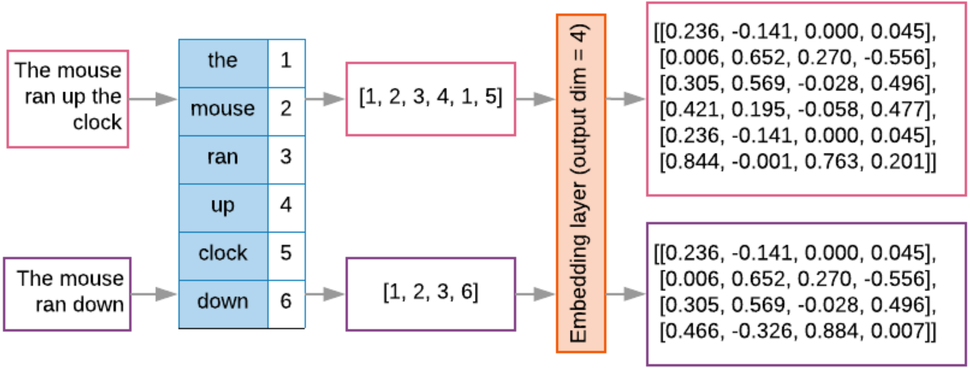
$$\boldsymbol{X} = \left[\begin{array}{ccc} x_1 & \dots & x_L \end{array}\right]$$

where

$$\underbrace{x_i}_{n_E \times 1} = \underbrace{\Omega'}_{n_E \times n_w} \cdot \underbrace{w_i}_{n_w \times 1}$$

- This $\boldsymbol{X}$ matrix is then flattened into a $L * n_E$ vector for input to the next layer.

Illustration



Examining the Embeddings

- See Jupyter notebook for examples on training and visualizing the embeddings with words as points.
  - Also examples for extracting vectors for words and computing cosine similarity between words.

Word Embeddings – Word2Vec, GloVe

- Word embeddings:
  - refers to a class of statistical models that **represent words or phrases as points in a vector space**.

Word Embeddings – Word2Vec, GloVe

- Word embeddings:
  - refers to a class of statistical models that **represent words or phrases as points in a vector space**.
- The key idea is to represent the meaning of words by the neighboring words – their **contexts**.

Word Embeddings – Word2Vec, GloVe

- Word embeddings:
  - refers to a class of statistical models that **represent words or phrases as points in a vector space**.
- The key idea is to represent the meaning of words by the neighboring words – their **contexts**.
- You might hear "word embeddings" and "word2vec" interchangeably, although word2vec technically refers to a particular implementation of a word embedding model.
  - the other well-known implementation is gloVe, which is faster but has similar performance/applications

Word Embeddings and Word2Vec

- Word2Vec , GloVe, and other popular embeddings vectors are trained the same way as the word embeddings we just made for citation counts.
  - rather than predicting some metadata (such as citations) they predict the co-occurence of neighboring words.

Why word vectors?

- Once words are represented as vectors, we can use linear algebra to understand the relationships between words:
  - Words that are geometrically close to each other are similar: e.g. “student” and “pupil.”

Why word vectors?

- Once words are represented as vectors, we can use linear algebra to understand the relationships between words:
  - Words that are geometrically close to each other are similar: e.g. "student" and "pupil."
  - More intriguingly, word2vec algebra can depict conceptual, analogical relationships between words.
    - Consider the analogy: **man is to king as woman is to _____**

Why word vectors?

- Once words are represented as vectors, we can use linear algebra to understand the relationships between words:
  - Words that are geometrically close to each other are similar: e.g. "student" and "pupil."
  - More intriguingly, word2vec algebra can depict conceptual, analogical relationships between words.
    - Consider the analogy: **man is to king as woman is to _____**
    - With word2vec, we have

$$vec(king) - vec(man) + vec(woman) \approx vec(queen)$$

How are word embeddings different from topic models?

- Ben Schmidt:
  - Topic models reduce words to core meanings to understand documents more clearly.

