Convolutional & recurrent networks

10-701 Introduction to Machine Learning

Geoff Gordon and Pradeep Ravikumar

Einstein summation

- Convention: a_{ij} means a matrix whose i, j element is a_{ij}
  - range of i, j can be left implicit or specified separately: e.g., $i \in \{1 \ldots 5\}, j \in \{1 \ldots 7\}$
- Similarly, a_{ijkl} means a 4-mode tensor
- So does $a_{ijk} b_{jkl}$ — the i, j, k, l element is $a_{ijk}\, b_{jkl}$
- To represent contraction, we raise one instance of index
  - e.g., $a_{ijk} b^k_{jl}$ is a 3-mode tensor
  - element i, j, l is $\sum_k a_{ijk} b_{jkl}$
- In general (Einstein summation), any index that appears both raised and lowered gets summed out
  - $a^k_{ij} b^l_{jk} c_{lm}$ has indices i, j, m, elements $\sum_k \sum_l a_{ijk} b_{jkl} c_{lm}$

Example

- If

$$a_i^j = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad b_j = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \qquad c^i = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

- then

$$a_i^j b_j =$$

$$a_i^j c^i =$$

Derivative as contraction/ einsum

- Revisiting convention from earlier: if we have two tensors y_{ij} and x_{klm} then derivative $\frac{dy}{dx}$ is a 5-mode tensor representing a linear function from dx to dy
- We can write this linear function as a tensor contraction or Einstein summation:

$$dy_{ij} = A_{ij}^{klm}\, dx_{klm}$$

  - where A is the 5-mode tensor of coefficients
- Indices are $ijklm$, with last 3 indices (the ones corresponding to input x) raised
- Ex: Jacobian is $dy_i = A_i^j\, dx_j$

Batching

- Tensors make it easier to work with minibatches
- Just add a mode (typically last) that indexes samples within a batch
  - 640×480 images in batches of 17 $\rightarrow$ $640 \times 480 \times 17$
- Works with Python indexing conventions: if we drop the last index, we work with all samples at once

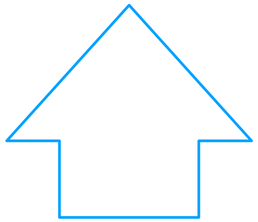
Convolutional neural networks

- We can use a convolution in a deep net layer
  - linear map $a \to a * b + c$ ($a, b \in \mathbb{R}^d$, $c \in \mathbb{R}^d$)
  - input a, weights b, bias c
  - learn b, c by SGD
- Nets with convolutions are *CNNs* or *convnets*
  - can also use the usual set of other blocks: fully connected linear, normalize, pointwise nonlinearity, …

Example: detect ones block of length 3

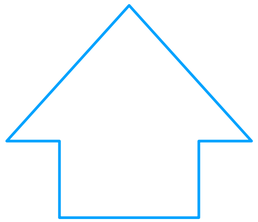
*unlike this example, a real network would **learn** the weights and biases*

$$z_3 = (0 \;\; 0 \;\; 0 \;\; 0 \;\; 1 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0)$$



pointwise nonlinearity: $[z - 1]_+$

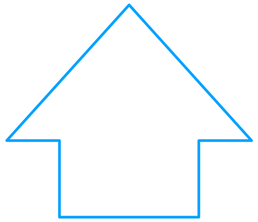
$$z_2 = (0 \;\; -1 \;\; 0 \;\; 0 \;\; 2 \;\; 0 \;\; 0 \;\; -1 \;\; 0 \;\; 0)$$



convolution: weights

$$(1 \;\; 0 \;\; 0 \;\; -1 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0)$$

$$z_1 = (0 \;\; 0 \;\; 0 \;\; 0 \;\; 1 \;\; 0 \;\; 0 \;\; -1 \;\; 0 \;\; 0)$$



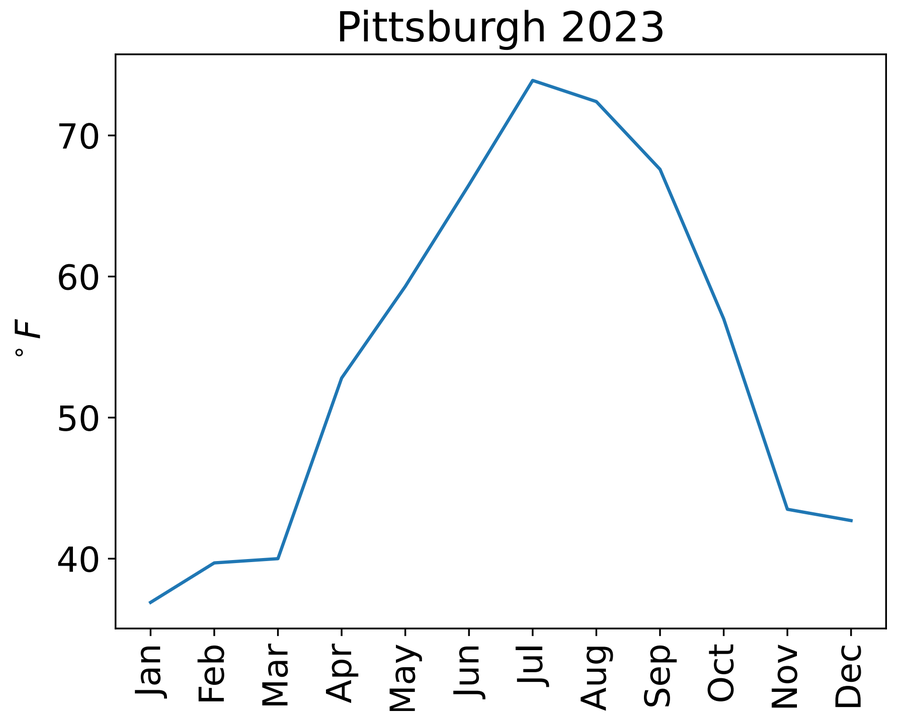
convolution: weights

$$(-1 \;\; 1 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0)$$

$$x = (0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 1 \;\; 1 \;\; 1 \;\; 0 \;\; 0)$$

could group these two

Pooling

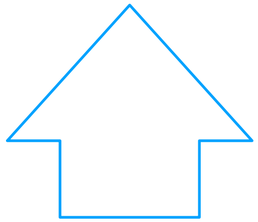


- Convolution gives us translation *equivariance*
- If we want *invariance*,
  - e.g., find the average value of the signal
  - e.g., check whether the max of the signal is ≥ threshold
- Need another step — called *pooling*
  - max-pool a group of input units = make a single unit whose value is max of inputs

Pooling in action

Change from ***localize*** *to* ***detect***

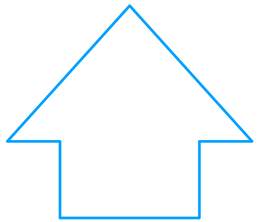
$$z_4 = (1)$$



Max pool: $\max(0,0,0,0,1,0,0,0,0,0) = 1$

Sum pool: $0+0+0+0+1+0+0+0+0+0 = 1$

$$z_3 = (0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0)$$



Convolution (w/ bias) followed by $[\,\cdot\,]_+$

$$x = (0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0)$$

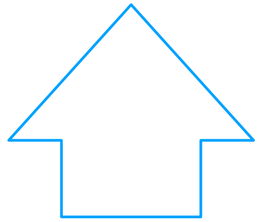
Local pooling

- Might want only *local* invariance (small shifts don't change output much)
- Can do *local* pooling — e.g., average or max of a block of adjacent values
- Can slide a window along signal as in convolution
- Or divide into blocks and pool each block (downsample)

Local pooling example: flexible-length ones block detector

from here, ReLU and pooling can detect or localize

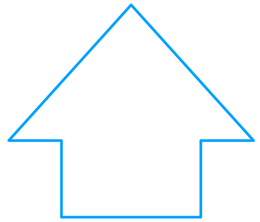
$$z_3 = (0 \;\; -1 \;\; 2 \;\; -1 \;\; 0)$$



convolution: weights

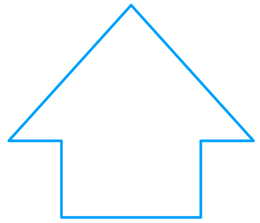
$$(1 \;\; -1)$$

$$z_2 = (0 \;\; 0 \;\; 1 \;\; -1 \;\; 0)$$



sum pool in blocks of 2

$$z_1 = (0 \;\; 0 \;\; 0 \;\; 0 \;\; 1 \;\; 0 \;\; 0 \;\; -1 \;\; 0 \;\; 0)$$



convolution: weights

$$(-1 \;\; 1)$$

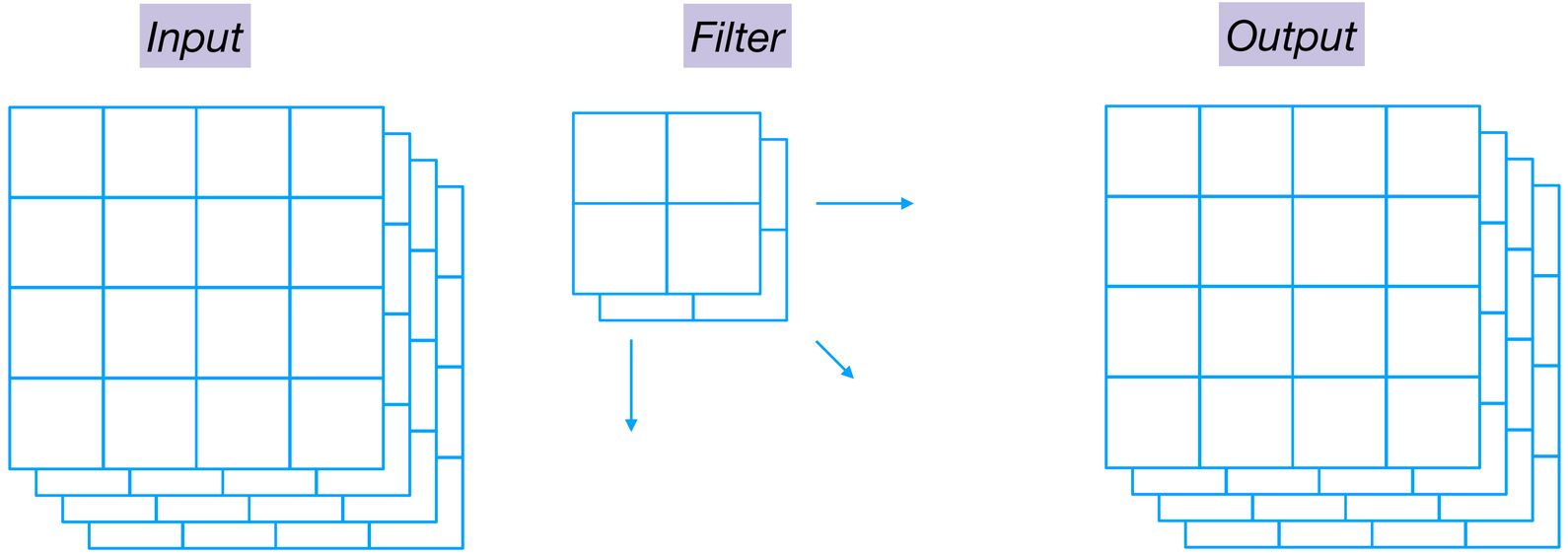
$$x = (0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 1 \;\; 1 \;\; 1 \;\; 0 \;\; 0)$$

Channels

- Can do multiple convolutions in parallel
  - $a \to (a * b_1 + c_1,\ a * b_2 + c_2,\ \ldots)$
  - each parameter set is a *head,* each output is a *channel*
  - different heads extract different kinds of info from signal
- Changes shape of activation tensor
  - e.g., if layer has 3d input: multiple heads → 4d output
  - e.g., image → (stack of band-pass images)
- If we think of a tensor as containing multiple channels, the channel index is also called *depth* or *layer* index

