Word Embeddings and Word2Vec

Deep Learning for NLP: Lecture 5

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilians-Universität München
poerner@cis.uni-muenchen.de

December 02, 2020

Outline

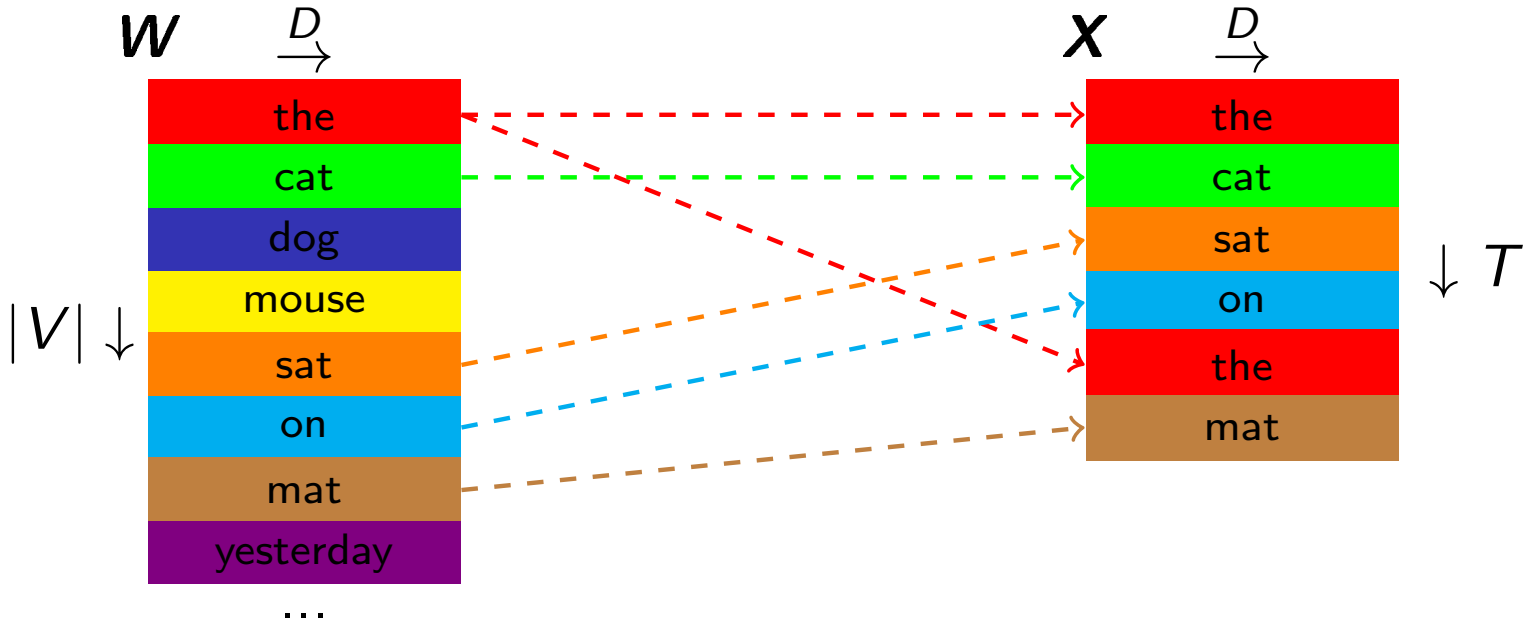
Outline

Motivation for word embeddings

- Deep Learning models understand numerical representations (vectors, matrices...)
- Texts are sequences of symbolic units (words, ngrams, ...)
- Step 1 in any NLP deep learning model: Turn symbols (e.g., words) into vectors

Word vector lookup

- For now, assume that we have finite word vocabulary V e.g.
  - $V = \{\text{the, cat, dog, mouse, sat, on, mat, yesterday, ...}\}$ with a bijective indexing function $\mathcal{I}_V : V \to \{1 \dots |V|\}$
- Let $\boldsymbol{W} \in \mathbb{R}^{|V| \times D}$ be a matrix whose rows are word vectors
- Represent word $x \in V$ as $\boldsymbol{w}^{(x)} = \boldsymbol{w}_{\mathcal{I}_V(x)}$
- Represent sentence $X = (x_1 \dots x_T)$ as matrix $\boldsymbol{X}$ with $\boldsymbol{x}_t = \boldsymbol{w}^{(x_t)}$.
- Then use $\boldsymbol{X}$ as input to downstream model (CNN, RNN ...)



Motivation for word embeddings

- So where do we get $\boldsymbol{W}$ from?
- Simple solution: Identity matrix, with $w_i^{(x)} = 1$ if $i = \mathcal{I}_V(x)$ and $w_i^{(x)} = 0$ otherwise

$$\boldsymbol{w}^{(\text{the})} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix} \quad \boldsymbol{w}^{(\text{cat})} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \end{bmatrix} \quad \boldsymbol{w}^{(\text{dog})} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

- All vectors are orthogonal to each other $\rightarrow$ no notion of word similarity
- That's not what we want...

Motivation for word embeddings

- Word embeddings are dense ($\neq$ sparse), trainable word vectors
- $\boldsymbol{w}^{(i)} \in \mathbb{R}^D$, where typically $50 \leq D \leq 1000 \ll |V|$
- Allows us to calculate similarities between words, e.g., with cosine similarity:

$$\text{sim}(\boldsymbol{w}^{(i)}, \boldsymbol{w}^{(j)}) = \frac{\boldsymbol{w}^{(i)T} \boldsymbol{w}^{(j)}}{\|\boldsymbol{w}^{(i)}\|_2 \cdot \|\boldsymbol{w}^{(j)}\|_2}$$

Word embeddings from scratch

- Word embeddings can be trained from scratch on supervised data:
  - Initialize $\boldsymbol{W}$ randomly
  - Use as input layer to some model (CNN, RNN, ...)
  - During training, back-propagate gradients from the model into the embedding layer, and update $\boldsymbol{W}$
- Result: Words that play similar roles in the training task get similar embeddings.
- For example: If our training task is sentiment analysis, we might expect

$$\text{sim}(\boldsymbol{w}^{(\text{great})}, \boldsymbol{w}^{(\text{awesome})}) \approx 1$$

Motivation for pretrained word embeddings

- Supervised training sets tend to be small (labeled data is expensive)
- Many words that are relevant at test time will be infrequent or unseen at training time
- The embeddings of infrequent words may have low quality, unseen words have no embeddings at all.

- We have more unlabeled than labeled data.
- So let's pretrain embeddings on the unlabeled data first.

Questions?

Take a moment to write down any questions you have for the QA session!

Outline

Word2Vec

- Proposed in Mikolov et al. (2013): Efficient estimation of word representations in vector space.
- The basis of many embedding pretraining models (including Word2Vec) is the **distributional hypothesis**:
- "a word is characterized by the company it keeps"' (Firth, 1957)

Outline

The skipgram task

- Let X be an unlabeled text (sequence of words), and let M be a hyperparameter
- Let x_t be the t'th word, and let $C_t = (x_{t-M} \dots x_{t-1}, x_{t+1} \dots x_{t+M})$ be a context window around x_t.
- Example:

$$M = 1, X = (\text{the}, \text{cat}, \text{sat}, \text{on}, \text{the}, \text{mat})$$

- $x_t = \text{the}, C_t = (\text{cat})$
- $x_t = \text{cat}, C_t = (\text{the}, \text{sat})$
- $x_t = \text{sat}, C_t = (\text{cat}, \text{on})$
- $x_t = \text{on}, C_t = (\text{sat}, \text{the})$
- ... and so on.

