Natural Language Processing

AI51701/CSE71001

Lecture 6

09/14/2023

Instructor: Taehwan Kim

Announcements

- Assignment 1 was out today
  - Due: Sep. 24 at 11:59:00pm

- No class on Sep 21 due to *2023 UNIST AI Technology Open Workshop*
  - You are encouraged to attend

Previous Quiz

- We use the following training set to estimate bigram probabilities:
  <s> I am Sam am I am </s>

- What is bigram estimates of sentence probabilities for the following sentence:

  <s> I am I am </s>

word2vec (Mikolov et al., 2013a)

Efficient Estimation of Word Representations in Vector Space

Tomas Mikolov
Google Inc., Mountain View, CA
tmikolov@google.com

Kai Chen
Google Inc., Mountain View, CA
kaichen@google.com

Greg Corrado
Google Inc., Mountain View, CA
gcorrado@google.com

Jeffrey Dean
Google Inc., Mountain View, CA
jeff@google.com

word2vec (Mikolov et al., 2013b)

Distributed Representations of Words and Phrases and their Compositionality

Tomas Mikolov
Google Inc.
Mountain View
mikolov@google.com

Ilya Sutskever
Google Inc.
Mountain View
ilyasu@google.com

Kai Chen
Google Inc.
Mountain View
kai@google.com

Greg Corrado
Google Inc.
Mountain View
gcorrado@google.com

Jeffrey Dean
Google Inc.
Mountain View
jeff@google.com

Learning word vectors

- Let’s use our classification framework
- We want to use unlabeled text to train the vectors
- We can convert our unlabeled text into a classification problem!
- How? (there are many possibilities)

skip-gram training data (window size = 5)

- skip-gram: predict context ("outside") words (position independent) given center word
- Corpus (English Wikipedia):

agriculture is the traditional mainstay of the cambodian economy .

but benares has been destroyed by an earthquake .

…

inputs (*x*)	outputs (*y*)
agriculture	<s>
agriculture	is
agriculture	the
is	<s>
is	agriculture
is	the
is	traditional
the	is
…	…

CBOW training data (window size = 5)

- Continuous Bag of Words (CBOW): predict center word from (bag of) context words
- corpus (English Wikipedia):

agriculture is the traditional mainstay of the cambodian economy .

but benares has been destroyed by an earthquake .

…

inputs (*x*)	outputs (*y*)
{<s>, is, the, traditional}	agriculture
{<s>, agriculture, the, traditional}	is
{agriculture, is, traditional, mainstay}	the
{is, the, mainstay, of}	traditional
{the, traditional, of, the}	mainstay
{traditional, mainstay, the, cambodian}	of
{mainstay, of, cambodian, economy}	the
…	…

skip-gram model

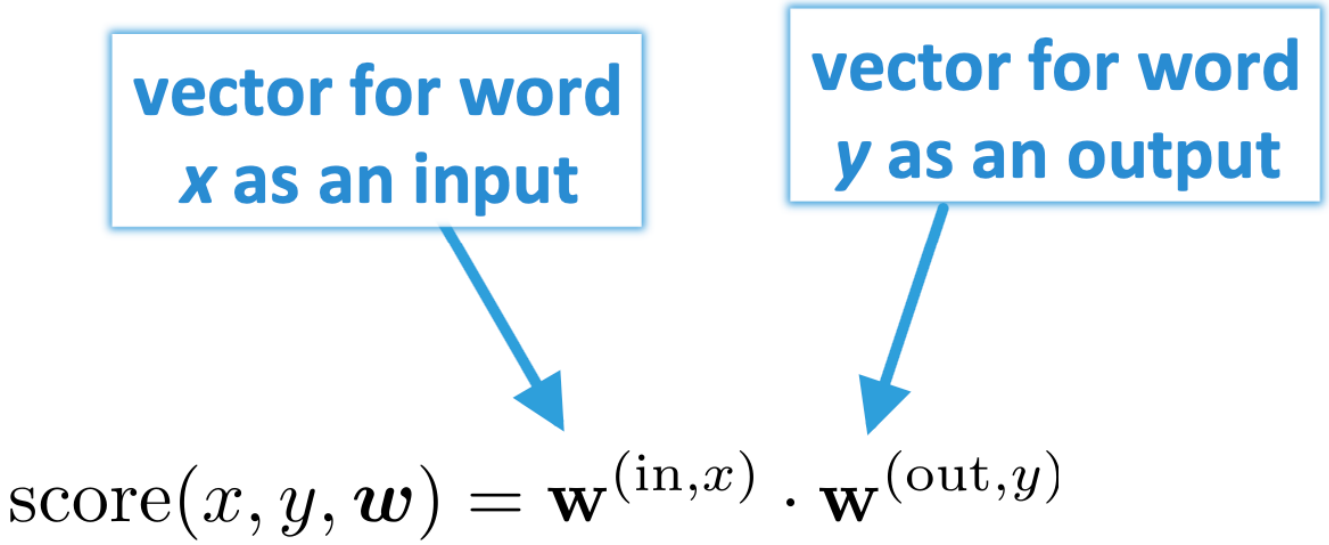
$$\text{classify}(x, \boldsymbol{w}) = \underset{y}{\text{argmax}}\ \text{score}(x, y, \boldsymbol{w})$$

- Here's our data:

inputs (*x*)	outputs (*y*)
agriculture	<s>
agriculture	is
agriculture	the
is	<s>
...	...

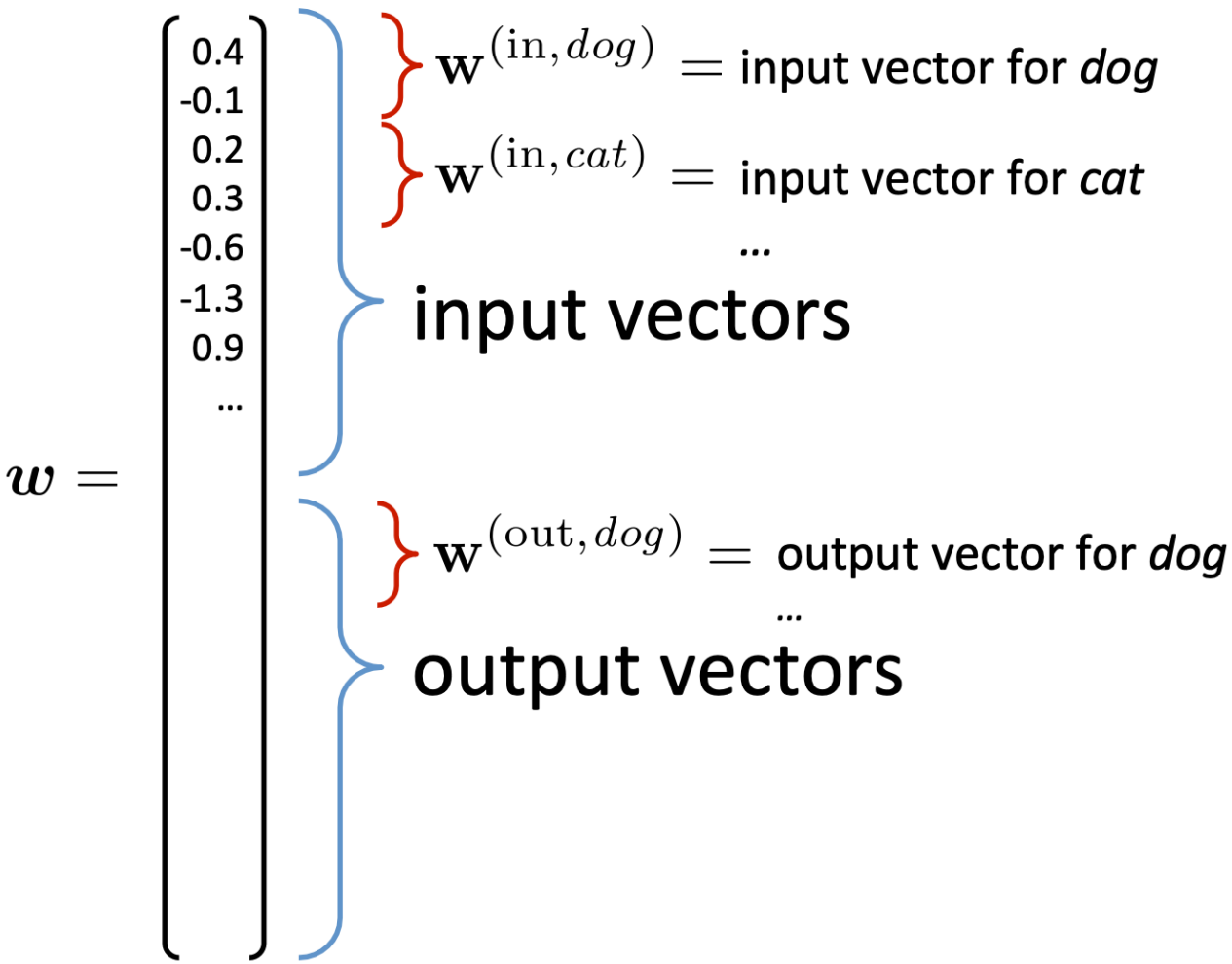
- How should we define the score function?

skip-gram score function: dot product



- Dot product of two vectors, one for each word
- Subtlety: different vector spaces for input and output
- No interpretation to vector dimensions (a **priori**)

skip-gram parameterization



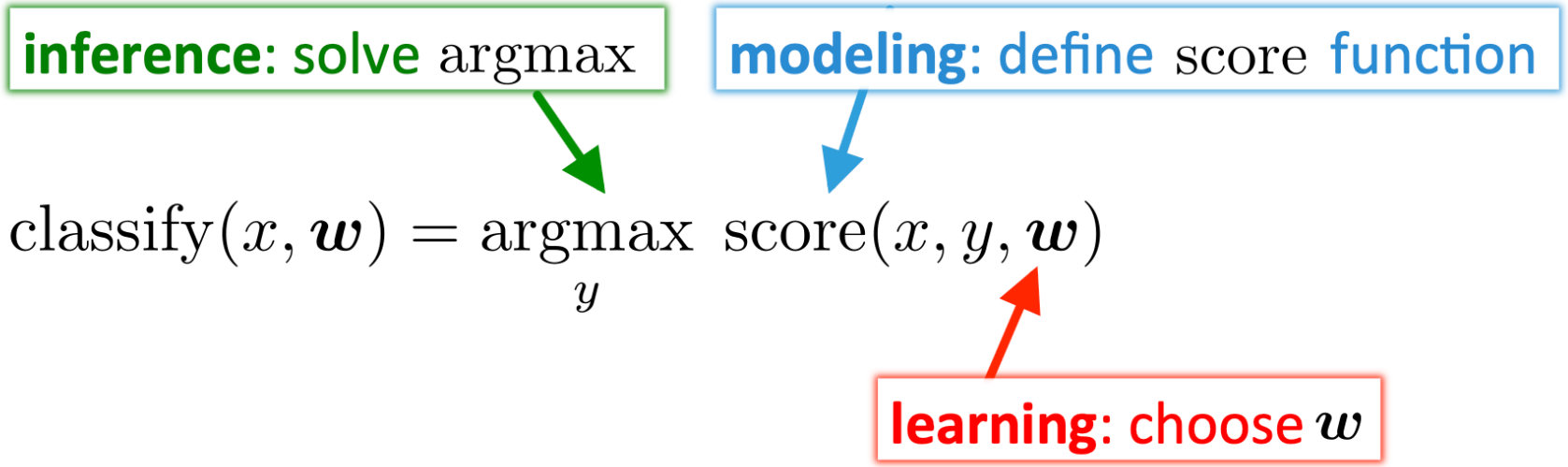
What will the skip-gram model learn?

- Corpus:
  an earthquake destroyed the city
  the town was destroyed by a tornado
- Sample of training pairs:

inputs (*x*)	outputs (*y*)
destroyed	earthquake
earthquake	destroyed
destroyed	tornado
tornado	destroyed
…	…

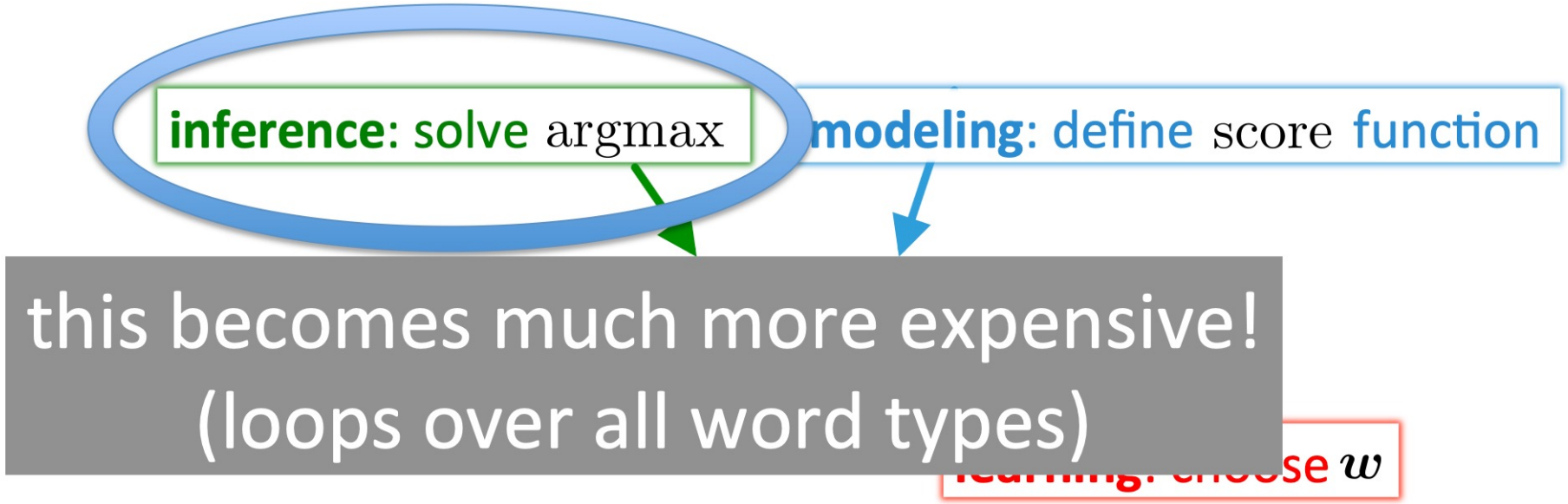
- Output vector for *destroyed* encouraged to be similar to input vectors of *earthquake* and *tornado*

Modeling, Inference, and Learning for Word Vectors



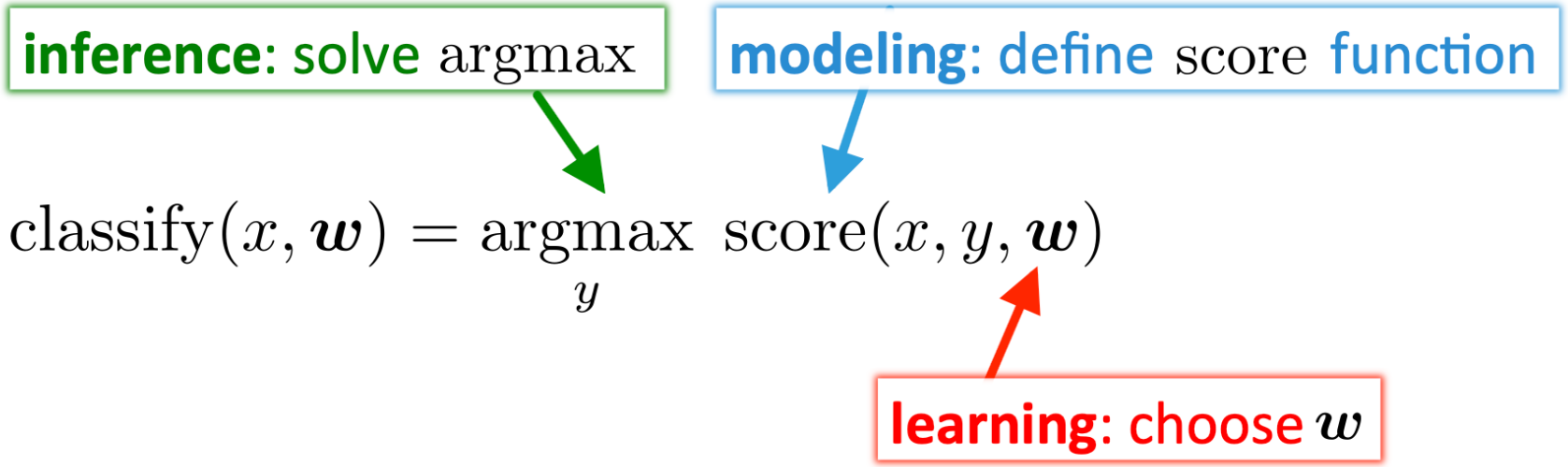
- **Inference**: How do we efficiently search over the space of all outputs?

Modeling, Inference, and Learning for Word Vectors



- **Inference**: How do we efficiently search over the space of all outputs?

Modeling, Inference, and Learning for Word Vectors



- **Learning**: How do we choose the weights w?

skip-gram

- Skip-gram objective: log loss

$$\min_{\boldsymbol{w}} \sum_{1 \le t \le |\mathcal{T}|} \sum_{-c \le j \le c, j \neq 0} -\log P_{\boldsymbol{w}}(x_{t+j} \mid x_t)$$

sum over positions in corpus

sum over context words in window

skip-gram

$$\min_{\boldsymbol{w}} \sum_{1 \le t \le |\mathcal{T}|} \sum_{-c \le j \le c, j \ne 0} -\log P_{\boldsymbol{w}}(x_{t+j} \mid x_t)$$

- From score to probability:

$$P_{\boldsymbol{w}}(y \mid x) \propto \exp\{\text{score}(x, y, \boldsymbol{w})\}$$

$$P_{\boldsymbol{w}}(y \mid x) \propto \exp\{\mathbf{w}^{(\text{in},x)} \cdot \mathbf{w}^{(\text{out},y)}\}$$

skip-gram

$$\min_{\boldsymbol{w}} \sum_{1 \le t \le |\mathcal{T}|} \sum_{-c \le j \le c, j \neq 0} -\log P_{\boldsymbol{w}}(x_{t+j} \mid x_t)$$

- Normalization requires sum over entire vocabulary:

$$P_{\boldsymbol{w}}(y \mid x) = \frac{\exp\{\mathbf{w}^{(\text{in},x)} \cdot \mathbf{w}^{(\text{out},y)}\}}{\sum_{y'} \exp\{\mathbf{w}^{(\text{in},x)} \cdot \mathbf{w}^{(\text{out},y')}\}}$$

