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

"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.

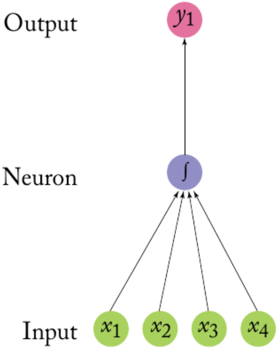
Will it last?

- 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:

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

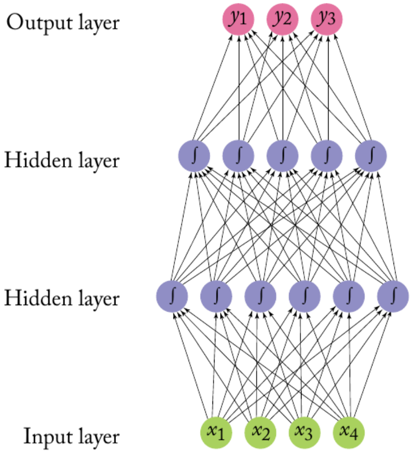
$$\boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega} \in \mathbb{R}^{n_x \times n_y}$$

  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

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

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

  - 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.

Multi-Layer Perceptron

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

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

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

  - 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

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

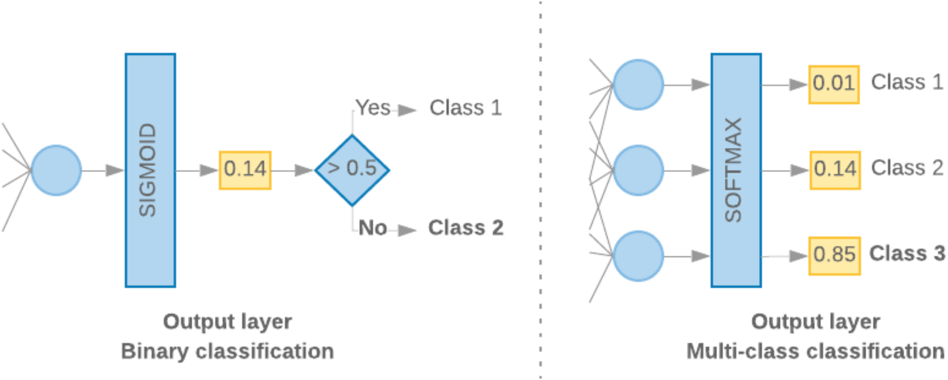
$$\boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega}_1 \in \mathbb{R}^{n_x \times n_1}, \boldsymbol{\omega}_2 \in \mathbb{R}^{n_1 \times n_2}, \boldsymbol{\omega}_3 \in \mathbb{R}^{n_2 \times n_y}$$

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

$$\begin{aligned} \text{MLP2}(\boldsymbol{x}) = & \\ \boldsymbol{h}_1 &= \boldsymbol{g}_1(\boldsymbol{x} \cdot \boldsymbol{\omega}_1) \\ \boldsymbol{h}_2 &= \boldsymbol{g}_2(\boldsymbol{h_1} \cdot \boldsymbol{\omega}_2) \\ \boldsymbol{y} &= \boldsymbol{h}_2 \cdot \boldsymbol{\omega}_3 \end{aligned}$$

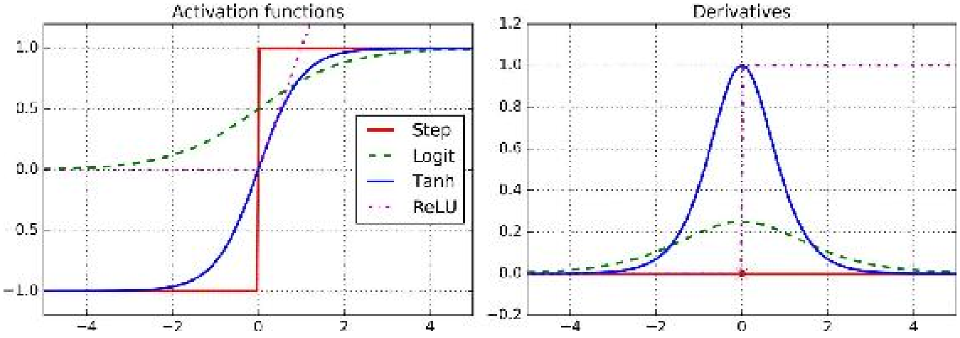
where $\boldsymbol{h}_l$ give hidden layers.

Constructing the Last Layer



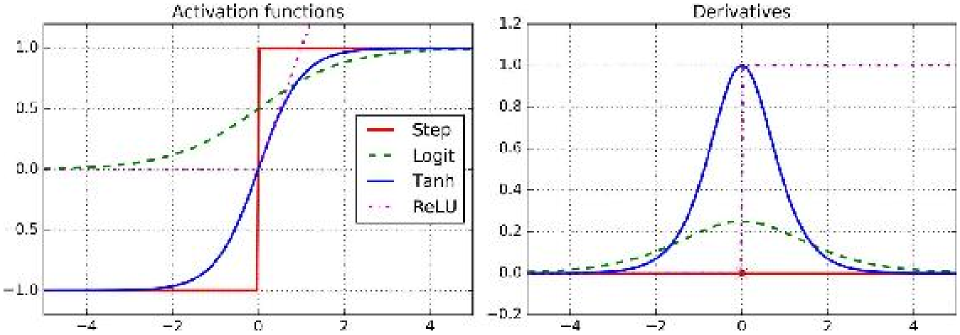
- MLPs will output a probability distribution across output classes.
  - can also output a real number, which would make a regression model.

What to pick for $g(\,)$



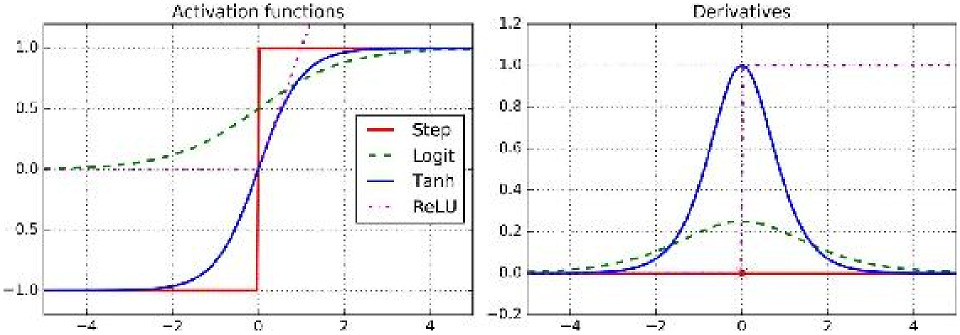
- logistic function: $\mathrm{logit}(z) = \frac{1}{1+\exp(-z)}$

What to pick for $g(\ )$



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

Backpropagation

- A crucial technology in deep learning is *backpropagation*, also known as *reverse-mode automatic differentiation (autodiff)*.
  - intuitively, computes the network's output error and how much each neuron contributes to the error, so they can be updated to reduce error.

Backpropagation

- A crucial technology in deep learning is *backpropagation*, also known as *reverse-mode automatic differentiation (autodiff).*
  - intuitively, computes the network's output error and how much each neuron contributes to the error, so they can be updated to reduce error.
  - Keras/TensorFlow will do this under the hood, just like how scikit-learn does gradient descent.
  - Appendix D of the Geron book has a decent explanation for how it works.

Outline

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.
   - A simple MLP is one of the models tried by Peterson and Spirling (2018).

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

Table 10-1. Typical regression MLP architecture

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Table 10-2. Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

Initializing Neuron Weights

- Neuron weights have to be initalized randomly; otherwise they are collinear and backprop won't distinguish their contributions to the output error.

Initializing Neuron Weights

- Neuron weights have to be initalized randomly; otherwise they are collinear and backprop won't distinguish their contributions to the output error.
- Standard practice is to use "Glorot" or "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)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Tuning NN Hyperparameters

- Number of neurons:
  - just pick 128 neurons per layer
  - overall, better to have too many neurons, and use regularization

Tuning NN Hyperparameters

- Number of neurons:
  - just pick 128 neurons per layer
  - overall, better to have too many neurons, and use regularization
- Number of hidden layers:
  - try between one and five.
  - adding layers usually helps more than adding neurons.

Tuning NN Hyperparameters

- Number of neurons:
  - just pick 128 neurons per layer
  - overall, better to have too many neurons, and use regularization
- Number of hidden layers:
  - try between one and five.
  - adding layers usually helps more than adding neurons.
- Activation functions:
  - ReLU is a good baseline in the hidden layers.

Tuning NN Hyperparameters

- Number of neurons:
  - just pick 128 neurons per layer
  - overall, better to have too many neurons, and use regularization
- Number of hidden layers:
  - try between one and five.
  - adding layers usually helps more than adding neurons.
- Activation functions:
  - ReLU is a good baseline in the hidden layers.
- See Geron, pp. 322-323 for tools to help with tuning, e.g. Keras Tuner.
  - See also Smith (2018).

Early stopping

An efficient training approach, that also works to regularize a model, is **early stopping:**

- Split data into three sets: training, validation, and test.

Early stopping

An efficient training approach, that also works to regularize a model, is **early stopping:**

- Split data into three sets: training, validation, and test.
- continually evaluate your model in validation set at regular intervals
- stop training when the validation-set accuracy starts to decrease.

Early stopping

An efficient training approach, that also works to regularize a model, is **early stopping:**

- Split data into three sets: training, validation, and test.
- continually evaluate your model in validation set at regular intervals
- stop training when the validation-set accuracy starts to decrease.
- evaluate model in test set.