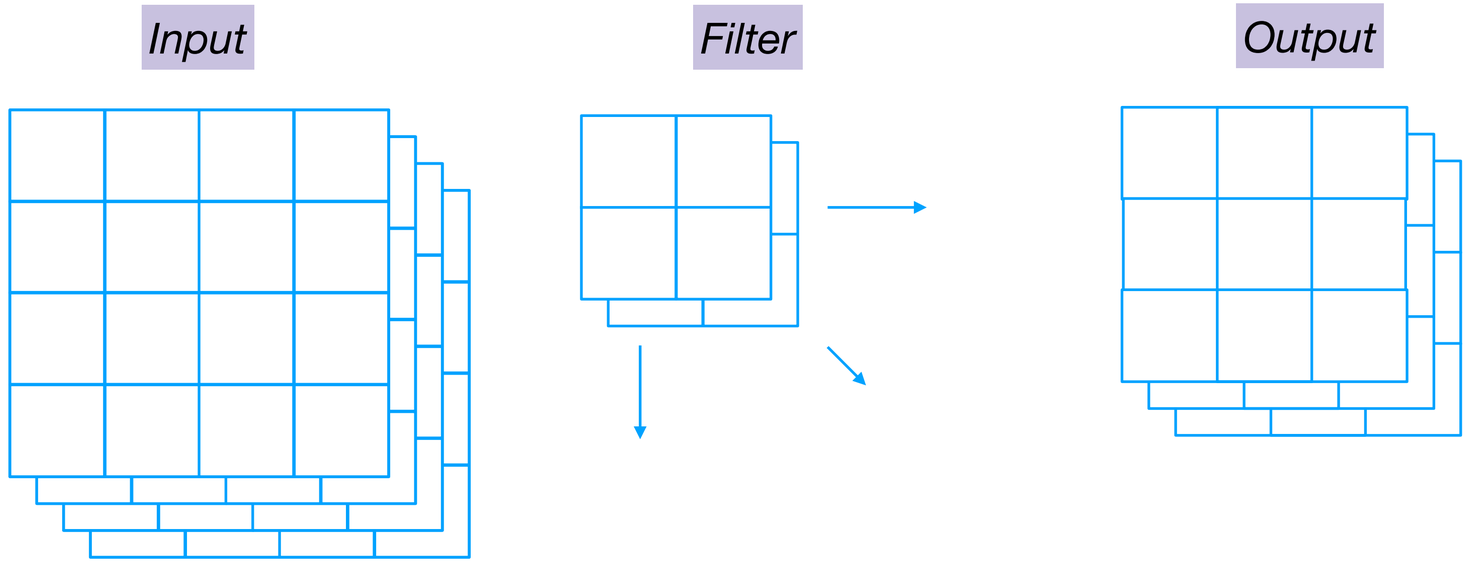
Full convolution

parameters like dilation and stride can be per-mode



- Without channels, the only option is a ***full*** convolution: filter has same number of modes as input
  - iterate over all positions in the input, placing the filter at each one and computing dot product
  - in this example, we're choosing the output to be the same size as the input (using padding or cyclic boundary conditions so that we don't have to skip filter placements near edges of input tensor)

Full convolution (no padding)

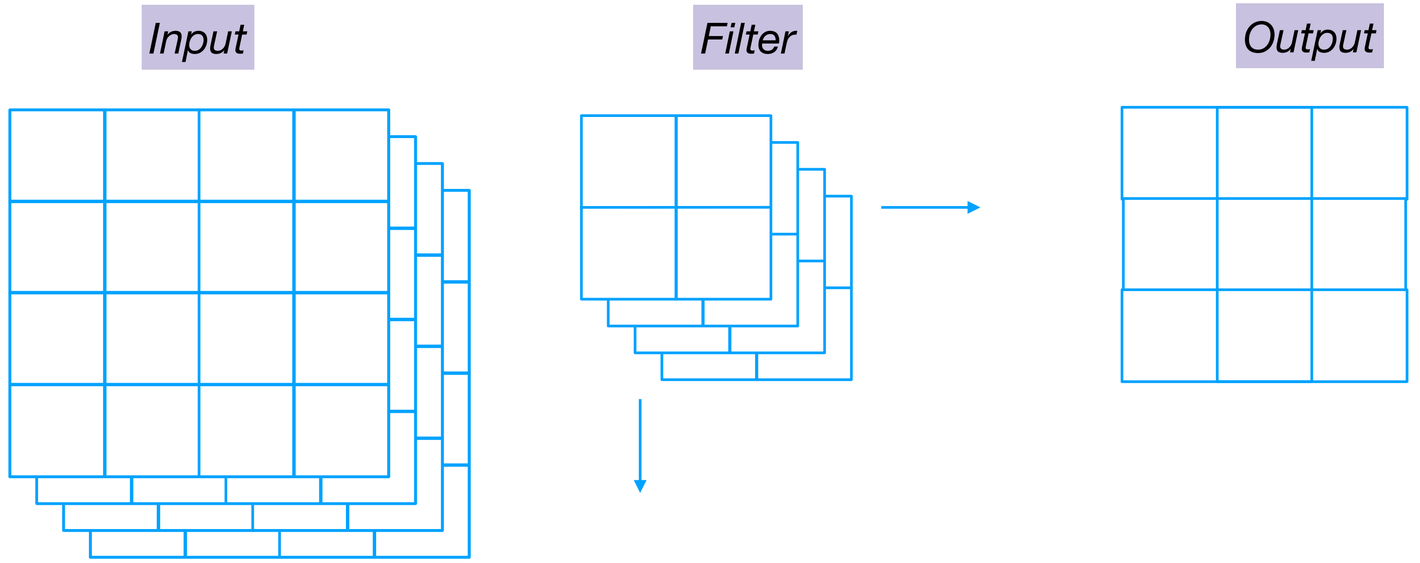


- If we use only ***valid*** positions (no padding or cyclic boundary), output is smaller than input
- Special case: if

```
filter.shape[-1] == input.shape[-1]
```

  - then last mode of output only has length 1
  - if desired we can drop this mode, reducing the number of modes of the output

Full convolution (no padding)

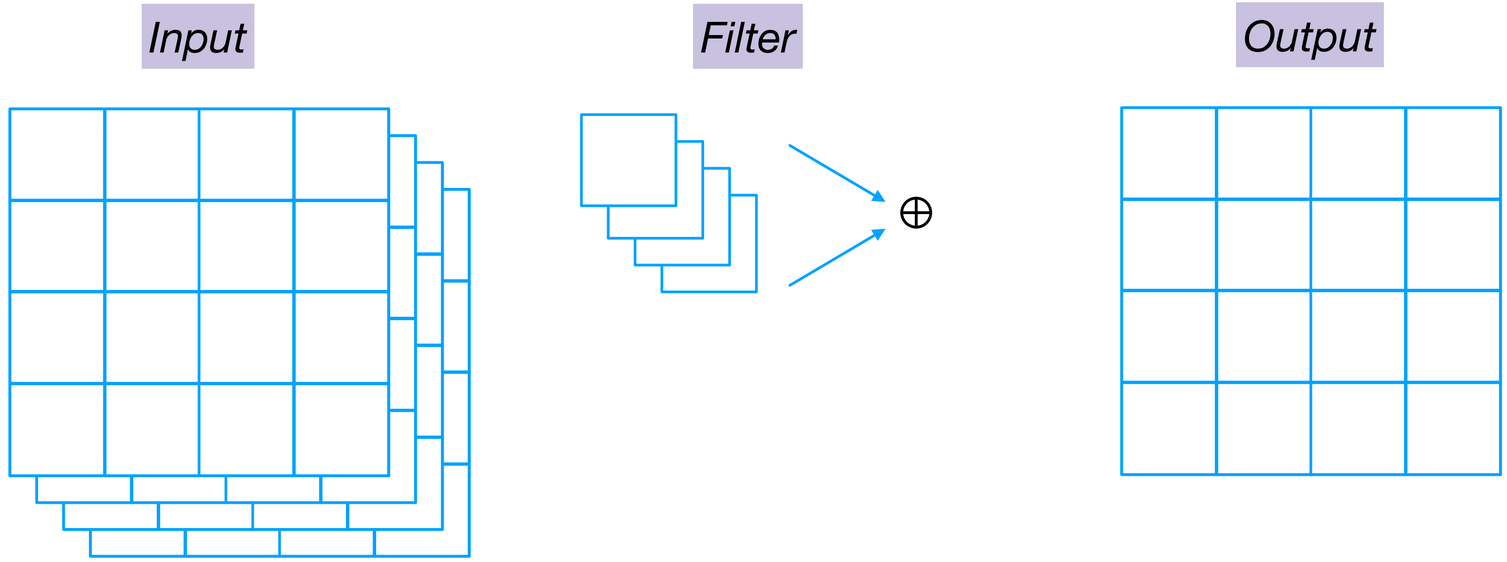


- If we use only ***valid*** positions (no padding or cyclic boundary), output is smaller than input
- Special case: if

```
filter.shape[-1] == input.shape[-1]
```

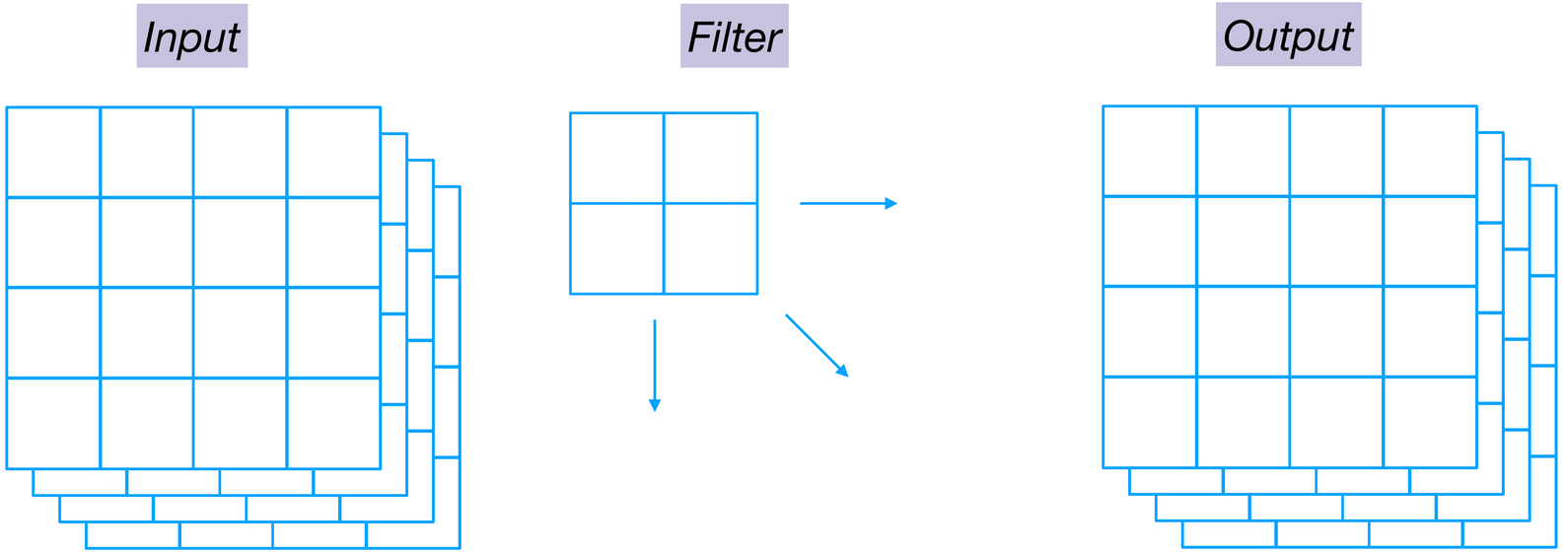
  - then last mode of output only has length 1
  - if desired we can drop this mode, reducing the number of modes of the output

Pointwise convolution



- Even specialer case: filter is $1 \times 1 \times \ldots \times 1 \times$ depth
  - keep all but one mode the same shape, lose last mode
- Called ***pointwise*** or (for images) 1×1 convolution
  - same as a weighted sum of slices / channels
- Makes a lot of sense if last mode is channel index: e.g., collapse RGB channels to get overall image intensity
  - common design: multi-head convolution adds a channel index, pointwise later removes it

Same filter on each channel

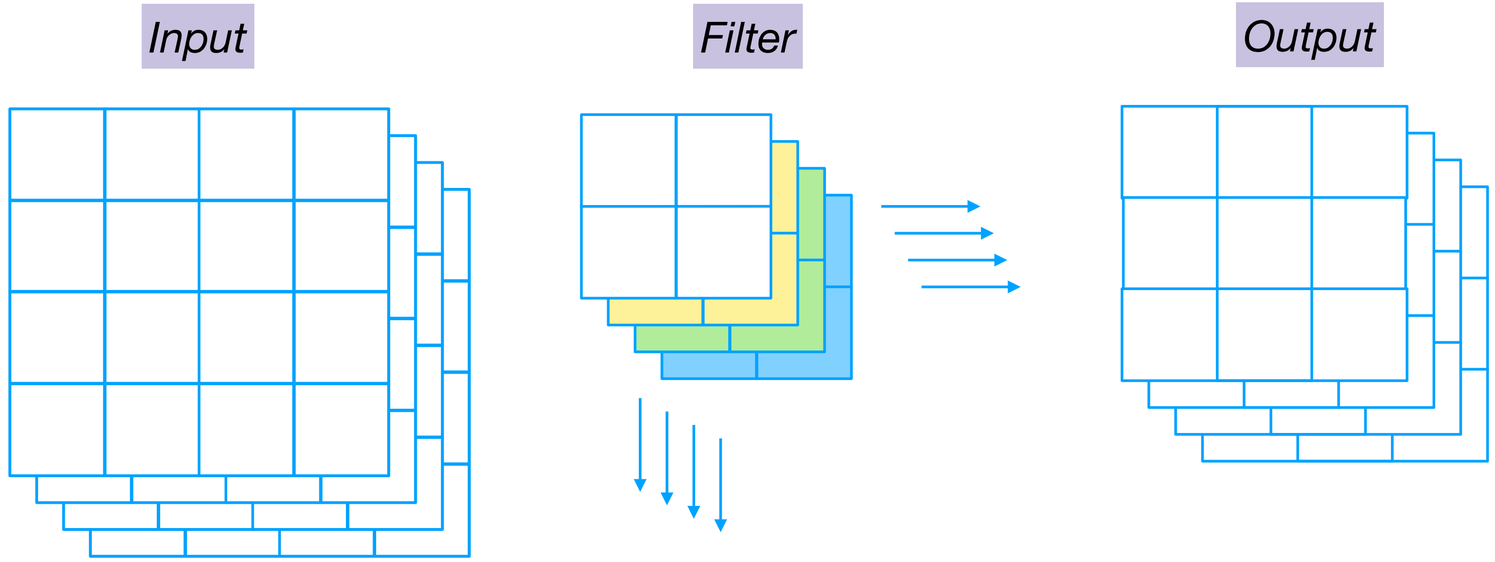


- If last mode is channel index, and if filter has just 1 channel, then full convolution applies the same filter independently to each channel
  - e.g., blur RGB channels independently by applying a filter like