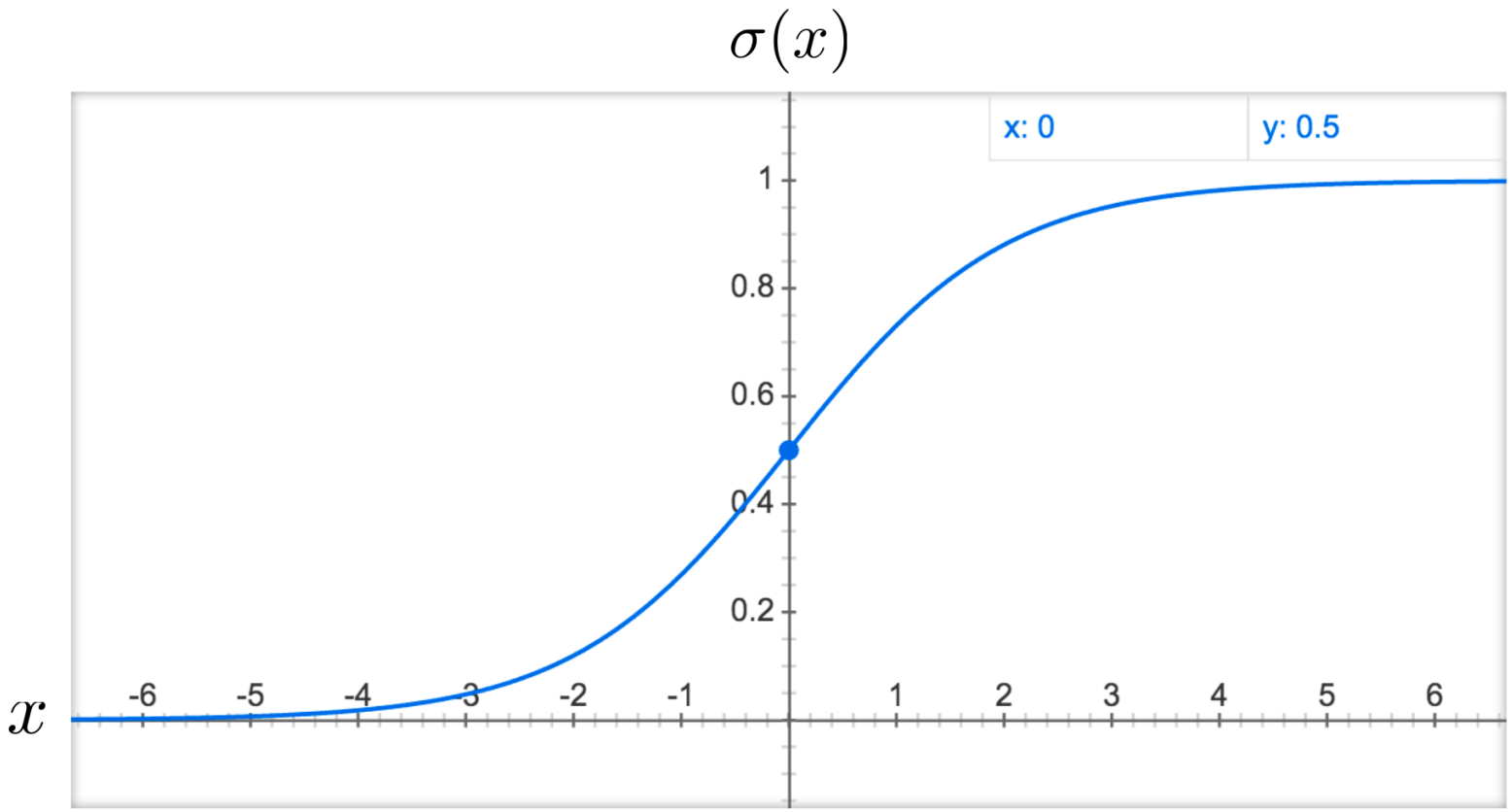
Negative Sampling (Mikolov et al., 2013)

- Rather than sum over entire vocabulary, generate samples and sum over them
- Main idea: train binary logistic regressions for a true pair (center word and a word in its context window) versus several "noise" pairs (the center word paired with a random word)

$$\min_{\boldsymbol{w}} \sum_{1 \leq t \leq |\mathcal{T}|} \sum_{-c \leq j \leq c, j \neq 0} -\log \sigma(\text{score}(x_t, x_{t+j}, \boldsymbol{w})) + \sum_{x \in \text{NEG}} \log \sigma(\text{score}(x_t, x, \boldsymbol{w}))$$

- Where sigma is logistic sigmoid function (see next slide)

(logistic) sigmoid: $\sigma(x) = \dfrac{1}{1 + \exp\{-x\}}$



- $\sigma(\text{score})$ often used to turn a score function into a probabilistic binary classifier, because its outputs range from 0 to 1

Negative Sampling (Mikolov et al., 2013)

$$\min_{\boldsymbol{w}} \sum_{1 \le t \le |\mathcal{T}|} \sum_{-c \le j \le c, j \ne 0} -\log \sigma(\text{score}(x_t, x_{t+j}, \boldsymbol{w})) + \sum_{x \in \text{NEG}} \log \sigma(\text{score}(x_t, x, \boldsymbol{w}))$$

- Maximize probability that real outside word appears;
- Minimize probability that random words appear around center word
- NEG contains 2-20 words sampled from some distribution
  - e.g., uniform, unigram, or smoothed unigram
  - smoothed: raise probabilities to power 3⁄4, renormalize to get a distribution
  - e.g., sample with $P(w)=U(w)^{3/4}/Z$, the unigram distribution U(w) raised to the 3/4 power.
  - The power makes less frequent words be sampled more often

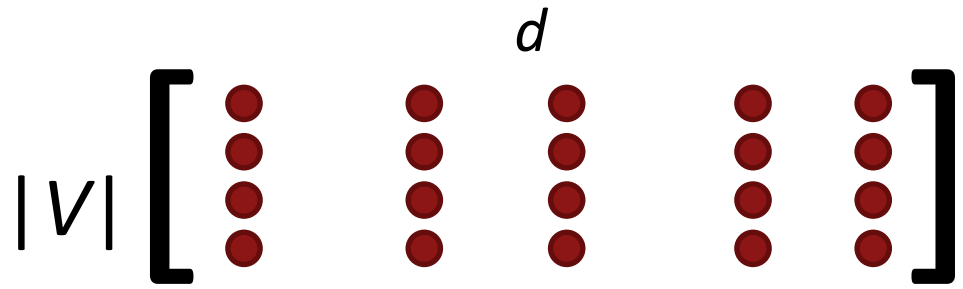
Stochastic gradients with negative sampling

- We iteratively take gradients at each window for SGD
- In each window, we only have at most *2m + 1* words plus *2km* negative words with negative sampling, so $\nabla_\theta J_t(\theta)$ is very sparse!

$$\nabla_\theta J_t(\theta) = \begin{bmatrix} 0 \\ \vdots \\ \nabla_{v_{like}} \\ \vdots \\ 0 \\ \nabla_{u_I} \\ \vdots \\ \nabla_{u_{learning}} \\ \vdots \end{bmatrix} \in \mathbb{R}^{2dV}$$

Stochastic gradients with negative sampling

- We might only update the word vectors that actually appear!
- Solution: either you need sparse matrix update operations to only update certain **rows** of full embedding matrices *U* and *V*, or you need to keep around a hash for word vectors



- If you have millions of word vectors and do distributed computing, it is important to not have to send gigantic updates around!

Two Ways to Represent Word Embeddings

- $\mathcal{V}$ = vocabulary, $|\mathcal{V}|$ = size of vocab
- 1: create $|\mathcal{V}|$-dimensional "one-hot" vector for each word, multiply by word embedding matrix:

$$emb(x) = \mathbf{W}\text{onehot}(\mathcal{V}, x)$$

- 2: store embeddings in a hash/dictionary data structure, do lookup to find embedding for word:

$$emb(x) = \text{lookup}(\mathbf{W}, x)$$

- These are equivalent, second can be much faster (though first can be fast if using sparse operations)

- We went through skip-gram in detail
- word2vec contains two models: skip-gram and continuous bag of words (CBOW)
- For CBOW: we can use the same loss and inference tricks as skip-gram, so we will just focus on the CBOW scoring function

word2vec Score Functions

- skip-gram:

$$\text{score}(x, y, \boldsymbol{w}) = \mathbf{w}^{(\text{in},x)} \cdot \mathbf{w}^{(\text{out},y)}$$

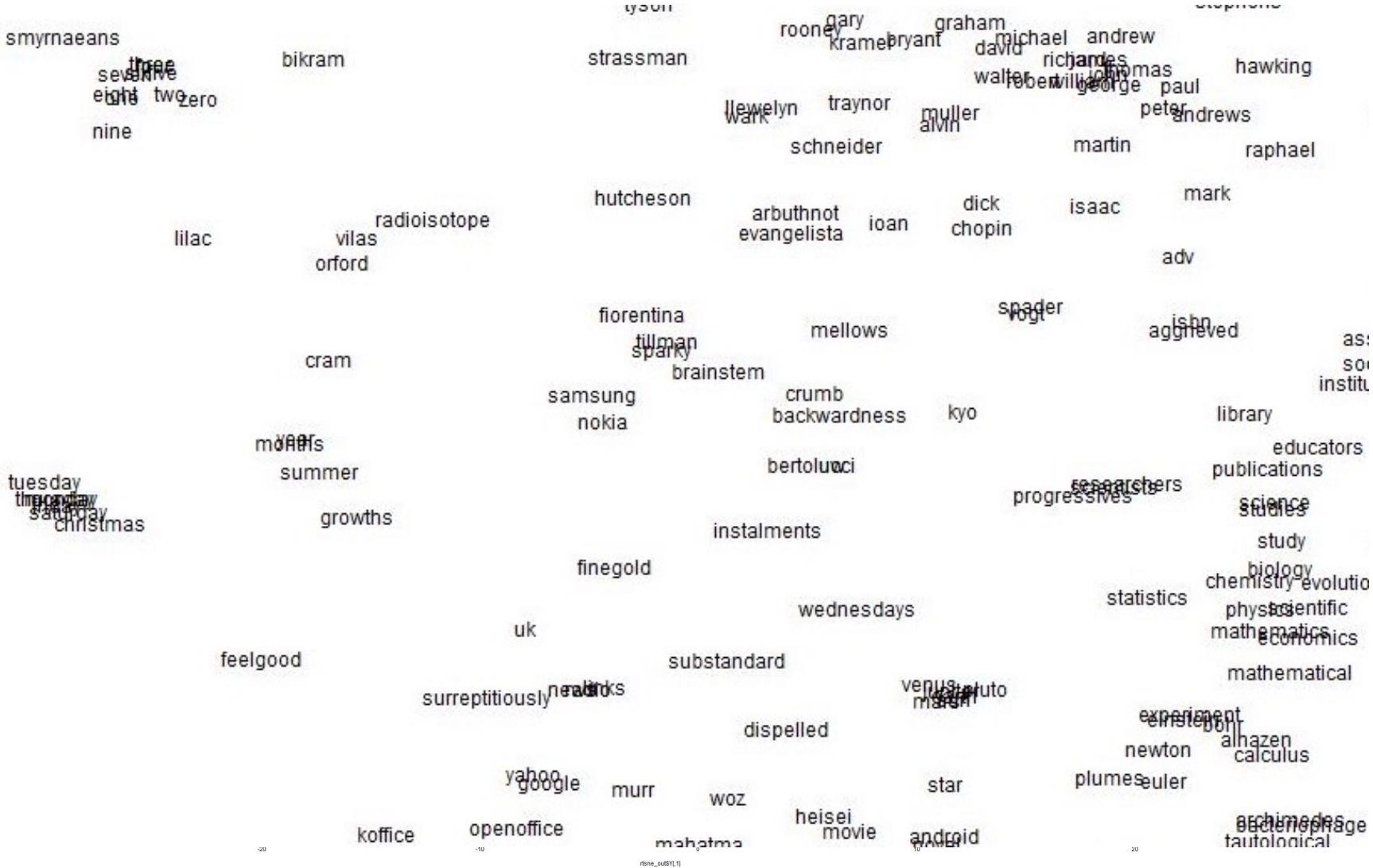
inputs (*x*)	outputs (*y*)
agriculture	<s>
agriculture	is
agriculture	the

- CBOW:

$$\text{score}(\boldsymbol{x}, y, \boldsymbol{w}) = \left(\frac{1}{|\boldsymbol{x}|} \sum_i \mathbf{w}^{(\text{in},x_i)} \right) \cdot \mathbf{w}^{(\text{out},y)}$$

inputs (*x*)	outputs (*y*)
{<s>, is, the, traditional}	agriculture
{<s>, agriculture, the, traditional}	is
{agriculture, is, traditional, mainstay}	the

Word2vec maximizes objective function by putting similar words nearby in space



word2vec

- word2vec toolkit implements training for skip- gram and CBOW models: https://code.google.com/archive/p/word2vec/
- Very fast to train, even on large corpora
- Pretrained embeddings available

A simple way to investigate the learned representations is to find the closest words for a user-specified word. The *distance* tool serves that purpose. For example, if you enter 'france', *distance* will display the most similar words and their distances to 'france', which should look like:

```
                Word       Cosine distance
------------------------------------------
               spain              0.678515
             belgium              0.665923
         netherlands              0.652428
               italy              0.633130
         switzerland              0.622323
          luxembourg              0.610033
            portugal              0.577154
              russia              0.571507
             germany              0.563291
           catalonia              0.534176
```

Using co-occurrence counts directly?

- Why don't we just accumulate all the statistics of what words appear near each other instead of iterating through the whole corpus (perhaps many times)?
- Building a co-occurrence matrix X

sugar, a sliced lemon, a tablespoonful of	**apricot**	preserve or jam, a pinch each of,
r enjoyment. Cautiously she sampled her first	**pineapple**	and another fruit whose taste she likened
well suited to programming on the digital	**computer**.	In finding the optimal R-stage policy from
for the purpose of gathering data and	**information**	necessary for the study authorized in the

	aardvark	computer	data	pinch	result	sugar	…
apricot	0	0	0	1	0	1	…
pineapple	0	0	0	1	0	1	…
digital	0	2	1	0	1	0	…
information	0	1	6	0	4	0	…
…							

Using co-occurrence counts directly?

- Building a co-occurrence matrix X
  - 2 options: windows vs. full document
  - Window: Similar to word2vec, use window around each word -> captures some syntactic and semantic information (“word space”)
  - Word-document co-occurrence matrix will give general topics (all sports terms will have similar entries) leading to “Latent Semantic Analysis” (“document space”)

Towards GloVe: Count based vs. direct prediction

- LSA, HAL (Lund & Burgess),
- COALS, Hellinger-PCA (Rohde et al, Lebret & Collobert)

- Fast training
- Efficient usage of statistics
- Primarily used to capture word similarity
- Disproportionate importance given to large counts

- Skip-gram/CBOW (Mikolov et al)
- NNLM, HLBL, RNN (Bengio et al; Collobert & Weston; Huang et al; Mnih & Hinton)

- Scales with corpus size
- Inefficient usage of statistics
- Generate improved performance on other tasks
- Can capture complex patterns beyond word similarity

From Stanford CS224n course

Encoding meaning components in vector differences (Pennington et al., EMNLP 2014)

GloVe: Global Vectors for Word Representation

Jeffrey Pennington, Richard Socher, Christopher D. Manning
Computer Science Department, Stanford University, Stanford, CA 94305
jpennin@stanford.edu, richard@socher.org, manning@stanford.edu

Encoding meaning components in vector differences (Pennington et al., EMNLP 2014)

- **Crucial insight**: Ratios of co-occurrence probabilities can encode meaning components

	x = solid	x = gas	x = water	x = random
$P(x\|\text{ice})$	large	small	large	small
$P(x\|\text{steam})$	small	large	large	small
$\frac{P(x\|\text{ice})}{P(x\|\text{steam})}$	large	small	~1	~1

Encoding meaning components in vector differences (Pennington et al., EMNLP 2014)

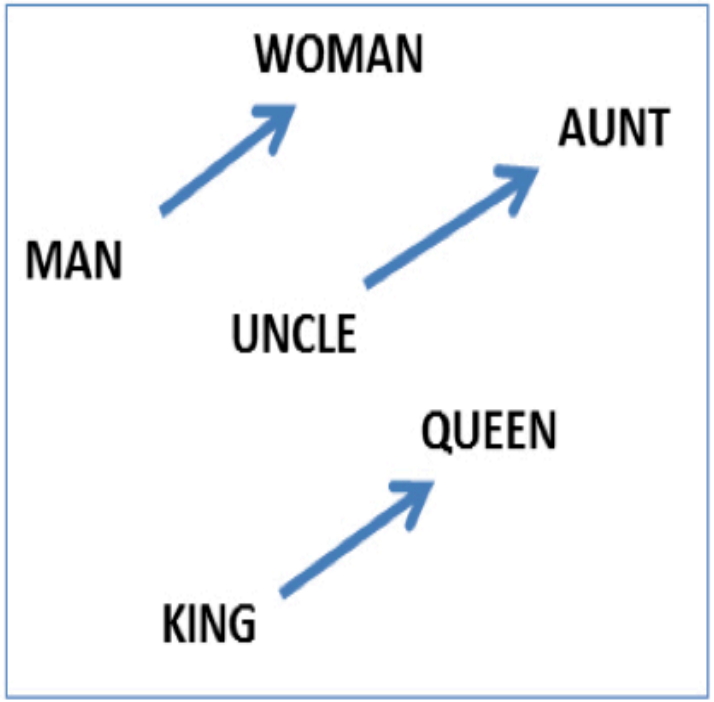
- **Crucial insight**: Ratios of co-occurrence probabilities can encode meaning components

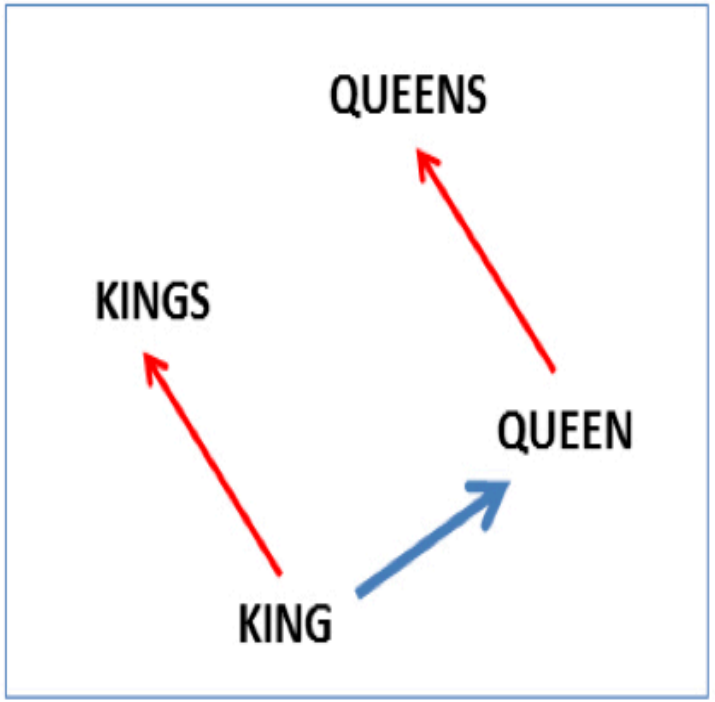
	x = solid	x = gas	x = water	x = fashion
$P(x\|\text{ice})$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(x\|\text{steam})$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$\frac{P(x\|\text{ice})}{P(x\|\text{steam})}$	8.9	8.5×10^{-2}	1.36	0.96

Embeddings capture relational meaning!

vector(*king*) – vector(*man*) + vector(*woman*) ≈ vector(*queen*)

vector(*Paris*) – vector(*France*) + vector(*Italy*) ≈ vector(*Rome*)





Encoding meaning components in vector differences (Pennington et al., EMNLP 2014)

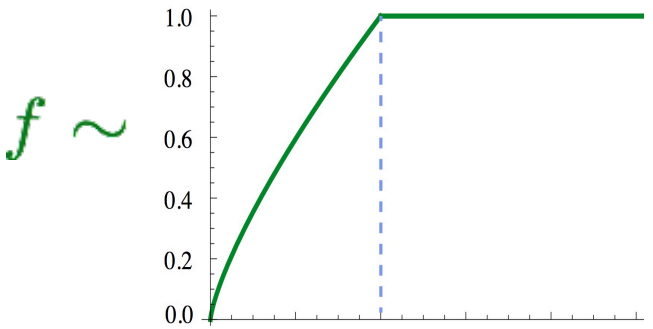
- How can we capture ratios of co-occurrence probabilities as linear meaning components in a word vector space?
  - Log-bilinear model: $w_i \cdot w_j = \log P(i|j)$
  - with vector differences $w_x \cdot (w_a - w_b) = \log \frac{P(x|a)}{P(x|b)}$

Encoding meaning components in vector differences (Pennington et al., EMNLP 2014)

$$w_i \cdot w_j = \log P(i|j)$$

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

- Fast training
- Scalable to huge corpora
- Good performance even with small corpus and small vectors



GloVe results

Nearest words to frog:

1. frogs
2. toad
3. litoria
4. leptodactylidae
5. rana
6. lizard
7. eleutherodactylus

litoria

leptodactylidae

rana

eleutherodactylus

How to evaluate word vectors?