Batch normalization

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

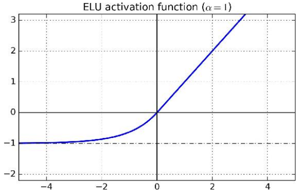
Batch normalization

- Another trick to increase performance, speed up training, and regularize the model:
  - in between layers, zero-center and normalize the inputs to variance one.
  - normally done before a non-linear activation function
- Does not work well in recurrent neural nets – can use gradient clipping (hard constraint on gradient) instead.

ELU and SELU

Exponential linear unit

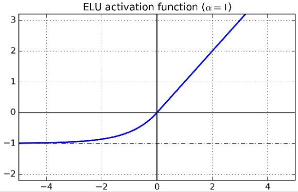
$$\text{ELU}(z) = \begin{cases} \alpha(\exp(z) - 1) & z < 0 \\ z & z \geq 0 \end{cases}$$



ELU and SELU

Exponential linear unit

$$\text{ELU}(z) = \begin{cases} \alpha(\exp(z) - 1) & z < 0 \\ z & z \geq 0 \end{cases}$$



SELU (Scaled ELU):

$$\text{SELU}(z) = \lambda \begin{cases} \alpha(\exp(z) - 1) & z < 0 \\ z & z \geq 0 \end{cases}$$

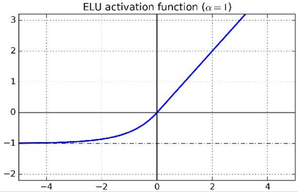
- Requires standardized inputs.
- Gaussian initialization of neuron weights:
  - mean zero and standard deviation $= n_x^{-1/2}$
- Set $\lambda \approx 1.0507, \alpha \approx 1.6732$.

- $\rightarrow$Then layers will be self-normalizing; weights will converge to mean zero and variance once (Klambauer et al 2017)

ELU and SELU

Exponential linear unit

$$\text{ELU}(z) = \begin{cases} \alpha(\exp(z) - 1) & z < 0 \\ z & z \geq 0 \end{cases}$$



SELU (Scaled ELU):

$$\text{SELU}(z) = \lambda \begin{cases} \alpha(\exp(z) - 1) & z < 0 \\ z & z \geq 0 \end{cases}$$

- Requires standardized inputs.
- Gaussian initialization of neuron weights:
  - mean zero and standard deviation $= n_x^{-1/2}$
- Set $\lambda \approx 1.0507, \alpha \approx 1.6732$.

- $\rightarrow$Then layers will be self-normalizing; weights will converge to mean zero and variance once (Klambauer et al 2017)

- In general, SELU $>$ ELU $>$ ReLU.

Optimizer and Learning Rate

- Choice of optimization algorithm is the topic of active research, which has shown that it can have a big impact on model performance.
  - See Geron, pp. 351-358. He recommends SGD with momentum, RMSProp, or Nadam.

Optimizer and Learning Rate

- Choice of optimization algorithm is the topic of active research, which has shown that it can have a big impact on model performance.
  - See Geron, pp. 351-358. He recommends SGD with momentum, RMSProp, or Nadam.
- On the learning rate, see Geron pp. 359-364, recommending 1cycle scheduling:
  - start at η_0, increase linearly up to η_1 halfway through training, then decrease linearly to η_0 at the end.

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.

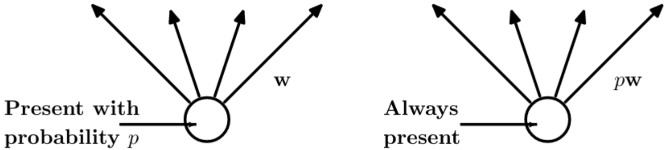
Regularization for Sparse Models

- 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.
- Another regularizer: “max-norm regularization”:
  - constrain each neuron’s weights ω by $||\omega||_2 \leq r$, where r is a hyperparameter.

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.
- Another regularizer: "max-norm regularization":
  - constrain each neuron's weights ω by $||\omega||_2 \leq r$, where r is a hyperparameter.
- But usually its better/simpler to use dropout.

Dropout



(a) At training time (b) At test time

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 anymore 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.

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.
- If using SELU activiation functions, use alpha dropout (Klambauer et al 2017).

Monte Carlo Dropout

- Normal dropout:
  - at test time, neurons dont get dropped but coefficients are down-weighted by p.

Monte Carlo Dropout

- Normal dropout:
  - at test time, neurons dont get dropped but coefficients are down-weighted by p.
- Monte Carlo dropout:
  - at test time, continue to allow dropout but produce 100 predictions, and average them.

Summary and Practical Guidelines

Table 11-3. Default DNN configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping (+ℓ_2 reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Table 11-4. DNN configuration for a self-normalizing net

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Outline

Application: Facts vs. Law

- A crucial distinction in law practice, and in applying caselaw, is to distinguish the law from the facts.
- In Cao, Ash, Chen (2018), we made a sample of fact and law sections in court cases and trained a model to predict them based on the text.

Application: Facts vs. Law

- A crucial distinction in law practice, and in applying caselaw, is to distinguish the law from the facts.
- In Cao, Ash, Chen (2018), we made a sample of fact and law sections in court cases and trained a model to predict them based on the text.
- Corpus:
  - all cases from courtlistener.com with an annotated "FACTS" section (extracted using regular expressions).
  - 23,497 sections, split into 1.3M paragraphs
  - 36% facts, 64% laws

Models and Results

DEPN: Document represented as a sequence of word-dependency pairs, fed to a **MLP classifier** with two 500-dim hidden layers.
DEPW: a variant of DEPN using Shulayeva et al.'s (2017) dependency features.
SMITH : Implementation of Smith's (2014) method: Bag of Words, but filter words that are statistically related to either fact statements or law statements. Fed to a Logistic Regression classifier.
WS: Laver et al.'s (2003) Wordscore algorithm (cf. Sarel and Demirtas, 2017). Bag of Words, but each word assigned a score based on relation to fact or law. Document is represented as a (re-scaled) score that sums up the pre- calculated scores of the words it contains, weighted by their frequencies.
BOW : Bag of words representation of documents, fed to a Logistic Regression classifier.
D2V: 500-dim Doc2Vec vectors (Le and Mikolov, 2014), fed to **MLP**.

Models and Results

DEPN: Document represented as a sequence of word-dependency pairs, fed to a **MLP classifier** with two 500-dim hidden layers.
DEPW: a variant of DEPN using Shulayeva et al.'s (2017) dependency features.
SMITH : Implementation of Smith's (2014) method: Bag of Words, but filter words that are statistically related to either fact statements or law statements. Fed to a Logistic Regression classifier.
WS: Laver et al.'s (2003) Wordscore algorithm (cf. Sarel and Demirtas, 2017). Bag of Words, but each word assigned a score based on relation to fact or law. Document is represented as a (re-scaled) score that sums up the pre- calculated scores of the words it contains, weighted by their frequencies.
BOW : Bag of words representation of documents, fed to a Logistic Regression classifier.
D2V: 500-dim Doc2Vec vectors (Le and Mikolov, 2014), fed to **MLP**.

	F1 for facts	**F1 for values**
DEPN	73.38	74.66
DEPW	72.77	73.79
SMITH	71.85	71.4
WS	**77.11**	**77.67**
BOW	72.57	73.29
D2V	67.18	65.36

Table 2: 5-fold cross validation.

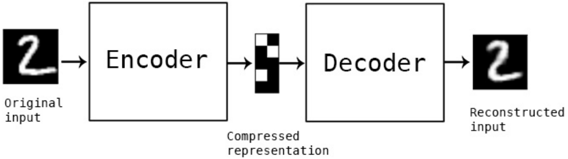
Outline

Efficient Data Representations

Which of the following number sequences do you find the easiest to memorize?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

Autoencoders $\leftrightarrow$ Optimal Compression Algorithms



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

PCA is a linear autoencoder

```python
from tensorflow import keras

encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=0.1))
```

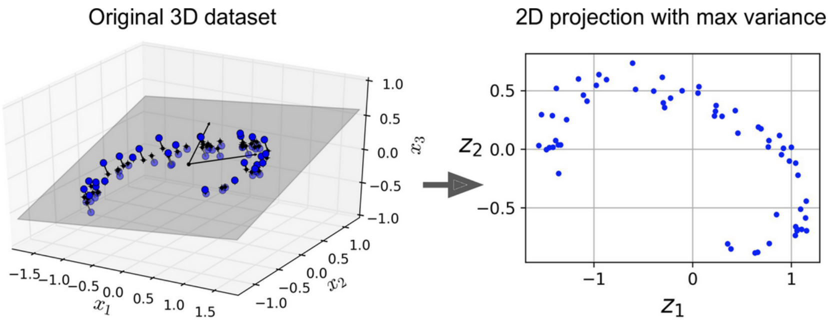


Figure 17-2. PCA performed by an undercomplete linear autoencoder

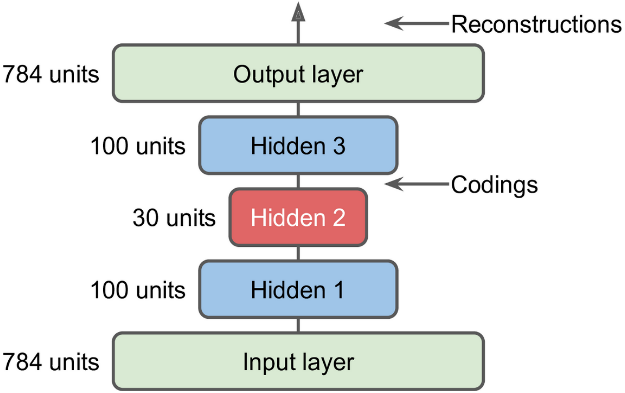


Figure 17-3. Stacked autoencoder

- 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.