Skipgram task

- The skipgram task is to maximize the likelihood of the context words, given their center word:

$$\text{argmax} \prod_{x_t, C_t} \prod_{x_{t'} \in C_t} P(x_{t'}|x_t)$$

- Expressed as a negative log likelihood loss:

$$\mathcal{J} = \sum_{x_t, C_t} \sum_{x'_t \in C_t} -\log(P(x_{t'}|x_t))$$

Outline

CBOW task

- The continuous bag of words (CBOW) task is to maximize the likelihood of the center words, given their context words:

$$\text{argmax} \prod_{x_t, C_t} P(x_t | C_t)$$

- Expressed as a negative log likelihood loss:

$$\mathcal{J} = \sum_{x_t, C_t} -\log(P(x_t | C_t))$$

Skipgram and CBOW

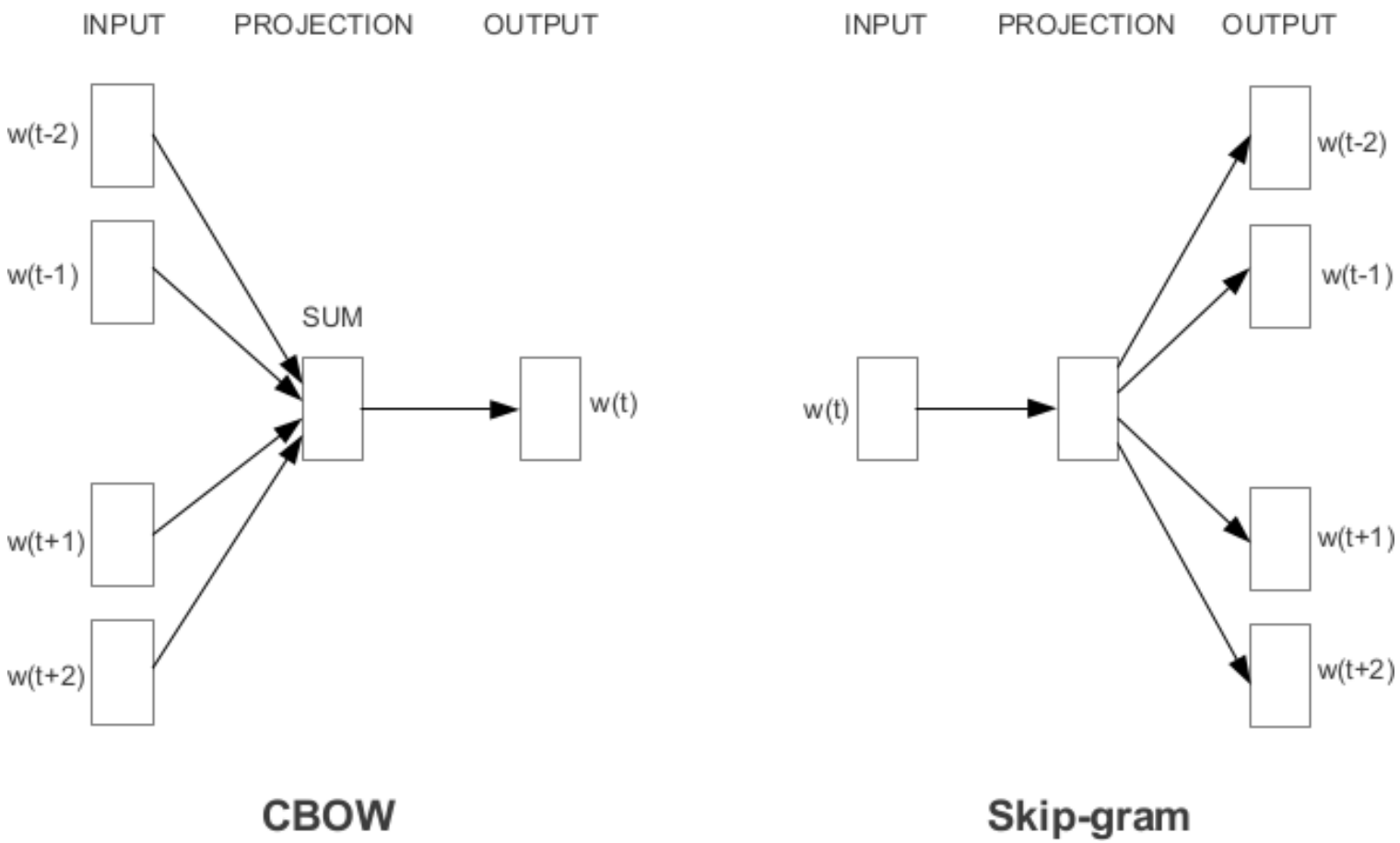


Figure from Mikolov et al. (2013): Efficient Estimation of Word Representations in Vector Space

Outline

Naive softmax model

- Let all words be represented as two types of vectors:
  - $\boldsymbol{v}^{(*)} \in \mathbb{R}^D$ (if they are being predicted)
  - $\boldsymbol{w}^{(*)} \in \mathbb{R}^D$ (if they are being conditioned on)
  - Total number of parameters: $2 \cdot |V| \cdot D$
- Skipgram:

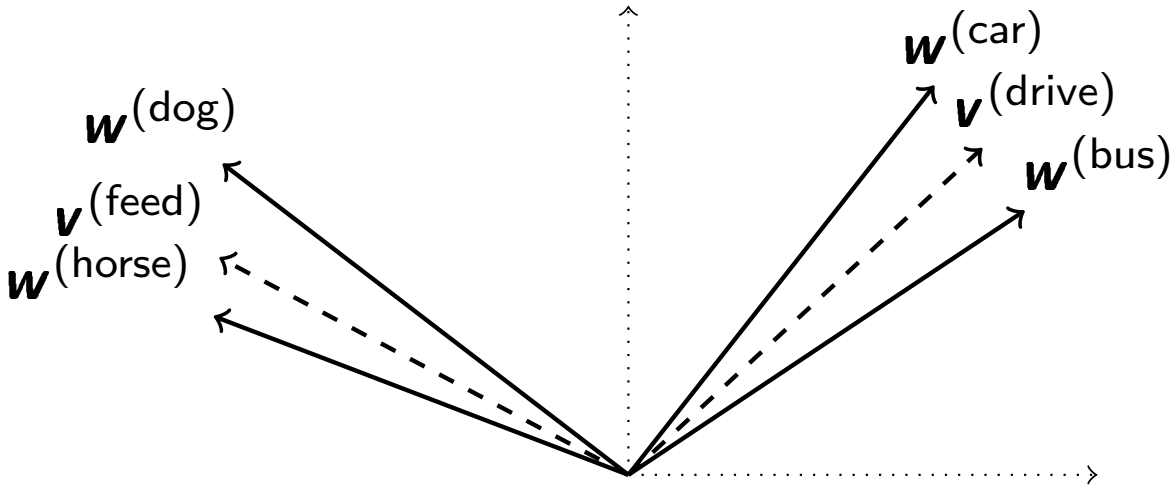
$$P(x_{t'}|x_t) = \text{softmax}(\boldsymbol{V}\boldsymbol{w}^{(x_t)})_{\mathcal{I}_V(x_{t'})} = \frac{exp(\boldsymbol{w}^{(x_t)T}\boldsymbol{v}^{(x_{t'})})}{\sum_{\bar{x}\in V} exp(\boldsymbol{w}^{(x_t)T}\boldsymbol{v}^{(\bar{x})})}$$

- CBOW:
  - Let $\boldsymbol{w}^{(C_t)}$ be shorthand for $\sum_{x_{t'}\in C_t} \boldsymbol{w}^{(x_{t'})}$. Then:

$$P(x_t|C_t) = \text{softmax}(\boldsymbol{V}\boldsymbol{w}^{(C_t)})_{\mathcal{I}_V(x_t)} = \frac{exp(\boldsymbol{w}^{(C_t)T}\boldsymbol{v}^{(x_t)})}{\sum_{\bar{x}\in V} exp(\boldsymbol{w}^{(C_t)T}\boldsymbol{v}^{(\bar{x})})}$$

Outcome

- $\boldsymbol{w}^{(*)}$ and $\boldsymbol{v}^{(*)}$ vectors express syntagmatic relations (e.g., "car" and "drive").
- $\boldsymbol{w}^{(*)}$ vectors express paradigmatic relations (e.g., "car" and "bus")
- Usually, $\boldsymbol{w}^{(*)}$ vectors are used for downstream applications