- Related to general evaluation in NLP: Intrinsic vs. extrinsic
- Intrinsic:
  - Evaluation on a specific/intermediate subtask
  - Fast to compute
  - Helps to understand that system
  - Not clear if really helpful unless correlation to real task is established
- Extrinsic:
  - Can take a long time to compute accuracy
  - Unclear if the subsystem is the problem or its interaction or other subsystems
  - If replacing exactly one subsystem with another improves accuracy -> Winning!

Intrinsic word vector evaluation

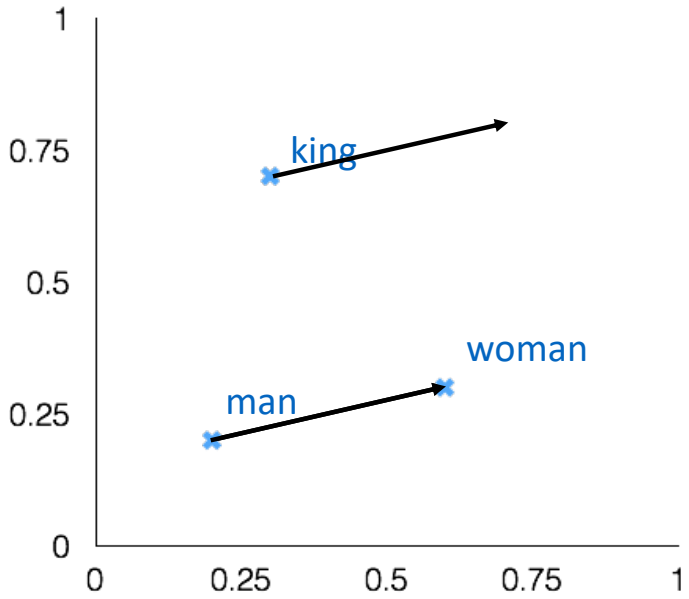
- Word Vector Analogies

a:b :: c:?

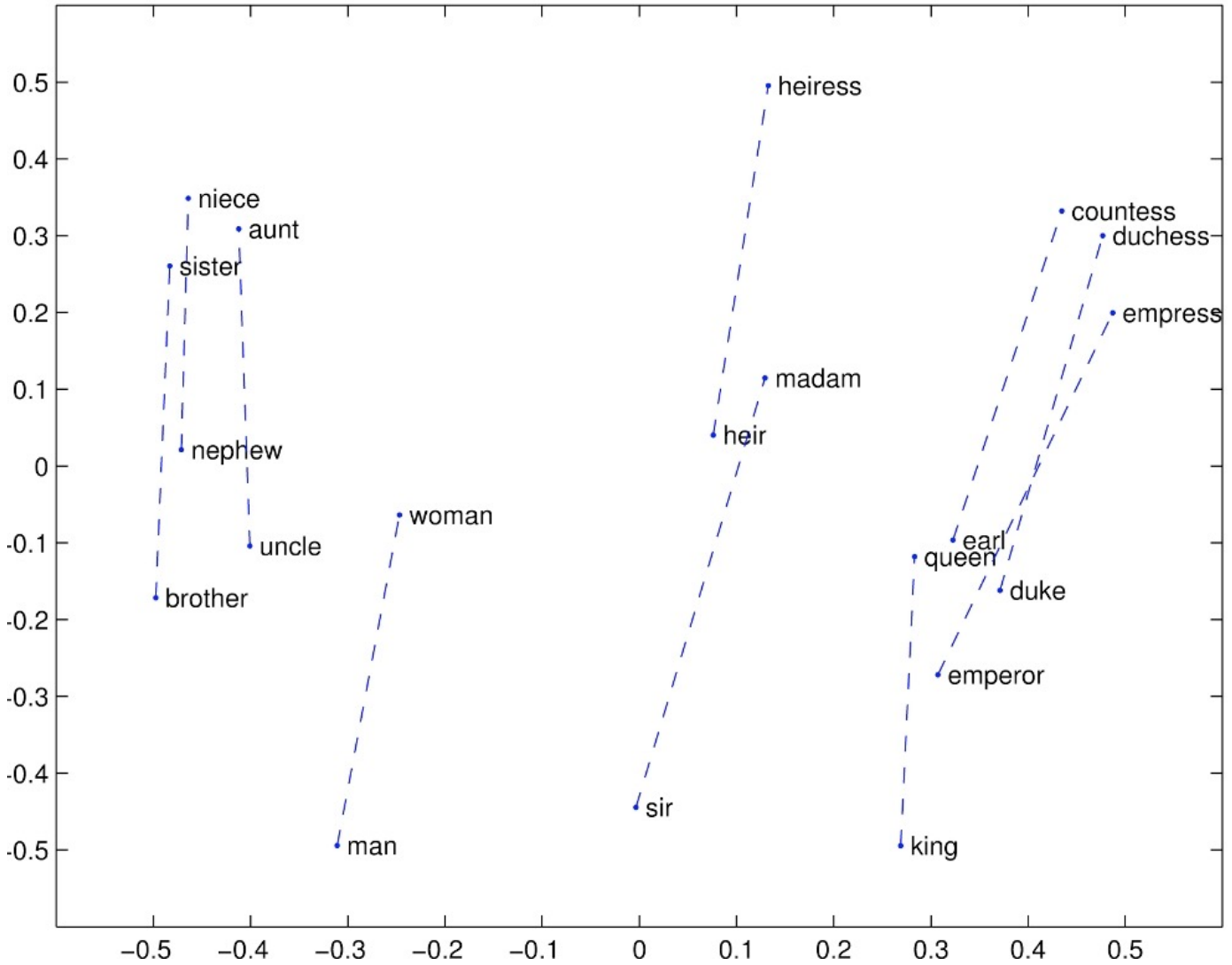
man:woman :: king:?

$$d = \arg\max_i \frac{(x_b - x_a + x_c)^T x_i}{||x_b - x_a + x_c||}$$

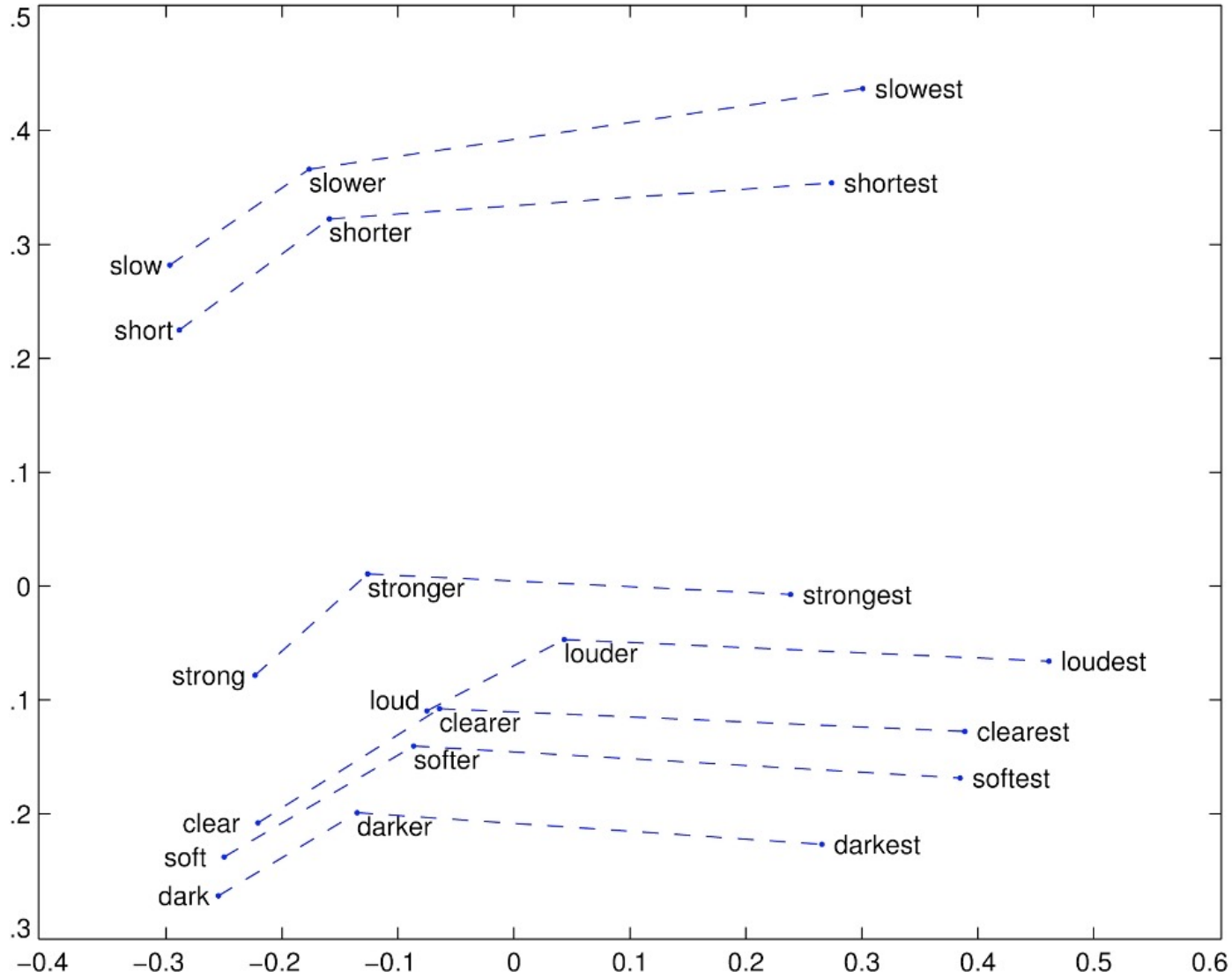
- Evaluate word vectors by how well their cosine distance after addition captures intuitive semantic and syntactic analogy questions
- Discarding the input words from the search (!)
- Problem: What if the information is there but not linear?



Glove Visualizations



Glove Visualizations: Comparatives and Superlatives



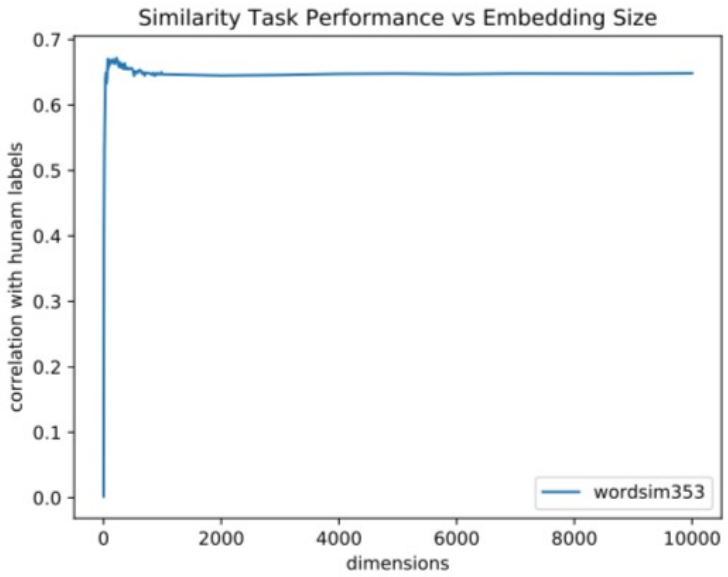
Analogy evaluation

- Glove word vectors evaluation

Model	Dim.	Size	Sem.	Syn.	Tot.
SVD	300	6B	6.3	8.1	7.3
SVD-S	300	6B	36.7	46.6	42.1
SVD-L	300	6B	56.6	63.0	60.1
CBOW†	300	6B	63.6	67.4	65.7
SG†	300	6B	73.0	66.0	69.1
GloVe	300	6B	77.4	67.0	71.7

On the Dimensionality of Word Embedding (Zi Yin and Yuanyuan Shen, NeurIPS 2018)

- https://papers.nips.cc/paper/7368-on-the-dimensionality-of-word-embedding.pdf
- Using matrix perturbation theory, reveal a fundamental bias-variance trade-off in dimensionality selection for word embeddings



(b) WordSim353 Test

Table 3: PIP loss minimizing dimensionalities and intervals for GloVe on Text8 corpus

Surrogate Matrix	arg min	+5% interval	+10% interval	+20% interval	+50% interval	WS353	MT771	Analogy
GloVe (log-count)	719	[290,1286]	[160,1663]	[55,2426]	[5,2426]	220	860	560

Another intrinsic word vector evaluation

- Word vector distances and their correlation with human judgments
- WordSim353 (Finkelstein et al., 2002)

word pair		similarity
journey	voyage	9.3
king	queen	8.6
computer	software	8.5
law	lawyer	8.4
forest	graveyard	1.9
rooster	voyage	0.6

Correlation evaluation

- Word vector distances and their correlation with human judgments
- Some ideas from Glove paper have been shown to improve skip-gram (SG) model also

Model	Size	WS353	MC	RG	SCWS	RW
SVD	6B	35.3	35.1	42.5	38.3	25.6
SVD-S	6B	56.5	71.5	71.0	53.6	34.7
SVD-L	6B	65.7	72.7	75.1	56.5	37.0
CBOW†	6B	57.2	65.6	68.2	57.0	32.5
SG†	6B	62.8	65.2	69.7	58.1	37.2
GloVe	6B	65.8	72.7	77.8	53.9	38.1
SVD-L	42B	74.0	76.4	74.1	58.3	39.9
GloVe	42B	**75.9**	**83.6**	**82.9**	**59.6**	**47.8**
CBOW*	100B	68.4	79.6	75.4	59.4	45.5

Extrinsic word vector evaluation

- One example where good word vectors should help directly: **named entity recognition**: identifying references to a person, organization or location

Some questioned if Tim Cook's first product would be a breakaway hit for Apple.

PERSON ORGANIZATION

Extrinsic word vector evaluation

- One example where good word vectors should help directly: **named entity recognition**: identifying references to a person, organization or location

Model	Dev	Test	ACE	MUC7
Discrete	91.0	85.4	77.4	73.4
SVD	90.8	85.7	77.3	73.7
SVD-S	91.0	85.5	77.6	74.3
SVD-L	90.5	84.8	73.6	71.5
HPCA	92.6	**88.7**	81.7	80.7
HSMN	90.5	85.7	78.7	74.7
CW	92.2	87.4	81.7	80.2
CBOW	93.1	88.2	82.2	81.1
GloVe	**93.2**	88.3	**82.9**	**82.2**

Other Work on Word Embeddings

- Active research area (probably too active)
- Other directions:
  - multiple embeddings for a single word corresponding to different word senses
  - using subword information (e.g., characters) in word embeddings
  - tailoring embeddings for different NLP tasks

Other ways to learn word vectors

- aside: any labeled dataset can be used to learn word vectors (depending on model/features)
- Can we use sentiment classifiers to produce word vectors?
- learned feature weights for a 5-way sentiment classifier (binary unigram features), for two words:

feel-good

label	weight
strongly positive	0.025
positive	0.035
neutral	-0.045
negative	0
strongly negative	-0.015

dull

label	weight
strongly positive	0
positive	0
neutral	-0.04
negative	0.015
strongly negative	0.025

Neural Network for Sentiment Classification

$$\mathbf{z}^{(1)} = g\left(\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{s} = \mathbf{U}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$

vector of label scores

Neural Network for Sentiment Classification

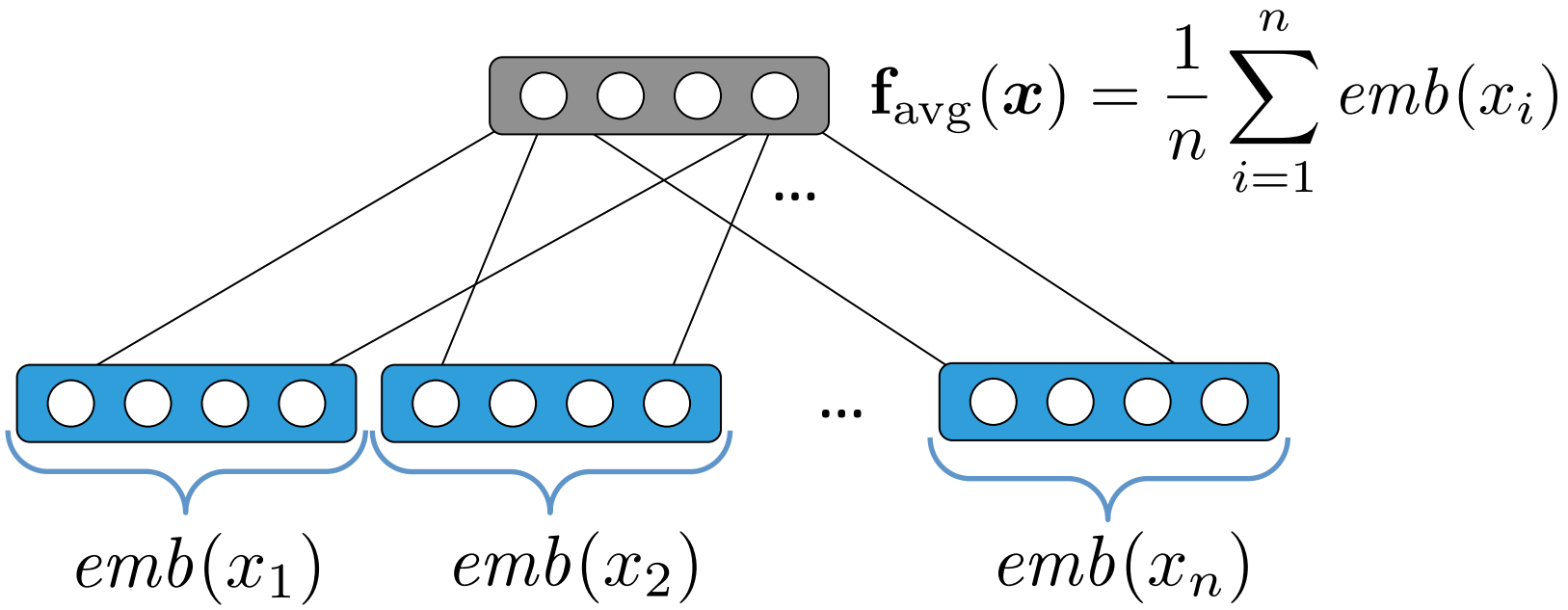
$$\mathbf{z}^{(1)} = g\left(\mathbf{U}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{s} = \mathbf{U}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$

$$\mathbf{s} = \begin{bmatrix} \text{score}(\boldsymbol{x}, \text{positive}, \boldsymbol{w}) \\ \text{score}(\boldsymbol{x}, \text{negative}, \boldsymbol{w}) \end{bmatrix}$$