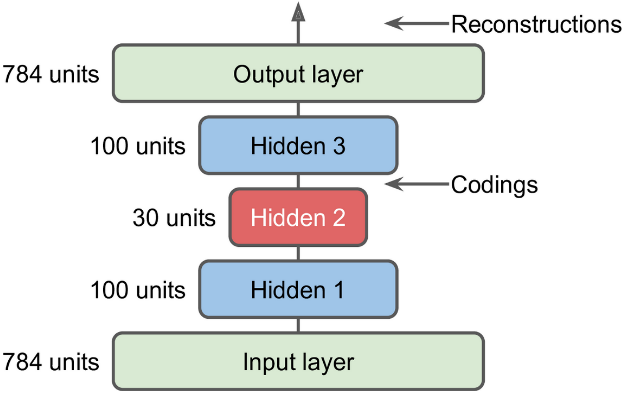


Figure 17-3. Stacked autoencoder

- 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.
  - for symmetic autoencoders, **tying weights** of the encoding and decoding segments will speed up training and tends to improve performance.

Reconstruction from encoded vector

Figure 17-4. Original images (top) and their reconstructions (bottom)

Autoencoders for Data Visualization

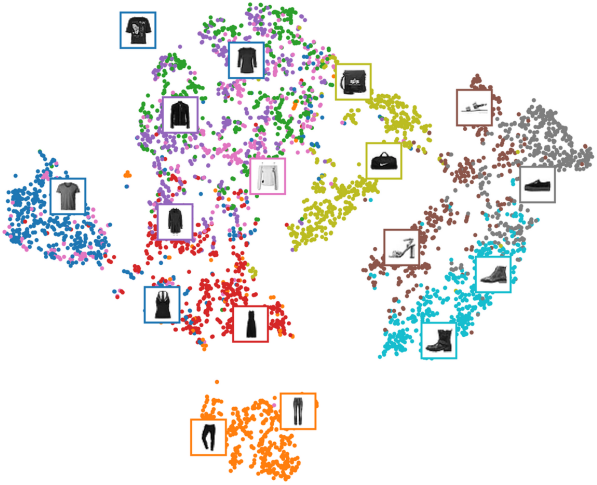


Figure 17-5. Fashion MNIST visualization using an autoencoder followed by t-SNE

- Decent baseline for visualizing the encodings:
  - 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.

Denoising Autoencoders

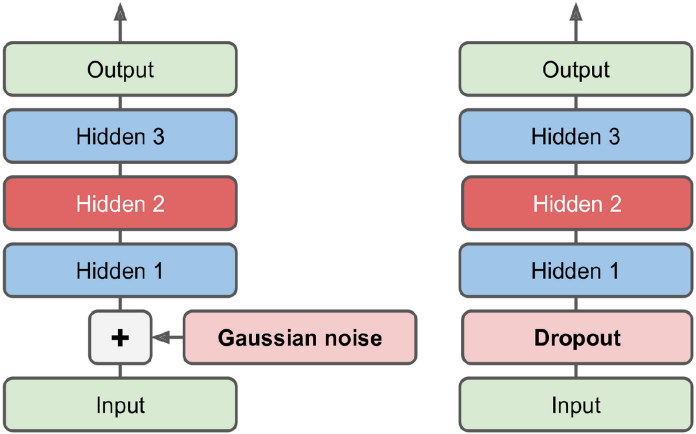


Figure 17-8. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

Outline

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

$$x = v(w)$$
$$\underbrace{x}_{n_E \times 1} = \underbrace{E}_{n_E \times n_w} \cdot \underbrace{w}_{n_w \times 1}$$

  - 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

$$x = v(w)$$
$$\underbrace{x}_{n_E \times 1} = \underbrace{E}_{n_E \times n_w} \cdot \underbrace{w}_{n_w \times 1}$$

  - 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

$$x = v(w)$$
$$\underbrace{x}_{n_E \times 1} = \underbrace{E}_{n_E \times n_w} \cdot \underbrace{w}_{n_w \times 1}$$

  - 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 function $v(\cdot)$, defined by matrix E, learnable by the DNN

The Embedding Matrix E

- 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.

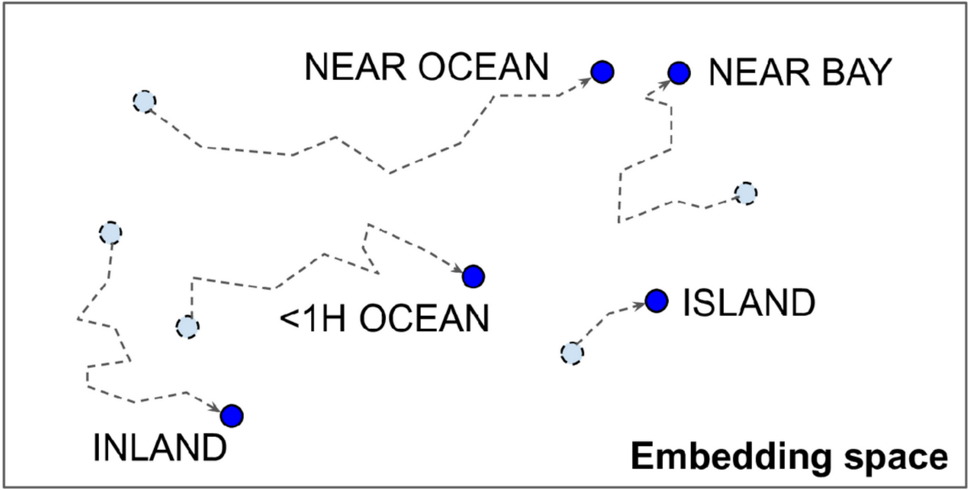
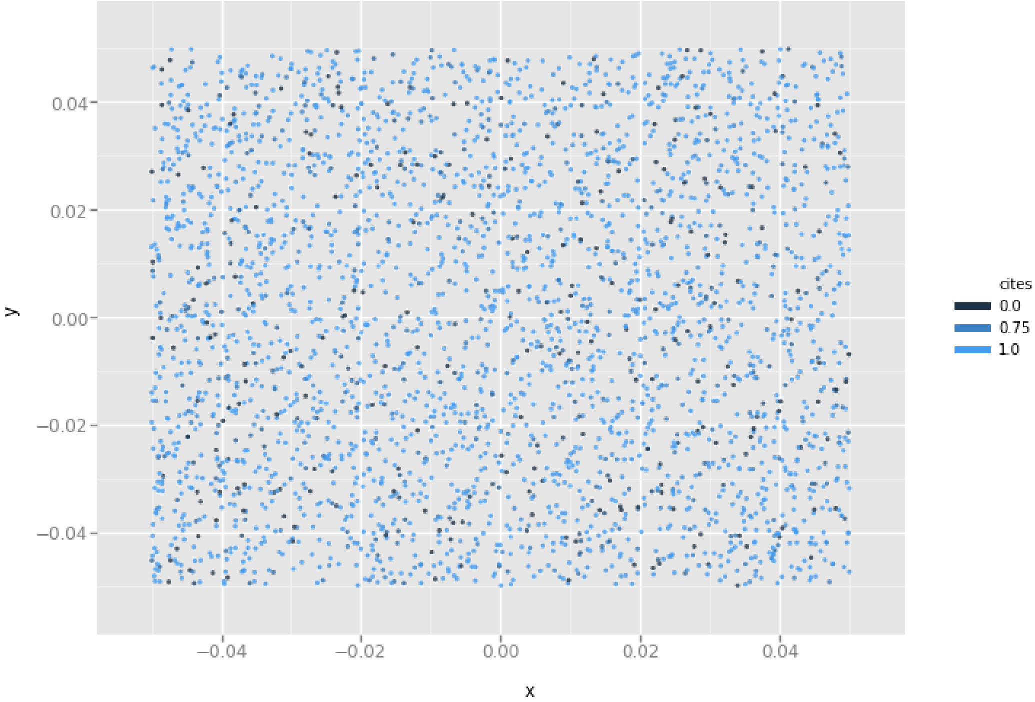
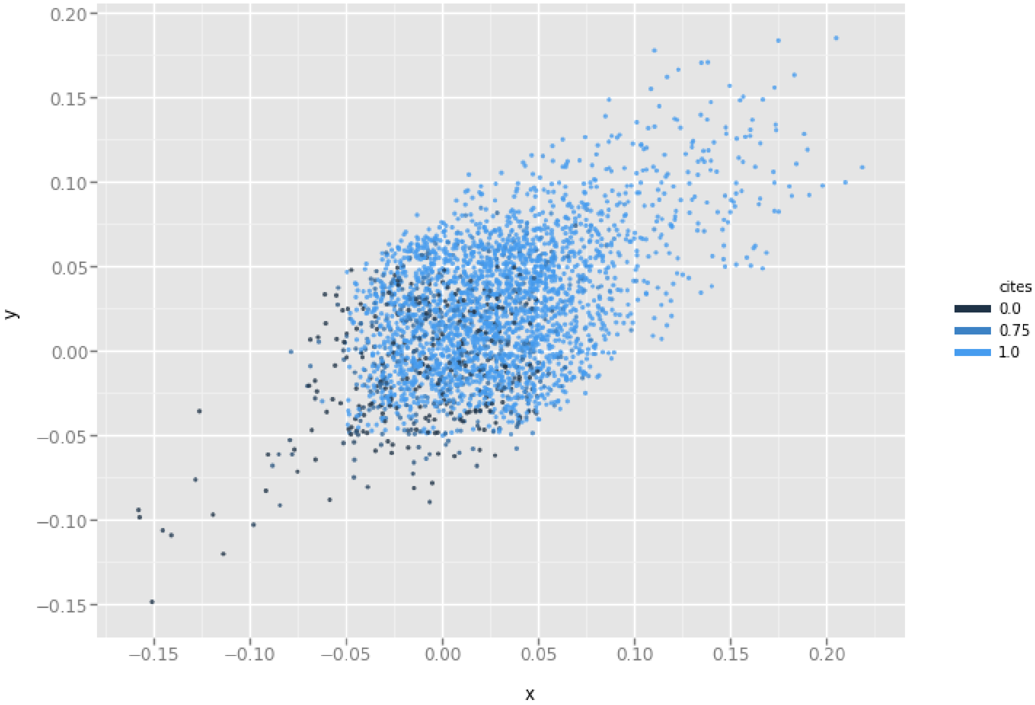
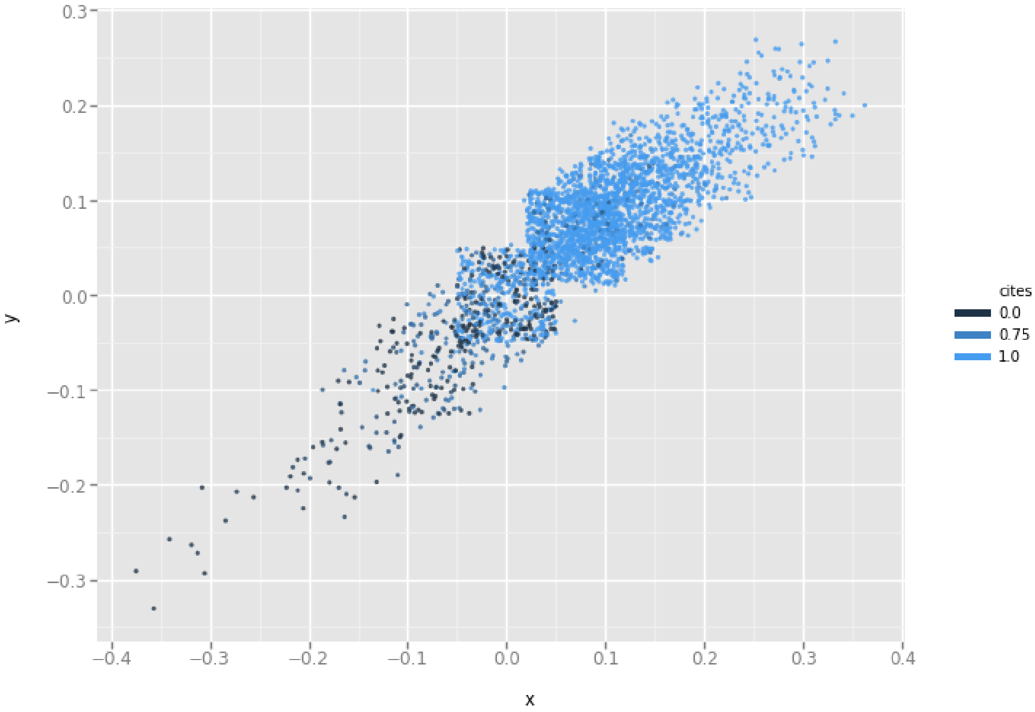