How are word embeddings different from topic models?

- Ben Schmidt:
  - Topic models reduce words to core meanings to understand documents more clearly.
  - Word embedding models ignore information about individual documents to better understand the relationships between words.

Word Function $\longleftrightarrow$ Word Neighbors

- "The meaning of a word is its use in the language"

  - Ludwig Wittgenstein, *Philosophical Investigation*, 1953

Word Function $\longleftrightarrow$ Word Neighbors

- "The meaning of a word is its use in the language"

  - Ludwig Wittgenstein, *Philosophical Investigation*, 1953

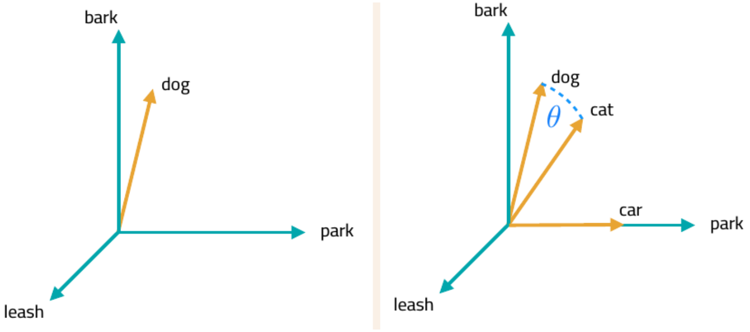
- "You shall know a word by the company it keeps"

  - J.R. Firth, Papers in Linguistics,1957

I've never seen this word before, but...

- He filled the **wampimuk**, passed it around and we all drunk some
- We found a little, hairy **wampimuk** sleeping behind the tree

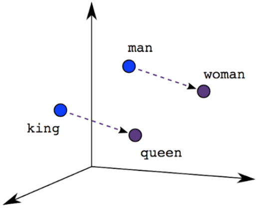
Words as Vectors



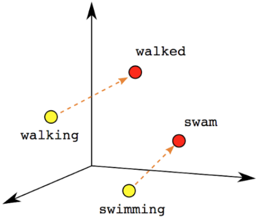
- Use cosine similarity as a measure of relatedness:

$$\cos\theta = \frac{v_1 \cdot v_2}{||v_1||||v_2||}$$

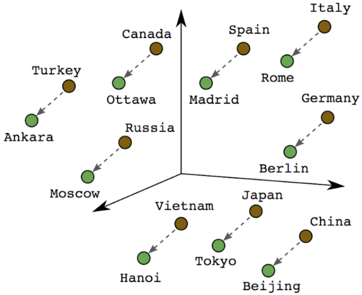
Vector Directions $\leftrightarrow$ Meaning