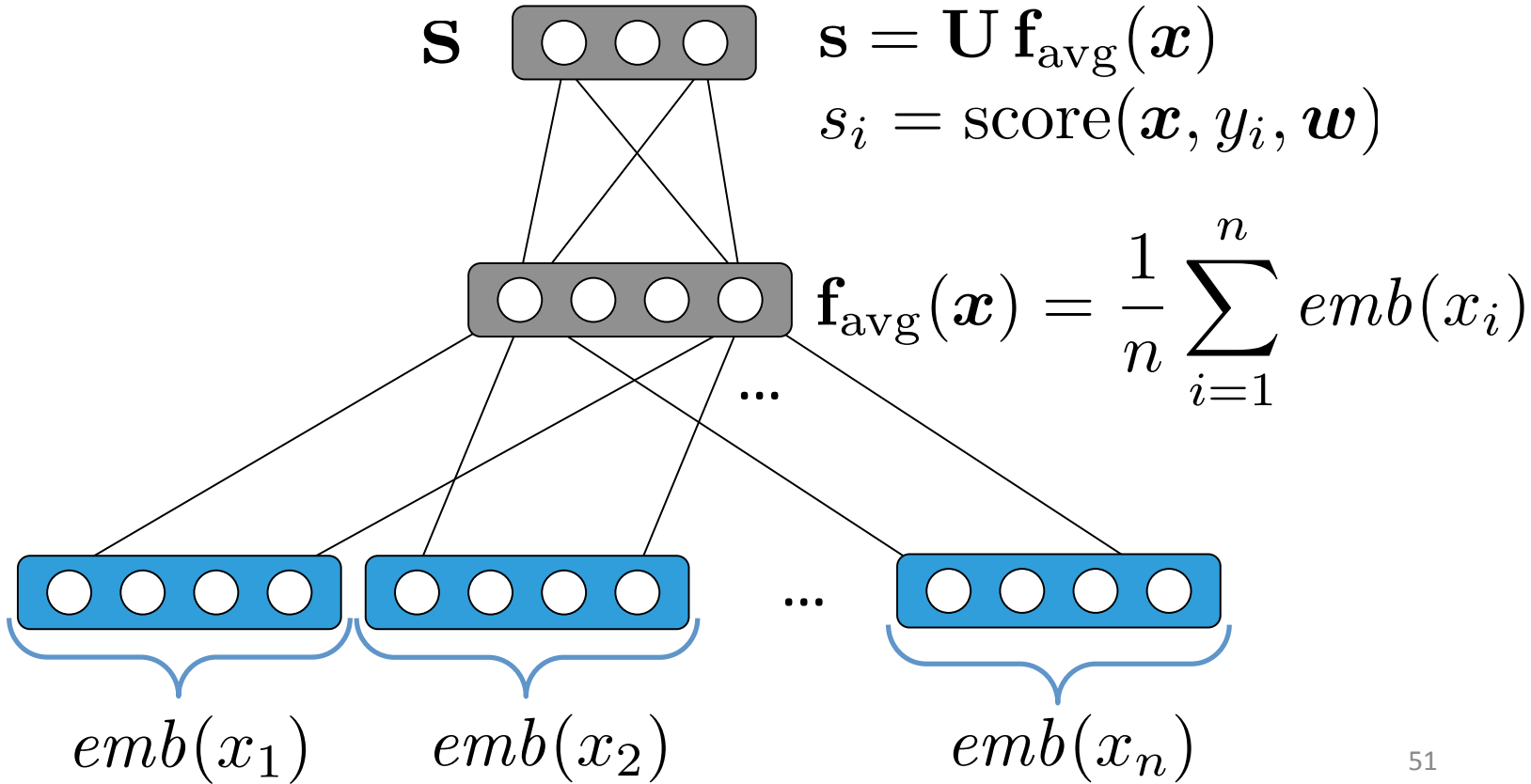
A Simple Neural Text Classification Model

- Given a word sequence ***x***, predict its label
- Represent ***x*** by averaging its word embeddings:



A Simple Neural Text Classification Model

- Represent x by averaging its word embeddings
- Output is a score vector over all possible labels:



Averaging Word Embeddings

- Effective encoder for text classification and many other tasks
- Sometimes called a neural bag of words (NBOW) model (Kalchbrenner et al., 2014)
- Or a deep averaging network (DAN), especially if hidden layers are used (Iyyer et al., 2015)

Encoders

- Encoder: a function to represent a word sequence as a vector
- Simplest: average word embeddings:

$$\mathbf{f}_{\text{avg}}(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} emb(x_i)$$

- Many other functions possible!
- Lots of recent work on developing better ways to encode word sequences

Deep Learning Classification Task: Named Entity Recognition (NER)

- The task: **find** and **classify** names in text, for example:

  Last night , Paris Hilton wowed in a sequin gown .

  PER PER

  Samuel Quinn was arrested in the Hilton Hotel in Paris in April 1989 .

  PER PER LOC LOC LOC DATE DATE

- Possible uses:
  - Tracking mentions of particular entities in documents
  - For question answering, answers are usually named entities
  - Relating sentiment analysis to the entity under discussion
- Often followed by Entity Linking/Canonicalization into a Knowledge Base such as Wikidata

Simple NER: Window classification using binary logistic classifier

- **Idea**: **classify each word in its context window** of neighboring words
- Train logistic classifier on hand-labeled data to classify center word {yes/no} for each class based on a **concatenation of word vectors** in a window
  - Really, we usually use multi-class softmax, but we're trying to keep it simple
- **Example:** Classify "Paris" as +/– location in context of sentence with window length 2:

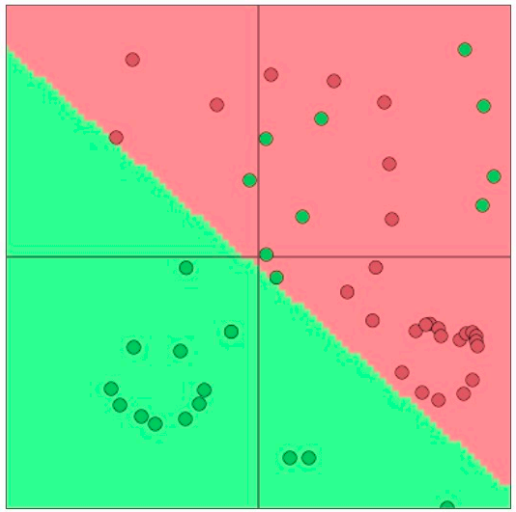
the museums in Paris are amazing to see .

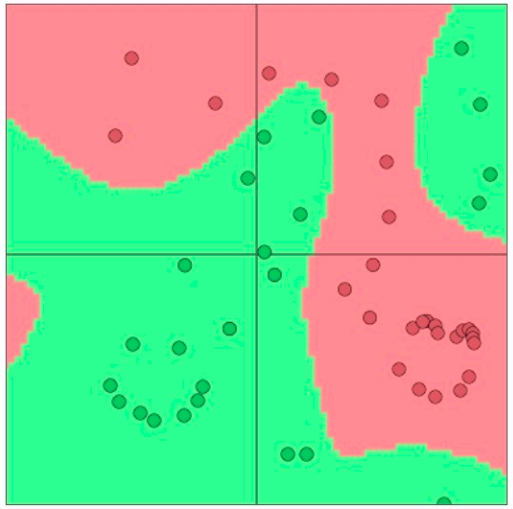
$$X_{window} = [\ x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}\ ]^T \qquad x \in R^{5d}$$

- To classify all words: run classifier for each class on the vector centered on each word in the sentence

Neural Network Classifiers

- Typical ML/stats softmax classifier: $$p(y|x) = \frac{\exp(W_y.x)}{\sum_{c=1}^{C} \exp(W_c.x)}$$
  - Learned parameters θ are just elements of W (not input representation x, which has sparse symbolic features)
  - Classifier gives linear decision boundary, which can be limiting
- Neural networks can learn much more complex functions with nonlinear decision boundaries!





From Stanford CS224n course

Neural Network Classifiers

- A neural network classifier differs in that:
  - We learn **both** *W* **and (distributed!)** representations for words
  - The word vectors *x* re-represent one-hot vectors, moving them around in an intermediate layer vector space, for easy classification with a (linear) softmax classifier
    - Conceptually, we have an embedding layer: $x=Le$
  - We use deep networks—more layers—that let us re-represent and compose our data multiple times, giving a non-linear classifier

Training with “cross entropy loss”

- Until now, our objective was stated as to **maximize the probability of the correct class** y or equivalently we can **minimize the negative log probability of that class**
- Now restated in terms of **cross entropy**, a concept from information theory
- Let the true probability distribution be *p*; let our computed model probability be *q*
- The cross entropy is:

$$H(p,q) = -\sum_{c=1}^{C} p(c) \log q(c)$$

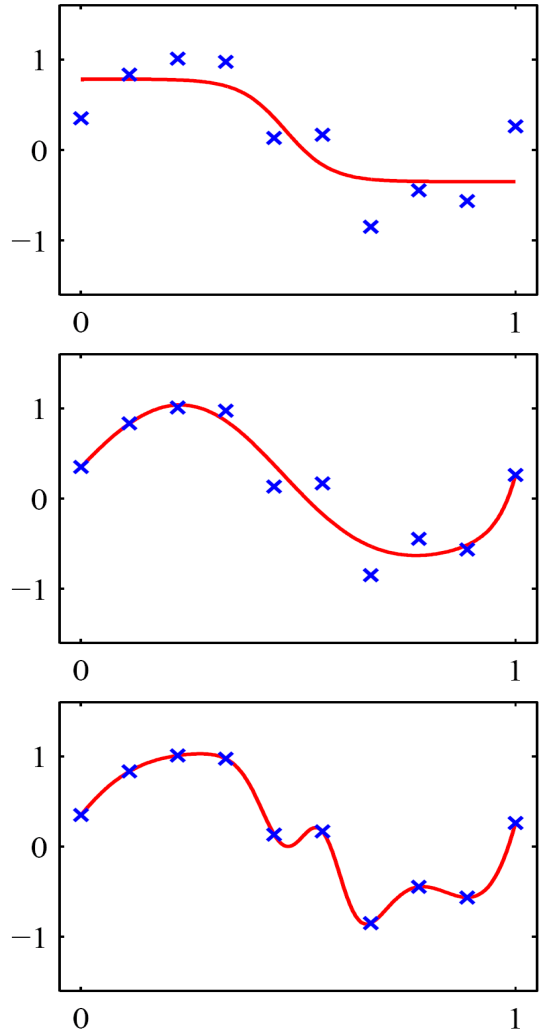
Training with “cross entropy loss”

- Let the true probability distribution be *p*; let our computed model probability be *q*
- The cross entropy is: $$H(p,q) = -\sum_{c=1}^{C} p(c) \log q(c)$$

- Assuming a ground truth (or true or gold or target) probability distribution that is 1 at the right class and 0 everywhere else, *p* = [0, ..., 0, 1, 0, ..., 0], then:
- **Because of one-hot *p*, the only term left is the negative log probability of the true class y_i:** $-\log p(y_i|x_i)$

Why nonlinearities?

- Neural networks do function approximation, e.g., regression or classification
  - Without non-linearities, deep neural networks can't do anything more than a linear transform
  - Extra layers could just be compiled down into a single linear transform: $W_1 W_2 x = Wx$
  - But, with more layers that include non-linearities, they can approximate more complex functions!

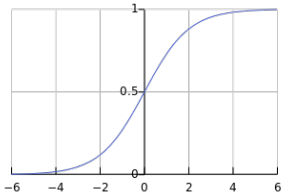


NER: Binary classification for center word being location

- We do supervised training and want high score if it's a location

$$J_t(\theta) = \sigma(s) = \frac{1}{1 + e^{-s}}$$

predicted model probability of class



$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$

f = Some element-wise non-linear function, e.g., logistic, tanh, ReLU

From Stanford CS224n course

Stochastic Gradient Descent

- Update equation: $\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$

  α = *step size* or *learning rate*

  - I.e., for each parameter: $\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial J(\theta)}{\partial \theta_j^{old}}$
  - In deep learning, θ includes the data representation (e.g., word vectors) too!
- How can we compute $\nabla_\theta J(\theta)$?
  - By hand
  - Algorithmically: the backpropagation algorithm

Gradients

- Given a function with 1 output and 1 input

$$f(x) = x^3$$

- It's gradient (slope) is its derivative

$$\frac{df}{dx} = 3x^2$$

  - "How much will the output change if we change the input a bit?"
    - At *x* = 1 it changes about 3 times as much: 1.013 = 1.03
    - At x = 4 it changes about 48 times as much: 4.013 = 64.48

Gradients

- Given a function with 1 output and n inputs

$$f(\boldsymbol{x}) = f(x_1, x_2, ..., x_n)$$

- Its gradient is a vector of partial derivatives with respect to each input

$$\frac{\partial f}{\partial \boldsymbol{x}} = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n}\right]$$

Jacobian Matrix: Generalization of the Gradient

- Given a function with ***m* outputs** and *n* inputs

$$f(\boldsymbol{x}) = f(x_1, x_2, ..., x_n)$$

- It's Jacobian is an ***m* x *n* matrix** of partial derivatives

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

$$\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)_{ij} = \frac{\partial f_i}{\partial x_j}$$

Jacobian Matrix: Generalization of the Gradient

- For composition of one-variable functions: **multiply derivatives**

$$z = 3y$$

$$y = x^2$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = (3)(2x) = 6x$$

- For multiple variables at once: **multiply Jacobians**

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \dots$$

Example Jacobian: Elementwise activation Function

$$\boldsymbol{h} = f(\boldsymbol{z}), \text{what is } \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}? \qquad \boldsymbol{h}, \boldsymbol{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

- Function has n outputs and n inputs → *n* by *n* Jacobian

Example Jacobian: Elementwise activation Function

$$\boldsymbol{h} = f(\boldsymbol{z}), \text{what is } \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}? \qquad \boldsymbol{h}, \boldsymbol{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

$$\left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i) \qquad \text{definition of Jacobian}$$

$$= \begin{cases} f'(z_i) & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases} \qquad \text{regular 1-variable derivative}$$

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} = \begin{pmatrix} f'(z_1) & & 0 \\ & \ddots & \\ 0 & & f'(z_n) \end{pmatrix} = \text{diag}(\boldsymbol{f}'(\boldsymbol{z}))$$

Other Jacobians

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{W}$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}) = \boldsymbol{I} \ \ (\text{Identity matrix})$$

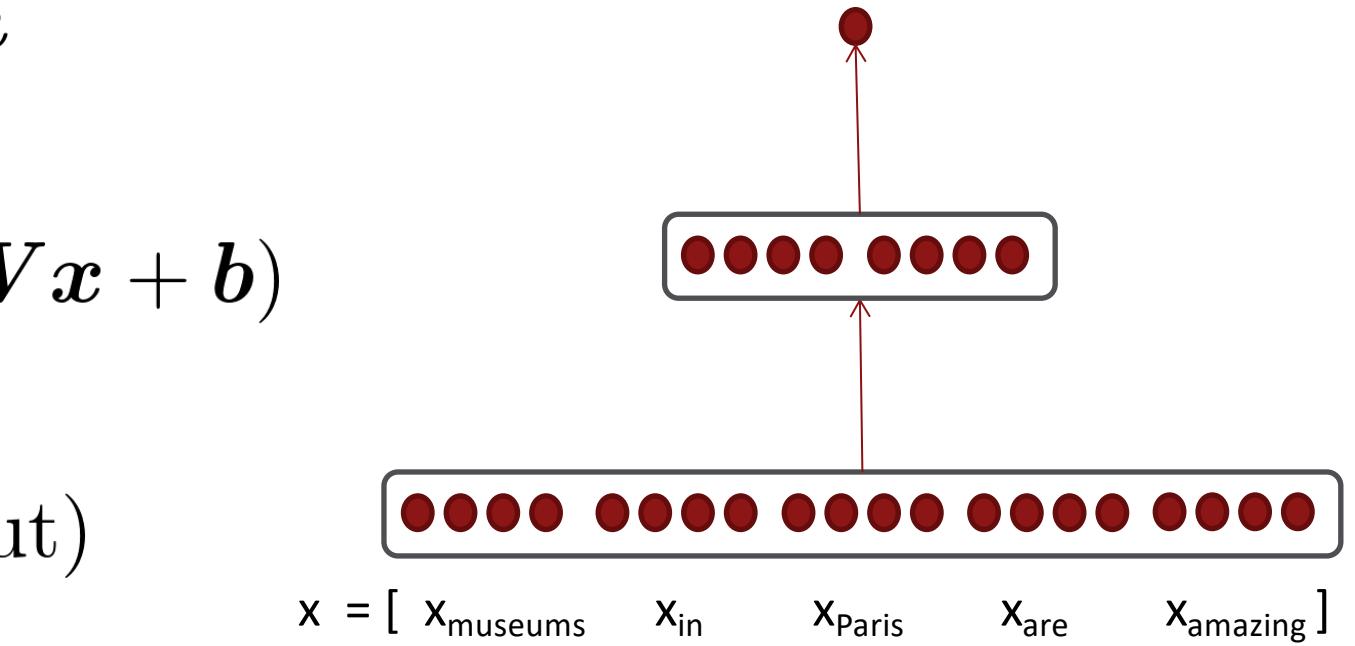
$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T\boldsymbol{h}) = \boldsymbol{h}^{\boldsymbol{T}}$$

Back to our Neural Net!

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad (\text{input})$$



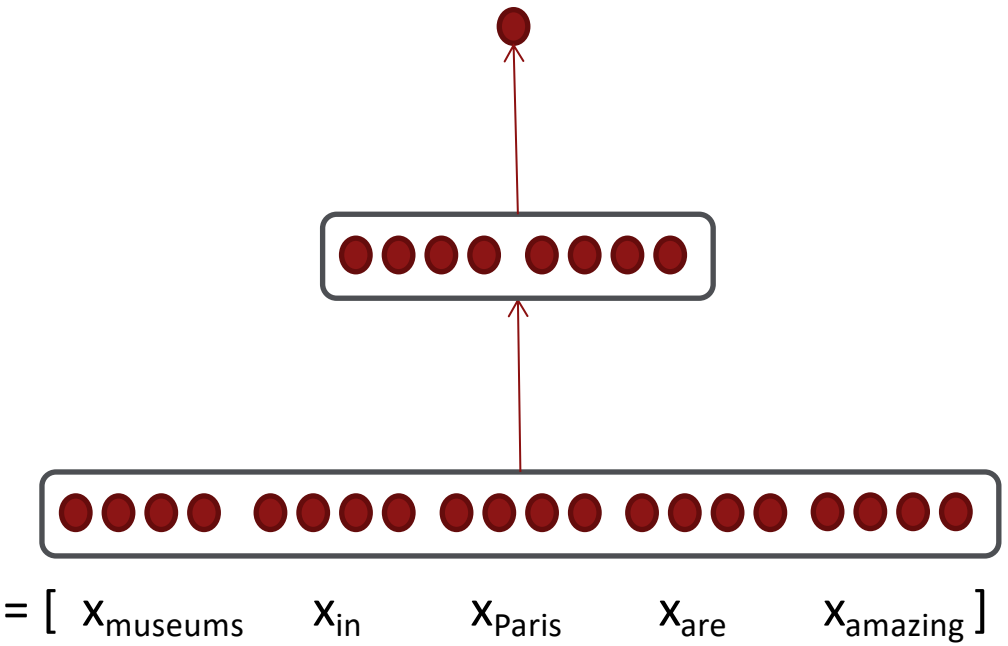
Back to our Neural Net!

- Let's find $\frac{\partial s}{\partial \boldsymbol{b}}$
- Really, we care about the gradient of the loss J_t but we will compute the gradient of the score for simplicity

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$



1. Break up equations into simple pieces

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\boldsymbol{x} \quad \text{(input)}$$

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad \text{(input)}$$

- Carefully define your variables and keep track of their dimensionality!

2. Apply the chain rule

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

2. Apply the chain rule

$$\boxed{s = \boldsymbol{u}^T \boldsymbol{h}}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \boxed{\frac{\partial s}{\partial \boldsymbol{h}}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

2. Apply the chain rule

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

2. Apply the chain rule

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

3. Write out the Jacobians

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \quad \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \quad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$

$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \text{diag}(f'(\boldsymbol{z}))$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

3. Write out the Jacobians

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

$$\downarrow$$

$$\boldsymbol{u}^T$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$

$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \text{diag}(f'(\boldsymbol{z}))$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

3. Write out the Jacobians

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

$$\boldsymbol{u}^T \operatorname{diag}(f'(\boldsymbol{z}))$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$

$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \operatorname{diag}(f'(\boldsymbol{z}))$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

3. Write out the Jacobians

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

$$= \boldsymbol{u}^T \operatorname{diag}(f'(\boldsymbol{z}))\boldsymbol{I}$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$

$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \operatorname{diag}(f'(\boldsymbol{z}))$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

3. Write out the Jacobians

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad (\text{input})$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

$$= \boldsymbol{u}^T \operatorname{diag}(f'(\boldsymbol{z}))\boldsymbol{I}$$

$$= \boldsymbol{u}^T \odot f'(\boldsymbol{z})$$

Useful Jacobians from previous slide

$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T \boldsymbol{h}) = \boldsymbol{h}^T$$

$$\frac{\partial}{\partial \boldsymbol{z}}(f(\boldsymbol{z})) = \operatorname{diag}(f'(\boldsymbol{z}))$$

$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{I}$$

$\odot$ = Hadamard product = element-wise multiplication of 2 vectors to give vector

Re-using Computation

- Suppose we now want to compute $\frac{\partial s}{\partial \boldsymbol{W}}$
  - Using the chain rule again:

$$\frac{\partial s}{\partial \boldsymbol{W}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$$

Re-using Computation

- Suppose we now want to compute $\frac{\partial s}{\partial \boldsymbol{W}}$
  - Using the chain rule again:

$$\frac{\partial s}{\partial \boldsymbol{W}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}$$

The same! Let's avoid duplicated computation ...