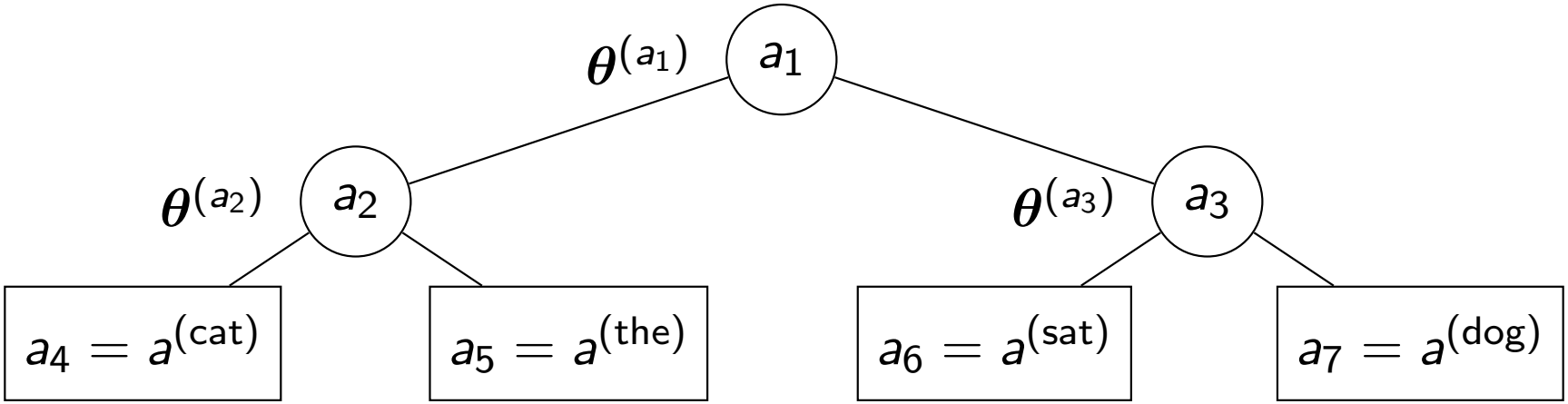


Naive softmax model: Complexity

- **Question:** How many dot products do we need in order to predict a single probability $P(n|m)$?
  - For both CBOW and skpigram, the softmax is defined over $|V|$ dot products.
  - So when V is big (which it usually is), then prediction (and training) is slow.
- Solution: Replace naive softmax with hierarchical softmax or negative sampling

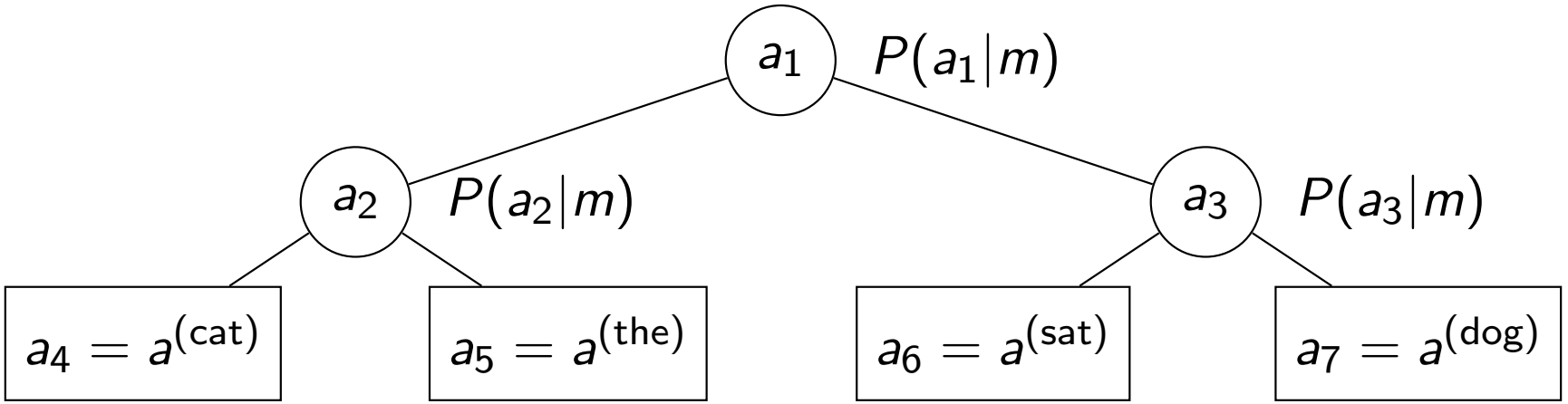
Outline

Hierarchical softmax model



- $\boldsymbol{w}^{(*)}$ vectors are defined like before
- $\boldsymbol{v}^{(*)}$ vectors are replaced by a binary tree with nodes $A = \{a_1, a_2 \ldots\}$
- Let $A' \subset A$ be the set of leaf nodes (rectangles). Every $a' \in A'$ corresponds to a unique word in V.
- Every non-leaf node (circles) has a parameter vector $\boldsymbol{\theta}^{(a)} \in \mathbb{R}^D$.
- We denote the parent of some node a as $q(a)$, its left child as $l(a)$ and its right child as $r(a)$, e.g., $a^{(2)} = l(a^{(1)}) = q(a^{(4)}) = q(a^{(5)})$

Hierarchical softmax model



$$P(\text{cat}|m) = P(a^{(\text{cat})}|m)\, P(\text{the}|m) = P(a^{(\text{the})}|m)\, P(\text{sat}|m) = P(a^{(\text{sat})}|m) P(\text{dog}|m) = P(a^{(\text{dog})}|m)$$

- Let m be the thing that we are conditioning on, and let $\boldsymbol{w}^{(m)}$ be its vector.
  - For skipgram, m is the center word x_t, and $\boldsymbol{w}^{(m)} = \boldsymbol{w}^{(x_t)}$
  - For CBOW, m is the context C_t, and $\boldsymbol{w}^{(m)} = \boldsymbol{w}^{(C_t)} = \sum_{x_{t'} \in C_t} \boldsymbol{w}^{(x_{t'})}$
- Every node has a conditional probability $P(a|m)$
- Let n be the word that we are trying to predict. Then we need to calculate the probability of the corresponding leaf node, given m: $P(n|m) = P(a^{(n)}|m)$

Hierarchical softmax model

- The conditional probability of the root node is always 1.
- The conditional probabilitiy of some node $a \in A$ is defined through recursion:

$$P(a|m) = P(q(a)|m)P(a|q(a), m)$$

- The probability of selecting the left child:

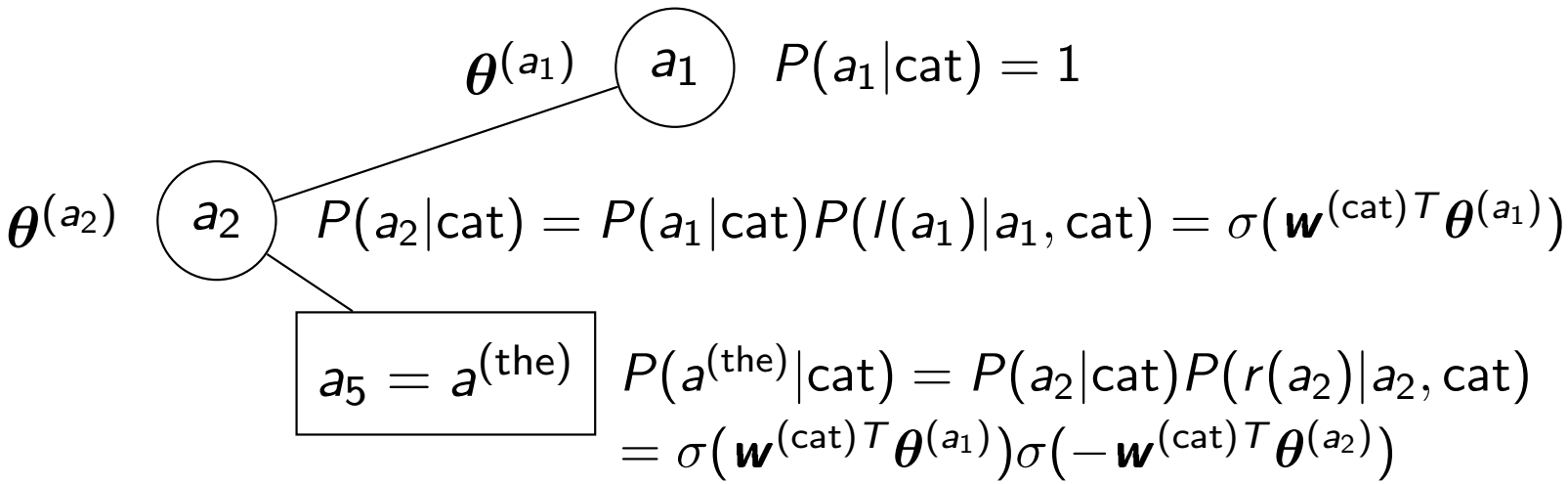
$$P(l(a)|a, m) = \sigma(\boldsymbol{w}^{(m)T}\boldsymbol{\theta}^{(a)})$$