Figure 13-4. Embeddings will gradually improve during training



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


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.

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.
- Geron (pg. 433) advises using one-hot encoding for less than 10 items, embedings for more than 50, and in between it's unclear.

MLP output layer as embedding matrix

- Consider the output layer for an MLP:
  - last hidden layer l, with dimensionality n_l
  - output classes with dimensionality n_y
  - layer $l = n_l \times n_y$ matrix of weights E_y

MLP output layer as embedding matrix

- Consider the output layer for an MLP:
  - last hidden layer l, with dimensionality n_l
  - output classes with dimensionality n_y
  - layer $l = n_l \times n_y$ matrix of weights E_y
- columns of E_y give n_y outcome class vectors.
  - vector similarities between columns indicate model's learned similarities between the output classes (Goldberg pg. 99)

Outline

Word Embeddings

- Word embeddings are NN layers that map word indexes to dense vectors.

Word Embeddings

- Word embeddings are NN layers that map 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

- Word embeddings are NN layers that map 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

- Word embeddings are NN layers that map 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

$$\boldsymbol{X} = \begin{bmatrix} x_1 & \dots & x_L \end{bmatrix}$$

where

$$\underbrace{x_j}_{n_E \times 1} = \underbrace{E}_{n_E \times n_w} \cdot \underbrace{w_j}_{n_w \times 1}$$

Word Embeddings

- Word embeddings are NN layers that map 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

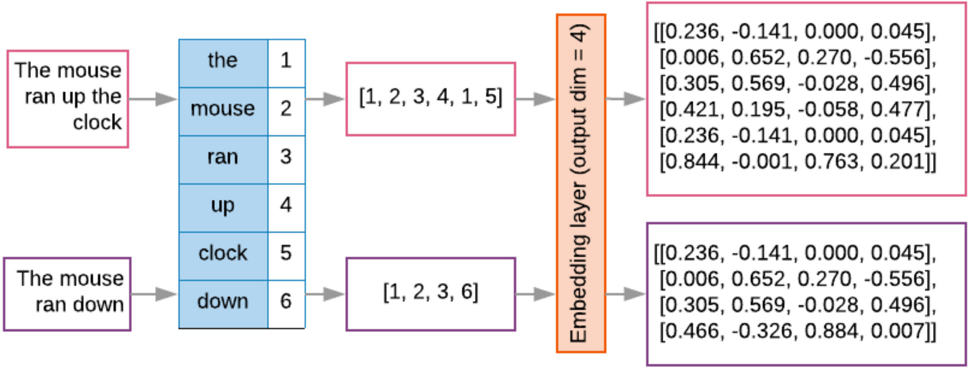
  $$\boldsymbol{X} = \left[\begin{array}{ccc} x_1 & \ldots & x_L \end{array} \right]$$

  where

  $$\underbrace{x_j}_{n_E \times 1} = \underbrace{E}_{n_E \times n_w} \cdot \underbrace{w_j}_{n_w \times 1}$$

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

Illustration



Continuous Bag-of-Words Representation

- Let w_j be the one-hot representation of feature j
- The dot product $w_j \cdot \boldsymbol{E}$ selects the row of $\boldsymbol{E}$ corresponding to j:

$$v(w_j) = \boldsymbol{w}_j \cdot \boldsymbol{E}$$

- And a document can be represented as

$$\boldsymbol{x}_i = \text{CBOW}(x_1, \ldots, x_{n_i}) = \sum_{j=1}^{n_i} w_j \cdot \boldsymbol{E} = (\sum_{j=1}^{n_i} w_j) \cdot \boldsymbol{E}$$

  - the sum over the word vectors in the document.

Word Embeddings can be used for any NLP Task

- For example, to predict political party from political speech:
  - could represent each document as concatenated embedding vectors for each word in the document.

Word Embeddings can be used for any NLP Task

- For example, to predict political party from political speech:
  - could represent each document as concatenated embedding vectors for each word in the document.
  - DNN will learn the optimal vector for each word in predicting the political party.

Practical Tip: Embedded Hashed N-Grams

Goldberg (2017) proposes the hashing vectorizer (**keras.preprocessing.text.hashing_trick**) as an efficient alternative to CNN's:

- Allocate $n_w \approx 1$ million rows to an embedding matrix $\boldsymbol{E}$
- Assign n-grams to embedding indexes with hashing function
- train MLP on top of embedding layer.

Practical Tip: Embedded Hashed N-Grams

Goldberg (2017) proposes the hashing vectorizer (**keras.preprocessing.text.hashing_trick**) as an efficient alternative to CNN's:

- Allocate $n_w \approx 1$ million rows to an embedding matrix $\boldsymbol{E}$
- Assign n-grams to embedding indexes with hashing function
- train MLP on top of embedding layer.
- Captures the local predictive power of n-grams without building vocabulary or costly training of CNN.

Outline

Word2Vec & GloVe

- "Word embeddings" often refers to Word2Vec or GloVe – these are particular (popular) models for producing word embeddings.
  - the goal: represent the meaning of words by the neighboring words – their **contexts**.

Word2Vec & GloVe

- "Word embeddings" often refers to Word2Vec or GloVe – these are particular (popular) models for producing word embeddings.
  - the goal: represent the meaning of words by the neighboring words – their **contexts**.
  - rather than predicting some metadata (such as citations) they predict the co-occurence of neighboring words.

Word2Vec & GloVe

- "Word embeddings" often refers to Word2Vec or GloVe – these are particular (popular) models for producing word embeddings.
  - the goal: represent the meaning of words by the neighboring words – their **contexts**.
  - rather than predicting some metadata (such as citations) they predict the co-occurence of neighboring words.
- "You shall know a word by the company it keeps":
  - "He filled the **wampimuk**, passed it around and we all drunk some."
  - "We found a little, hairy **wampimuk** sleeping behind the tree."

Words and Contexts

A long line of NLP research aims to capture the distributional properties of words using a **word-context matrix** $\boldsymbol{M}$:

- each row w represents a **word** (e.g. "income"), each column c represents a linguistic **context** in which words can occur (e.g. "corporate ___ tax").
  - A matrix entry $\boldsymbol{M}_{[w,c]}$ quantifies the strength of association between a word and a context in a large corpus.