$$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

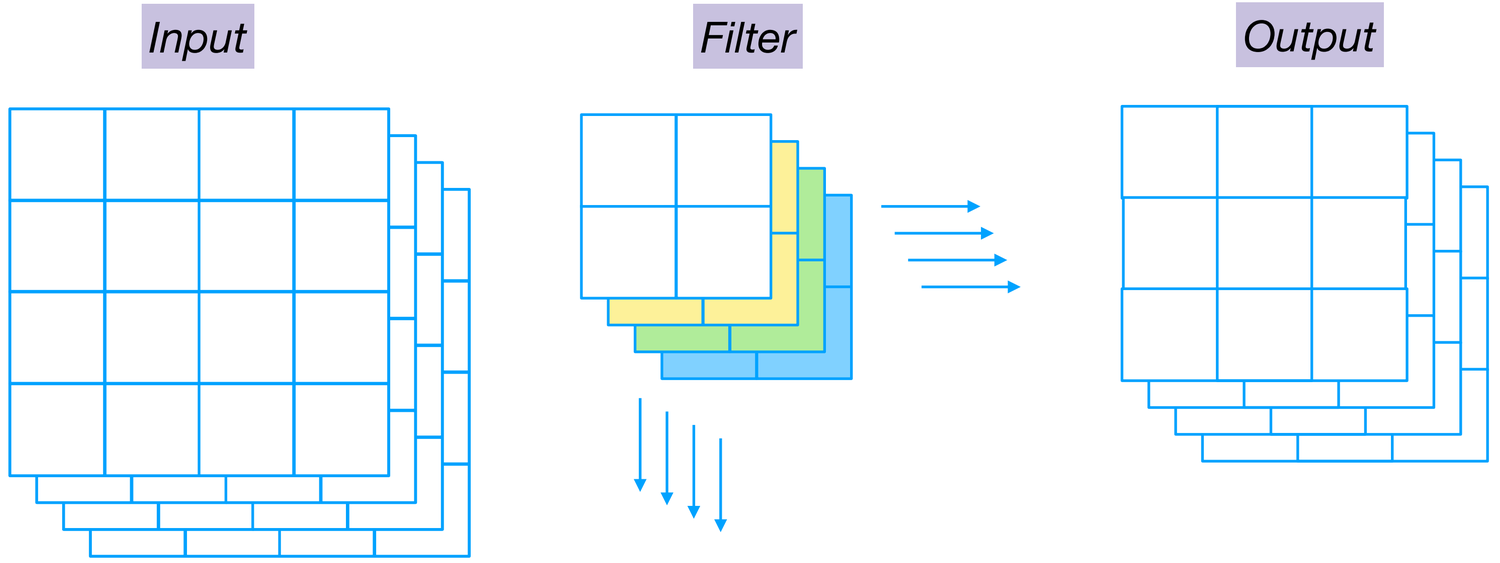
    to each one

Layerwise convolution

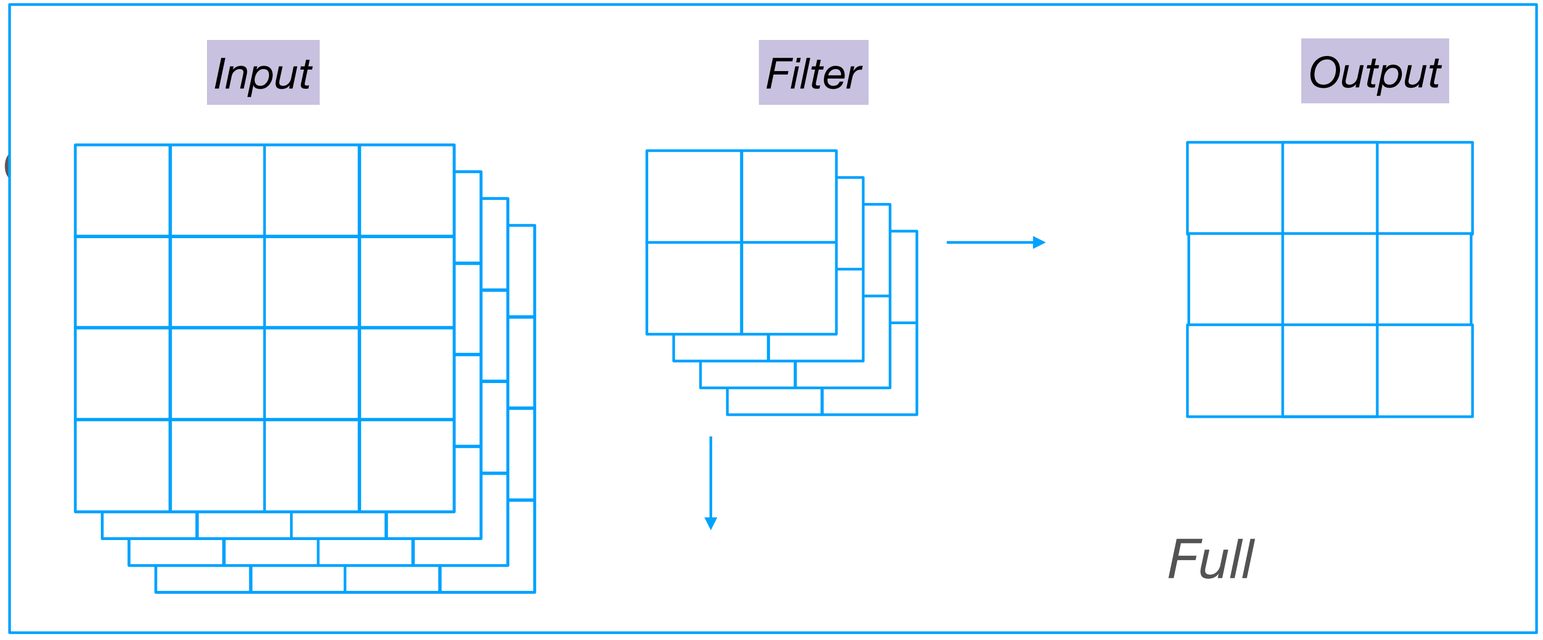


- If last mode is channel index, we might also want to filter each channel separately
  - filter has same number of channels as input
  - this is *different* from a full convolution

Layerwise convolution

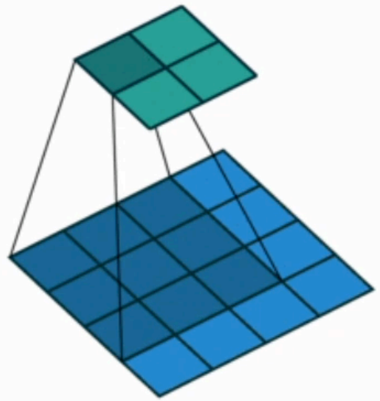


Layerwise

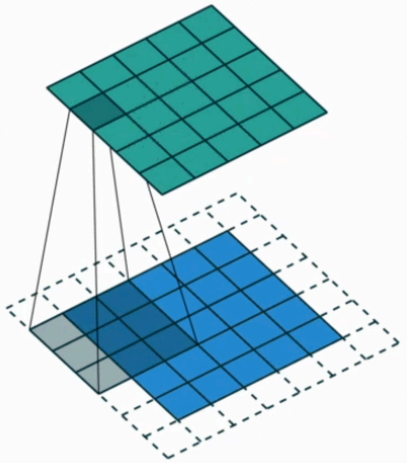


Visualizations

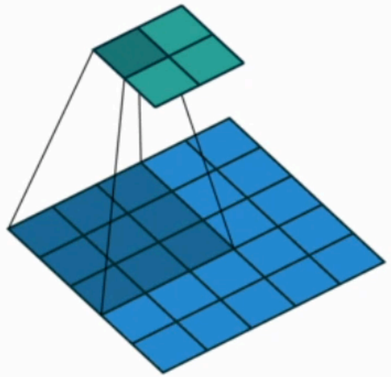
padding = 0, stride = 1



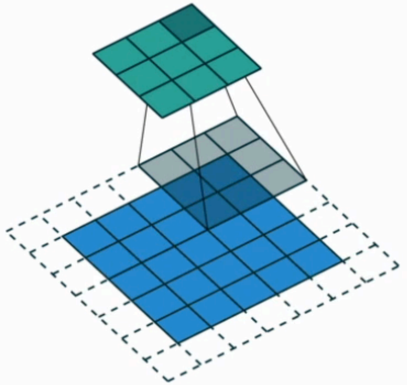
padding = 1, stride = 1



padding = 0, stride = 2

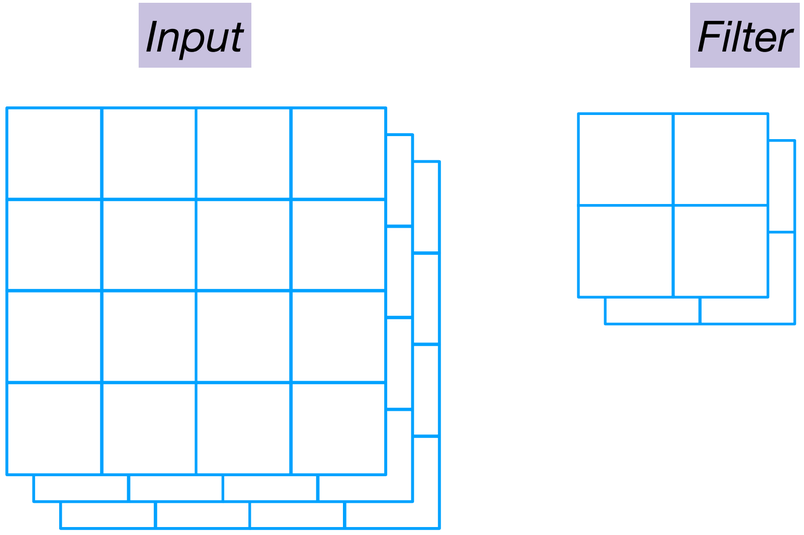


padding = 1, stride = 2



link to animations

Calculating sizes



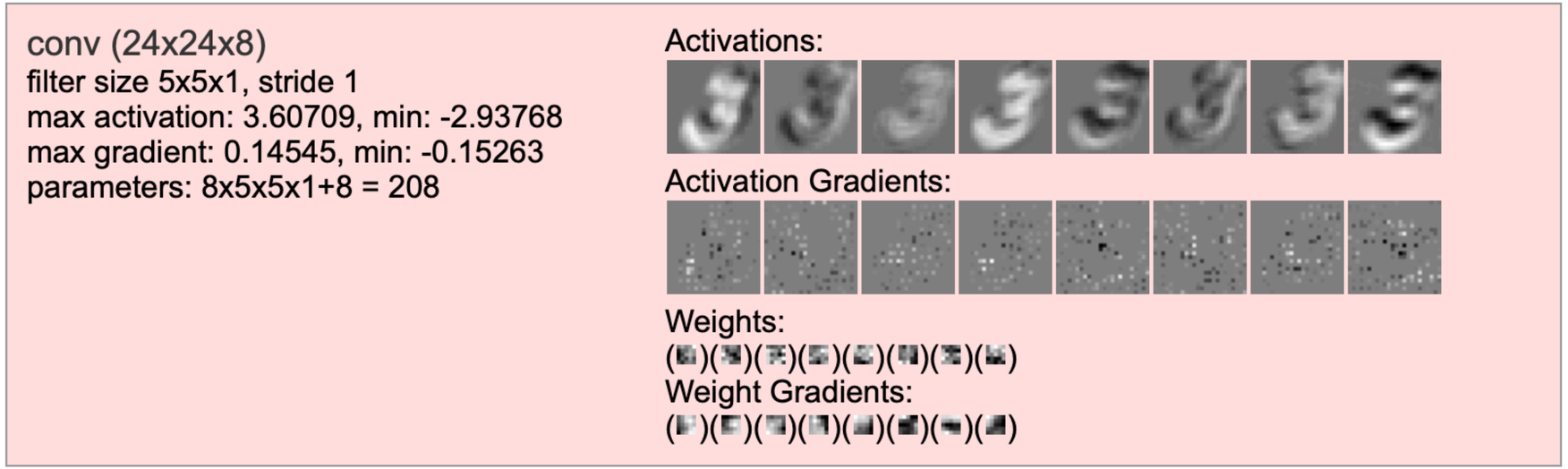
- With all the options, it can be tricky to figure out what size the output of a convolution will be
  - e.g., given $4 \times 4 \times 3$ input and $2 \times 2 \times 2$ filter
  - what size is output for a default (full, valid) convolution?
  - what if we zero-pad the last (depth) mode by 1 at end?
  - what if we use stride 2, 2, 1 (with no padding)?