- The probability of selecting the right child:

$$P(r(a)|a, m) = 1 - P(l(a)|a, m) = \sigma(-\boldsymbol{w}^{(m)T}\boldsymbol{\theta}^{(a)})$$

Hierarchical softmax model

Example: Let's calculate $P(\text{the}|\text{cat})$



Hierarchical softmax model: Complexity

- **Question:** How many dot products do we need in order to compute $P(n|m)$ (assuming that the tree is balanced)? How does this compare to the naive softmax?
  - Depth of the tree, i.e., $\log_2(|A'|) = \log_2(|V|)$
  - $\log_2(|V|) \ll |V|$ (e.g., $\log_2(1000000) \approx 20$)

- **Question:** Show that $\sum_{n \in V} P(n|m) = 1$.
- There is a one-to-one correspondence between V and the set of leaf nodes A', so we must show that $\sum_{a' \in A'} P(a'|m) = 1$

- Imagine re-building the tree top-down. Let A'_j be the set of leaf nodes at step j, and let $A'_0 = \{a_1\}$. Let J be the number of steps required to finish the tree, i.e., $A' = A'_J$.

- Building procedure:
  - For every step $1 \leq j \leq J$, we select a leaf node $a'_j \in A'_{j-1}$ and split it into two new leaf nodes $l(a'_j), r(a'_j)$. Of course, a'_j stops being a leaf node at this point.
  - Formally: $A'_j = A'_{j-1} \cup \{l(a'_j), r(a'_j)\} \setminus \{a'_j\}$

- Base case:

$$\sum_{a' \in A'_0} P(a'|m) = P(a_1|m) = 1$$

- Induction step: We must show that:

$$\sum_{a' \in A'_{j-1}} P(a'|m) = 1 \implies \sum_{a' \in A'_j} P(a'|m) = 1$$

- Let's try:

$$\begin{aligned}\sum_{a' \in A'_j} P(a'|m) &= \sum_{a' \in A'_{j-1} \cup \{l(a'_j), r(a'_j)\} \setminus \{a'_j\}} P(a'|m) \\ &= \Big(\sum_{a' \in A'_{j-1}} P(a'|m) \Big) + P(l(a'_j)|m) + P(r(a'_j)|m) - P(a'_j|m)\end{aligned}$$

- If we could show that $P(l(a'_j)|m) + P(r(a'_j)|m) - P(a'_j|m) = 0$, it would follow that $\sum_{a' \in A'_j} P(a'|m) = \sum_{a' \in A'_{j-1}} P(a'|m)$. This would would fulfill the induction step.

- Recall from our model definition:

$$P(a|m) = P(q(a)|m)P(a|q(a), m)$$
$$P(r(a)|a, m) = 1 - P(l(a)|a, m)$$

- Also, keep in mind that $q(l(a)) = q(r(a)) = a$.
- Then:

$$\begin{aligned}
& P(l(a)|a, m) + P(r(a)|a, m) = 1 \\
\iff & P(a|m)P(l(a)|a, m) + P(a|m)P(r(a)|a, m) = P(a|m) \\
\iff & P(q(l(a))|m)P(l(a)|a, m) + P(q(r(a))|m)P(r(a)|a, m) = P(a|m) \\
\iff & P(l(a)|m) + P(r(a)|m) = P(a|m) \\
\iff & P(l(a)|m) + P(r(a)|m) - P(a|m) = 0
\end{aligned}$$

Questions?

Take a moment to write down any questions you have for the QA session!

Outline

Negative sampling model

- Another trick: **negative sampling**
- Instead of predicting a probability distribution over whole vocabulary, predict binary probabilities for a small number of word pairs.
- This is not a language model anymore...
- ... but since we only care about the word vectors, and not the skip gram/CBOW predictions, that's not really a problem.

Negative sampling model

- Let B be a hyperparameter.
- For every positive (observed) center-context (skipgram) or context-center (CBOW) pair (m, n), create B negative pairs $((m, n'_1) \dots (m, n'_B))$, where $n'_b \sim V$ is a random word
  - Detail: We sample as a function of word frequency, with undersampling for very frequent words
- Binary classification: predict whether a pair is positive or negative.
- As negative log likelihood:

$$\mathcal{J} = \sum_{m,n} \Big(-\log(P(\text{pos}|m, n)) - \sum_{b=1}^{B} \log(P(\text{neg}|m, n'_b)) \Big)$$

- Can be rewritten as:

$$\mathcal{J} = \sum_{m,n,y} \Big(-y\log(P(\text{pos}|m, n)) - (1-y)\log(P(\text{neg}|m, n)) \Big)$$

- where $y = 1$ if (m, n) is a positive pair and $y = 0$ if (m, n) is a negative pair.

Negative sampling model

- Parameters:
  - $\boldsymbol{v}^{(*)} \in \mathbb{R}^D$
  - $\boldsymbol{w}^{(*)} \in \mathbb{R}^D$

$$P(\text{pos}|m,n) = \sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})$$

$$P(\text{neg}|m,n) = 1 - P(\text{pos}|m,n) = 1 - \sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)}) = \sigma(-\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})$$

- So the loss function becomes:

$$\mathcal{J} = \sum_{m,n,y} \Big(-y\log(\sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})) - (1-y)\log(\sigma(-\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)}))\Big)$$

- On the next slides, we will show how to perform a gradient update for one tuple (m, n, y).

Negative sampling model: Gradients

$$J = -y\log(\sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)}))-(1-y)\log(\sigma(-\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)}))$$

$$\begin{aligned}
\nabla_{\boldsymbol{w}^{(m)}} J =& -y\frac{1}{\sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})}\sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})\sigma(-\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})\boldsymbol{v}^{(n)} \\
& -(1-y)\frac{1}{\sigma(-\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})}\sigma(-\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})\sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})(-1)\boldsymbol{v}^{(n)} \\
=& -y(1-\sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)}))\boldsymbol{v}^{(n)} + (1-y)\sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})\boldsymbol{v}^{(n)} \\
=& -y\boldsymbol{v}^{(n)} + y\sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})\boldsymbol{v}^{(n)} + \sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})\boldsymbol{v}^{(n)} - y\sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})\boldsymbol{v}^{(n)} \\
=& \sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)})\boldsymbol{v}^{(n)} - y\boldsymbol{v}^{(n)} = (\sigma(\boldsymbol{w}^{(m)T}\boldsymbol{v}^{(n)}) - y)\boldsymbol{v}^{(n)} \\
=& (P(\text{pos}|m,n) - y)\boldsymbol{v}^{(n)}
\end{aligned}$$

Similarly:

$$\nabla_{\boldsymbol{v}^{(n)}} J = (P(\text{pos}|n,m) - y)\boldsymbol{w}^{(m)}$$

Derivation cheat sheet: $\frac{dcx}{dx} = c$ $\frac{d\log(x)}{dx} = \frac{1}{x}$ $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x)) = \sigma(x)\sigma(-x)$ $\frac{\partial \boldsymbol{x}^T\boldsymbol{y}}{\partial x_i} = y_i$