Words and Contexts

A long line of NLP research aims to capture the distributional properties of words using a **word-context matrix** $\boldsymbol{M}$:

- each row w represents a **word** (e.g. "income"), each column c represents a linguistic **context** in which words can occur (e.g. "corporate ___ tax").
  - A matrix entry $\boldsymbol{M}_{[w,c]}$ quantifies the strength of association between a word and a context in a large corpus.
- each word (row) w has a vector $\boldsymbol{M}_{[w,:]}$ giving a distribution over contexts.
  - normally, these vectors have a spatial interpretation $\rightarrow$ geometric distances between word vectors reflect semantic distances between words.

Words and Contexts

A long line of NLP research aims to capture the distributional properties of words using a **word-context matrix** $\boldsymbol{M}$:

- each row w represents a **word** (e.g. "income"), each column c represents a linguistic **context** in which words can occur (e.g. "corporate ___ tax").
  - A matrix entry $\boldsymbol{M}_{[w,c]}$ quantifies the strength of association between a word and a context in a large corpus.
- each word (row) w has a vector $\boldsymbol{M}_{[w,:]}$ giving a distribution over contexts.
  - normally, these vectors have a spatial interpretation $\rightarrow$ geometric distances between word vectors reflect semantic distances between words.
  - Different definitions of contexts and different measures of association $\rightarrow$ different types of word vectors.

Defining the context

- The simplest definition of context is neighboring words:
  - for “the tabby cat”: we get ($w =$ "the”, $c =$ "tabby"), ($w =$ "tabby”, $c =$ "the"), ($w =$ "tabby”, $c =$ "cat"), and ($w =$ "cat”, $c =$ "tabby")

Defining the context

- The simplest definition of context is neighboring words:
  - for "the tabby cat": we get $(w = \text{"the"}, c = \text{"tabby"})$, $(w = \text{"tabby"}, c = \text{"the"})$, $(w = \text{"tabby"}, c = \text{"cat"})$, and $(w = \text{"cat"}, c = \text{"tabby"})$
  - note the symmetry between words and contexts in this case.

Defining the context

- The simplest definition of context is neighboring words:
  - for "the tabby cat": we get $(w = \text{"the"}, c = \text{"tabby"})$, $(w = \text{"tabby"}, c = \text{"the"})$, $(w = \text{"tabby"}, c = \text{"cat"})$, and $(w = \text{"cat"}, c = \text{"tabby"})$
  - note the symmetry between words and contexts in this case.
- Could extend this to words within a window of two:
  - add $(w = \text{"the"}, c = \text{"cat"})$, $(w = \text{"cat"}, c = \text{"the"})$

Defining the context

- The simplest definition of context is neighboring words:
  - for "the tabby cat": we get $(w = \text{"the"}, c = \text{"tabby"})$, $(w = \text{"tabby"}, c = \text{"the"})$, $(w = \text{"tabby"}, c = \text{"cat"})$, and $(w = \text{"cat"}, c = \text{"tabby"})$
  - note the symmetry between words and contexts in this case.
- Could extend this to words within a window of two:
  - add $(w = \text{"the"}, c = \text{"cat"})$, $(w = \text{"cat"}, c = \text{"the"})$
  - word2vec and glove generally use 5- or 10-word windows.

Defining the context

- The simplest definition of context is neighboring words:
  - for "the tabby cat": we get $(w = \text{"the"}, c = \text{"tabby"})$, $(w = \text{"tabby"}, c = \text{"the"})$, $(w = \text{"tabby"}, c = \text{"cat"})$, and $(w = \text{"cat"}, c = \text{"tabby"})$
  - note the symmetry between words and contexts in this case.
- Could extend this to words within a window of two:
  - add $(w = \text{"the"}, c = \text{"cat"})$, $(w = \text{"cat"}, c = \text{"the"})$
  - word2vec and glove generally use 5- or 10-word windows.
- The context could be the tuple of preceding and subsequent words:
  - $(w = \text{"tabby"}, c = (\text{"the"}, \text{"cat"}))$

Defining the context

- The simplest definition of context is neighboring words:
  - for "the tabby cat": we get $(w = \text{"the"}, c = \text{"tabby"})$, $(w = \text{"tabby"}, c = \text{"the"})$, $(w = \text{"tabby"}, c = \text{"cat"})$, and $(w = \text{"cat"}, c = \text{"tabby"})$
  - note the symmetry between words and contexts in this case.
- Could extend this to words within a window of two:
  - add $(w = \text{"the"}, c = \text{"cat"})$, $(w = \text{"cat"}, c = \text{"the"})$
  - word2vec and glove generally use 5- or 10-word windows.
- The context could be the tuple of preceding and subsequent words:
  - $(w = \text{"tabby"}, c = (\text{"the"}, \text{"cat"}))$
- Could include all words in the same sentence.

Defining the context

- The simplest definition of context is neighboring words:
  - for "the tabby cat": we get $(w = \text{"the"}, c = \text{"tabby"})$, $(w = \text{"tabby"}, c = \text{"the"})$, $(w = \text{"tabby"}, c = \text{"cat"})$, and $(w = \text{"cat"}, c = \text{"tabby"})$
  - note the symmetry between words and contexts in this case.
- Could extend this to words within a window of two:
  - add $(w = \text{"the"}, c = \text{"cat"})$, $(w = \text{"cat"}, c = \text{"the"})$
  - word2vec and glove generally use 5- or 10-word windows.
- The context could be the tuple of preceding and subsequent words:
  - $(w = \text{"tabby"}, c = (\text{"the"}, \text{"cat"}))$
- Could include all words in the same sentence.
  - or same paragraph
  - or nouns in the same sentence
  - or syntactically connected words (from the parse tree)

Defining the context

- The simplest definition of context is neighboring words:
  - for "the tabby cat": we get $(w = \text{"the"}, c = \text{"tabby"})$, $(w = \text{"tabby"}, c = \text{"the"})$, $(w = \text{"tabby"}, c = \text{"cat"})$, and $(w = \text{"cat"}, c = \text{"tabby"})$
  - note the symmetry between words and contexts in this case.
- Could extend this to words within a window of two:
  - add $(w = \text{"the"}, c = \text{"cat"})$, $(w = \text{"cat"}, c = \text{"the"})$
  - word2vec and glove generally use 5- or 10-word windows.
- The context could be the tuple of preceding and subsequent words:
  - $(w = \text{"tabby"}, c = (\text{"the"}, \text{"cat"}))$
- Could include all words in the same sentence.
  - or same paragraph
  - or nouns in the same sentence
  - or syntactically connected words (from the parse tree)
  - ...
- Etc.

Association Measures

- Let $\boldsymbol{M}_{[w,c]} = f_M(w, c)$ where w and c are lookups to words in the w vocabulary and c vocabulary.
  - "word" could also mean phrases or more complicated objects.

Association Measures

- Let $\boldsymbol{M}_{[w,c]} = f_M(w,c)$ where w and c are lookups to words in the w vocabulary and c vocabulary.
  - "word" could also mean phrases or more complicated objects.
- **counts**: $f_M(w,c) = \#(w,c)$, the number of times w appeared along with context c.
- **document frequencies**: $f_M(w,c) = \frac{\#(w,c)}{n_D}$

Association Measures

- Let $\boldsymbol{M}_{[w,c]} = f_M(w, c)$ where w and c are lookups to words in the w vocabulary and c vocabulary.
  - "word" could also mean phrases or more complicated objects.
- **counts**: $f_M(w, c) = \#(w, c)$, the number of times w appeared along with context c.
- **document frequencies**: $f_M(w, c) = \frac{\#(w,c)}{n_D}$
  - puts high weight on very common contexts that are shared across many words (e.g., "the cat" will be weighted higher than "tabby cat")

Association Measures

- Let $\boldsymbol{M}_{[w,c]} = f_M(w,c)$ where w and c are lookups to words in the w vocabulary and c vocabulary.
  - "word" could also mean phrases or more complicated objects.
- **counts**: $f_M(w,c) = \#(w,c)$, the number of times w appeared along with context c.
- **document frequencies**: $f_M(w,c) = \frac{\#(w,c)}{n_D}$
  - puts high weight on very common contexts that are shared across many words (e.g., "the cat" will be weighted higher than "tabby cat")
- **Point-wise mutual information** (PMI) normalized for high frequency:

$$(w,c) = \log \frac{\Pr(w,c)}{\Pr(w)\Pr(c)} = \log \frac{\frac{\#(w,c)}{n_D}}{\frac{\#(w)}{n_D}\frac{\#(c)}{n_D}} = \log \frac{n_D\#(w,c)}{\#(w)\#(c)}$$

where $\#(w)$ and $\#(c)$ are the corpus counts for w and c, respectively.

Issues with PMI

$$f_M(w,c) = \mathrm{PMI}(w,c) = \log \frac{\#(w,c)}{\#(w)\#(c)}$$

- Issue 1: with PMI($\cdot$), unseen (w,c) pairs will take value $-\infty$.
  - So positive PMI (PPMI) is preferred:

$$\mathrm{PPMI}(w,c) = \max(\mathrm{PMI}(w,c), 0)$$

Issues with PMI

$$f_M(w,c) = \mathrm{PMI}(w,c) = \log \frac{\#(w,c)}{\#(w)\#(c)}$$

- Issue 1: with PMI($\cdot$), unseen (w,c) pairs will take value $-\infty$.
  - So positive PMI (PPMI) is preferred:

$$\mathrm{PPMI}(w,c) = \max(\mathrm{PMI}(w,c), 0)$$

- Issue 2: PMI assigns high value to rare word-context pairs.
  - so impose a minimum count threshold on (w,c) pairs; below the threshold, set to zero.