Re-using Computation

- Suppose we now want to compute $\frac{\partial s}{\partial \boldsymbol{W}}$
  - Using the chain rule again:

$$\frac{\partial s}{\partial \boldsymbol{W}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$$

$$\frac{\partial s}{\partial \boldsymbol{b}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} = \boldsymbol{\delta}$$

$$\boldsymbol{\delta} = \frac{\partial s}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} = \boldsymbol{h}^T \circ f'(\boldsymbol{z})$$

δ is the local error signal

Derivative with respect to Matrix

- What does $\frac{\partial s}{\partial \boldsymbol{W}}$ look like? $\boldsymbol{W} \in \mathbb{R}^{n \times m}$
- 1 output, *nm* inputs: 1 by *nm* Jacobian?
  - Inconvenient to then do $\theta^{new} = \theta^{old} - \alpha \nabla_\theta J(\theta)$

- Instead, we leave pure math and use the shape convention: the shape of the gradient is the shape of the parameters!

  - So $\frac{\partial s}{\partial \boldsymbol{W}}$ is *n* by *m*:

$$\begin{bmatrix} \frac{\partial s}{\partial W_{11}} & \cdots & \frac{\partial s}{\partial W_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial s}{\partial W_{n1}} & \cdots & \frac{\partial s}{\partial W_{nm}} \end{bmatrix}$$

Derivative with respect to Matrix

- What is $\frac{\partial s}{\partial \boldsymbol{W}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$
  - The other term should be $\boldsymbol{x}$ because $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$

- Answer is: $\frac{\partial s}{\partial \boldsymbol{W}} = \boldsymbol{\delta}^T \boldsymbol{x}^T$

δ is local error signal at z; x is local input signal

Deriving local input gradient in backprop

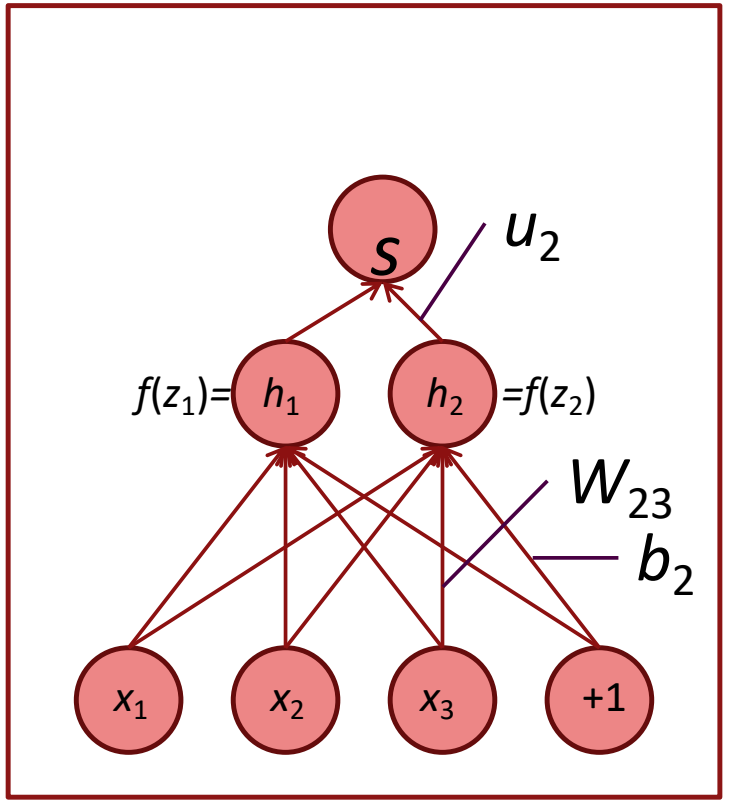
- For $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$ in our equation:

$$\frac{\partial s}{\partial \boldsymbol{W}} = \boldsymbol{\delta}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \boldsymbol{\delta}\frac{\partial}{\partial \boldsymbol{W}}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

- Let's consider the derivative of a single weight W_{ij}
- W_{ij} only contributes to z_i
- For example: W_{23} is only used to compute z_2 not z_1

$$\frac{\partial z_i}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}}\boldsymbol{W}_{i.}\boldsymbol{x} + b_i$$

$$= \frac{\partial}{\partial W_{ij}}\sum_{k=1}^{d} W_{ik}x_k = x_j$$



Why the Transposes?

$$\frac{\partial s}{\partial \boldsymbol{W}} = \boldsymbol{\delta}^T \quad \boldsymbol{x}^T$$

$$[n \times m] \quad [n \times 1][1 \times m]$$

$$= \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_n \end{bmatrix} [x_1, ..., x_m] = \begin{bmatrix} \delta_1 x_1 & \dots & \delta_1 x_m \\ \vdots & \ddots & \vdots \\ \delta_n x_1 & \dots & \delta_n x_m \end{bmatrix}$$

- Hacky answer: this makes the dimensions work out!
  - Useful trick for checking your work!
  - Each input goes to each output – you want to get outer product

3. Backpropagation

- We've almost shown you backpropagation
  - It's taking derivatives and using the (generalized, multivariate, or matrix) chain rule
- Other trick:
  - We **re-use** derivatives computed for higher layers in computing derivatives for lower layers to minimize computation

Computation Graphs and Backpropagation

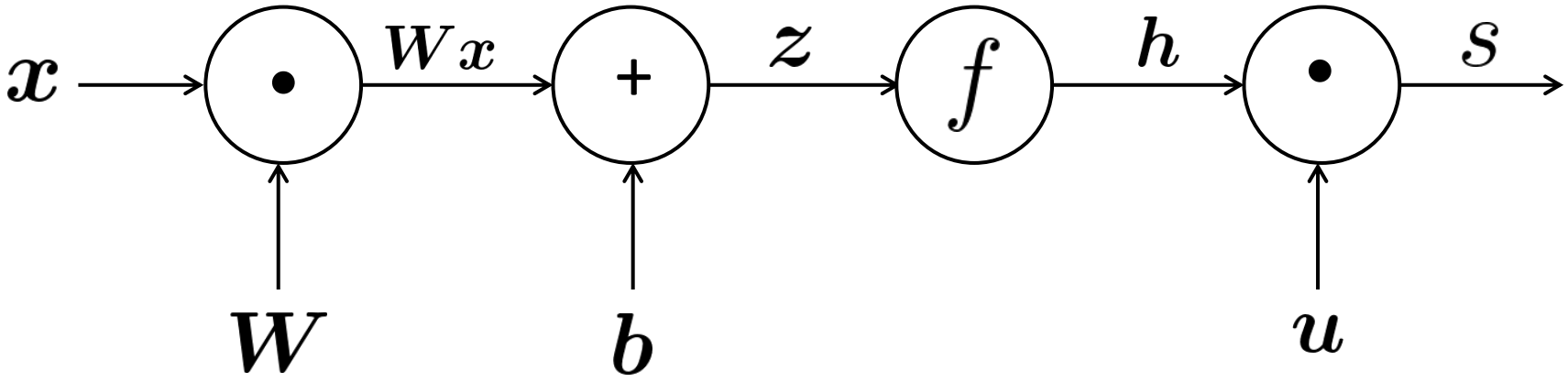
- Software represents our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations
  - Edges pass along result of the operation

$$s = \boldsymbol{u}^T \boldsymbol{h}$$

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} \quad (\text{input})$$

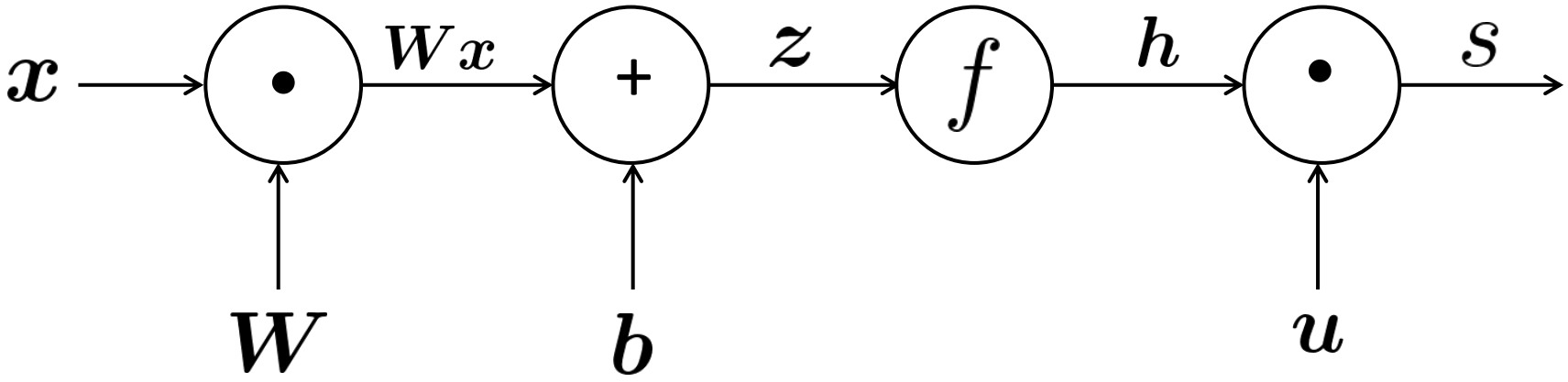


Computation Graphs and Backpropagation

- Software represents our neural net equations as a graph
  - Source nodes: inputs
  - Interior r
  - Edges pa

"Forward Propagation"

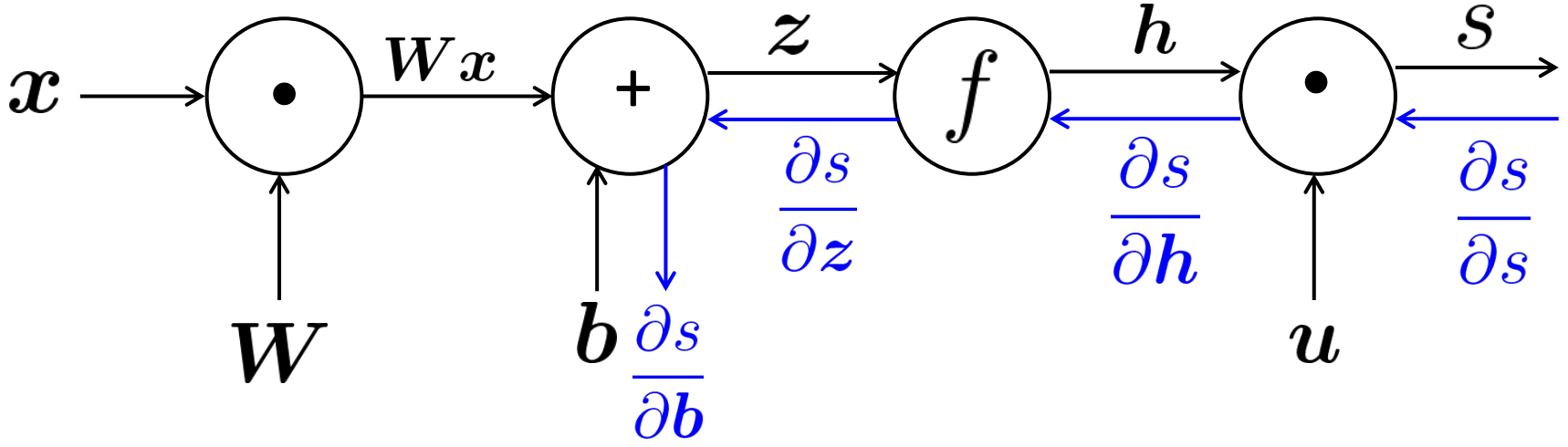
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



Computation Graphs and Backpropagation

- Then go backwards along edges
  - Pass along **gradients**

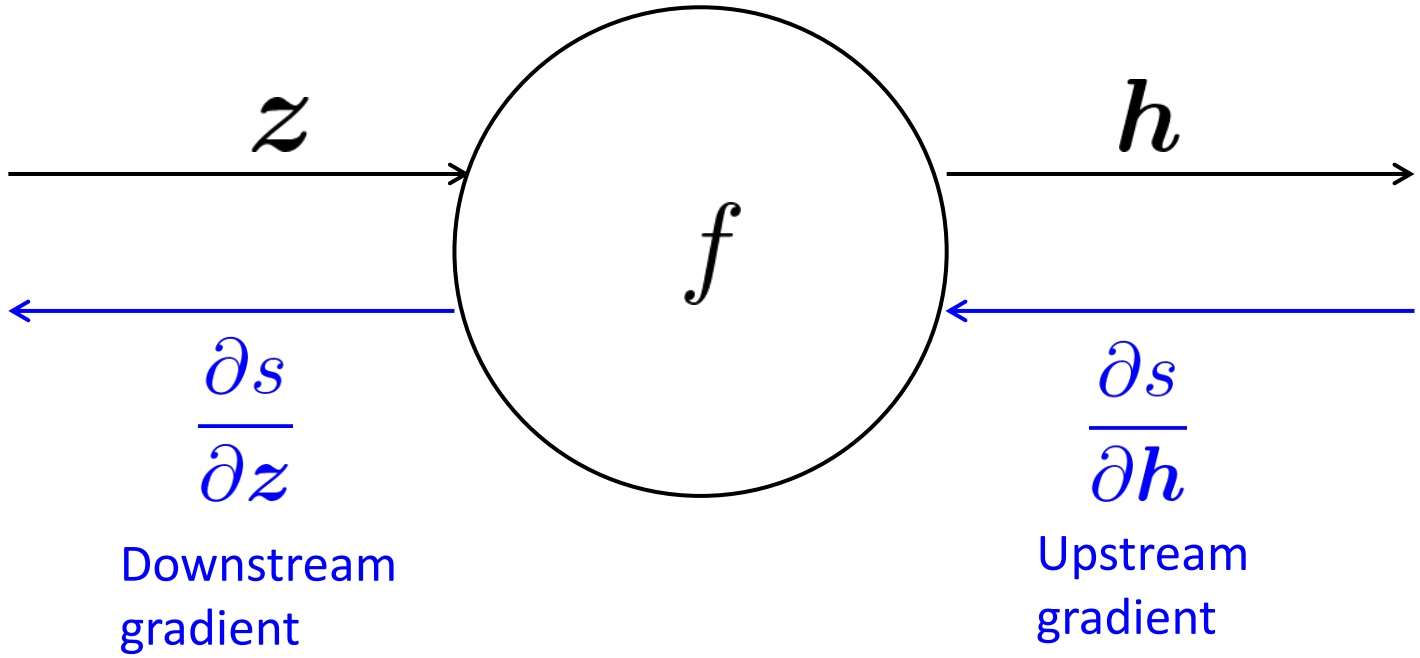
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



Backpropagation: Single Node

- Node receives an “upstream gradient”
- Goal is to pass on the correct “downstream gradient”

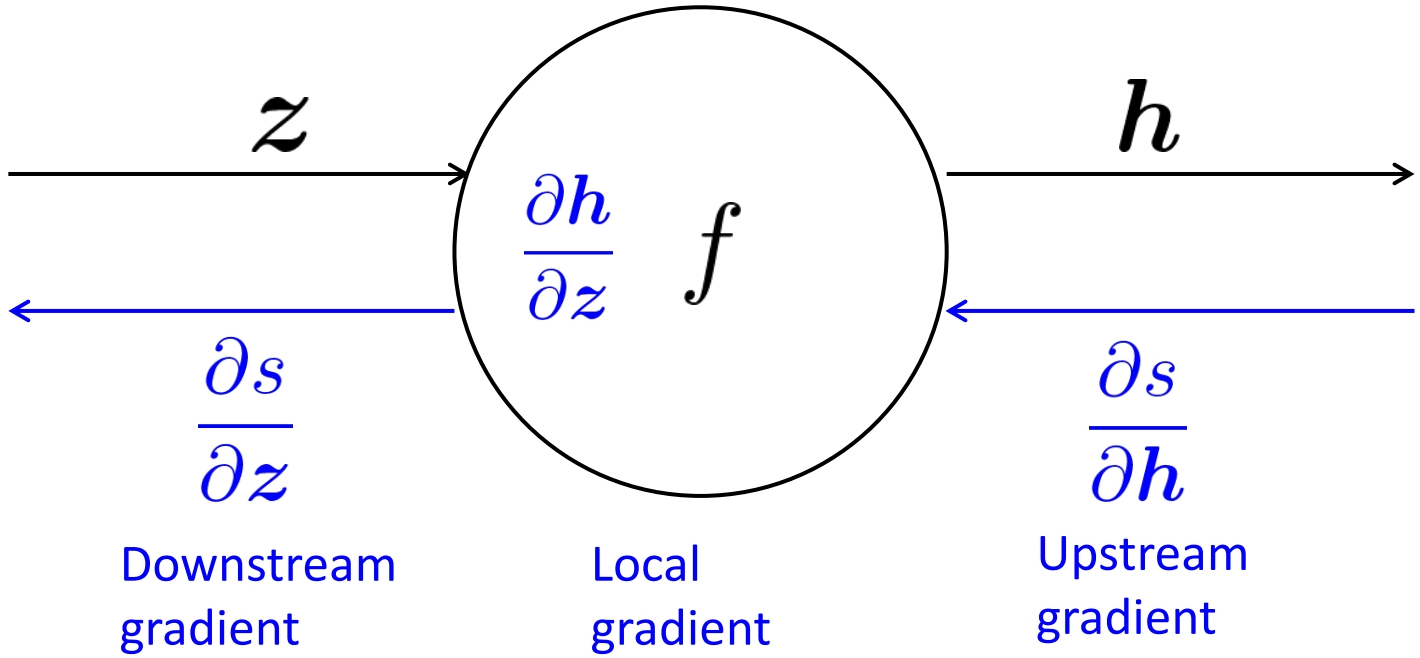
$$\boldsymbol{h} = f(\boldsymbol{z})$$



Backpropagation: Single Node

- Each node has a local gradient
  - The gradient of its output with respect to its input