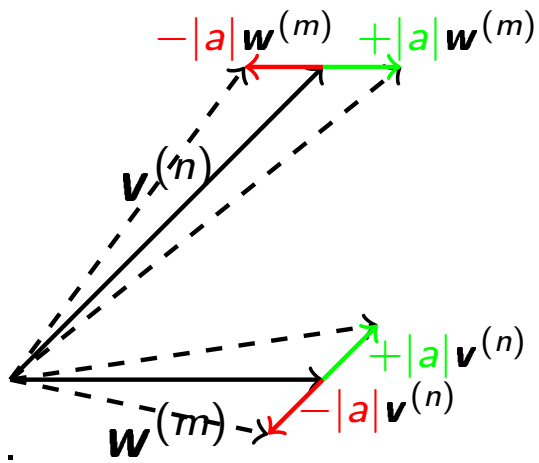
Negative sampling model: Gradient update

$$\boldsymbol{w}^{(m)}_{updated} = \boldsymbol{w}^{(m)} - \eta \nabla_{\boldsymbol{w}^{(m)}} J = \boldsymbol{w}^{(m)} + (-1)\eta(P(\text{pos}|m,n) - y)\boldsymbol{v}^{(n)}$$

$$\boldsymbol{v}^{(n)}_{updated} = \boldsymbol{v}^{(n)} - \eta \nabla_{\boldsymbol{v}^{(n)}} J = \boldsymbol{v}^{(n)} + (-1)\eta(P(\text{pos}|m,n) - y)\boldsymbol{w}^{(m)}$$

Intuition:

- Let $a = (-1)\eta(P(\text{pos}|m,n) - y)$
- True word pair: $y = 1 \wedge \eta > 0 \implies a \geq 0 \implies a = +|a|$
  - $|a|\boldsymbol{v}^{(n)}$ is added to $\boldsymbol{w}^{(m)}$ and vice versa $\rightarrow$ vectors become more similar.
- Random word pair: $y = 0 \wedge \eta > 0 \implies a \leq 0 \implies a = -|a|$
  - $|a|\boldsymbol{v}^{(n)}$ is subtracted from $\boldsymbol{w}^{(m)}$ and vice versa $\rightarrow$ vectors become less similar.



Negative sampling model: Complexity

- **Question:** How many dot products do we need to calculate per positive pair (m, n)? How does this compare to the naive and hierarchical softmax?
  - $B + 1$ (B negative pairs and 1 positive pair)
  - $B + 1 \approx \log_2(|V|) \ll |V|$ (for $B = 20, |V| = 1000000$)

Questions?

Take a moment to write down any questions you have for the QA session!

Outline

FastText

- Even if we train Word2Vec on a very large corpus, we will still encounter unknown words at test time
- Subwords can often help us:
- $\boldsymbol{w}^{(\text{remuneration})}$ should be similar to
  - $\boldsymbol{w}^{(\text{remunerate})}$ (same stem)
  - $\boldsymbol{w}^{(\text{iteration})}, \boldsymbol{w}^{(\text{consideration})} \ldots$ (same suffix $\rightarrow$ may be same part of speech)

FastText

- Extension of Word2Vec by Bojanowski et al. (2017): Enriching Word Vectors with Subword Information.
- Model:
  - No $\boldsymbol{w}^{(*)}$ vectors
  - Instead: $\boldsymbol{u}^{(*)} \in \mathbb{R}^D$ vectors, one vector per word in $x \in V$, and one for every character n-gram (typically 3- to 6-grams)
  - Function that enumerates the character n-grams of a word:

$$\mathcal{S}(\text{remuneration}) = \{\#\text{re}, \text{rem}, \#\text{rem}, \ldots \text{ration}, \text{ation}\$\}$$

$$\boldsymbol{w}^{(m)} = \begin{cases} \frac{1}{|\mathcal{S}(m)|+1}\left[\boldsymbol{u}^{(m)} + \sum_{m' \in \mathcal{S}(m)} \boldsymbol{u}^{(m')}\right] & \text{if } m \in V \\ \frac{1}{|\mathcal{S}(m)|} \sum_{m' \in \mathcal{S}(m)} \boldsymbol{u}^{(m')} & \text{otherwise} \end{cases}$$

- Plug new definition of $\boldsymbol{w}^{(m)}$ into one of the Word2Vec models.
- During training, gradients of loss w.r.t. $\boldsymbol{w}^{(m)}$ are evenly distributed to $\boldsymbol{u}^{(m)}$ and its subword vectors $\boldsymbol{u}^{(m')}$

Questions?

Take a moment to write down any questions you have for the QA session!

Outline

Outline

Initialization of word embedding layers

- Use pretrained $\boldsymbol{w}^{(*)}$ vectors to initialize the word embedding layer of a CNN, RNN ... (see first slides)
- Design choice: Fine-tune embeddings on task or freeze them?
  - Pro fine-tuning: Can learn/strengthen features that are important for the task
  - Pro freezing: We might overfit on training set and mess up the relationships between seen and unseen words
  - Both options are popular

Model	MR	SST-1	SST-2	Subj	TREC	CR	MPQA
CNN-rand (randomly initialized)	76.1	45.0	82.7	89.6	91.2	79.8	83.4
CNN-static (pretrained+frozen)	81.0	45.5	86.8	93.0	92.8	84.7	**89.6**
CNN-non-static (pretrained+fine-tuned)	**81.5**	48.0	87.2	93.4	93.6	84.3	89.5
CNN-multichannel (combination)	81.1	47.4	**88.1**	93.2	92.2	**85.0**	89.4

Table from Kim 2014: Convolutional Neural Networks for Sentence Classification.

Outline

Word analogies

$$\boldsymbol{w}^{(a)} - \boldsymbol{w}^{(b)} \approx \boldsymbol{w}^{(c)} - \boldsymbol{w}^{(d)}$$

$$\hat{d} = \underset{d}{\operatorname{argmax}} \quad \operatorname{sim}(\boldsymbol{w}^{(d)}, \boldsymbol{w}^{(c)} - \boldsymbol{w}^{(a)} + \boldsymbol{w}^{(b)})$$