$\boldsymbol{M}$ is too high-dimensional

- $\boldsymbol{M}$ is $n_w \times n_c$
  - if c is drawn from from the vocabulary of a reasonably large corpus, the associated word vectors $\{v_1 = \boldsymbol{M}_{[w_1,:]}, v_2 = \boldsymbol{M}_{[w_2,:]}, \dots\}$ are too high-dimensional to be useful.

$\boldsymbol{M}$ is too high-dimensional

- $\boldsymbol{M}$ is $n_w \times n_c$
  - if c is drawn from from the vocabulary of a reasonably large corpus, the associated word vectors $\{v_1 = \boldsymbol{M}_{[w_1,:]}, v_2 = \boldsymbol{M}_{[w_2,:]}, \ldots\}$ are too high-dimensional to be useful.
- The standard approach is singular value decomposition (SVD):
  - factorize $\boldsymbol{M} \in \mathbb{R}^{n_w \times n_c}$ into a word matrix $\boldsymbol{W} \in \mathbb{R}^{n_w \times n_E}$ and context matrix $\boldsymbol{C} \in \mathbb{R}^{n_c \times n_E}$
  - such that $\tilde{\boldsymbol{M}} = \boldsymbol{W}\boldsymbol{C}'$ is the best rank-n_E approximation of $\boldsymbol{M}$.

$\boldsymbol{M}$ is too high-dimensional

- $\boldsymbol{M}$ is $n_w \times n_c$
  - if c is drawn from from the vocabulary of a reasonably large corpus, the associated word vectors $\{v_1 = \boldsymbol{M}_{[w_1,:]}, v_2 = \boldsymbol{M}_{[w_2,:]}, \ldots\}$ are too high-dimensional to be useful.
- The standard approach is singular value decomposition (SVD):
  - factorize $\boldsymbol{M} \in \mathbb{R}^{n_w \times n_c}$ into a word matrix $\boldsymbol{W} \in \mathbb{R}^{n_w \times n_E}$ and context matrix $\boldsymbol{C} \in \mathbb{R}^{n_c \times n_E}$
  - such that $\tilde{\boldsymbol{M}} = \boldsymbol{W}\boldsymbol{C}'$ is the best rank-n_E approximation of $\boldsymbol{M}$.
  - $\tilde{\boldsymbol{M}}$ can be seen as a "smoothed" version of $\boldsymbol{M}$; "missing" values are filled in, etc.

$\boldsymbol{M}$ is too high-dimensional

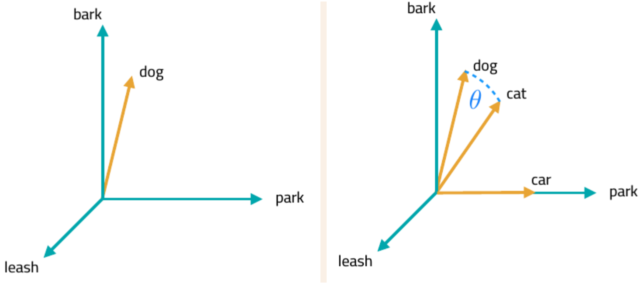
- $\boldsymbol{M}$ is $n_w \times n_c$
  - if c is drawn from from the vocabulary of a reasonably large corpus, the associated word vectors $\{v_1 = \boldsymbol{M}_{[w_1,:]}, v_2 = \boldsymbol{M}_{[w_2,:]}, \ldots\}$ are too high-dimensional to be useful.
- The standard approach is singular value decomposition (SVD):
  - factorize $\boldsymbol{M} \in \mathbb{R}^{n_w \times n_c}$ into a word matrix $\boldsymbol{W} \in \mathbb{R}^{n_w \times n_E}$ and context matrix $\boldsymbol{C} \in \mathbb{R}^{n_c \times n_E}$
  - such that $\tilde{\boldsymbol{M}} = \boldsymbol{W}\boldsymbol{C}'$ is the best rank-n_E approximation of $\boldsymbol{M}$.
  - $\tilde{\boldsymbol{M}}$ can be seen as a "smoothed" version of $\boldsymbol{M}$; "missing" values are filled in, etc.
- $\boldsymbol{W}$ is the matrix of word vectors (word embeddings):
  - relatively low-dimensional ($n_E << n_w$, typically between 50 and 300)
  - dense, rather than sparse.

$\boldsymbol{M}$ is too high-dimensional

- $\boldsymbol{M}$ is $n_w \times n_c$
  - if c is drawn from from the vocabulary of a reasonably large corpus, the associated word vectors $\{v_1 = \boldsymbol{M}_{[w_1,:]}, v_2 = \boldsymbol{M}_{[w_2,:]}, \ldots\}$ are too high-dimensional to be useful.
- The standard approach is singular value decomposition (SVD):
  - factorize $\boldsymbol{M} \in \mathbb{R}^{n_w \times n_c}$ into a word matrix $\boldsymbol{W} \in \mathbb{R}^{n_w \times n_E}$ and context matrix $\boldsymbol{C} \in \mathbb{R}^{n_c \times n_E}$
  - such that $\tilde{\boldsymbol{M}} = \boldsymbol{W}\boldsymbol{C}'$ is the best rank-n_E approximation of $\boldsymbol{M}$.
  - $\tilde{\boldsymbol{M}}$ can be seen as a "smoothed" version of $\boldsymbol{M}$; "missing" values are filled in, etc.
- $\boldsymbol{W}$ is the matrix of word vectors (word embeddings):
  - relatively low-dimensional ($n_E << n_w$, typically between 50 and 300)
  - dense, rather than sparse.
  - similarity measures between rows of $\boldsymbol{W}$ approximate similarity measures between rows of $\boldsymbol{M}$

Word Similarity

- Once words are represented as vectors $\{v_1 = \boldsymbol{M}_{[w_1,:]},\ v_2 = \boldsymbol{M}_{[w_2,:]}, \ldots\}$, we can use linear algebra to understand the relationships between words:
  - Words that are geometrically close to each other are similar: e.g. "dog" and "cat":



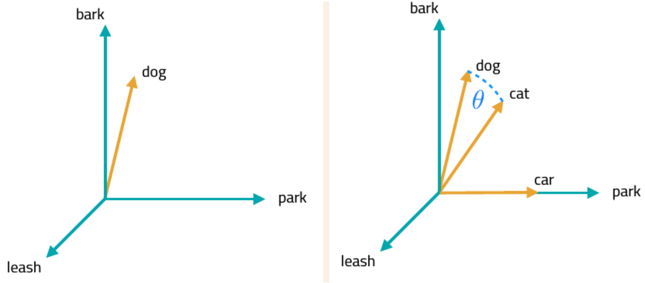
- The standard metric for comparing vectors is cosine similarity:

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

  - alternatives include e.g. Jaccard similarity (Goldberg 2017)

Word Similarity

- Once words are represented as vectors $\{v_1 = \boldsymbol{M}_{[w_1,:]},\ v_2 = \boldsymbol{M}_{[w_2,:]}, \dots\}$, we can use linear algebra to understand the relationships between words:
  - Words that are geometrically close to each other are similar: e.g. "dog" and "cat":



- The standard metric for comparing vectors is cosine similarity:

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

  - alternatives include e.g. Jaccard similarity (Goldberg 2017)
- Thanks to linearity, can compute similarities between groups of words by averaging the groups.

Word2Vec

- When people mention “word2vec”, they are usually talking about a particular word-embedding model with good performance on a range of analogy and prediction tasks.

Word2Vec

- When people mention "word2vec", they are usually talking about 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"?
  - By comparing true instances of the word fox ("The quick brown **fox** jumps over the lazy dog")
  - to fake (randomly sampled) ones ("The prescription of **fox** is advised for this diagnosis")

Word2Vec

- When people mention "word2vec", they are usually talking about 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"?
  - By comparing true instances of the word fox ("The quick brown **fox** jumps over the lazy dog")
  - to fake (randomly sampled) ones ("The prescription of **fox** is advised for this diagnosis")
- Word2Vec learns embedding vectors for the target word ("fox") and context words (neighbors of "fox") to distinguish true from false samples.

Word2Vec Objective

- The dataset is a collection of context pairs indexed by i:
  - $y_i = 1$ means correct (it appeared in the corpus)
  - $y_i = 0$ means incorrect (it was randomly drawn).

Word2Vec Objective

- The dataset is a collection of context pairs indexed by i:
  - $y_i = 1$ means correct (it appeared in the corpus)
  - $y_i = 0$ means incorrect (it was randomly drawn).
  - This random drawing of incorrect pairs is called "**negative sampling**". It is a form of unsupervised learning.

Word2Vec Objective

- The dataset is a collection of context pairs indexed by i:
  - $y_i = 1$ means correct (it appeared in the corpus)
  - $y_i = 0$ means incorrect (it was randomly drawn).
  - This random drawing of incorrect pairs is called "**negative sampling**". It is a form of unsupervised learning.
- Let $\hat{y}_i(w, c; \theta)$ be the predicted probability of a correct pair. Word2Vec minimizes the binary cross-entropy

$$L(\theta) = -\sum_{i=1}^{n_D} [y_i \log \hat{y}_i(w, c; \theta) + [1 - y_i] \log(1 - \hat{y}_i(w, c; \theta))]$$

The neural net

We have an embedding matrix for target words, $\boldsymbol{E}_w$, and for context words, $\boldsymbol{E}_c$.

- Let $v_w(w) = \boldsymbol{E}_w \cdot w$ and $v_c(c) = \boldsymbol{E}_c \cdot c$ be the associated lookup functions.

The neural net

We have an embedding matrix for target words, $\boldsymbol{E}_w$, and for context words, $\boldsymbol{E}_c$.