Putting it together

- Convolution and pooling work well together: tunable between equivariance and invariance
- Often do convolutions alternating with nonlinearities, occasional pooling to reduce size of a layer, occasional multiple heads or pointwise convolutions to change number of modes of tensor
- Sometimes increase number of channels as we downsample (else we force the net to discard a lot of information)
- Can use vec or unvec to glue between convolutional layers and plain MLP layers
- Add in residual connections to help gradients not explode or vanish

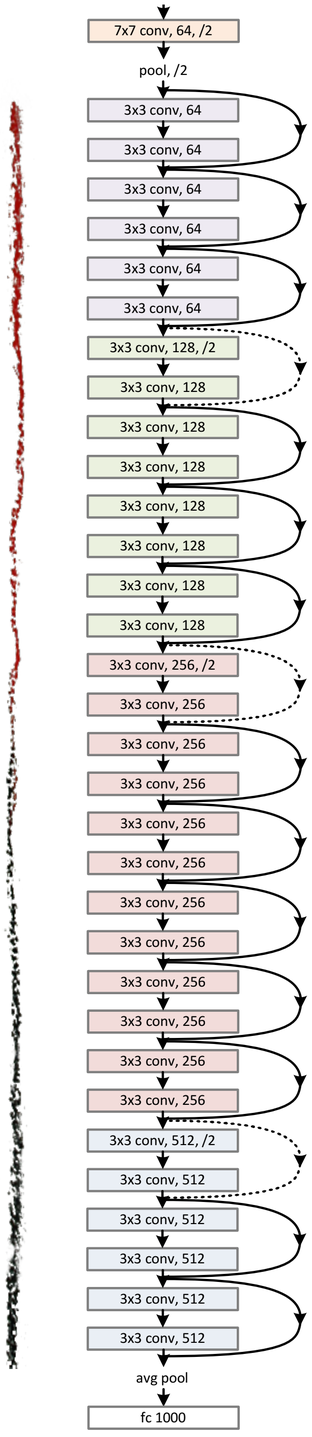
Example

https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

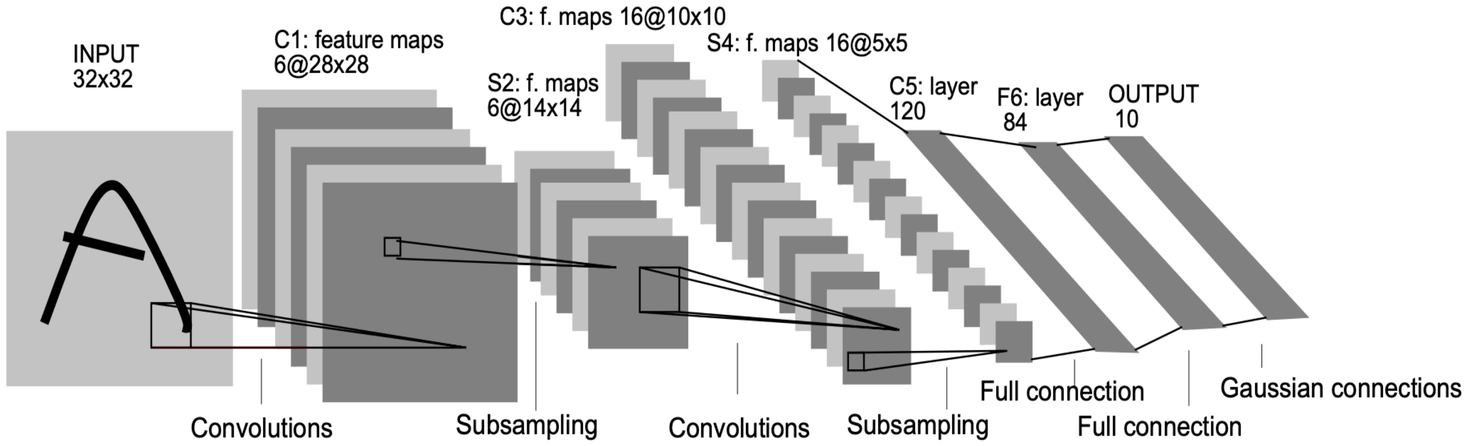


- CNN for MNIST, trained by Javascript in browser
- 2 × (convolution + ReLU + max-pool) + FC + softmax
- Enough for 99% validation accuracy

Examples



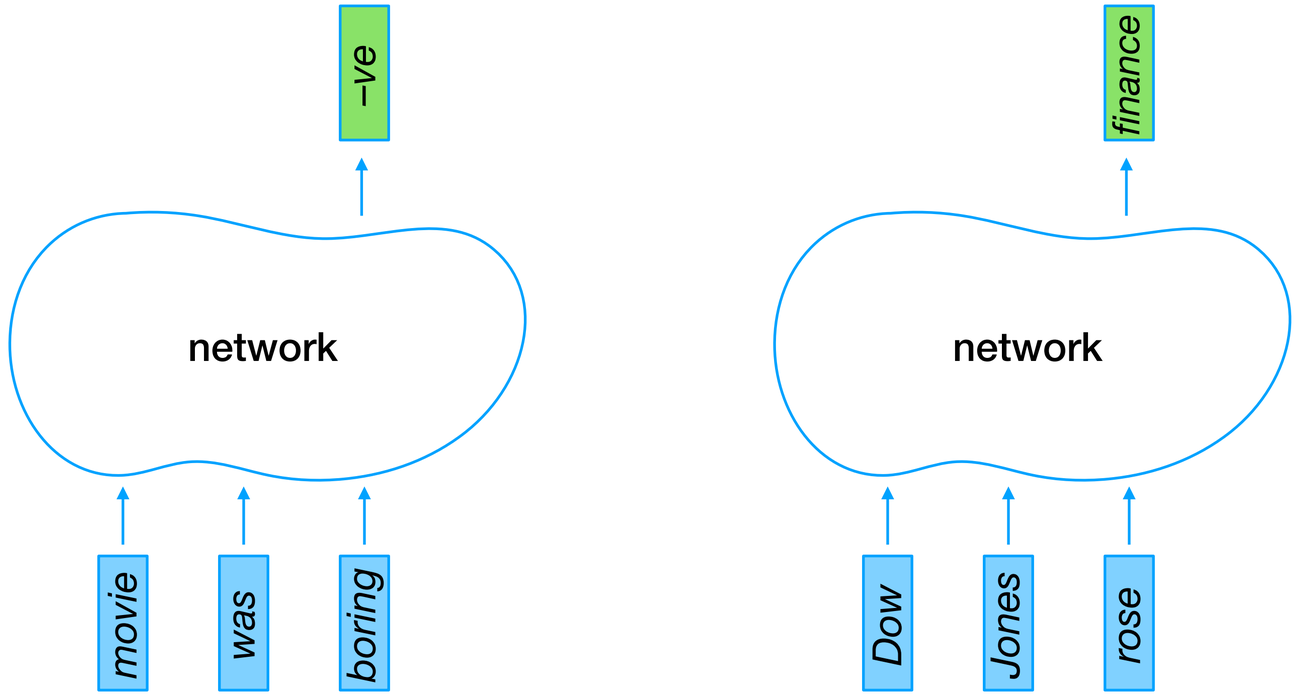
[He et al., 2015]



[LeCun et al., 1998]

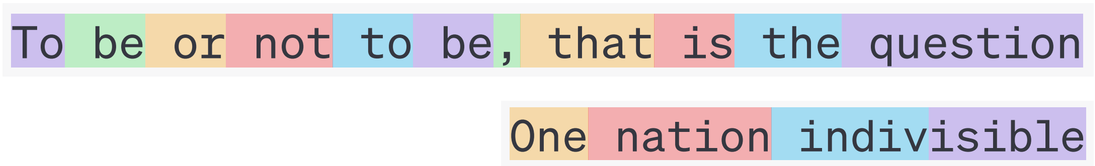
- These features have appeared in *many* famous models (SOTA at their time)
  - e.g., LeNet (convolution and pooling, for MNIST digit classification)
  - e.g., ResNet (convolution, pooling, batch norm, residual connections, for ImageNet)

Sequence prediction



- Models up to now have *fixed-shape* inputs
- Many language tasks (like sentiment classification, topic classification) take *variable-length* sequence as input
- Need models that can scale to fit

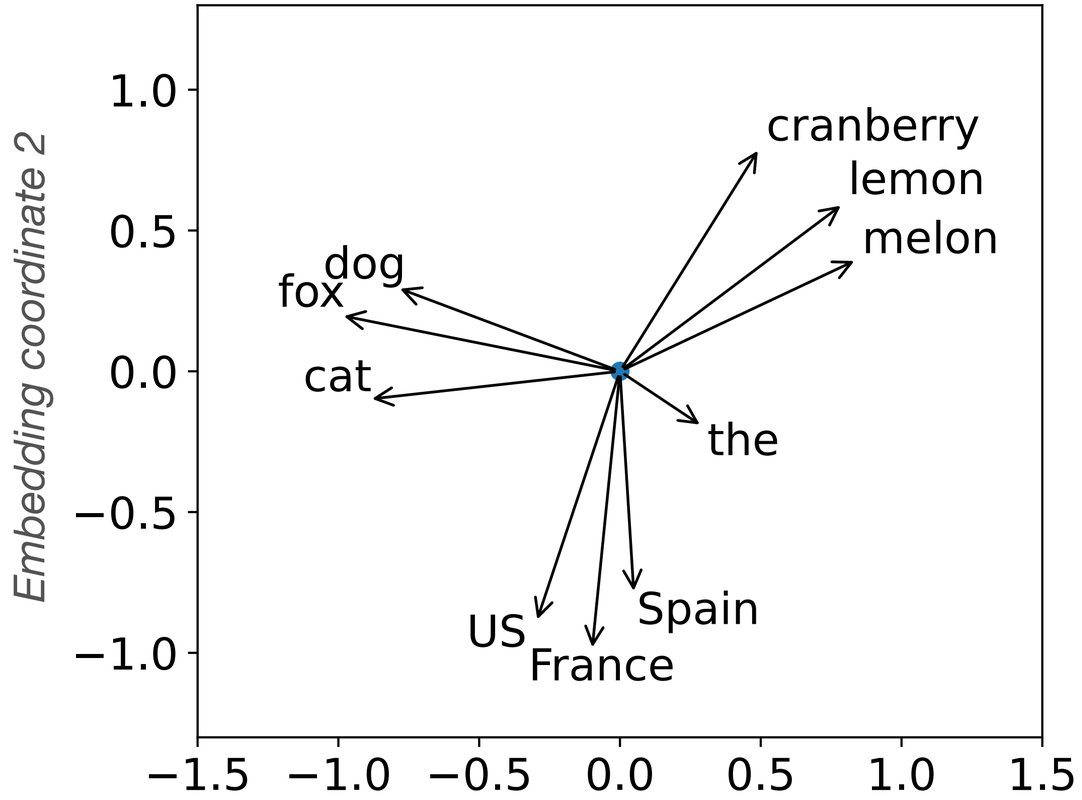
Discrete-valued sequences



credit: OpenAI tokenizer

- Another way language differs: ***discrete*** tokens instead of continuous pixel values
  - characters
  - words
  - word parts
  - class labels — potentially 1000s with relationships among them

Discrete-valued sequences



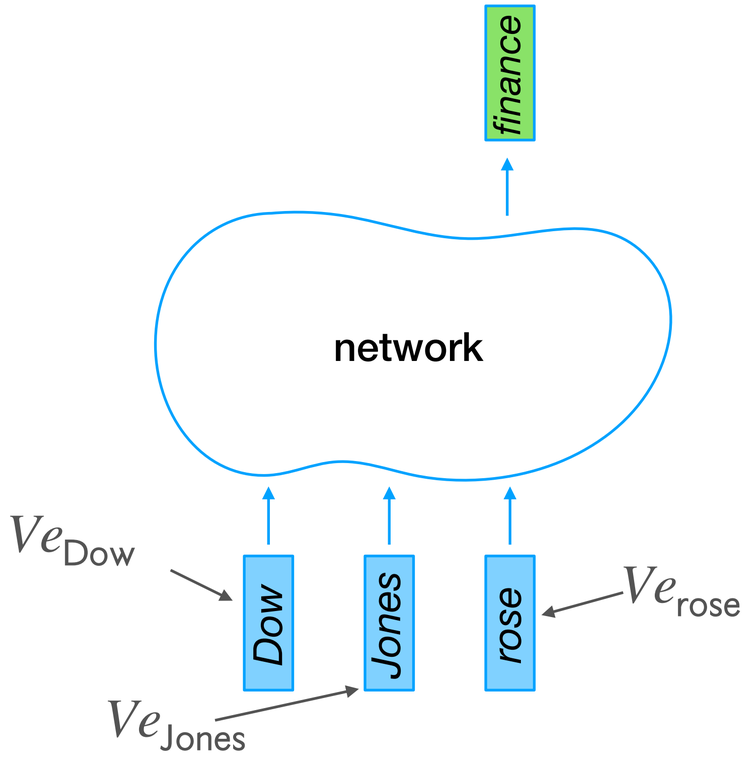
- Handle tokens by *embedding*: replace by vector $\in \mathbb{R}^d$ (d might be 512, 2048, 10k — large but $\ll$ vocabulary size)
  - ideally: relationships among tokens reflected in vectors
  - dog and cat embedded near each other
  - Paris – France + Spain $\approx$ Madrid