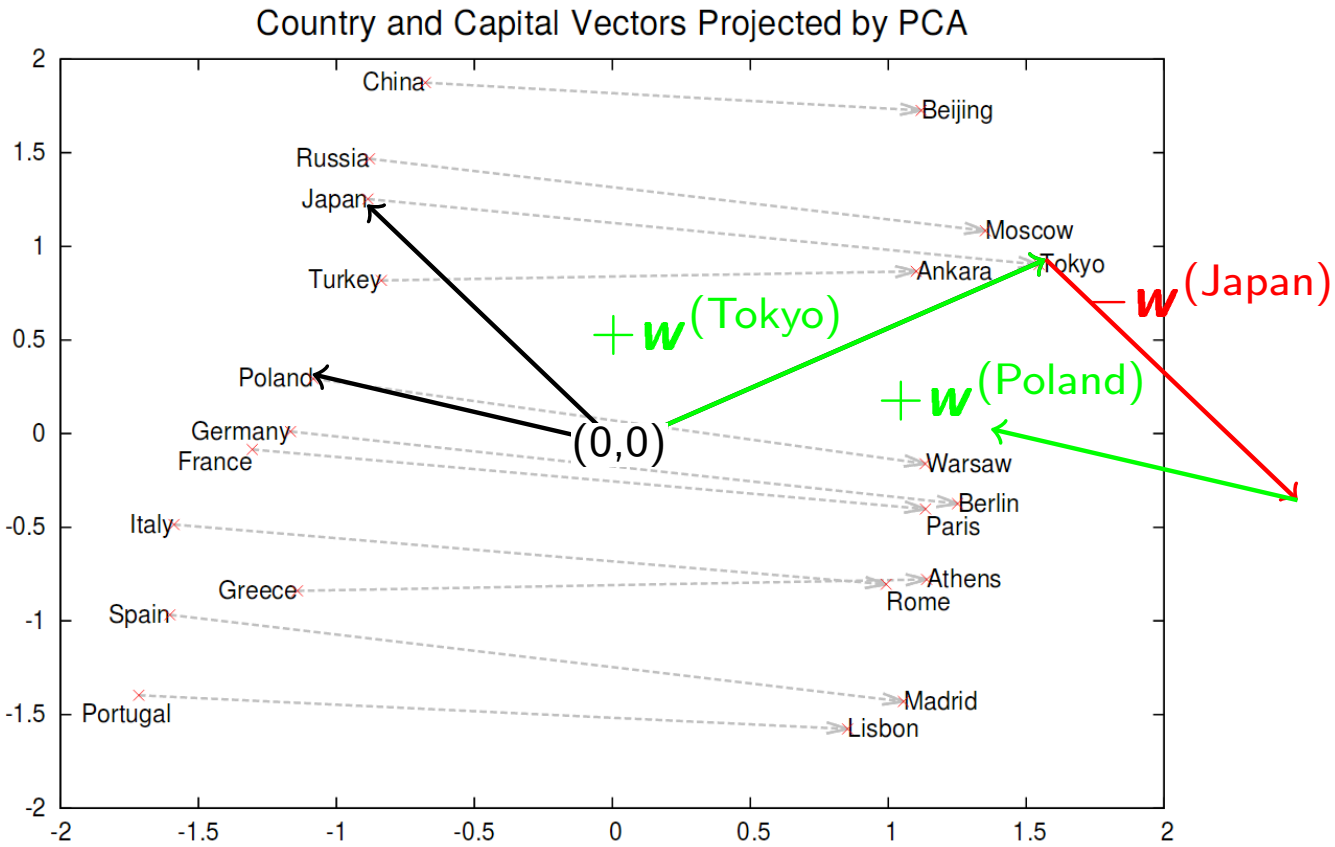


Figure from Mikolov et al. (2013): Distributed Representations of Words and Phrases and their Compositionality

Word analogies

country-capital

$$\boldsymbol{w}^{(\text{Tokio})} - \boldsymbol{w}^{(\text{Japan})} + \boldsymbol{w}^{(\text{Poland})} \approx \boldsymbol{w}^{(\text{Warsaw})}$$

opposite

$$\boldsymbol{w}^{(\text{unacceptable})} - \boldsymbol{w}^{(\text{acceptable})} + \boldsymbol{w}^{(\text{logical})} \approx \boldsymbol{w}^{(\text{illogical})}$$

Nationality-adjective

$$\boldsymbol{w}^{(\text{Australian})} - \boldsymbol{w}^{(\text{Australia})} + \boldsymbol{w}^{(\text{Switzerland})} \approx \boldsymbol{w}^{(\text{Swiss})}$$

Outline

Cross-lingual Embedding Spaces

- Monolingual embedding spaces of two languages: $\boldsymbol{W}_{L1}$, $\boldsymbol{W}_{L2}$
- Dictionary of a few translations: $D = ((\text{cat, katze}), (\text{dog, hund}), \ldots)$
- Learn function f with parameters θ that minimizes:

$$\text{argmin}_\theta \sum_{i,j \in D} ||f(\boldsymbol{w}_{L1}^{(i)}; \theta) - \boldsymbol{w}_{L2}^{(j)}||_2^2$$

  - e.g., linear transformation with parameter matrix $\boldsymbol{A}$: $f(\boldsymbol{w}_{L1}^{(i)}; \boldsymbol{A}) = \boldsymbol{A}\boldsymbol{w}_{L1}^{(i)}$

- Given a word k in L1 with unknown translation, translate as:

$$\underset{\bar{j}}{\text{argmax}} \quad \text{sim}(\boldsymbol{w}_{L2}^{(\bar{j})}, f(\boldsymbol{w}_{L1}^{(k)})$$

Cross-lingual Embedding Spaces

Spanish word	Computed English Translations	Dictionary Entry
emociones	emotions emotion feelings	emotions
protegida	wetland undevelopable protected	protected
imperio	dictatorship imperialism tyranny	empire
determinante	crucial key important	determinant
preparada	prepared ready prepare	prepared
millas	kilometers kilometres miles	miles
hablamos	talking talked talk	talk

Table from Mikolov et al. 2013: Exploiting Similarities among Languages for Machine Translation

Questions?

Take a moment to write down any questions you have for the QA session!

Introduction to numpy

Deep Learning for NLP: Lecture 5(b)

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilians-Universität München
poerner@cis.uni-muenchen.de

December 02, 2020

What is numpy?

- Acronym for "Numeric Python"
- Open source library for python for efficient computation of arrays (vectors, matrices, higher-order arrays)
- Used by many scientific computing and machine learning packages, such as scipy or sklearn.
- Not a specialized library for deep learning (unlike tensorflow or pytorch). But the APIs of deep learning libraries are often similar to numpy.
- Useful for data handling and preprocessing.
- `www.numpy.org`

Core python lists vs. numpy

- Using python nested lists as "arrays":
  - \+ can contain any type of object, or mix of data types (e.g., ints and floats)
  - \+ dynamic sizing, cheap insertion/deletion
  - \- array operations are not very straightforward (e.g., taking the "dot product" of two lists)
  - \- no built-in checks for shapes and data types
  - \- operations are generally not very efficient
- Numpy arrays:
  - \- no mixing of data types
  - \- data type and shape of array must be declared beforehand
  - \+ commonly used operations are functions or overloaded operators
  - \+ built-in checks for shapes and types
  - \+ array operations are efficient

A simple numpy example

- Declaring a "vector" **y** and a "matrix" **X** as nested python lists:

```
>>> l_y = [25.0, 33.3]
>>> l_X = [[6.3, 5.6], [4.5, 4.4], [8.0, 1.0]]
>>> l_y
[25.0, 33.3]
```

- Calculating **Xy** with core python is possible, but annoying:

```
>>> [sum([a*b for a,b in zip(l_y, row)]) for row in l_X]
[343.97999999999996, 259.02, 233.3]
```

- No shape check: If `l_y` had more dimensions than the rows, extra elements would just be ignored by the `zip` function

- To convert the lists into numpy arrays, use `np.array`

```
>>> import numpy as np # Convention: use np for numpy
>>> n_y = np.array(l_y)
>>> n_X = np.array(l_X)
>>> n_y
array([25.  , 33.3])
```

- Then you can simply calculate **Xy** as:

```
>>> n_X.dot(n_Y)
array([343.98, 259.02, 233.3 ])
```

Numpy arrays: dtypes

- An array has exactly one datatype (dtype), e.g., `np.int32`, `np.float32`, `np.float64`, `np.bool`...
- When you create an array from a nested list, numpy will try to infer the dtype. If a list mixes several dtypes, it will resolve it by casting (e.g., int to float):

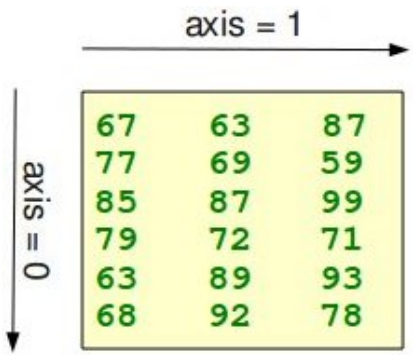
```
>>> np.array([[1, 2, 3], [10, 2, 1]]) # ints only
array([[ 1,  2,  3],
       [10,  2,  1]])
>>> np.array([[1.0, 2.0, 3.0], [10, 2, 1]]) # ints and floats
array([[ 1.,  2.,  3.],
       [10.,  2.,  1.]])
```

- Or you can specify the dtype explicitly:.