Male-Female

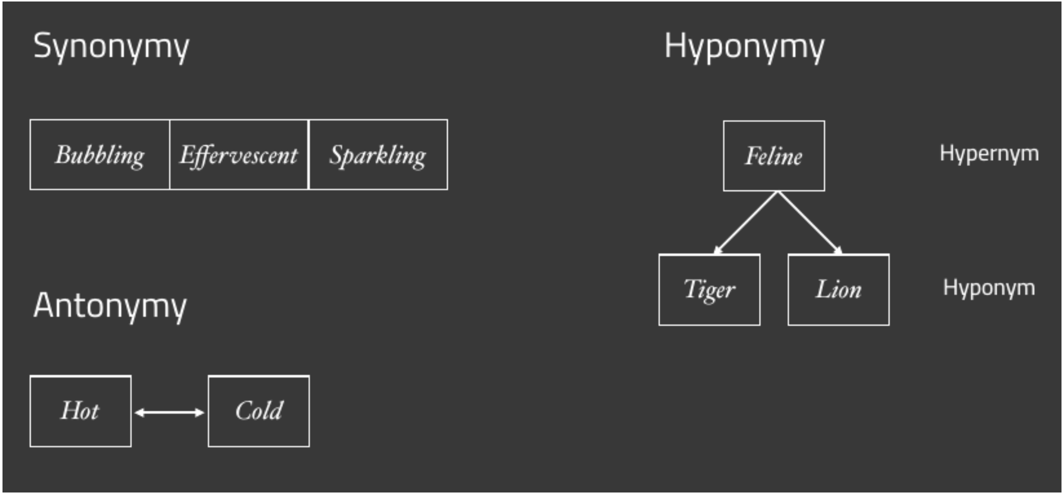


Verb Tense

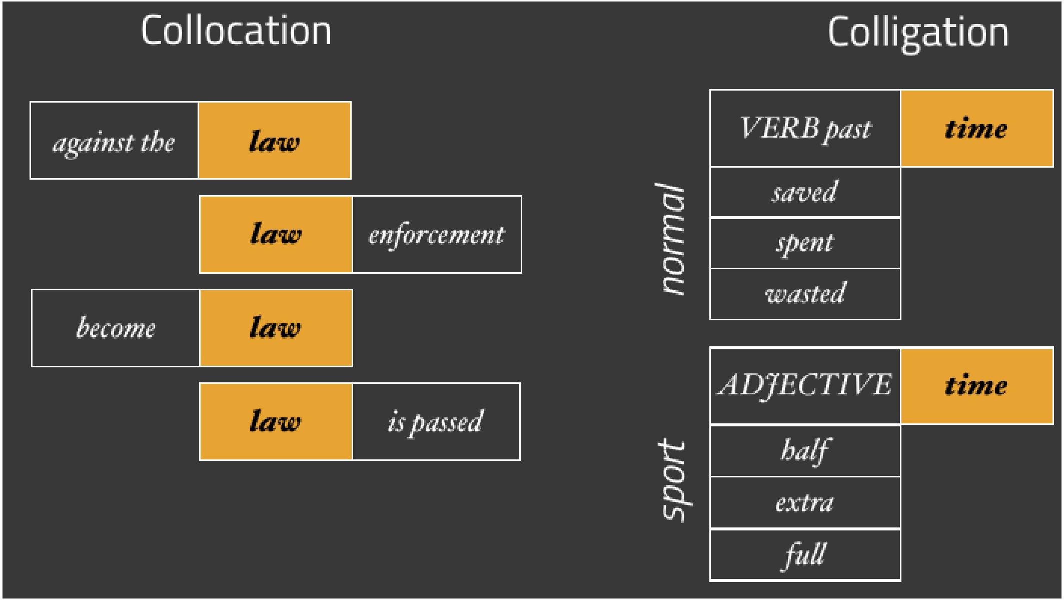


Country-Capital

Linguistic Relations



Collocational Relations



Similarity vs. Relatedness

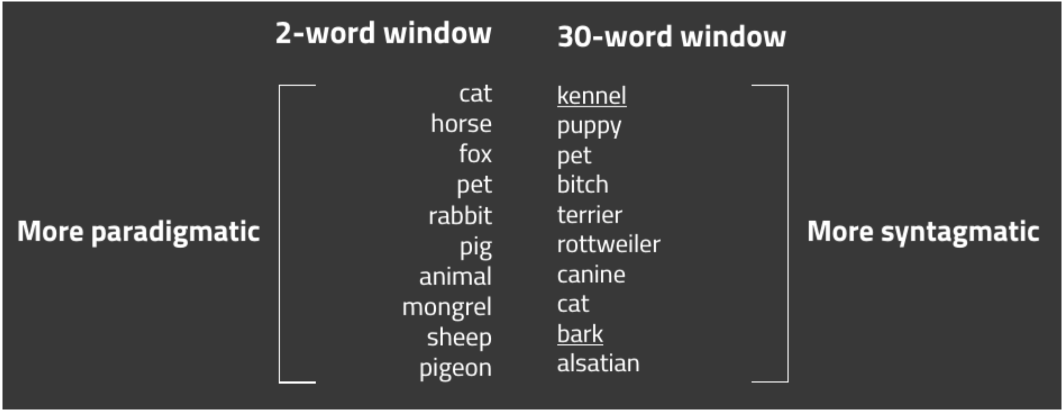
- Semantic **similarity**: words sharing salient attributes / features
  - synonymy (car / automobile)
  - hypernymy (car / vehicle)
  - co-hyponymy (car / van / truck)