Embedding

- If each raw token = one-hot vector, and each embedded token = arbitrary vector, this is a ***linear map***
- $\mathbb{R}^{\text{vocabulary size}} \to \mathbb{R}^{\text{embedding dimension}}$
- Embedding of “dog” = Ve_{dog}

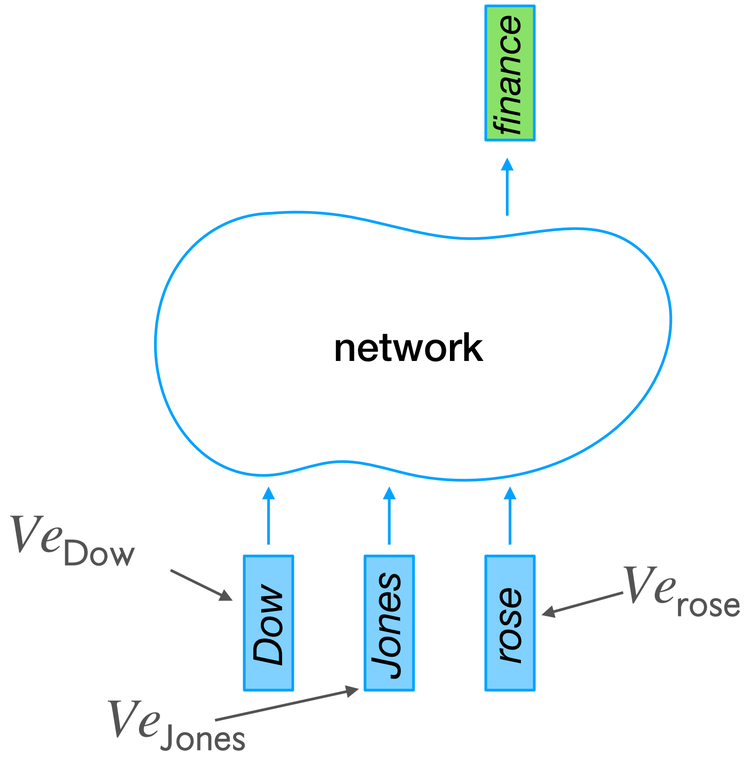
$$V = \left| \left| \left| \left| \quad \cdots \right. \right. \right. \right.$$

Sequences of tokens



- Putting it together:
  - first we split the input sequence into tokens
  - then we embed each token
  - and pass the tokens to the model one by one
  - finally read off prediction at the output node

Sequences of tokens



$W^\top e_{\text{lbl\_finance}}$

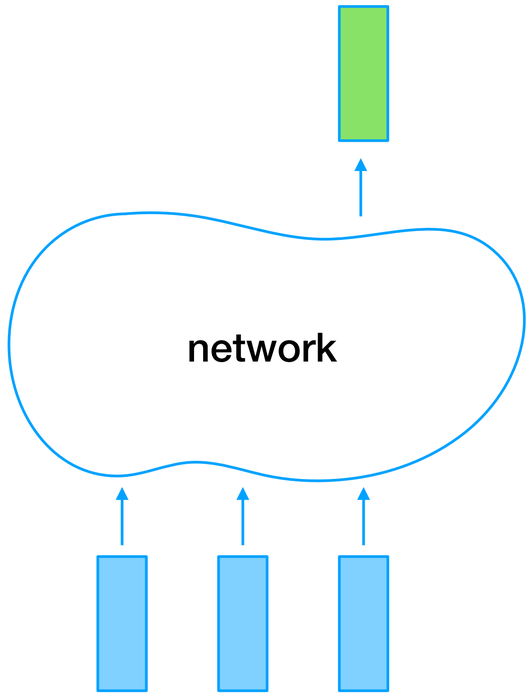
$W^\top e_{\text{lbl\_sports}}$

$W^\top e_{\text{lbl\_celebrities}}$

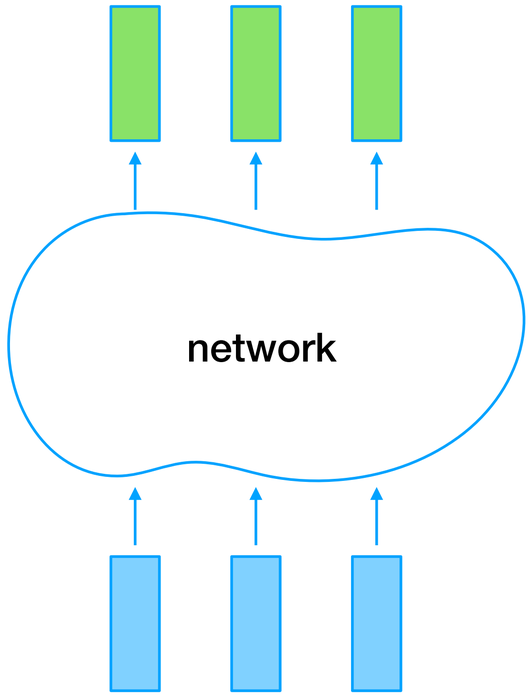
...

- Can even interpret class labels as tokens with embeddings
  - if penultimate layer activations are z
  - and last layer is $\text{softmax}(Wz)$
  - then we can think of ith row of W as embedding of ith label
  - and we put highest probability on the label whose embedding has highest dot product with z

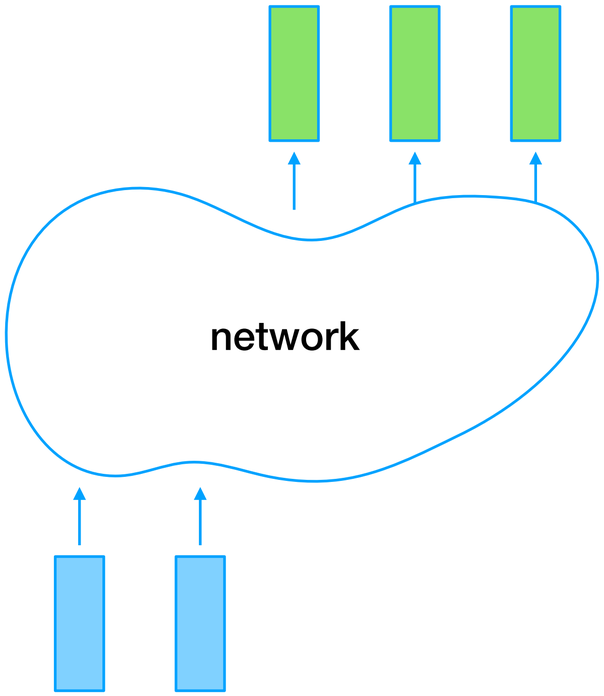
Sequence prediction tasks



many → one
classification, regression
(e.g., sentiment analysis)



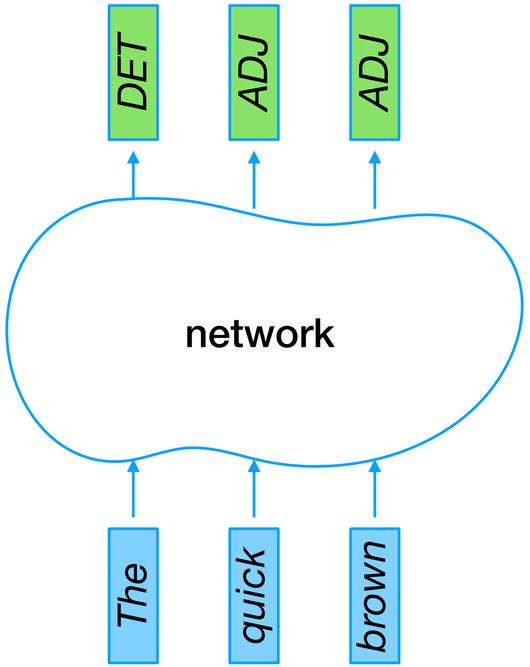
many → many (corresp.)
transduction
(e.g., POS tagging)



many → many
generation
(e.g., language model)

Part of speech tagging

- Example application: POS tagging
- Map each word (in context) to its part of speech (determiner, adjective, noun, verb, …)
- A transduction task: output tokens are matched with corresponding input tokens

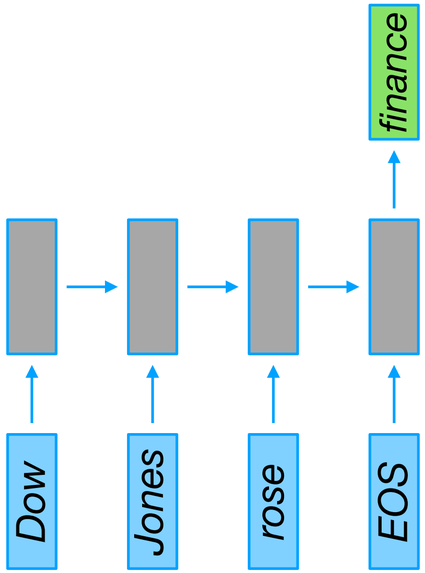


Another transduction task: OCR

- Could try to recognize each character independently, but accuracy will be higher if we use context

figure credit: (Chatzis & Demiris, 2013)

RNN for many:1 tasks



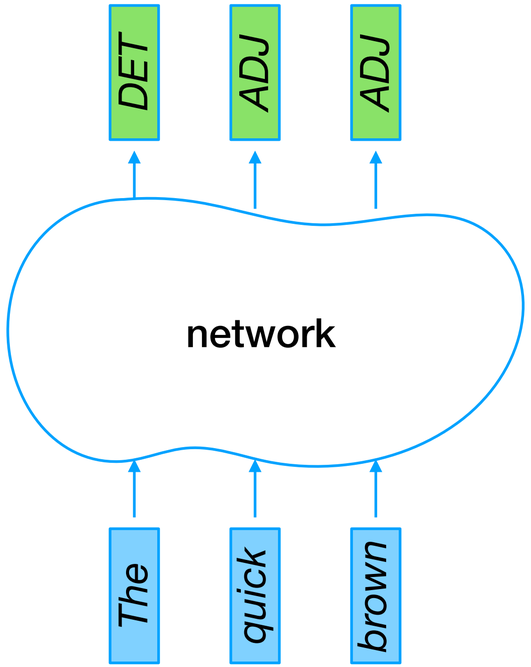
- To handle many:1 tasks like classification, most common setup is to end with a special token like EOS (end-of-sentence), predict label based on corresponding hidden vector

Recurrent neural network (RNN)

- Oldest sequence model
- Replicate a common network block for each step
  - input x_t, hidden state z_t, output y_t
- Connect hidden units at each step as inputs to next
  - initialize hidden state to zero before first step

$$z_0 = 0$$

$$z_t = \sigma(Vz_{t-1} + Wx_t + b) \quad t = 1..T$$

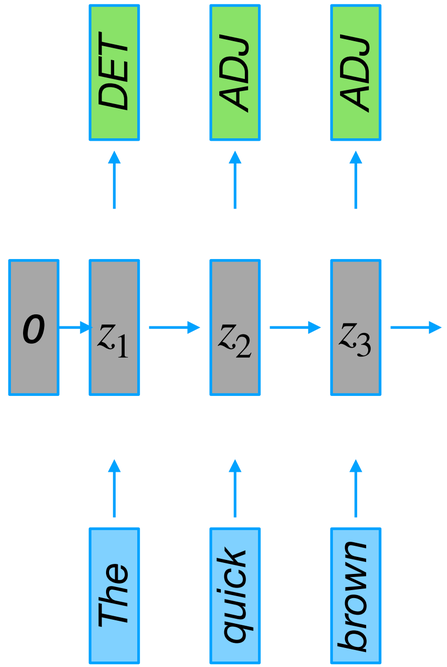


Recurrent neural network (RNN)