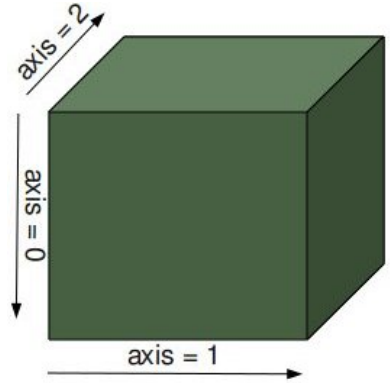
```
>>> np.array([0,0,1], dtype=np.bool)
array([False, False,  True])
>>> np.array([0,0,1], dtype=np.float64)
array([0., 0., 1.])
```

Numpy arrays: shapes

- Every numpy array has a shape (tuple of integers)
- The length of the shape denotes the number of axes:
  - 0 = scalar, 1 = vector, 2 = matrix, 3 or more = higher-dimensional array (tensor)



← matrix 3D tensor →



- The integers in the tuple denote the number of dimensions per axis
- When you create an array from a nested python list, numpy tries to infer the shape:

```
>>> np.array([[1.0, 2.0, 3.0], [10.0, 2.0, 1.0]]).shape
(2,3)
```

- Careful: If the nested list is not a valid shape (e.g., different row lengths), you get an "object" array instead of a numerical array:

```
>>> np.array([[1.0, 2.0, 3.0], [10.0, 2.0]])
array([list([1.0, 2.0, 3.0]), list([10.0, 2.0])], dtype=object)
```

Numpy arrays: shapes

- When declaring an array from scratch, you usually specify its shape:

```
>> np.zeros(shape=()) # scalar
0.0
>>> np.zeros(shape=(2,)) # vector
array([0., 0.])
>>> np.zeros(shape=(2,3)) # matrix
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> np.zeros(shape=(2,3,4)) # tensor (3 axes)
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

- Careful: An axis with one dimension is not the same as no axis:

```
>>> np.zeros(shape=(1,1)) # matrix
array([[0.]])
>>> np.zeros(shape=(1,)) # vector
array([0.])
>>> np.zeros(shape=()) # scalar
array(0.)
```

Exercise

- Given a shape and dtype, think of an example of what the array might represent.

shape	dtype	example
(100,)	bool	yes/no votes by 100 voters
(32, 2,)	int	coordinates of 32 chess pieces on a chess board (8x8 grid)
(1080, 720,)	float	picture with height 1080, width 720, grayscale (0.0-1.0)
(240, 1080, 720, 3)	float	movie with 240 frames, height 1080, width 720, three channels per pixel (red, green, blue)

Some useful numpy functionalities

- This is not meant to be an exhaustive guide!
- For more info, check out
  `https://numpy.org/devdocs/user/quickstart.html`
- Or look on stack overflow!

Zeros and ones

- To create an array of all-zeros/all-ones:

```
>>> zeros = np.zeros(shape=(2,3))
>>> zeros
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> ones = np.ones(shape=(2,3))
```

- You can also create arrays that have the same shape and dtype as a different array, but are filled with zeros or ones:

```
>>> np.zeros_like(ones)
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> np.ones_like(zeros)
```

- To create an identity matrix of shape NxN:

```
>>> np.eye(N=3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

- Padding with zeros:

```
>>> a = np.ones((3,))
>>> np.pad(a, (2,))
array([0., 0., 1., 1., 1., 0., 0.])
```

Ranges in numpy

- Numpy has an equivalent of the core python range function. The default dtype is `np.int64`.

```
>>> np.arange(start=2, stop=10, step=2)
array([2, 4, 6, 8])
```

- To create a vector of N evenly spaced floats between two points:

```
>>> np.linspace(start=2, stop=10, num=6)
array([ 2. ,  3.6,  5.2,  6.8,  8.4, 10. ])
```

Changing the shape

- Changing `shape` value of an existing array:

```
>>> a = np.arange(12)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a.shape = (2,6)
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

- `reshape` creates new array:

```
>>> b = a.reshape((3,4))
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

- Obviously, the product of the dimensions must match number of elements in the array.
- If a dimension is specified as -1, numpy tries to derive it from the other dimensions:

```
>>> b.reshape((2,2,-1)).shape
(2, 2, 3)
```

Transposing an array

- `a.transpose(...)` takes a tuple of indices, indicating which axis of the old (input) array is used for each axis of the new (output) array.
- Note: numpy counts axes starting from 0

```
>>> a = np.zeros((2,3,4))
>>> a.shape
(2, 3, 4)
>>> b=a.transpose((2,0,1))
>>> b.shape
(4, 2, 3)
```

- Interpretation: Axis 0 of *a* becomes axis 1 of *b*, axis 1 of *a* becomes axis 2 of *b*, and axis 2 of *a* becomes axis 0 of *b*.
- To reverse the order of the axes, you can also use:

```
>>> a = np.zeros((2,3,4))
>>> b = a.T
>>> b.shape
(4,3,2)
```

Basic Operations

- Usually, operators on arrays apply *elementwise*:

```
>>> a = np.array([20, 30, 40, 50])
>>> b = np.array([0, 1, 2, 3])
>>> a-b
array([20, 29, 38, 47])
```

- In particular, the *elementwise multiplication* ...

```
>>> a * b
array([ 20,  60, 120, 200])
```

- ... should not be confused with the *dot product*:

```
>>> a.dot(b)
400
```

- Numpy implements many standard unary (elementwise) functions:

```
>>> np.exp(b)
>>> np.sqrt(b)
>>> np.log(b)
```

Reduction operators over axes

- Some functions reduce the array, by taking the sum (or mean, maximum, ...) over a specified axis.
- If no axis is specified, the entire array is reduced to a scalar.

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> b.sum(axis=0)
array([12, 15, 18, 21])
>>> b.sum(axis=1)
array([ 6, 22, 38])
>>> b.sum() # sum over all axes
66
>>> b.mean(axis=0)
>>> b.max(axis=1)
>>> b.min()
```

- To keep the reduced axis as a 1D axis:

```
>>> b.sum(axis=0, keepdims=True)
array([[12, 15, 18, 21]])
```

Indexing elements

- Indexing single elements:

```
>>> b = np.array([[[111, 112], [121, 122]],
                  [[211, 212], [221, 222]],
                  [[311, 312], [321, 322]]])
>>> b[2,1,0]  # this would also work: b[2][1][0]
321
>>> b[-1,-2,-1] # indexing from the end
312
```

- Indexing a sub-array:

```
>>> b[1]
array([[211, 212],
       [221, 222]])
```

- Assignment by index:

```
>>> b[2,1,0] = -100
>>> b
np.array([[[111, 112],
           [121, 122]],
          [[211, 212],
           [221, 222]],
          [[311, 312],
           [-100, 322]]])
```

Indexing with arrays

- You can index several elements at once with an array / list of indices:

```
>>> a = np.arange(12)**2
>>> i = np.array( [ 1,1,3,8,5 ] ) # or: i = [ 1,1,3,8,5 ]
>>> a[i]
array([ 1,  1,  9, 64, 25])
```

- To index a matrix or higher, use a tuple of index arrays (one array per axis):

```
>>> a.shape = (3,4)
>>> a
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121]])
>>> i = (np.array([0,0,0,2,1]), np.array([1,1,3,0,1]))
>>> a[i]
array([ 1,  1,  9, 64, 25])
```

Boolean indexing

- Boolean indexing is done with a boolean array of the *same shape* as the indexed array.

```
>>> a = np.arange(12).reshape((3,4))
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b = a > 4
>>> b
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])

>>> a[b]
array([ 5,  6,  7,  8,  9, 10, 11])
```

Slicing

- Syntax for slicing lists and tuples (start:stop:step) can be applied to multiple axes of arrays.
- Example with one axis:

```
>>> a = np.arange(12)
>>> a[3:8:2]
array([3, 5, 7])
>>> a[:4]
array([0, 1, 2, 3])
>>> a[4:]
array([4, 5, 6, 7, 8, 9, 10, 11])
>>> a[:] # ''dummy'' slice
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

- Example with several axes:

```
>>> b = a.reshape((4,3))
>>> b
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> b[:2,:2]
array([[0, 1],
       [3, 4]])
```

Slicing: Caveat

- Slicing only creates a new **view**: the underlying data is shared with the original array.

```
>>> a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[2:6]
>>> b[0] = -10000
>>> b[1] = -10000
>>> a
array([    0,     1, -10000, -10000,     4,     5,     6,     7,     8,     9])
```

- If you want a copy, use:

```
>>> a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[2:6].copy()
>>> b[0] = -10000
>>> b[1] = -10000
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Creating random arrays

- Array of floats uniformly drawn from the interval $[0, 1)$:

```
>>> np.random.rand(shape=(2,3))
array([[ 0.53604809,  0.54046081,  0.84399025],
       [ 0.59992296,  0.51895053,  0.09988041]])
```

- Generate floats drawn from standard normal distribution $\mathcal{N}(0, 1)$:

```
>>> np.random.randn(shape=(2,3))
array([[-1.28520219, -1.02882158, -0.20196267],
       [ 0.48258382, -0.2077209 , -2.03846176]])
```

- For reproducibility, initialize the random seed at the beginning of your script:

```
>>> np.random.seed(0)
```

- Otherwise, it will be initialized differently at every run (from system clock), leading to different results each time.

Iterating over arrays

- You can iterate over arrays the way you would iterate over a nested list:

```
>>> b = np.arange(9).reshape(3,3)
>>> for row in b:
...     print(row)
...     for elem in row:
...         print(elem)

[0 1 2]
0
1
2
[3 4 5]
3
4
5
[6 7 8]
6
7
8
```

- For efficiency, use array operations instead of iterating, whenever you can.

Finding, sorting and reversing

- To find the indices of an array where a condition holds true:

```
>>> a = np.arange(10, 20).reshape(2,5)
>>> a
array([[10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> indices = np.where(a % 2 == 0)
>>> indices
(array([0, 0, 0, 1, 1]), array([0, 2, 4, 1, 3]))
>>> a[indices]
array([10, 12, 14, 16, 18])
```

- You can sort arrays over a specified axis:

```
>>> a = np.random.random((3,3))
>>> a.sort(axis=1) # sort inside rows
>>> a
array([[0.60930752, 0.85081993, 0.95468601],
       [0.27631831, 0.57972567, 0.88009649],
       [0.28681042, 0.65898285, 0.69914403]])
```

- ... and reverse the order of axes:

```
>>> a[::-1] # reverse the order of the rows
array([[0.28681042, 0.65898285, 0.69914403],
       [0.27631831, 0.57972567, 0.88009649],
       [0.60930752, 0.85081993, 0.95468601]])
```

Concatenating arrays

- Two or more arrays can be concatenated:

```
>>> a = np.array([[1,2],[3,4]])
>>> b = np.array([[11,22],[33,44]])
>>> c = np.array([[-1,-2]])
>>> np.concatenate((a,b,c), axis=0)
array([[ 1,  2],
       [ 3,  4],
       [11, 22],
       [33, 44],
       [-1, -2]])
>>> np.concatenate((a,b), axis=1)
array([[ 1,  2, 11, 22],
       [ 3,  4, 33, 44]])
```

- Note that concatenation requires that (a) all arrays have the same number of axes and (b) all axes (except the one used for concatenation) have the same dimensionality.

```
>>> np.concatenate((a,b,c), axis=1)
ValueError: all the input array dimensions for the concatenation
```

Stacking arrays

- Stacking arrays means treating them as sub-arrays of a new array. This requires that all stacked arrays have the same shape. With stacking, a new axis is introduced.

```
>>> np.stack((a,b), axis=0) # add new axis before first
array([[[ 1,  2],
        [ 3,  4]],

       [[11, 22],
        [33, 44]]])
>>> np.stack((a,b), axis=-1) # add new axis after last
array([[[ 1, 11],
        [ 2, 22]],

       [[ 3, 33],
        [ 4, 44]]])
```

Inserting and deleting dummy axes

- For broadcasting (see next slide), it can be useful to add "dummy axes" to our arrays:

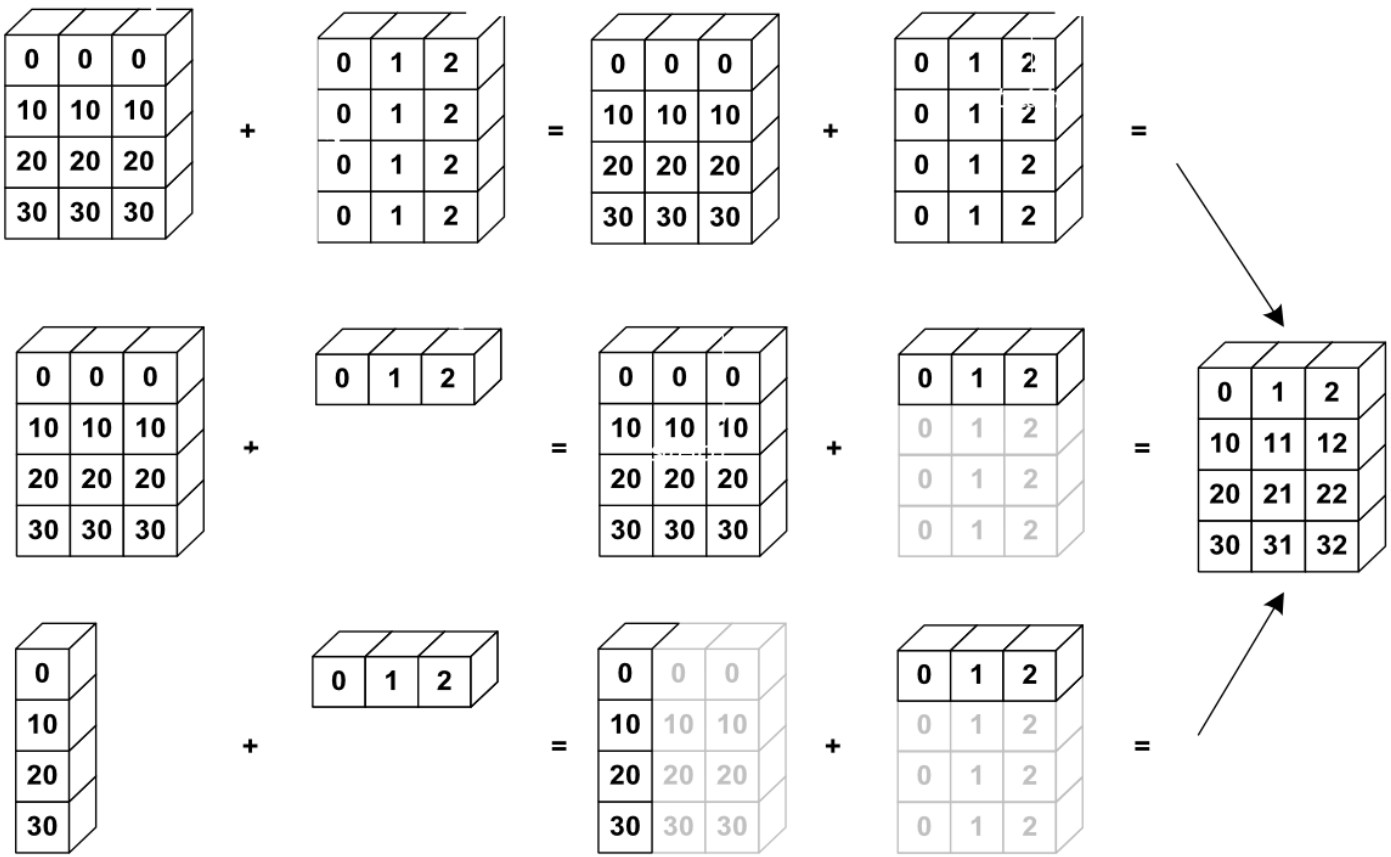
```
>>> a = np.array([5,6,7])
>>> a
array([5, 6, 7])
>>> a.shape
(3,)
>>> b = np.expand_dims(a, axis=0)
>>> b
array([[5, 6, 7]])
>>> b.shape
(1,3)
>>> c = np.expand_dims(a, axis=1)
>>> c.shape
(3, 1)
```

- To delete a dummy axis:

```
>>> b.squeeze(axis=0)
array([5, 6, 7])
```

Broadcasting

Let $\mathbf{X} \in \mathbb{R}^{m \times n}$ and $\mathbf{Y} \in \mathbb{R}^{m \times 1}$. In mathematical notation, the elementwise sum $\mathbf{X} + \mathbf{Y}$ is not legal, because their shapes don't match. But in numpy, it is possible through *broadcasting*.



- This is useful for the outer vector product:

```
>>> x = np.arange(3)
>>> y = np.arange(4)*10
>>> x
array([0, 1, 2])
>>> y
array([ 0, 10, 20, 30])
>>> np.expand_dims(x, axis=0) * np.expand_dims(y, axis=1)
array([[ 0,  0,  0],
       [ 0, 10, 20],
       [ 0, 20, 40],
       [ 0, 30, 60]])
```