- Let $v_w(w) = \boldsymbol{E}_w \cdot w$ and $v_c(c) = \boldsymbol{E}_c \cdot c$ be the associated lookup functions.

Embedding layer for instance i:

- $\boldsymbol{w}_i = v_w(w_i)$

The neural net

We have an embedding matrix for target words, $\boldsymbol{E}_w$, and for context words, $\boldsymbol{E}_c$.

- Let $v_w(w) = \boldsymbol{E}_w \cdot w$ and $v_c(c) = \boldsymbol{E}_c \cdot c$ be the associated lookup functions.

Embedding layer for instance i:

- $\boldsymbol{w}_i = v_w(w_i)$
- **Skip-gram** step:
  - draw one word c_i from i's context window (n_{cw} words before or after w_i)
  - look up $\boldsymbol{c}_i = v_c(c_i)$

The neural net

We have an embedding matrix for target words, $\boldsymbol{E}_w$, and for context words, $\boldsymbol{E}_c$.

- Let $v_w(w) = \boldsymbol{E}_w \cdot w$ and $v_c(c) = \boldsymbol{E}_c \cdot c$ be the associated lookup functions.

Embedding layer for instance i:

- $\boldsymbol{w}_i = v_w(w_i)$
- **Skip-gram** step:
  - draw one word c_i from i's context window (n_{cw} words before or after w_i)
  - look up $\boldsymbol{c}_i = v_c(c_i)$

The embedding $[\boldsymbol{w}; \boldsymbol{c}]$ is input to MLP1 (no activation) with sigmoid output:

$$\hat{y}(w, c) = \text{sigmoid}(([\boldsymbol{w}; \boldsymbol{c}] \cdot \omega_0) \cdot \omega_1)$$

The neural net

We have an embedding matrix for target words, $\boldsymbol{E}_w$, and for context words, $\boldsymbol{E}_c$.

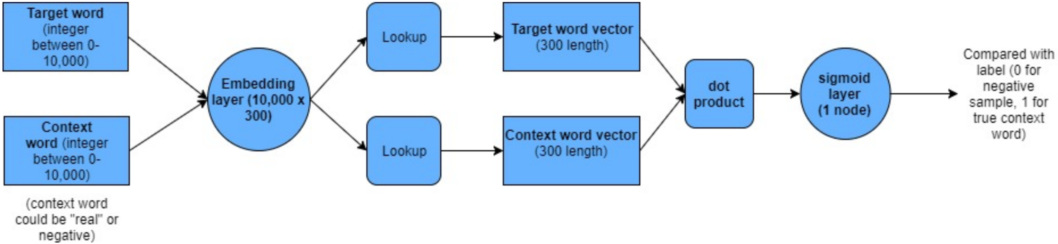
- Let $v_w(w) = \boldsymbol{E}_w \cdot w$ and $v_c(c) = \boldsymbol{E}_c \cdot c$ be the associated lookup functions.

Embedding layer for instance i:

- $\boldsymbol{w}_i = v_w(w_i)$
- **Skip-gram** step:
  - draw one word c_i from i's context window (n_{cw} words before or after w_i)
  - look up $\boldsymbol{c}_i = v_c(c_i)$

The embedding $[\boldsymbol{w}; \boldsymbol{c}]$ is input to MLP1 (no activation) with sigmoid output:

$$\hat{y}(w, c) = \text{sigmoid}(([\boldsymbol{w}; \boldsymbol{c}] \cdot \omega_0) \cdot \omega_1)$$



Does does this relate to $\boldsymbol{M}$?

- Word2Vec produces embedding matrices $\boldsymbol{E}_w$ and $\boldsymbol{E}_c$.
  - generally, context embeddings are discarded after training.

Does does this relate to $\boldsymbol{M}$?

- Word2Vec produces embedding matrices $\boldsymbol{E}_w$ and $\boldsymbol{E}_c$.
  - generally, context embeddings are discarded after training.
- If we take $\tilde{M} = E_w E_c'$, Levy and Goldberg (2014) show that word2vec is equivalent to factorizing a matrix $\boldsymbol{M}$ with items

$$\boldsymbol{M}_{[w,c]} = \mathrm{PMI}(w, c) - \log a$$

where a is a constant calibrating the amount of negative sampling.

GloVe Embeddings

- Pennington et al (2014) (GloVe = Global Vectors) take a different approach:
  - that does not require a neural net
- Let $\#(w, c)$ be the co-occurence counts within some window, e.g. 10 words.

GloVe Embeddings

- Pennington et al (2014) (GloVe = Global Vectors) take a different approach:
  - that does not require a neural net
- Let $\#(w, c)$ be the co-occurence counts within some window, e.g. 10 words.
- Learn $\boldsymbol{W}$ and $\boldsymbol{C}$ (containing vectors $\boldsymbol{w}$ and $\boldsymbol{c}$ corresponding to w and c, respectively), to solve

$$\min_{\boldsymbol{W},\boldsymbol{C}} \sum_{(w,c)} \phi(\#(w,c))(\boldsymbol{w}\cdot\boldsymbol{c} - \log(\#(w,c)))^2$$

  - $\phi(\cdot)$ is weighting function to down-weight frequent words.

GloVe Embeddings

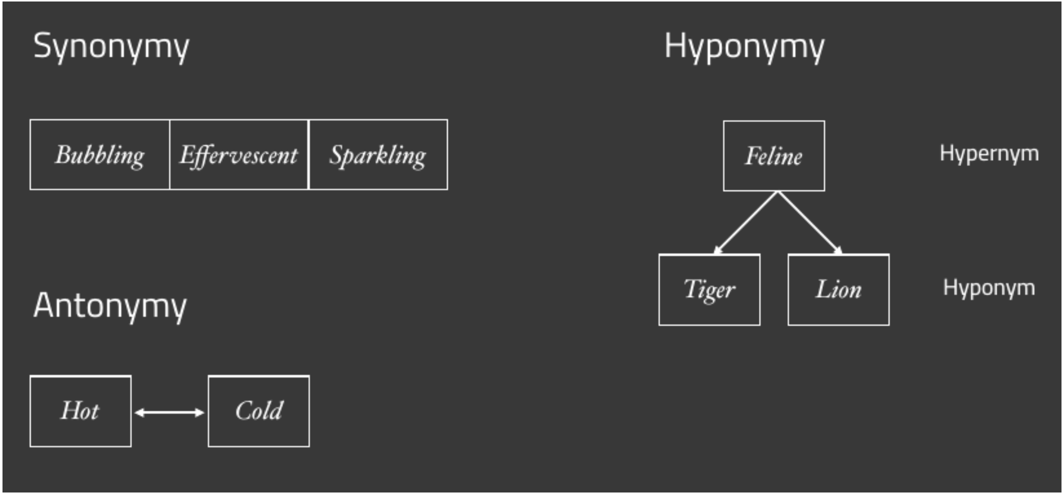
- Pennington et al (2014) (GloVe = Global Vectors) take a different approach:
  - that does not require a neural net
- Let $\#(w, c)$ be the co-occurence counts within some window, e.g. 10 words.
- Learn $\boldsymbol{W}$ and $\boldsymbol{C}$ (containing vectors $\boldsymbol{w}$ and $\boldsymbol{c}$ corresponding to w and c, respectively), to solve

$$\min_{\boldsymbol{W},\boldsymbol{C}} \sum_{(w,c)} \phi(\#(w,c))\left(\boldsymbol{w}\cdot\boldsymbol{c} - \log\left(\#(w,c)\right)\right)^2$$

  - $\phi(\cdot)$ is weighting function to down-weight frequent words.
- Minimizes **squared difference** between**:**
  - **dot product of word vectors,** $\boldsymbol{w}\cdot\boldsymbol{c}$
  - **empirical co-occurrence,** $\log(\#(w,c))$
- Intuitively: words that co-occur have high correlation (dot product)

Word Embeddings Encode Linguistic Relations

Word Embeddings Encode Linguistic Relations



Similarity vs. Relatedness (Budansky and Hirst, 2006)

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

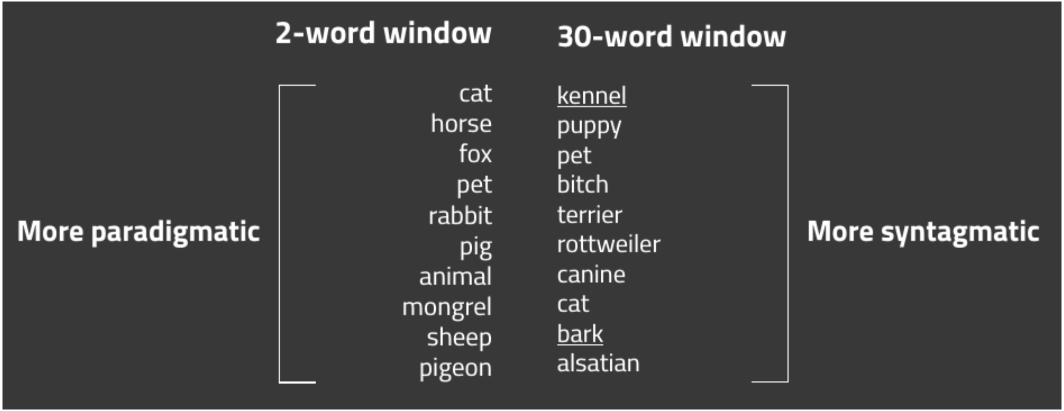
Similarity vs. Relatedness (Budansky and Hirst, 2006)

- 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)

Similarity vs. Relatedness (Budansky and Hirst, 2006)

- 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)
- Word embeddings will recover one or both of these relations, depending on how contexts and associated are constructed.