- Oldest sequence model
- Replicate a common network block for each step
  - input x_t, hidden state z_t, output y_t
- Connect hidden units at each step as inputs to next
  - initialize hidden state to zero before first step

$$z_0 = 0$$

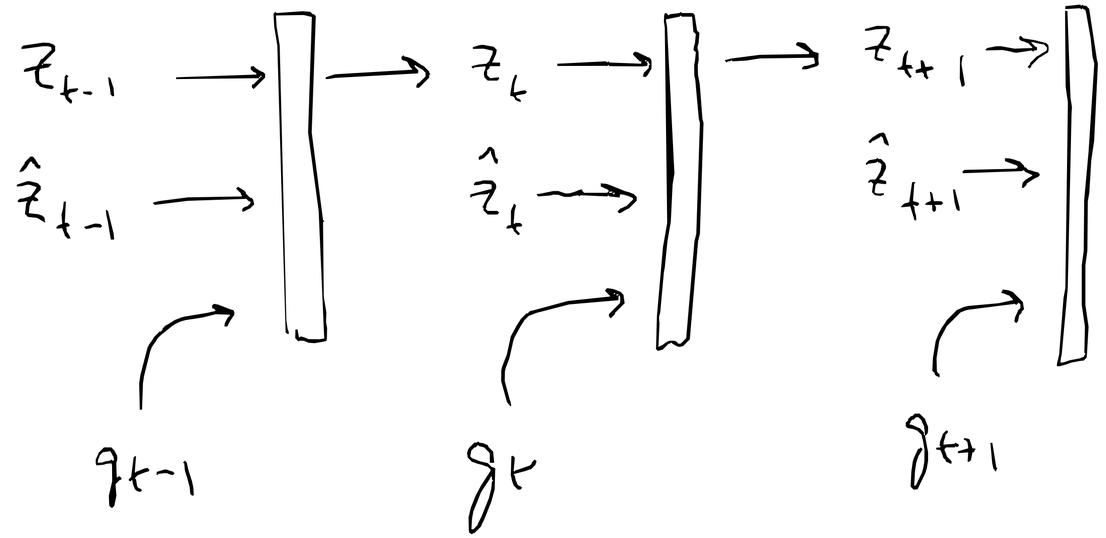
$$z_t = \sigma(Vz_{t-1} + Wx_t + b) \quad t = 1..T$$



Training RNNs

- Once we unroll it, RNN is just like any other network
  - forward pass from $t = 1..T$
  - backward pass from $t = T..1$ (*backprop through time)*
  - SGD step
- Not much difference from deep feedforward net
  - main one: same parameters V, W, b used at every step
  - autodiff handles parameter sharing automatically
  - or we can manually sum gradients (total derivative rule)
- Vanishing / exploding gradients!
  - same tools as above: initialize well, use residual connections — and …

Memory



minimal gated unit (MGU)

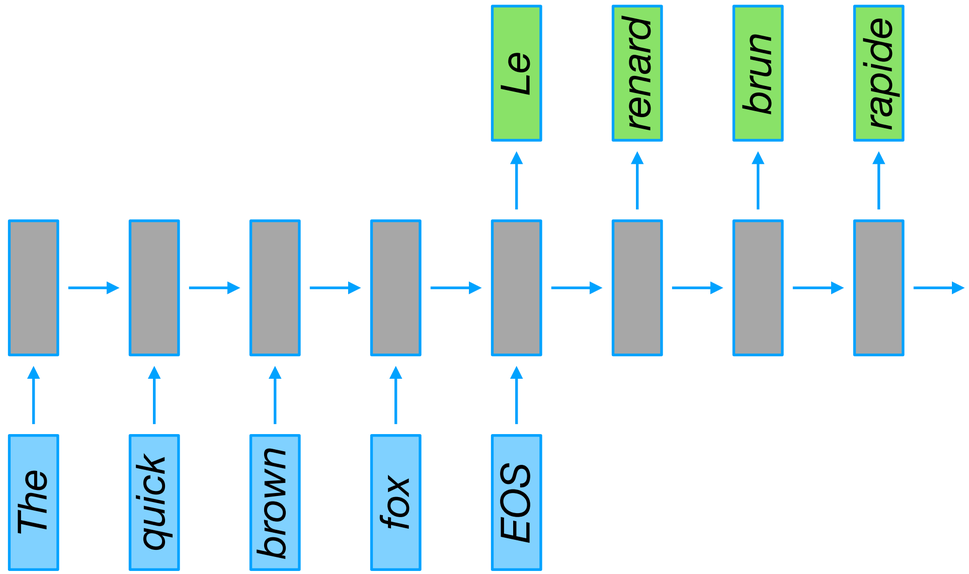
gating: $z_{t+1} = g_t \circ z_t + (1 - g_t) \circ \hat{z}_t$

both gate g_t *and candidate update* $\hat{z}_t$ *are of form* $f(Vz_{t-1} + Wx_t + b)$

output y_t *could depend on* z_t, x_t

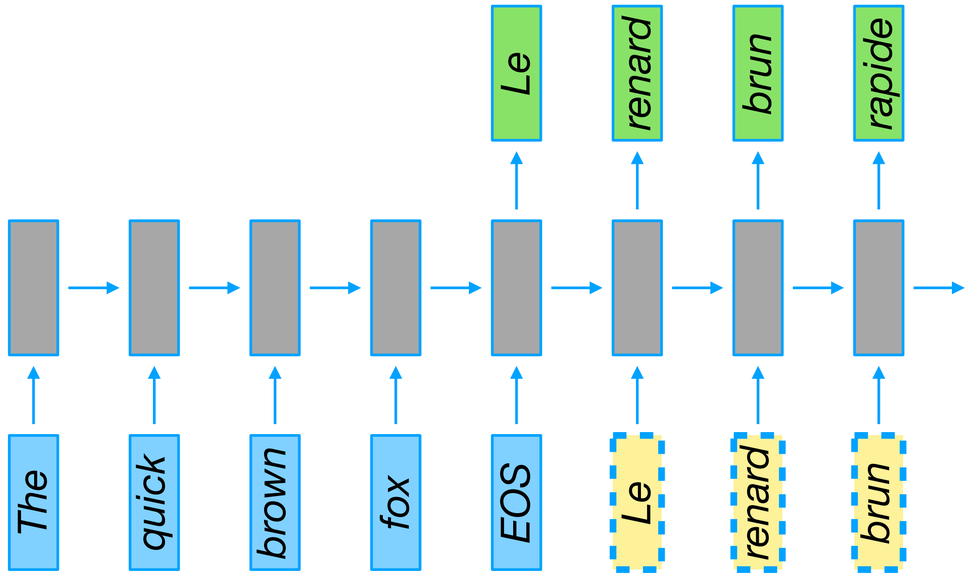
- Use *gating* to update hidden state
  - differentiable decision to store into z_t
  - like a controllable residual connection that lets us connect storage to retrieval
- Examples: LSTM, GRU, MGU

RNN generation



- To generate an arbitrary-length output, we can use RNN as a generative model
- Process input tokens (the *prompt*), mark end w/ special token, or pass a flag to start generating
- Copy outputs back to next step's input to make it easier to generate coherent sequences

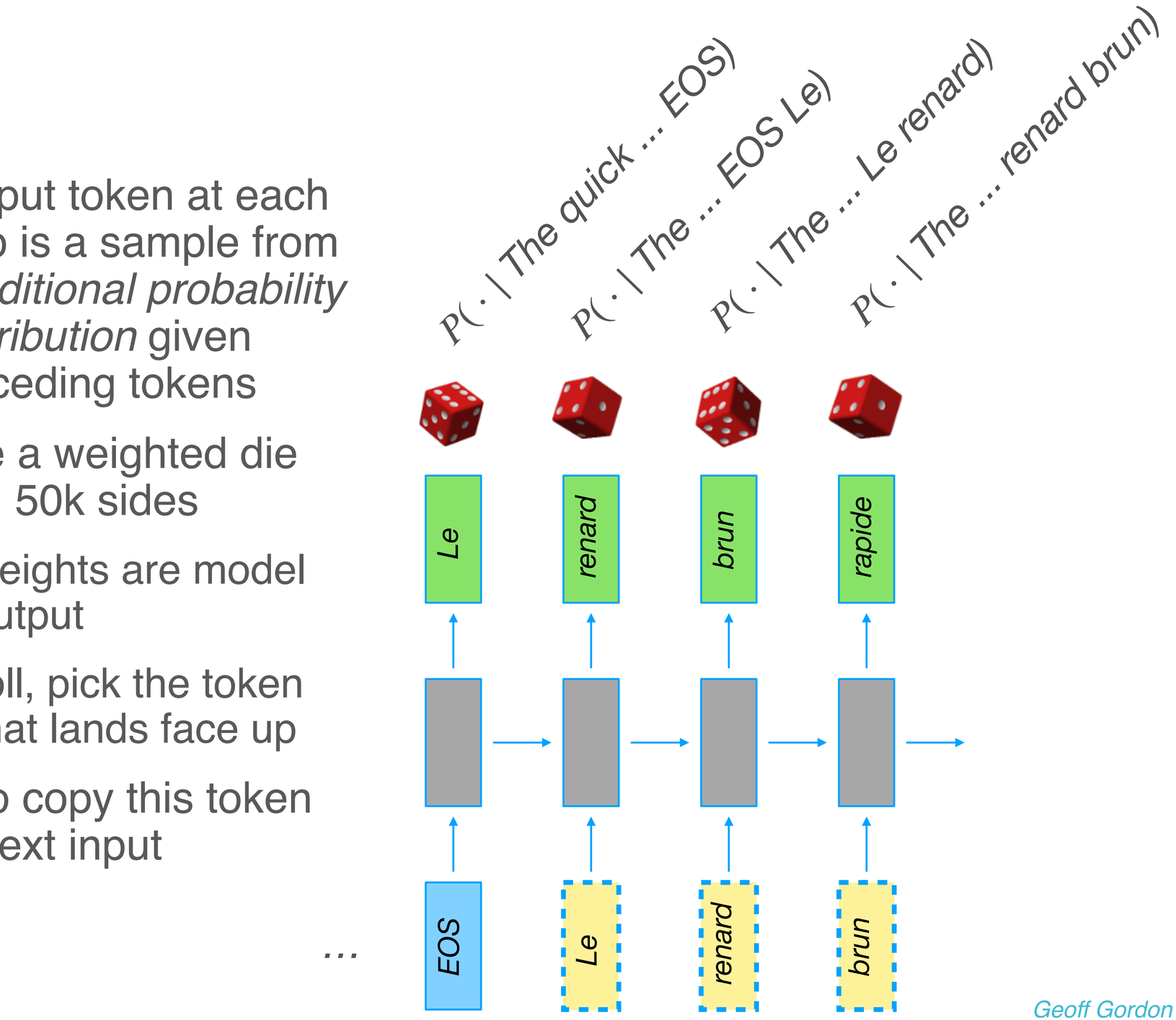
RNN generation



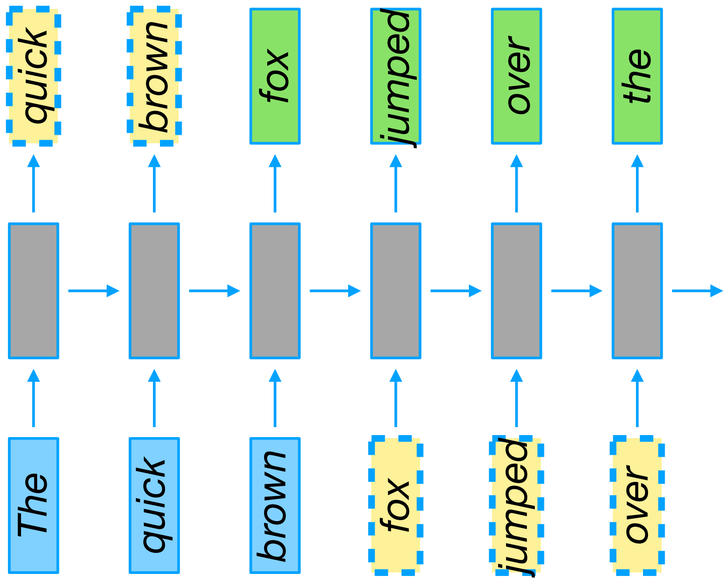
- To generate an arbitrary-length output, we can use RNN as a generative model
- Process input tokens (the *prompt*), mark end w/ special token, or pass a flag to start generating
- Copy outputs back to next step's input to make it easier to generate coherent sequences

Generating from a language model

- Output token at each step is a sample from *conditional probability distribution* given preceding tokens
- Like a weighted die with 50k sides
  - weights are model output
  - roll, pick the token that lands face up
- Also copy this token to next input



RNN generative model for sequence completion



- If outputs are shifted inputs, called *next token prediction*
  - results in a model that can generate a new sequence by repeatedly generating next token
- When we process inputs, outputs are training targets
- During prediction, copy outputs back to next step's input

Sampling from an RNN-LM

Training data (Shakespeare's As You Like It)

VIOLA: Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR: O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.

Sampling from an RNN-LM

Sample (RNN-LM)

CHARLES: Marry, do I, sir; and I came to acquaint you with a matter. I am given, sir, secretly to understand that your younger brother Orlando hath a disposition to come in disguised against me to try a fall. To-morrow, sir, I wrestle for my credit; and he that escapes me without some broken limb shall acquit him well. Your brother is but young and tender; and, for your love, I would be loath to foil him, as I must, for my own honour, if he come in: therefore, out of my love to you, I came hither to acquaint you withal, that either you might stay him from his intendment or brook such disgrace well as he shall run into, in that it is a thing of his own search and altogether against my will.

TOUCHSTONE: For my part, I had rather bear with you than bear you; yet I should bear no cross if I did bear you, for I think you have no money in your purse.