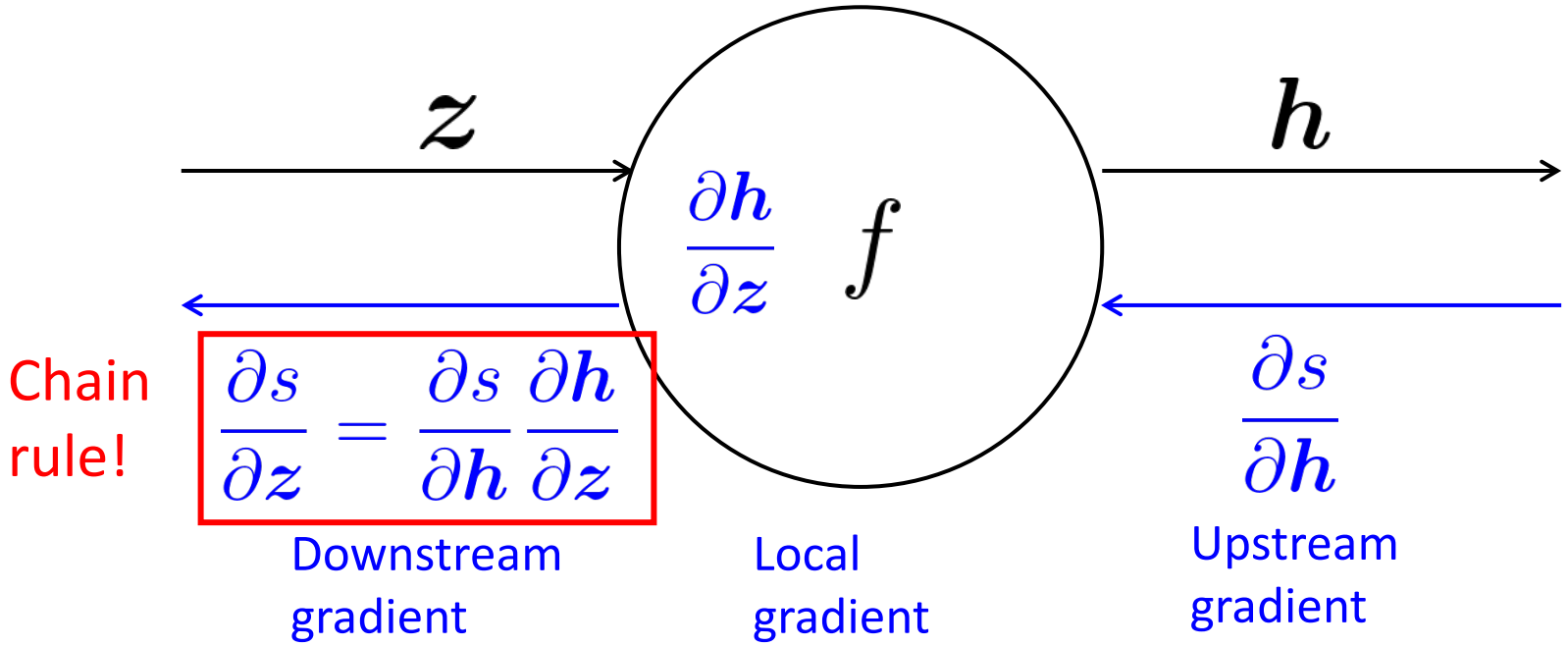
$$\boldsymbol{h} = f(\boldsymbol{z})$$



Backpropagation: Single Node

- Each node has a local gradient
  - The gradient of its output with respect to its input

$$\boldsymbol{h} = f(\boldsymbol{z})$$

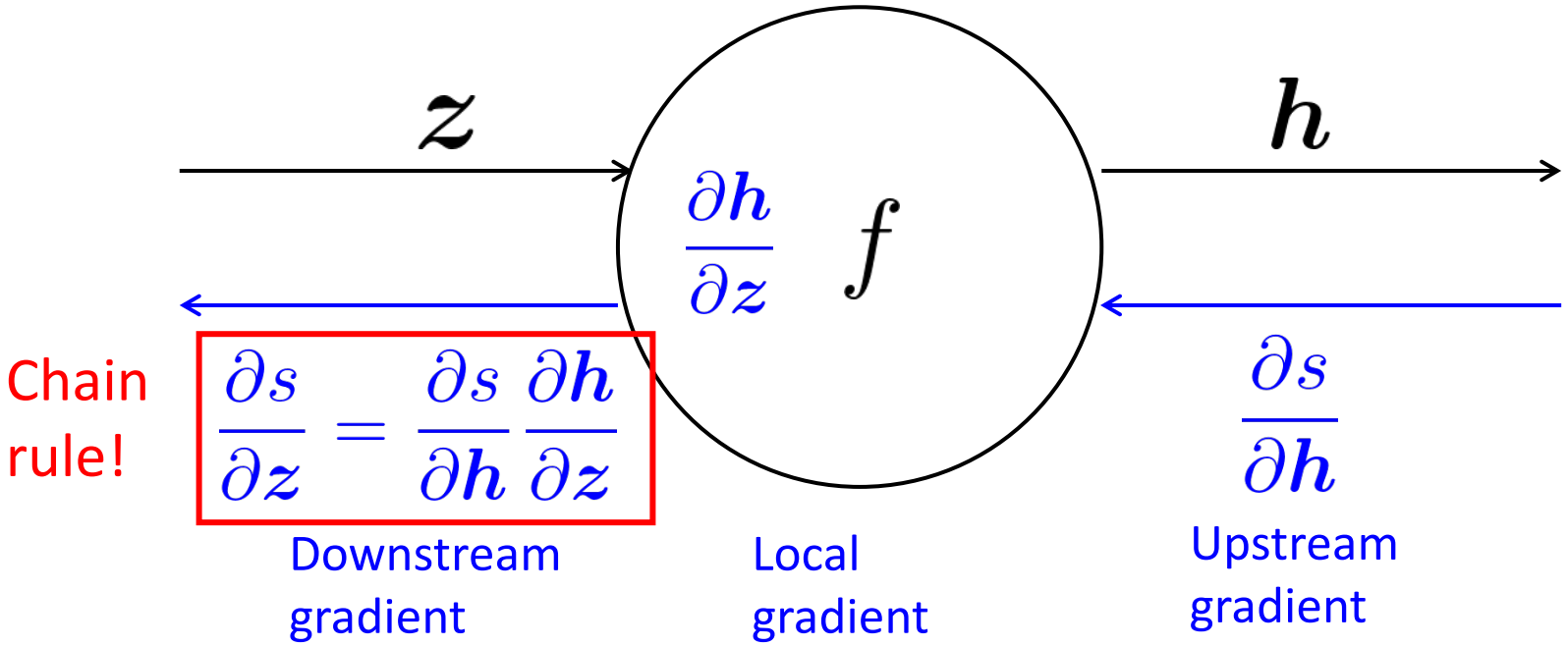


Backpropagation: Single Node

- Each node has a local gradient
  - The gradient of its output with respect to its input

$$\boldsymbol{h} = f(\boldsymbol{z})$$

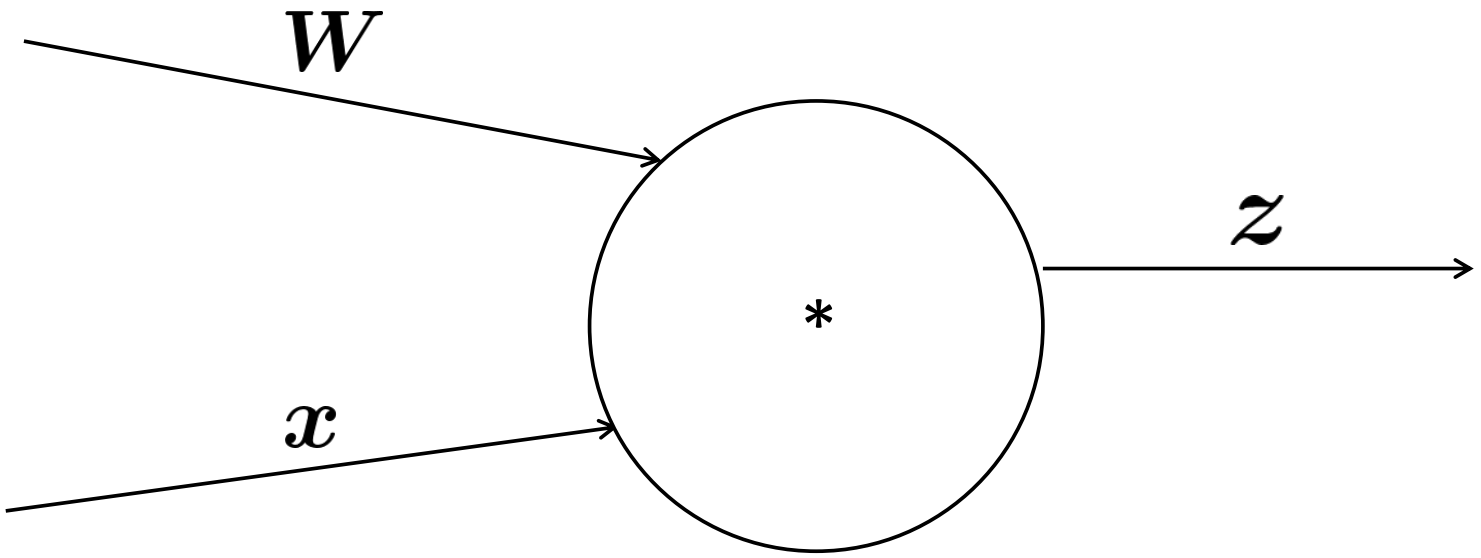
- [downstream gradient] = [upstream gradient] x [local gradient]



Backpropagation: Single Node

- What about nodes with multiple inputs?

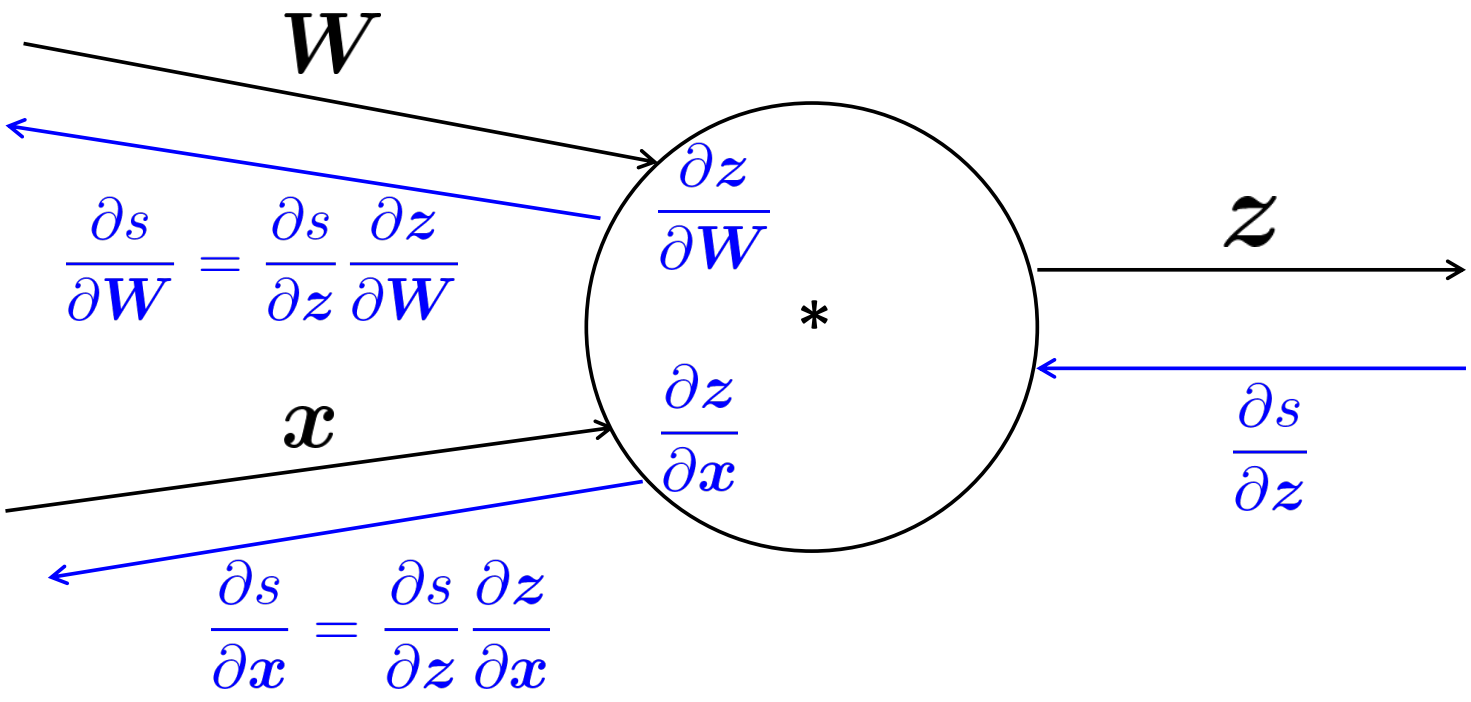
$$z = Wx$$



Backpropagation: Single Node

- Multiple inputs → multiple local gradients

$$z = Wx$$



Downstream gradients

Local gradients

Upstream gradient

An Example

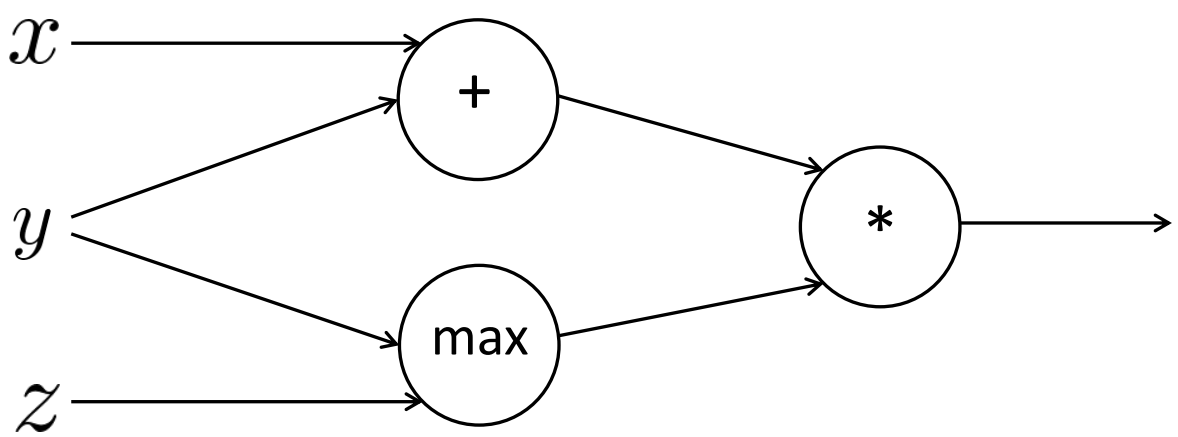
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

$$b = \max(y, z)$$

$$f = ab$$



An Example

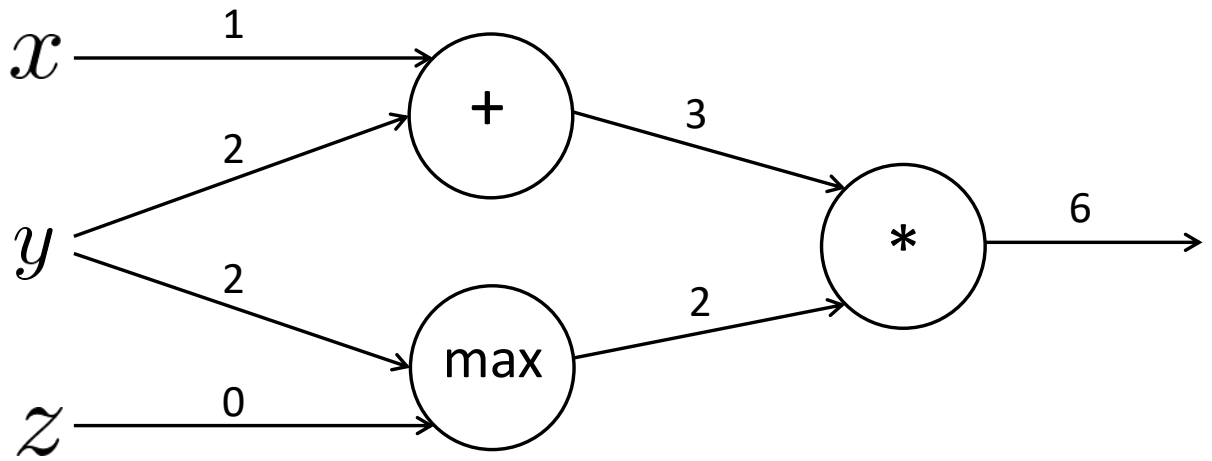
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$

$$b = \max(y, z)$$

$$f = ab$$



An Example

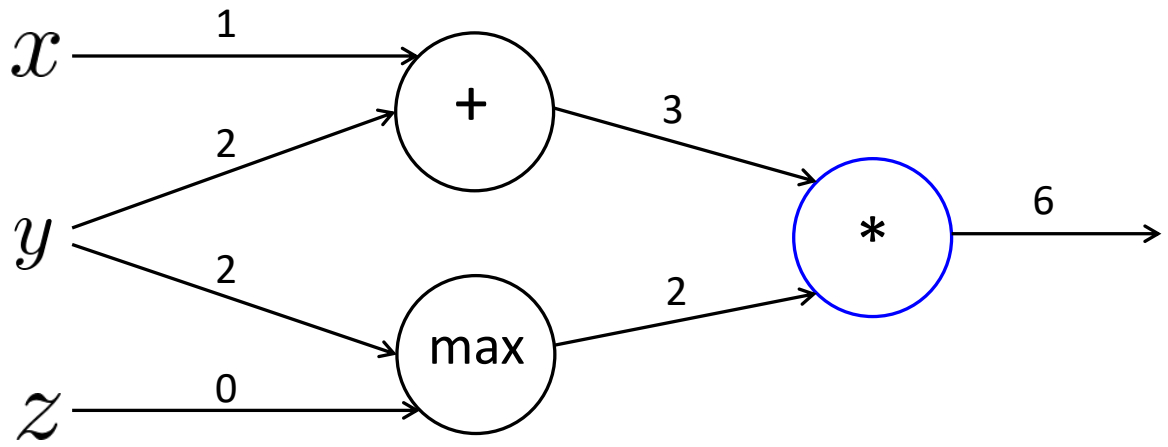
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



An Example

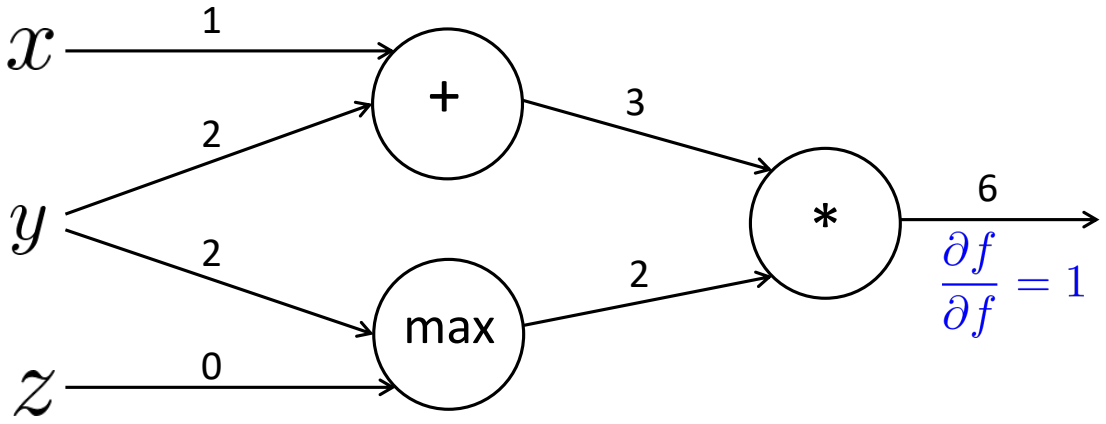
$$f(x, y, z) = (x + y) \max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