Most similar words to dog, depending on context window size



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

Target Word	BoW5	BoW2	Deps
batman	nightwing aquaman catwoman superman manhunter	superman superboy aquaman catwoman batgirl	superman superboy supergirl catwoman aquaman
hogwarts	dumbledore hallows half-blood malfoy snape	evernight sunnydale garderobe blandings collinwood	sunnydale collinwood calarts greendale millfield
turing	nondeterministic non-deterministic computability deterministic finite-state	non-deterministic finite-state nondeterministic buchi primality	pauling hotelling heting lessing hamming
florida	gainesville fla jacksonville tampa lauderdale	fla alabama gainesville tallahassee texas	texas louisiana georgia california carolina
object-oriented	aspect-oriented smalltalk event-driven prolog domain-specific	aspect-oriented event-driven objective-c dataflow 4gl	event-driven domain-specific rule-based data-driven human-centered
dancing	singing dance dances dancers tap-dancing	singing dance dances breakdancing clowning	singing rapping breakdancing miming busking

Parts of Speech and Phrases

- In the default model multiple senses of a word are merged.
    - e.g. "I like a bird" (verb) and "I am like a bird" (preposition).

Parts of Speech and Phrases

- In the default model multiple senses of a word are merged.
  - e.g. "I like a bird" (verb) and "I am like a bird" (preposition).
- Can significantly improve the quality of embeddings in these cases by attaching the POS to the word (e.g. "like:verb", "like:prep") before training.

Parts of Speech and Phrases

- In the default model multiple senses of a word are merged.
  - e.g. “I like a bird” (verb) and “I am like a bird” (preposition).

- Can significantly improve the quality of embeddings in these cases by attaching the POS to the word (e.g. “like:verb”, “like:prep”) before training.
- The default model only works by word, but “new york $\neq$ ”new” + “york”
  - it makes sense to tokenize phrases together (see Week 2 lecture) before training.

The black sheep problem

- The trivial or obvious features of a word are not mentioned in standard corpora.

The black sheep problem

- The trivial or obvious features of a word are not mentioned in standard corpora.
- For example, although most sheep are white, you rarely see the phrase “white sheep”.
    - so word2vec tells you sim(black,sheep) $>$ sim(white,sheep).

The black sheep problem

- The trivial or obvious features of a word are not mentioned in standard corpora.
- For example, although most sheep are white, you rarely see the phrase "white sheep".
  - so word2vec tells you sim(black,sheep) $>$ sim(white,sheep).
- This is really important when we will use embeddings to analysis social biases. And I don't see a solution to it.

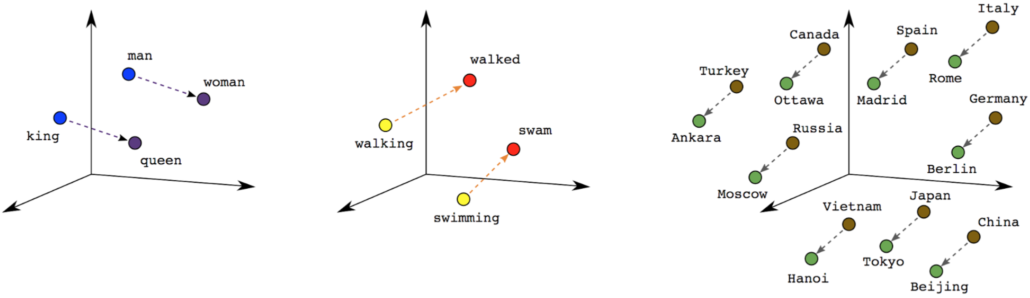
The black sheep problem

- The trivial or obvious features of a word are not mentioned in standard corpora.
- For example, although most sheep are white, you rarely see the phrase “white sheep”.
  - so word2vec tells you sim(black,sheep) $>$ sim(white,sheep).
- This is really important when we will use embeddings to analysis social biases. And I don’t see a solution to it.

- Relatedly, antonyms are often rated similarly, be careful with that.

Vector Directions ↔ Meaning

- Intriguingly, word2vec algebra can depict conceptual, analogical relationships between words:



Male-Female Verb Tense Country-Capital

Word Embeddings for Analogies

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

Word Embeddings for Analogies

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

- More generally: The analogy $a_1 : b_1 :: a_2 : b_2$ can be solved (that is, find b_2 given a_1, b_1, a_2) by

$$\arg\max_{b_2 \in V} \cos(b_2, a_2 - a_1 + b_1)$$

where V excludes (a_1, b_1, a_2).

Word Embeddings for Analogies

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

- More generally: The analogy $a_1 : b_1 :: a_2 : b_2$ can be solved (that is, find b_2 given a_1, b_1, a_2) by

$$\arg\max_{b_2 \in V} \cos(b_2, a_2 - a_1 + b_1)$$

  where V excludes (a_1, b_1, a_2).
- Often works better with normalized vectors (so that one long vector doesnt wash out the others)

Word Embeddings for Analogies

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

- More generally: The analogy $a_1 : b_1 :: a_2 : b_2$ can be solved (that is, find b_2 given a_1, b_1, a_2) by
$$\arg\max_{b_2 \in V} \cos(b_2, a_2 - a_1 + b_1)$$
where V excludes (a_1, b_1, a_2).
- Often works better with normalized vectors (so that one long vector doesnt wash out the others)
- Levy and Goldberg (2014) recommend the following "CosMul" metric which tends to perform better:
$$\arg\max_{b_2 \in V} \frac{\cos(b_2, a_2)\cos(b_2, b_1)}{\cos(b_2, a_1) + \epsilon}$$
    - requires normalized, non-negative vectors (can transform using $(x + 1)/2$)
    - ϵ is a small smoothing parameter.

Word embeddings vs topic models

Ben Schmidt:

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

Word embeddings vs 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.

Tokenizing for Embeddings

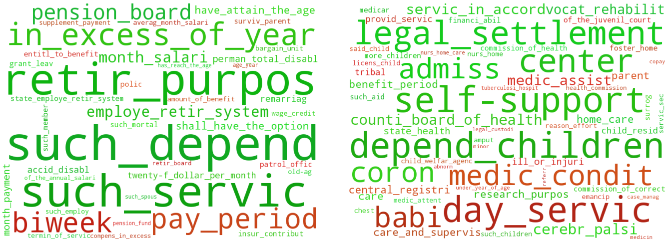
- embeddings work better with more information about the placement of words in sentences.
  - don't drop stopwords/function-words
  - should include tokens for start of sentence and end of sentence
  - should include a special token for out-of-vocabulary words
    - or replace with the part of speech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

Word Dropout

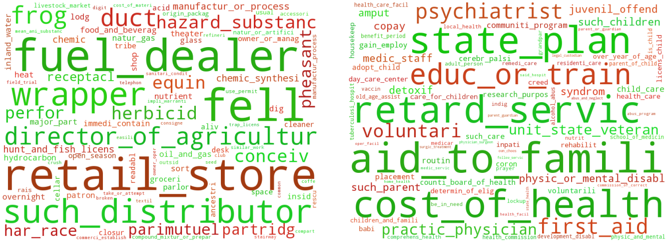
- When training models, words can be randomly replaced with the unknown symbol with some small probability (Iyyer et al 2015).
- Prevents models from relying too much on particular words.

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.

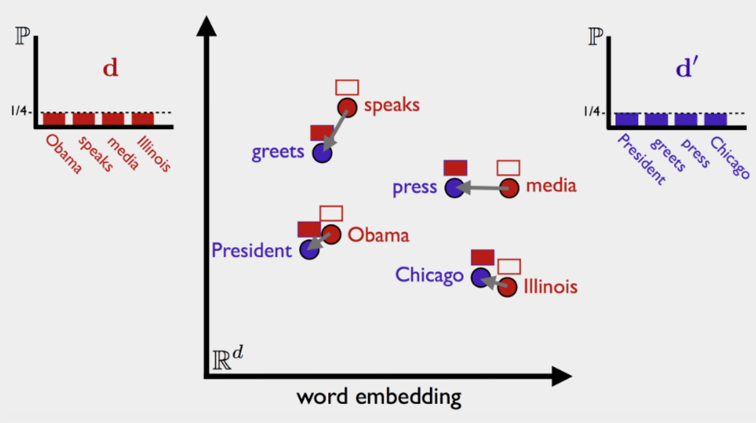
Word Mover Distance

- 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

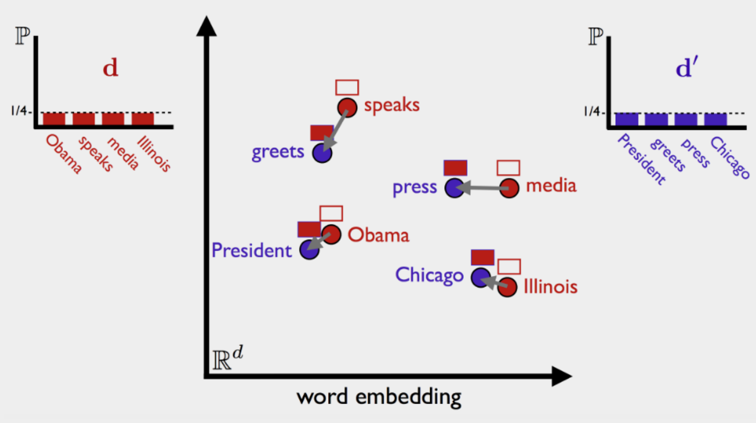
- Cosine distance treats synonyms as just as close as totally unrelated words.
- Word mover distance between two texts is given by:
  - total amount of "mass" needed to move words from one side into the other
  - multiplied by the distance the words need to move
  - Kusner, Sun, Kolkin, and Weinberger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