Similarity vs. Relatedness

- Semantic **similarity**: words sharing salient attributes / features
  - synonymy (car / automobile)
  - hypernymy (car / vehicle)
  - co-hyponymy (car / van / truck)
- Semantic **relatedness**: words semantically associated without necessarily being similar
  - function (car / drive)
  - meronymy (car / tire)
  - location (car / road)
  - attribute (car / fast)

  (Budansky and Hirst, 2006)

Most similar words to dog, depending on window size



- Small windows pick up substitutable words; large windows pick up topics.

Evaluation of Word Embeddings

- Intrinsic:
  - evaluate word-pairs similarities → compare with similarity judgments given by humans
  - evaluate on analogy tasks ("Paris is to France as Tokyo is to ____")

Evaluation of Word Embeddings

- Intrinsic:
  - evaluate word-pairs similarities $\rightarrow$ compare with similarity judgments given by humans
  - evaluate on analogy tasks ("Paris is to France as Tokyo is to ____")
- Extrinsic:
  - use the vectors in a downstream task (classification, translation, ...) and evaluate the final performance on the task

SGNS: Skip-gram with negative sampling

- When people mention "word2vec", they are usually talking about SGNS: "skip gram with negative sampling."
  - This is a particular word-embedding model with good performance on a range of analogy and prediction tasks.

SGNS: Skip-gram with negative sampling

- When people mention “word2vec”, they are usually talking about SGNS: “skip gram with negative sampling.”
  - This is a particular word-embedding model with good performance on a range of analogy and prediction tasks.

- How does it learn the meaning of the word “fox”:

The <u>quick brown</u> **fox** <u>jumps over</u> the lazy dog

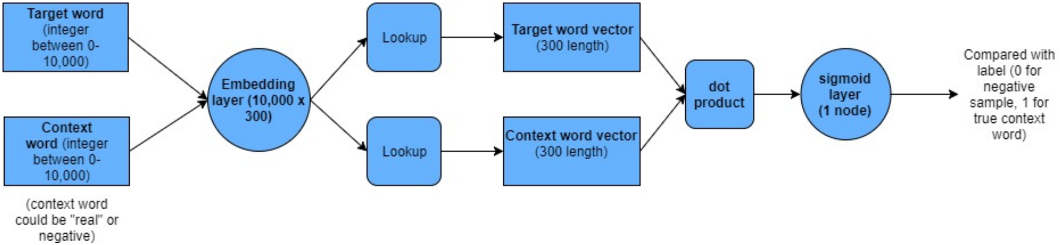
SGNS: Skip-gram with negative sampling

- When people mention "word2vec", they are usually talking about SGNS: "skip gram with negative sampling."
  - This is a particular word-embedding model with good performance on a range of analogy and prediction tasks.
- How does it learn the meaning of the word "fox":

  The <u>quick brown</u> **fox** <u>jumps over</u> the lazy dog

- Word2Vec reads in every example of the word "fox" and tries to predict what other words will be in the context window.
  - the prediction weights on these other words (after dimension reduction) are the word vectors

Word2Vec Schema



Tokenizing for Embeddings

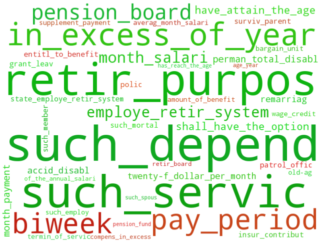
- embeddings work better with more information about the placement of words in sentences.
  - don't drop stopwords/function-words
  - should include tokens for start of sentence and end of sentence
  - should include a special token for out-of-vocabulary words
    - or replace with the part of speech tag

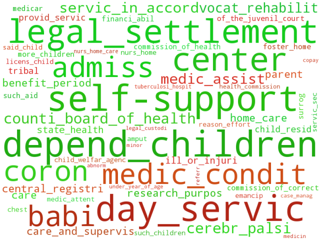
Word Dropout

- When training models, words can be randomly replaced with the unknown symbol with some small probability (Iyyer et al 2015).
- Prevents models from relying too much on particular words.