An Example

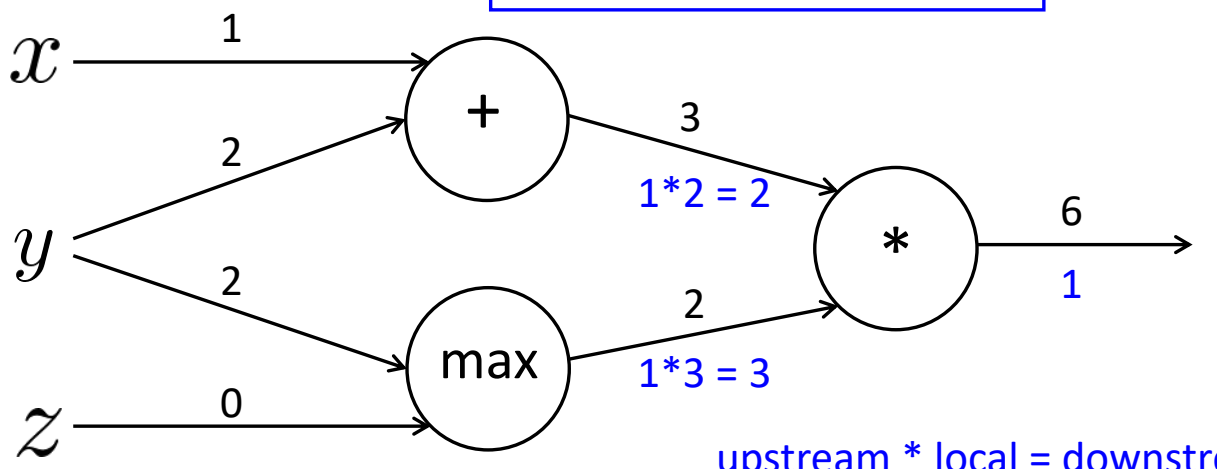
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



upstream * local = downstream

An Example

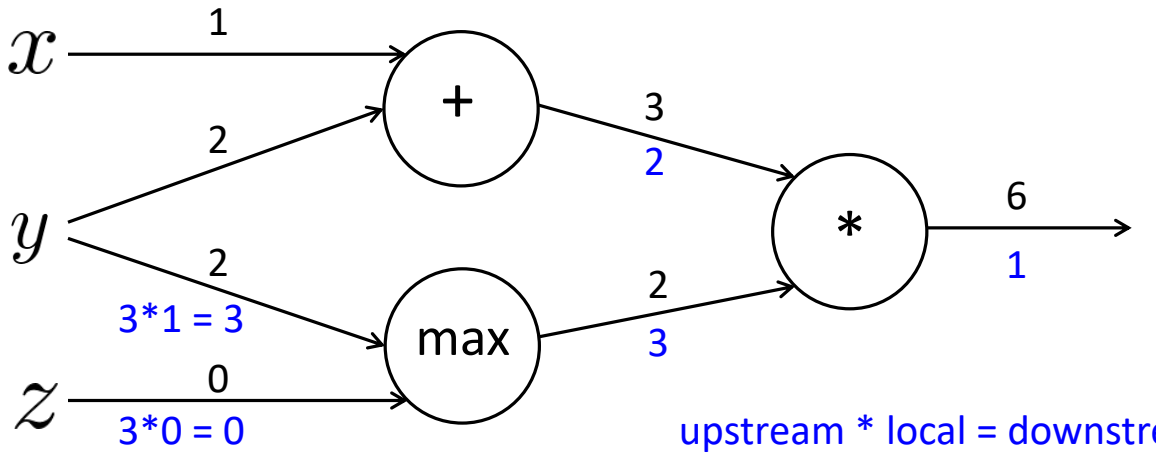
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



An Example

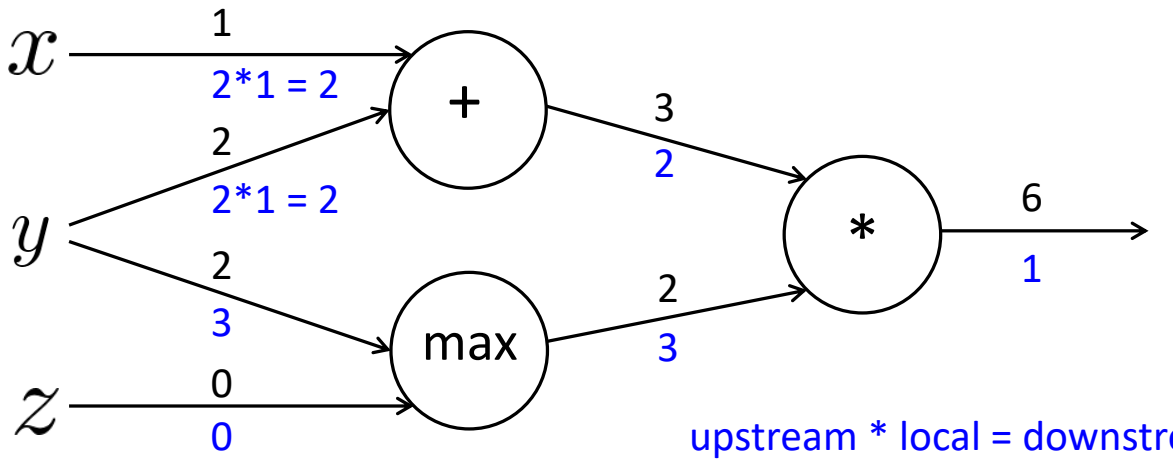
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



An Example

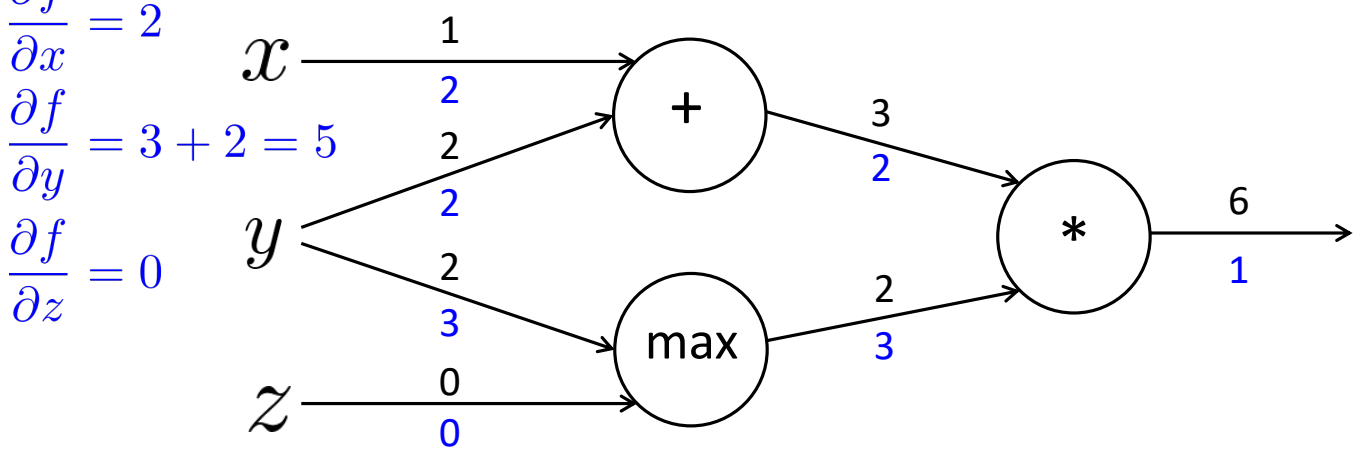
$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps

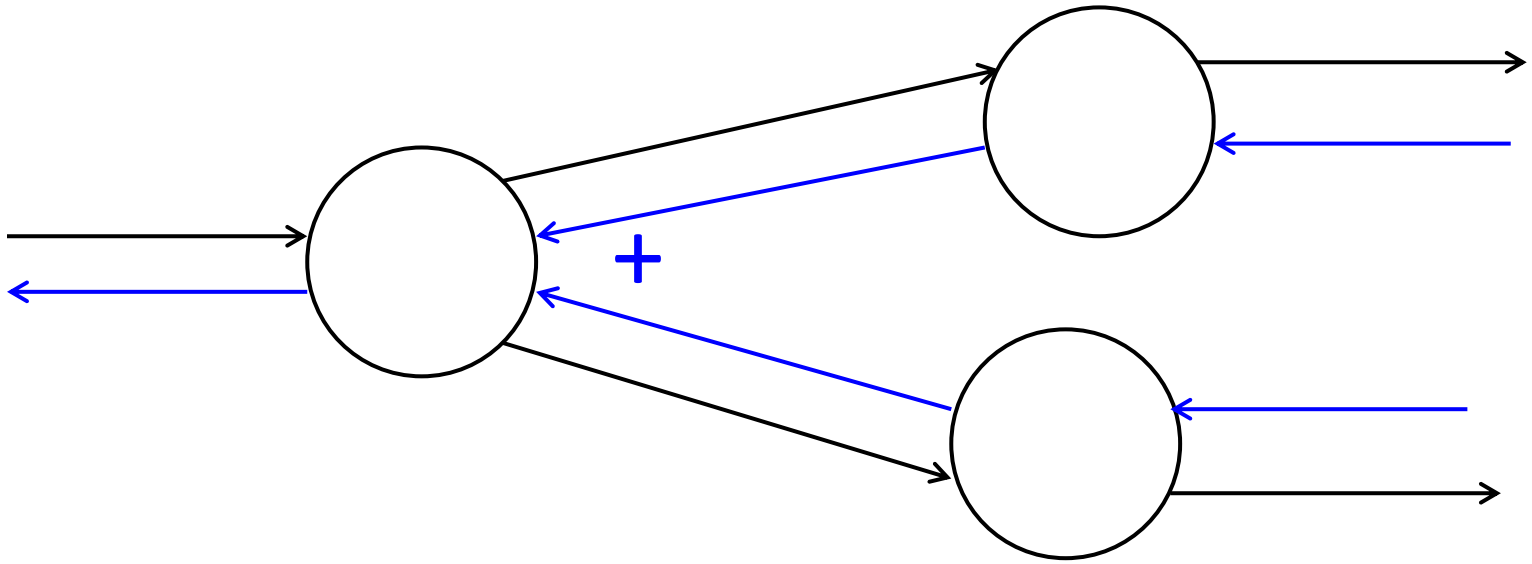
$$a = x + y$$
$$b = \max(y, z)$$
$$f = ab$$

Local gradients

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial y} = 1$$
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$
$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$



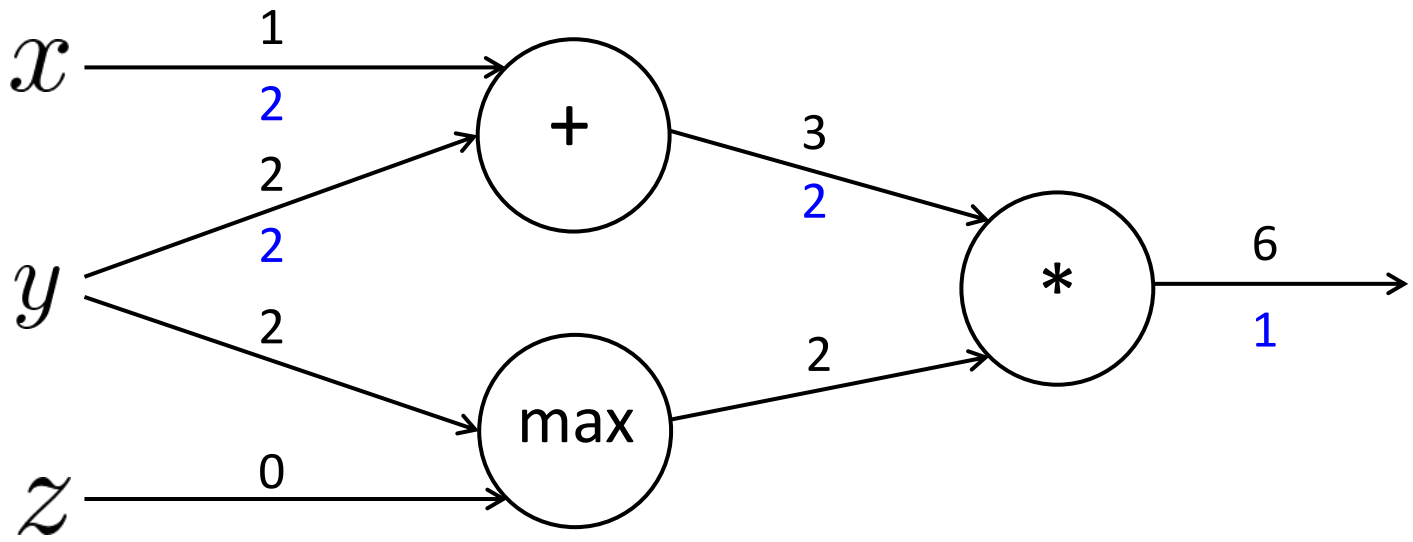
Gradients sum at outward branches



Node Intuitions

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

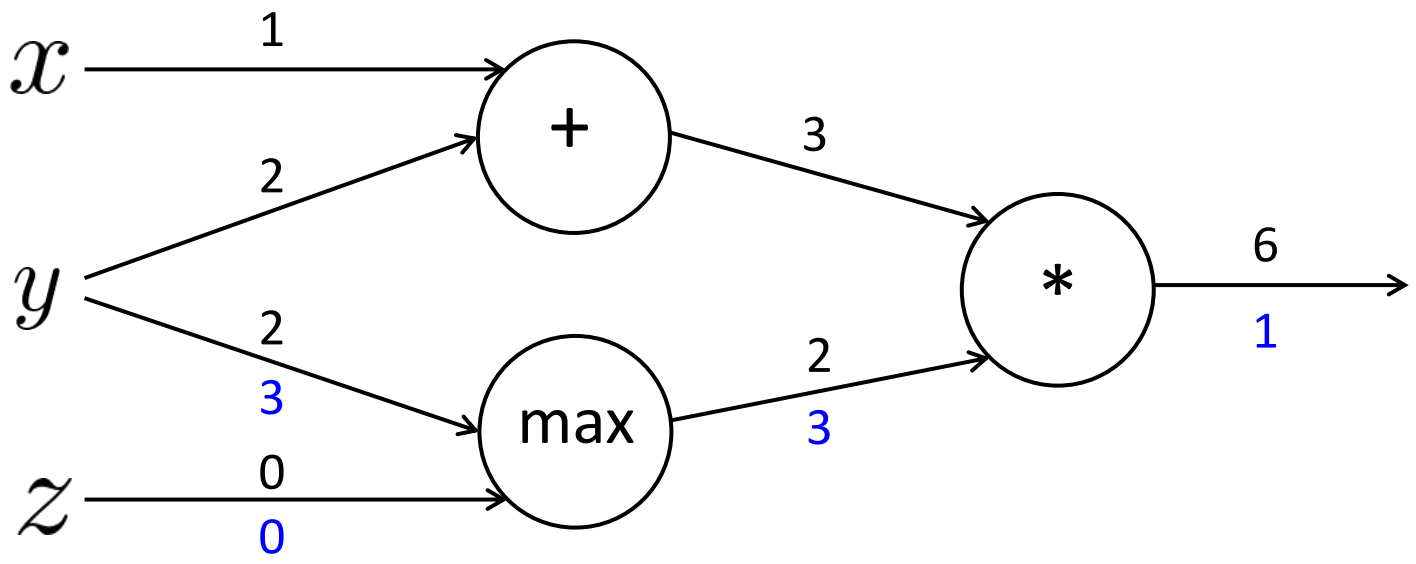
- + "distributes" the upstream gradient to each summand



Node Intuitions

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

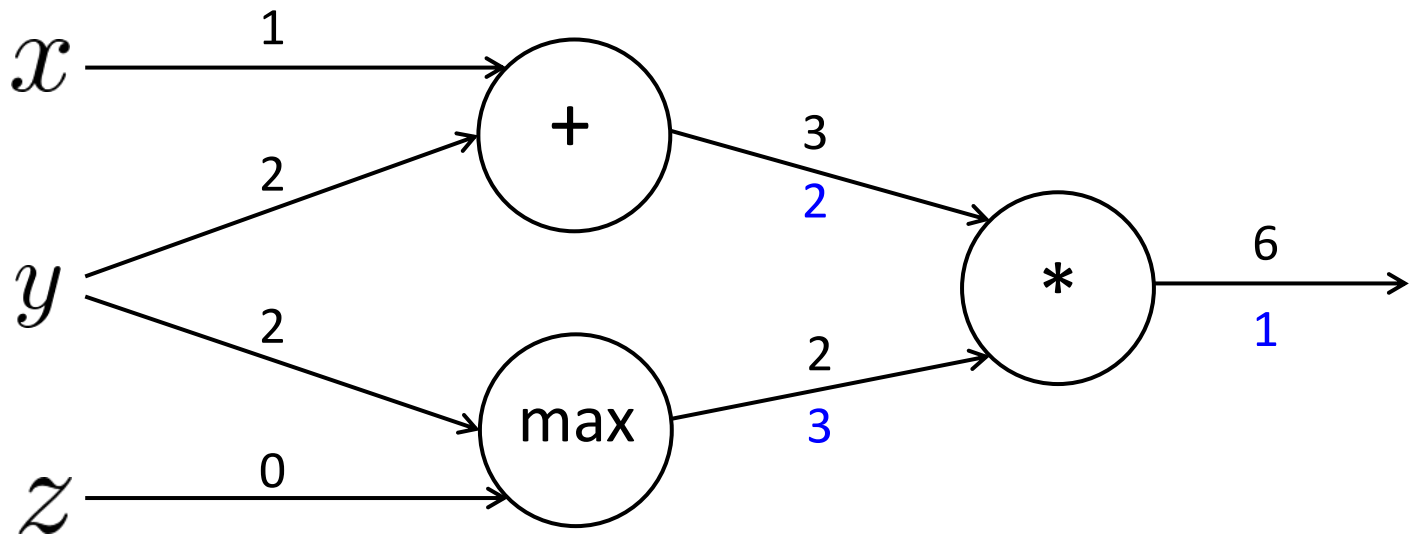
- + "distributes" the upstream gradient to each summand
- max "routes" the upstream gradient



Node Intuitions

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

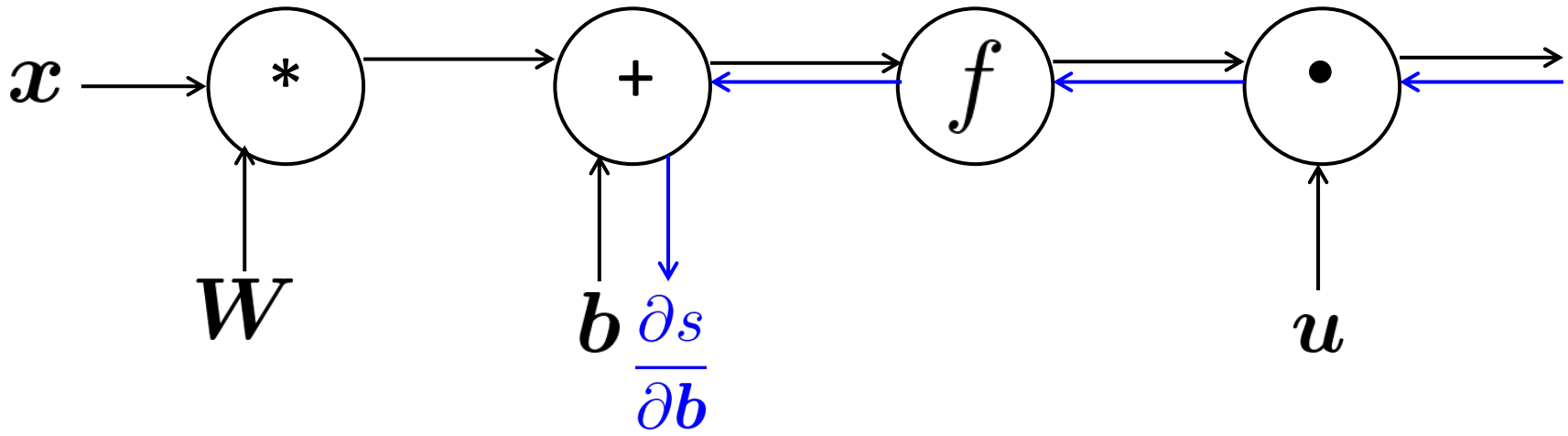
- + “distributes” the upstream gradient to each summand
- max “routes” the upstream gradient
- * “switches” the upstream gradient



Efficiency: compute all gradients at once

- Incorrect way of doing backprop:
  - First compute $\frac{\partial s}{\partial \boldsymbol{b}}$

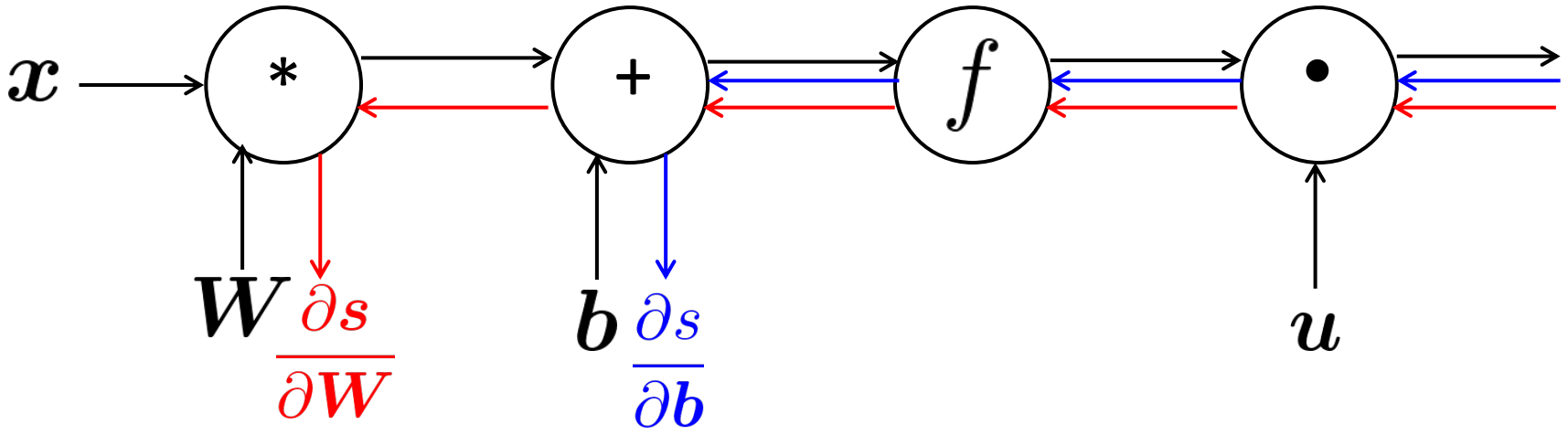
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



Efficiency: compute all gradients at once

- Incorrect way of doing backprop:
  - First compute $\frac{\partial s}{\partial \boldsymbol{b}}$
  - Then independently compute $\frac{\partial \boldsymbol{s}}{\partial \boldsymbol{W}}$
  - Duplicated computation!