Example from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Sampling from an RNN-LM

Sample (RNN-LM)

CHARLES: Marry, do I, sir; and I came to acquaint you with a matter. I am given, sir, secretly to understand that your younger brother Orlando hath a disposition to come in disguised against me to try a fall. To-n… capes me without s… her is but young and… il him, as I must, for… y love to you, I car… night stay him from… shall run into, in th… gainst my

… actually, this is the training data, and the previous one was the sample

TOUCHSTONE: For my part, I had rather bear with you than bear you; yet I should bear no cross if I did bear you, for I think you have no money in your purse.

Example from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Word2vec

masked token prediction

The quick brown fox [???] over the lazy dog

- Early successful embedding learner
- Only parameters: embedding matrix V
- Train: masked token prediction
  - other, later models: mask more than 1 token
- Two layer network:
  - first layer: embed and sum tokens near mask index i

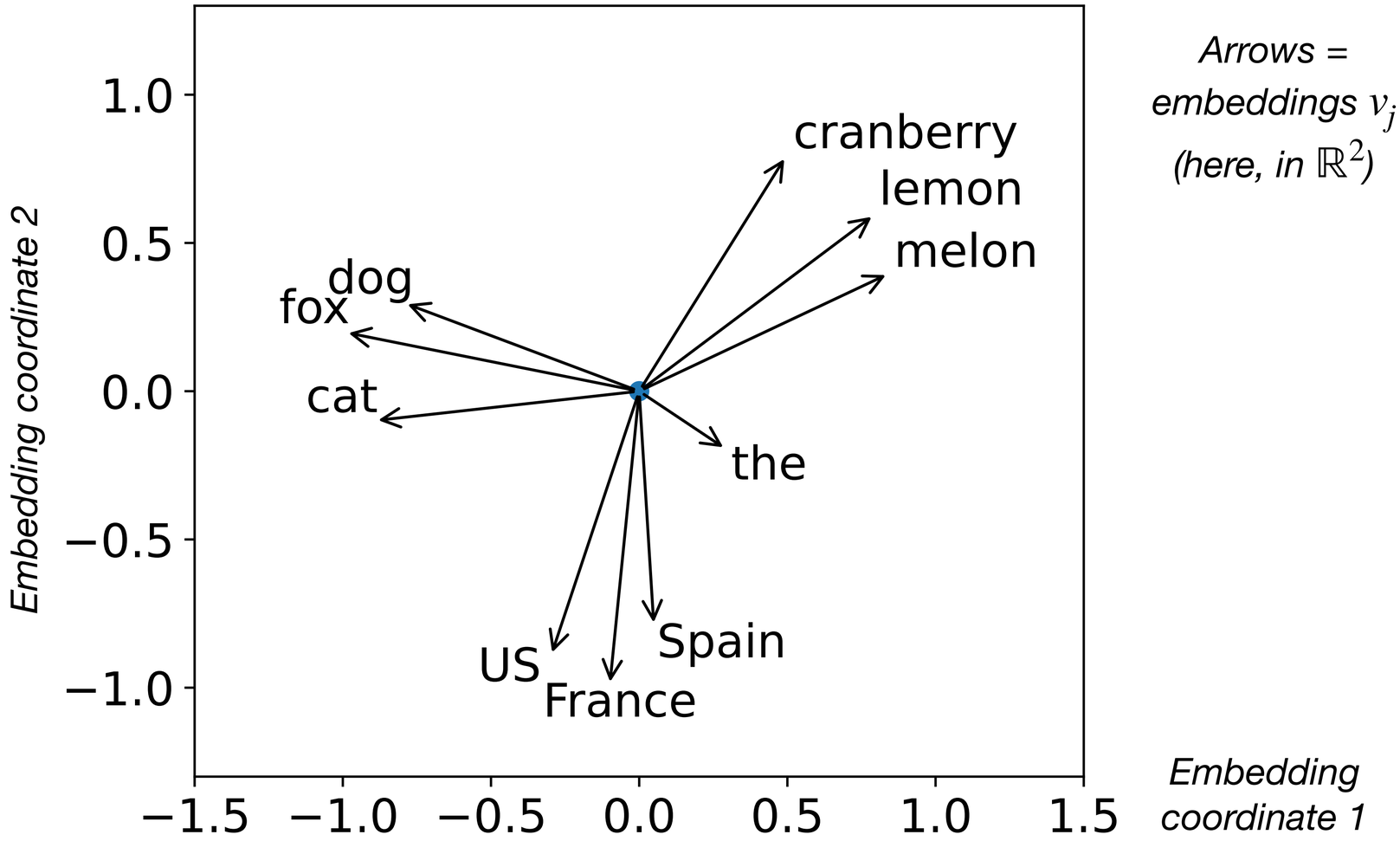
$$z = \sum_{j=i-k,\, j\neq 0}^{i+k} Vx_j$$

  - second layer: search over vocabulary

$$P(y \mid \text{input}) = \frac{\exp(z \cdot Vy)}{\sum_{y'} \exp(z \cdot Vy')}$$

  - note: approximations to make it fast

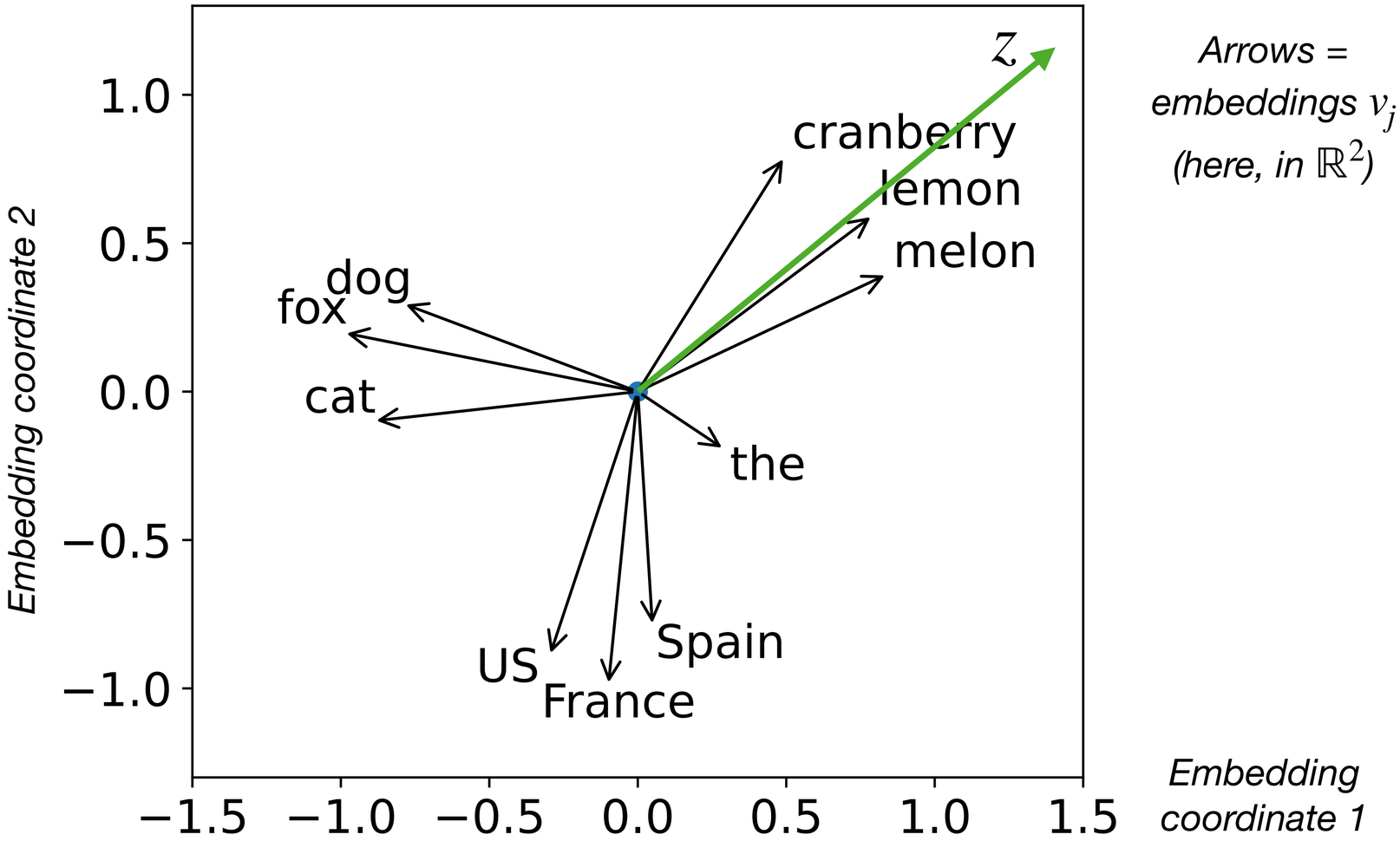
First layer



prediction task: … cranberry [???] melon …

- z = average of nearby tokens · window length

First layer

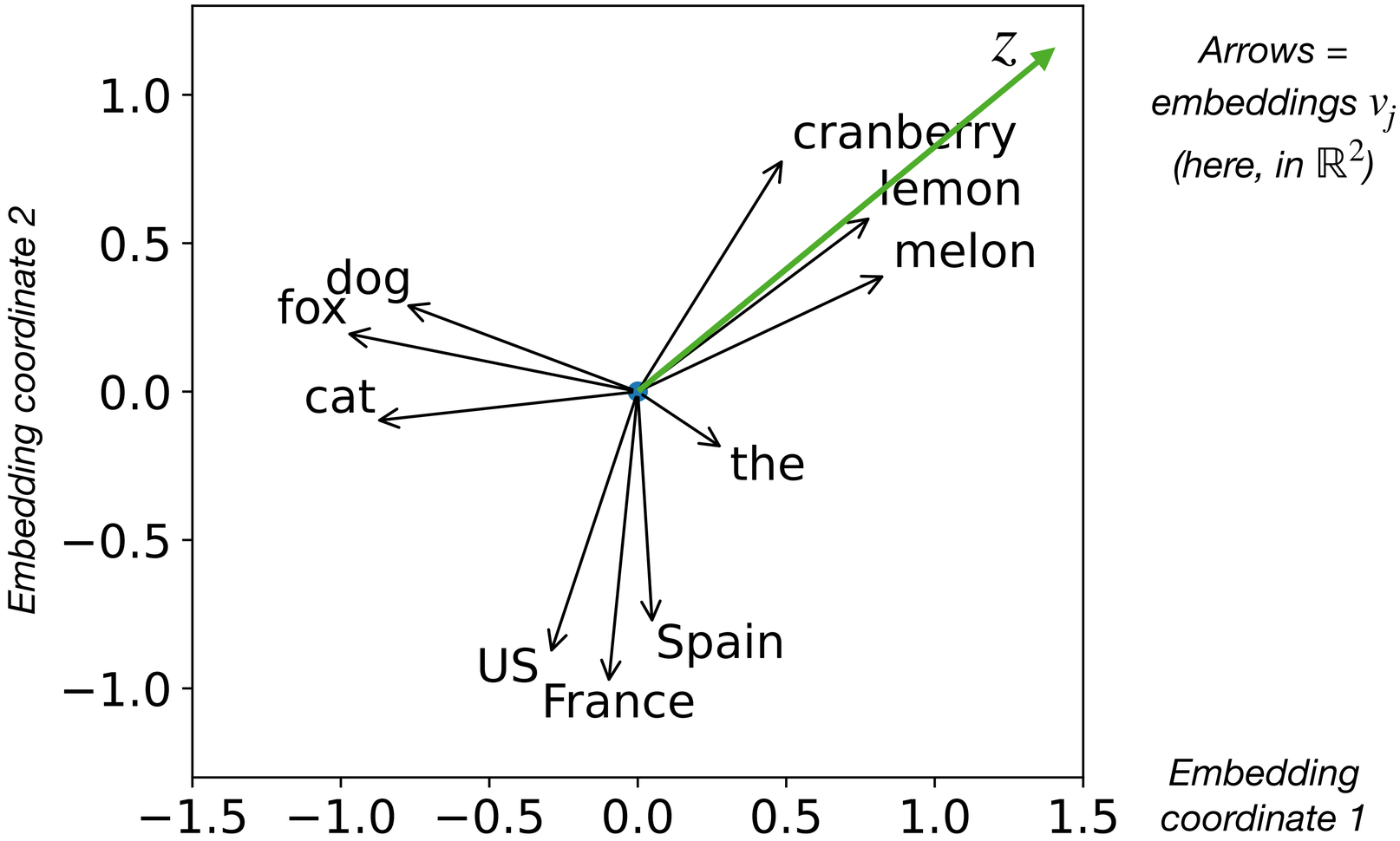


Arrows = embeddings v_j (here, in $\mathbb{R}^2$)

prediction task: … cranberry [???] melon …

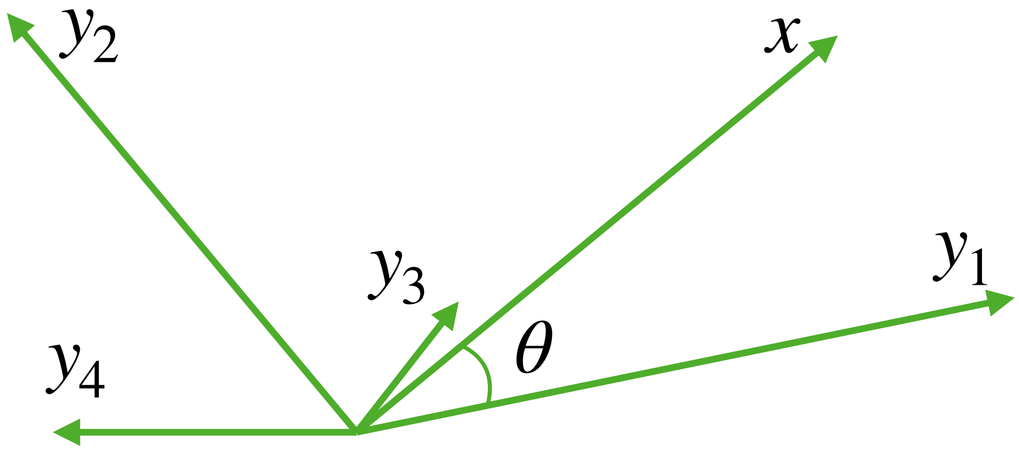
- z = average of nearby tokens · window length