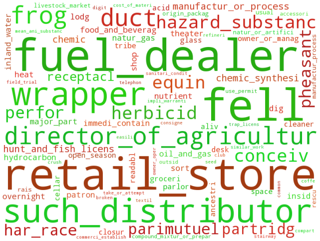
K-means clustering with Word Embeddings

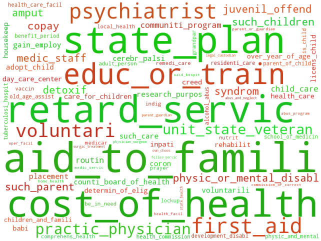
Income Tax (Pensions Topic and Health Care Topic)





Sales Tax (Retail Topic and Health Care Topic)





- Clustered phrases affecting tax revenues (Ash 2018); Green words tend to increase revenues; red words tend to decrease revenues.

Word Mover Distance

- Cosine distance treats synonyms as just as close as totally unrelated words.

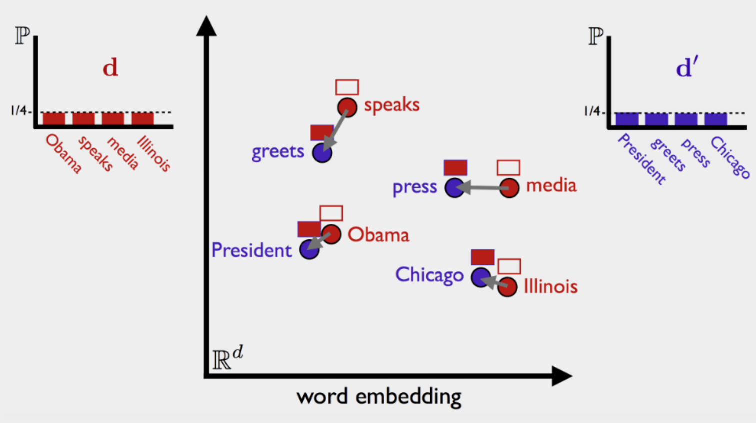
Word Mover Distance

- Cosine distance treats synonyms as just as close as totally unrelated words.
- Word mover distance between two texts is given by:
  - total amount of "mass" needed to move words from one side into the other
  - multiplied by the distance the words need to move
  - Kusner, Sun, Kolkin, and Weinberger (ICML 2015)

Word Mover Distance

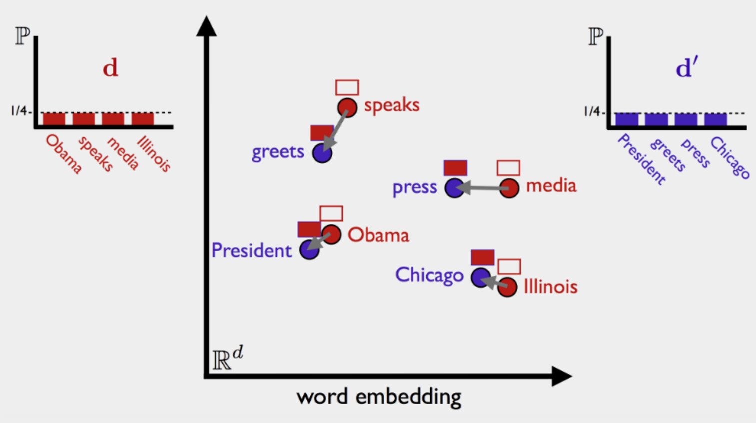
- Cosine distance treats synonyms as just as close as totally unrelated words.
- Word mover distance between two texts is given by:
  - total amount of "mass" needed to move words from one side into the other
  - multiplied by the distance the words need to move
  - Kusner, Sun, Kolkin, and Weinberger (ICML 2015)
- Requires measure of distance between words (word embeddings).
  - see wmd package in Python.