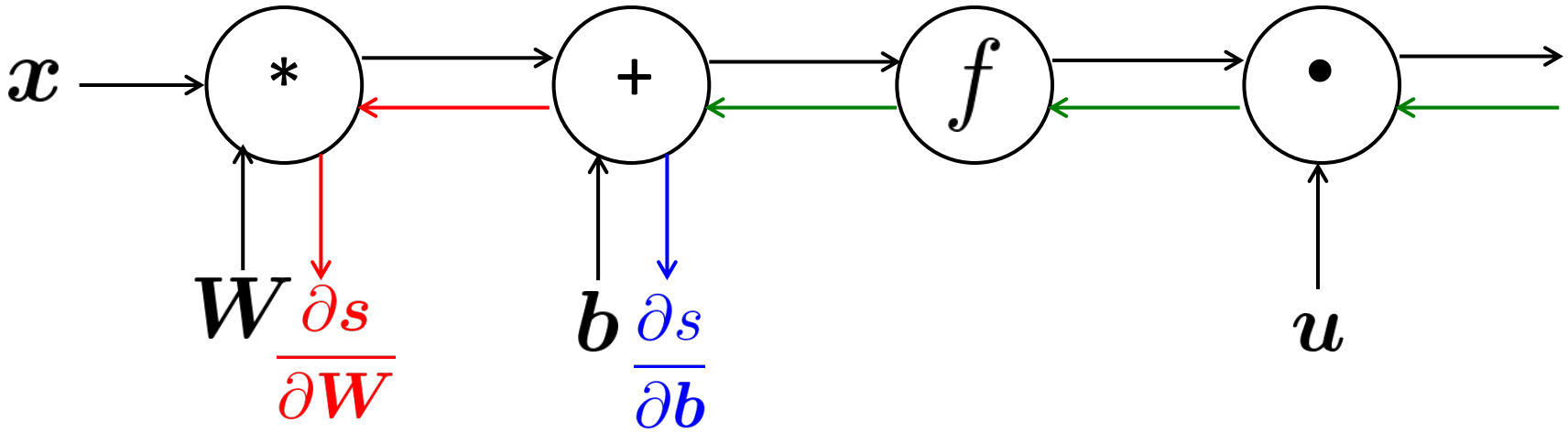
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



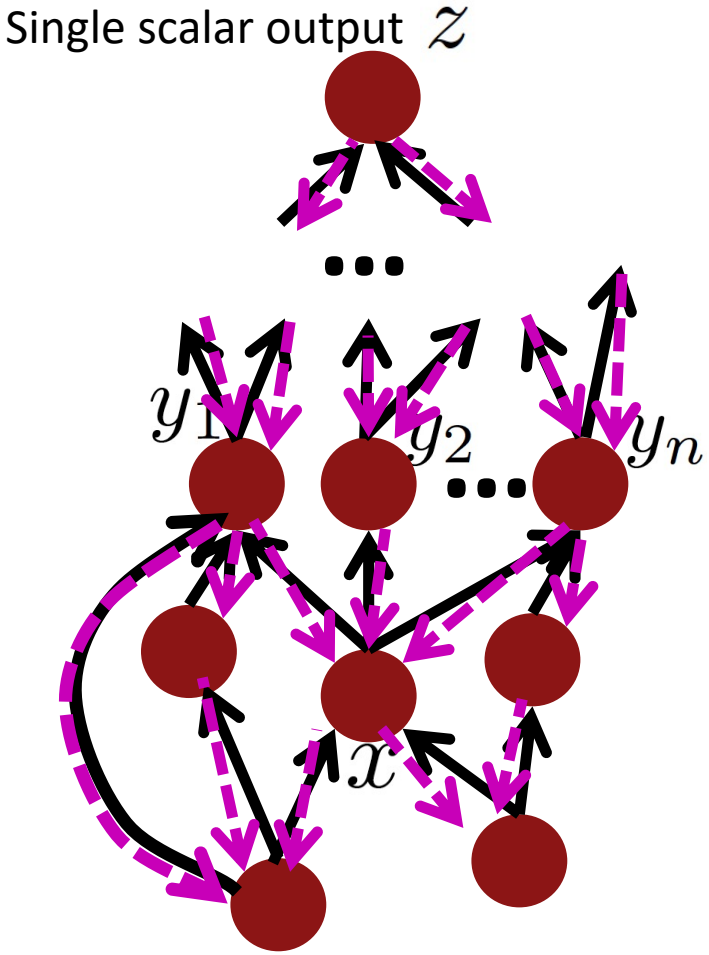
Efficiency: compute all gradients at once

- Correct way:
  - Compute all the gradients at once
  - Analogous to using $\boldsymbol{\delta}$ when we computed gradients by hand

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



Back-Prop in General Computation Graph



1. Fprop: visit nodes in topological sort order
   - Compute value of node given predecessors
2. Bprop:
   - initialize output gradient = 1
   - visit nodes in reverse order:
   Compute gradient wrt each node using gradient wrt successors

$\{y_1, y_2, \ldots y_n\}$ = successors of x

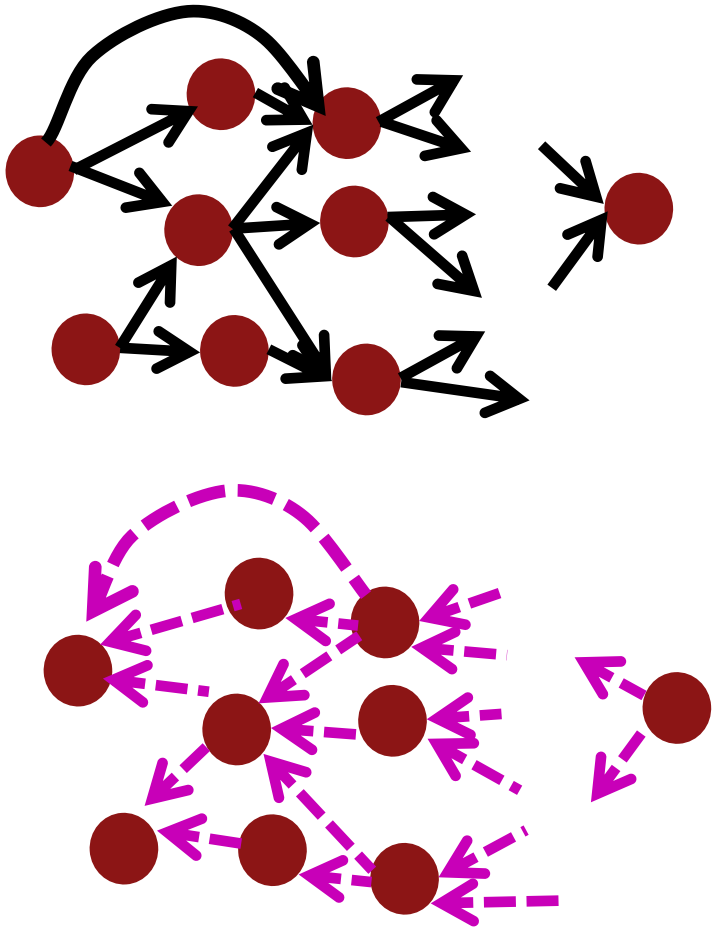
$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Done correctly, big O() complexity of fprop and bprop is **the same**

In general, our nets have regular layer-structure and so we can use matrices and Jacobians…

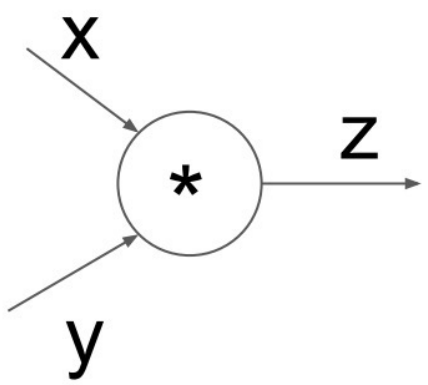
From Stanford CS224n course

Automatic Differentiation



- The gradient computation can be automatically inferred from the symbolic expression of the fprop
- Each node type needs to know how to compute its output and how to compute the gradient w.r.t. its inputs given the gradient w.r.t. its output
- Modern DL frameworks (PyTorch, Tensorflow, etc.) do backpropagation for you but mainly leave layer/node writer to hand-calculate the local derivative

:ward API



(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$\frac{\partial L}{\partial z}$

$\frac{\partial L}{\partial x}$

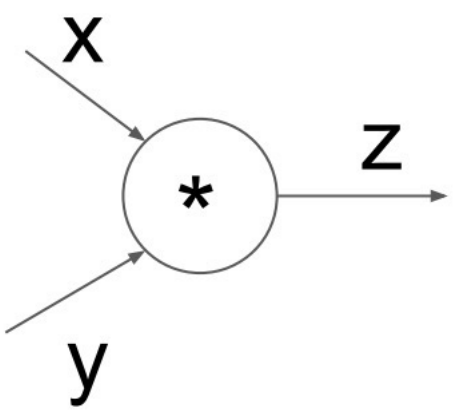
(x,y,z are scalars)

```
        # dy = ... #todo
        return [dx, dy]
```

$\frac{\partial L}{\partial z}$

$\frac{\partial L}{\partial x}$

Implementation: forward/backward API



(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Manual Gradient checking: Numeric Gradient

- For small h (≈ 1e-4),

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

- Easy to implement correctly
- But approximate and **very** slow:
  - You have to recompute *f* for **every parameter** of our model
- Useful for checking your implementation
  - In the old days, we hand-wrote everything, doing this everywhere was the key test
  - Now much less needed; you can use it to check layers are correctly implemented

Summary

- Backpropagation: recursively (and hence efficiently) apply the chain rule along computation graph
  - [downstream gradient] = [upstream gradient] x [local gradient]

- Forward pass: compute results of operations and save intermediate values
- Backward pass: apply chain rule to compute gradients

Why learn all these details about gradients?

- Modern deep learning frameworks compute gradients for you!
  - There will a lecture for PyTorch soon

- But why take a class on compilers or systems when they are implemented for you?
  - Understanding what is going on under the hood is useful!

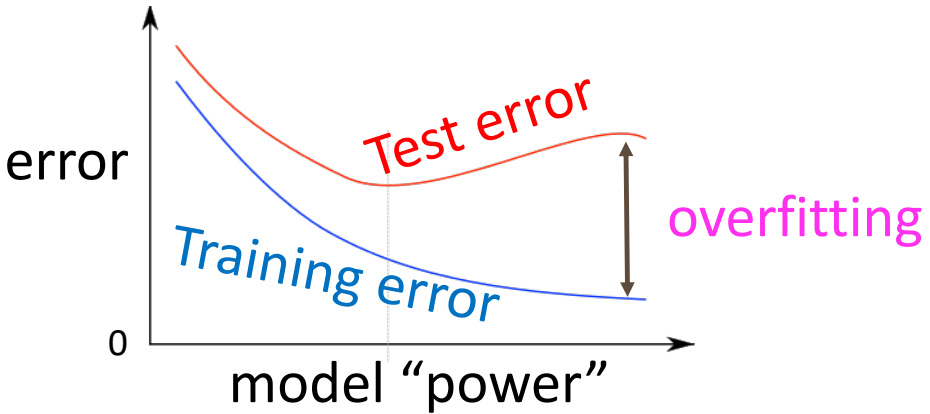
- Backpropagation doesn't always work perfectly out of the box
  - Understanding why is crucial for debugging and improving models
  - See Karpathy article: https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b
  - Example in future lecture: exploding and vanishing gradients

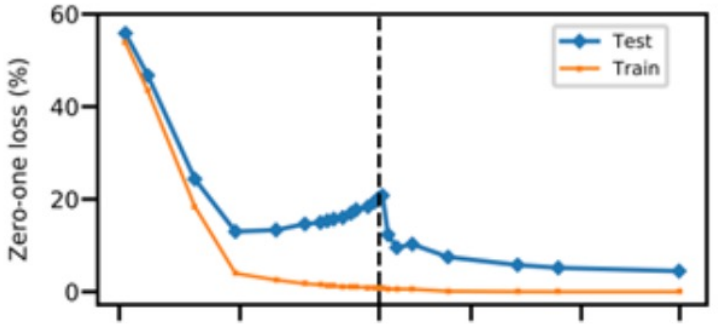
We have models with many parameters! Regularization!

- A full loss function includes **regularization** over all parameters θ, e.g., L2 regularizati

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) + \lambda \sum_k \theta_k^2$$

- Classic view: Regularization works to prevent **overfitting** when we have a lot of features (or later a very powerful/deep model, etc.)
- Now: Regularization **produces models that generalize well** when we have a "big" model
  - We do not care that our models overfit on the training data, even though they are **hugely** overfit!





Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov 2012/JMLR 2014)

- Preventing Feature Co-adaptation = Good Regularization Method! Use it widely!
  - Training time: at each instance of evaluation (in online SGD-training), randomly set 50% of the inputs to each neuron to 0
  - Test time: halve the model weights (now twice as many)
  - (Except usually only drop first layer inputs a little (~15%) or not at all)
  - This prevents feature co-adaptation: A feature cannot only be useful in the presence of particular other features
  - In a single layer: A kind of middle-ground between Naïve Bayes (where all feature weights are set independently) and logistic regression models (where weights are set in the context of all others)
  - Can be thought of as a form of model bagging (i.e., like an ensemble model)
  - Nowadays usually thought of as strong, feature-dependent regularizer [Wager, Wang, & Liang 2013]

“Vectorization”

- E.g., looping over word vectors versus concatenating them all into one large matrix and then multiplying the softmax weights with that matrix:

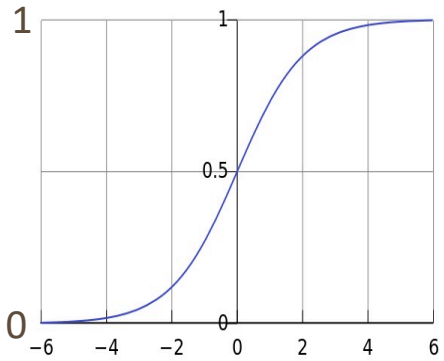
```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

- 1000 loops, best of 3: 639 µs per loop
  10000 loops, best of 3: 53.8 µs per loop <- Now using a single a C x N matrix
- Matrices are awesome!!! Always try to use vectors and matrices rather than for loops!
- The speed gain goes from 1 to 2 orders of magnitude with GPUs!

Non-linearities, old and new

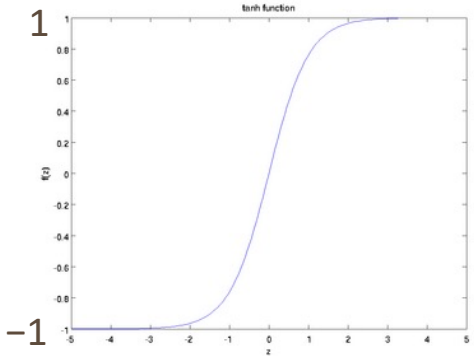
logistic ("sigmoid")

$$f(z) = \frac{1}{1 + \exp(-z)}.$$



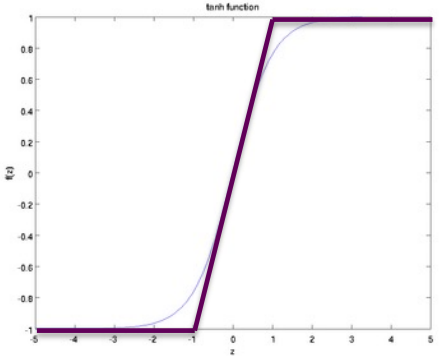
tanh

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$



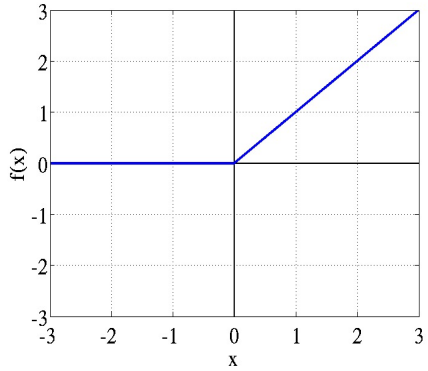
hard tanh

$$\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$



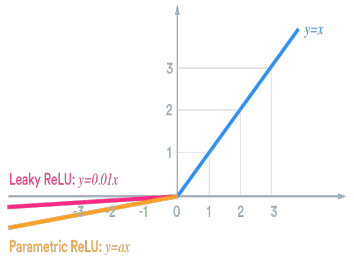
ReLU (Rectified Linear Unit)

$$\text{ReLU}(z) = \max(z, 0)$$

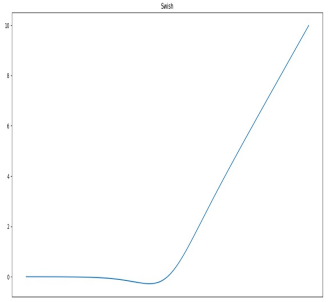


- Both logistic and tanh are still used in various places (e.g., to get a probability), but are no longer the defaults for making deep networks
- For building a deep network, the first thing you should try is ReLU — it trains quickly and performs well due to good gradient backflow

Leaky ReLU / Parametric ReLU



Swish [Ramachandran, Zoph & Le 2017]



Parameter Initialization

- You normally must initialize weights to small random values (i.e., not zero matrices!)
  - To avoid symmetries that prevent learning/specialization
- Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target)
- Initialize **all other weights** ~ Uniform(*–r, r*), with r chosen so numbers get neither too big or too small (later the need for this is removed with use of layer normalization)
- Xavier initialization has variance inversely proportional to fan-in nin (previous layer size) and fan-out n_{out} (next layer size):

$$\mathrm{Var}(W_i) = \frac{2}{n_{\mathrm{in}} + n_{\mathrm{out}}}$$

Optimizers

- Usually, plain SGD will work just fine!
  - However, getting good results will often require hand-tuning the learning rate
    - E.g., start it higher and halve it every *k* epochs (passes through full data, **shuffled** or sampled)
- For more complex nets and situations, or just to avoid worry, you often do better with one of a family of more sophisticated "adaptive" optimizers that scale the adjustment to individual parameters by an accumulated gradient.
  - These models give differential per-parameter learning rates
    - Adagrad
    - RMSprop
    - Adam <- A fairly good, safe place to begin in many cases
    - AdamW
    - SparseAdam
    - ...
  - Can just start them with an initial learning rate, around 0.001