Illustration



- d (obama speaks media illinois) is orthogonal to d' (president greets press chicago):
  - cosine similarity is zero

Illustration



- d (obama speaks media illinois) is orthogonal to d' (president greets press chicago):
  - cosine similarity is zero
  - Word mover distance sums the shortest distances between the words in the documents.

Pre-trained word embeddings

- For some NLP tasks, you might not need to train your own vectors.
  - or your corpus might be too small to do so.

Pre-trained word embeddings

- For some NLP tasks, you might not need to train your own vectors.
  - or your corpus might be too small to do so.
- In these cases, you can use a pre-trained model, like spaCy's GloVe embeddings:
  - one million vocabulary entries
  - 300-dimensional vectors
  - trained on the Common Crawl corpus

Pre-trained word embeddings

- For some NLP tasks, you might not need to train your own vectors.
  - or your corpus might be too small to do so.
- In these cases, you can use a pre-trained model, like spaCy's GloVe embeddings:
  - one million vocabulary entries
  - 300-dimensional vectors
  - trained on the Common Crawl corpus
- Can initialize prediction model using pre-trained embeddings.

Tips for using pre-trained embeddings

- Split training in two steps:
  - in first run, train the model with the first layer (the pre-trained embeddings) frozen.

Tips for using pre-trained embeddings

- Split training in two steps:
  - in first run, train the model with the first layer (the pre-trained embeddings) frozen.
  - in second run, un-freeze the embedding layer for fine tuning.

Outline

Intro to Neural Nets

Practicalities

Autoencoders

Embedding Layers

Word Embeddings

Kozlowski, Evans, and Taddy (2019)

Kozlowski, Evans, and Taddy (2019)

- Data-set: Google 5-grams.
  - n-grams of length five for U.S. and U.K. publications.
  - provides counts for each year

Kozlowski, Evans, and Taddy (2019)

- Data-set: Google 5-grams.
  - n-grams of length five for U.S. and U.K. publications.
  - provides counts for each year
- Extract language dimensions:
  - get the complete list of WordNet antonym pairs (e.g, “weak/strong”, “tall/short”)
    - filter on document frequency to 428 pairs.
  - map the dimensional shifts between the antonyms.

Kozlowski, Evans, and Taddy (2019)

- Data-set: Google 5-grams.
  - n-grams of length five for U.S. and U.K. publications.
  - provides counts for each year
- Extract language dimensions:
  - get the complete list of WordNet antonym pairs (e.g, “weak/strong”, “tall/short”)
    - filter on document frequency to 428 pairs.
  - map the dimensional shifts between the antonyms.
  - compare this vector shift to the one between men and women.

Mapping gender, class, and race

Gender	Class	Race[†]
man – woman	rich – poor	black – white
men – women	richer – poorer	blacks – whites
he – she	richest – poorest	Blacks – Whites
him – her	affluence – poverty	Black – White
his – her	affluent – impoverished	African – European
his – hers	expensive – inexpensive	African – Caucasian
boy – girl	luxury – cheap	
boys – girls	opulent – needy	
male – female		
masculine – feminine		

Matching antonyms to gender/class

Gender dimension nearest neighbors	
1. rugged–delicate	.219 (.213, .224)
2. soft–loud	-.209 (-.216, -.201)
3. tender–tough	-.202 (-.210, -.197)
4. timid–bold	-.181 (-.186, -.174)
5. soft–hard	-.161 (-.168, -.158)

Class dimension nearest neighbors	
1. weak-strong	-.292 (-.301, -.287)
2. fortunate-unfortunate	.291 (.286, .297)
3. unhappy-happy	-.259 (-.266, -.254)
4. beautiful-ugly	.242 (.238, .245)
5. potent_impotent	.234 (.227, .244)

Mapping musical genres to race/class