Second layer



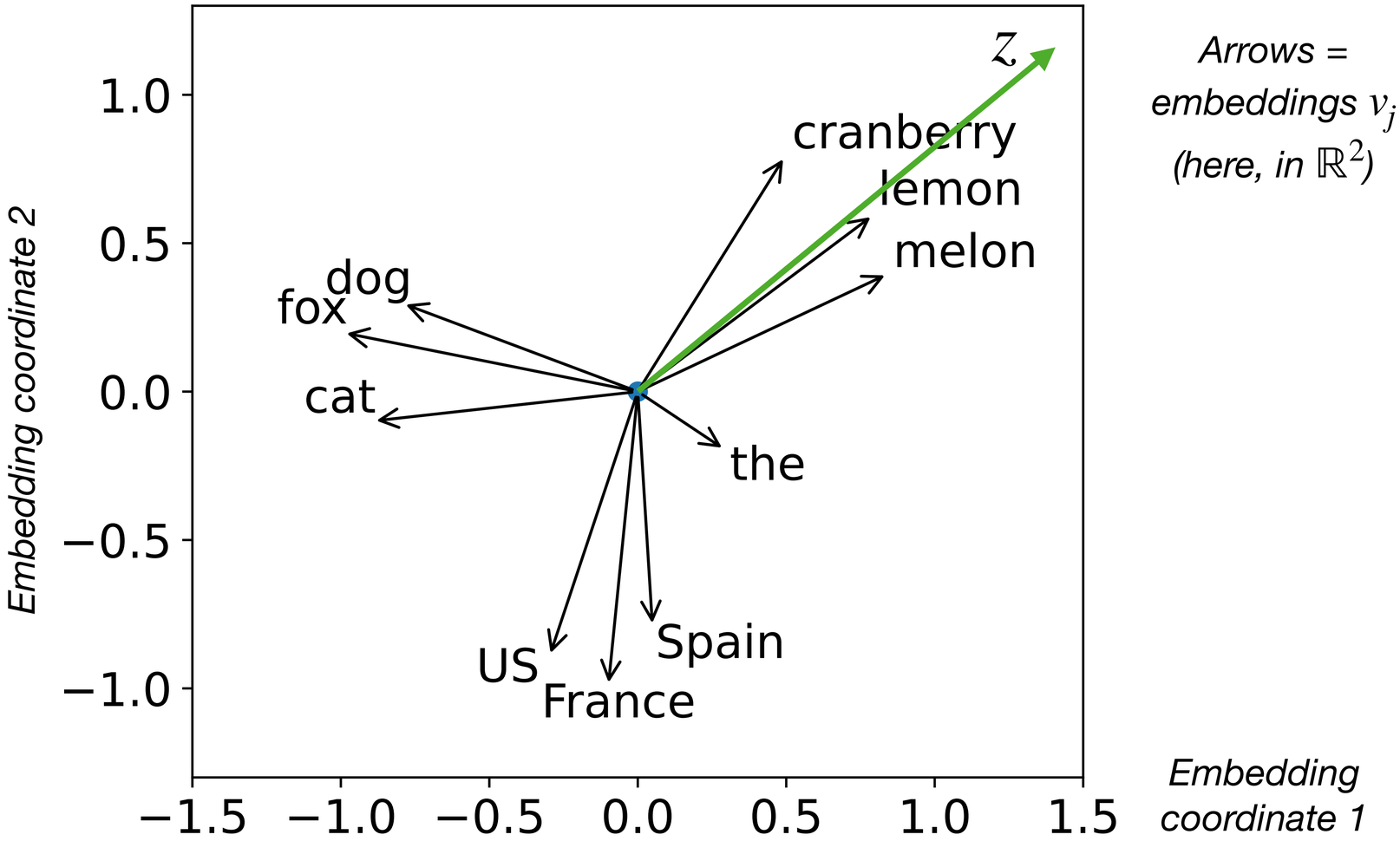
- Input to second layer = a vector z, called the *query*
- Goal of second layer: search through a set of vectors (*keys* — here, vocabulary) to find similar ones to query

Reminder: dot product



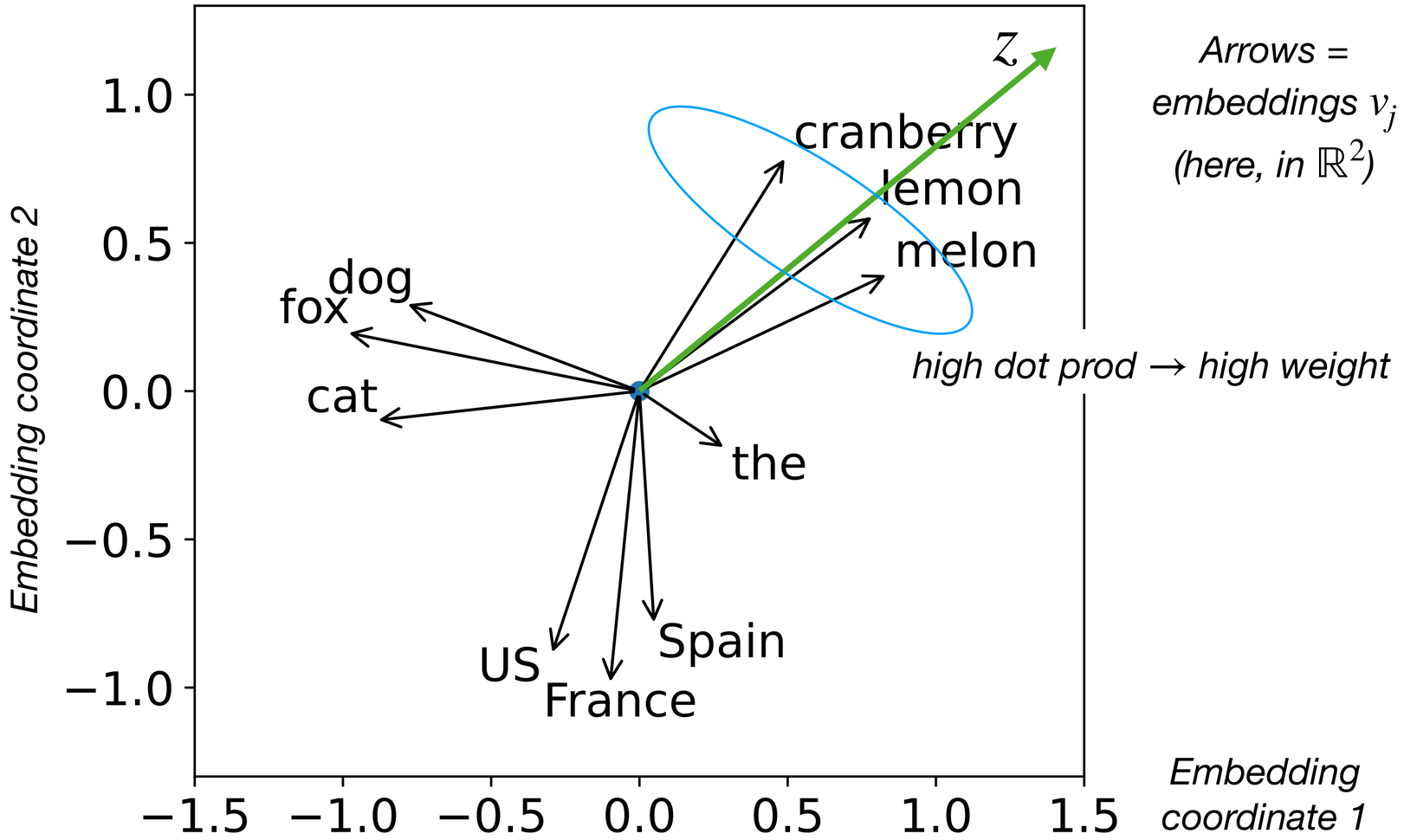
- Dot product $x \cdot y$ depends on both *norm* and *direction* of input vectors x and y
  - $x \cdot y = \|x\| \, \|y\| \cos\theta$
- Large dot product means
  - x and y point in in approximately the *same direction*
  - x and y both have *large norm*

Second layer



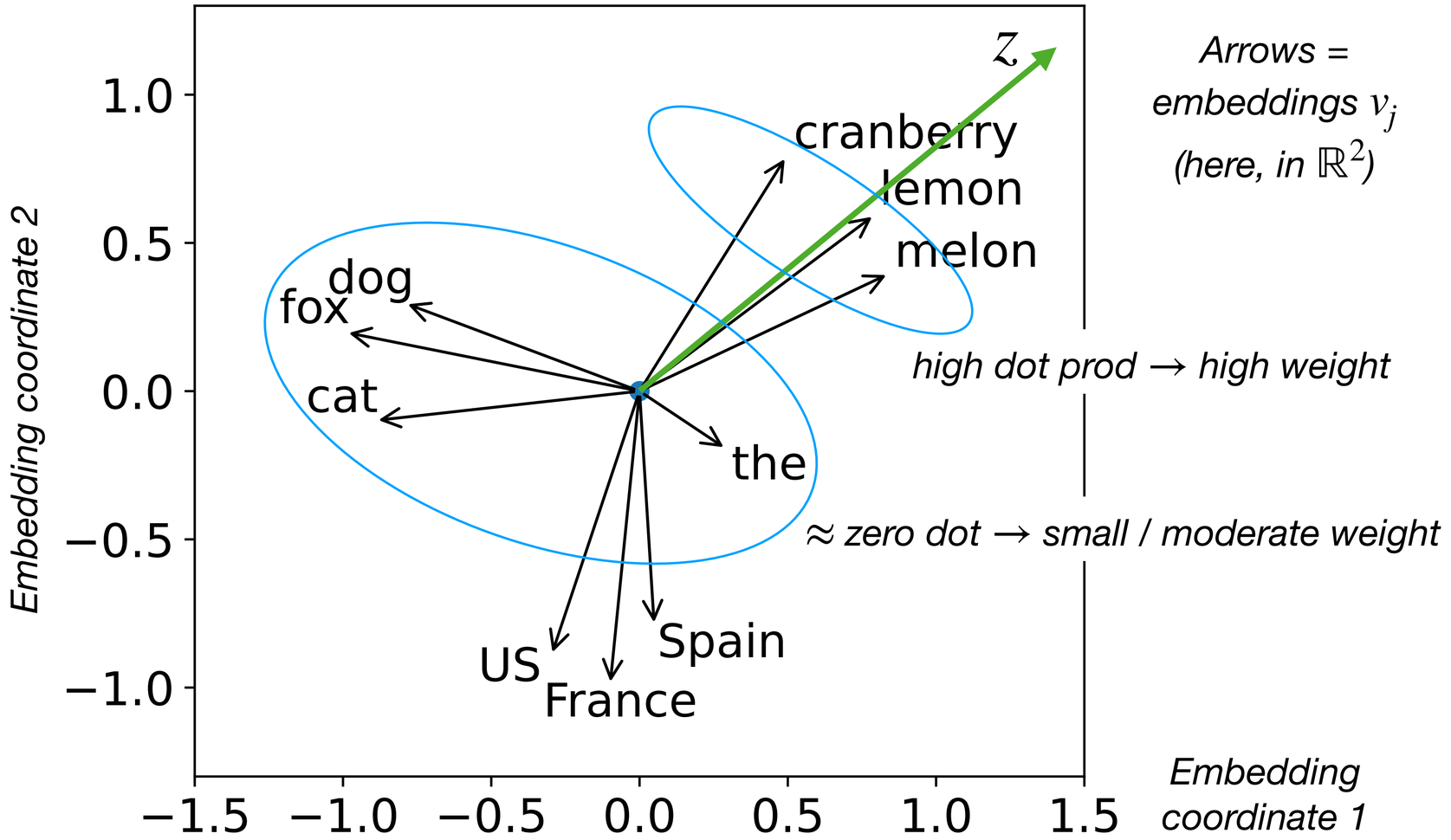
- Dot product $v_j \cdot z$ is highest for token embeddings that are both *high norm* and *similar direction* to z
- The $\exp$ in softmax further emphasizes highest values

Second layer



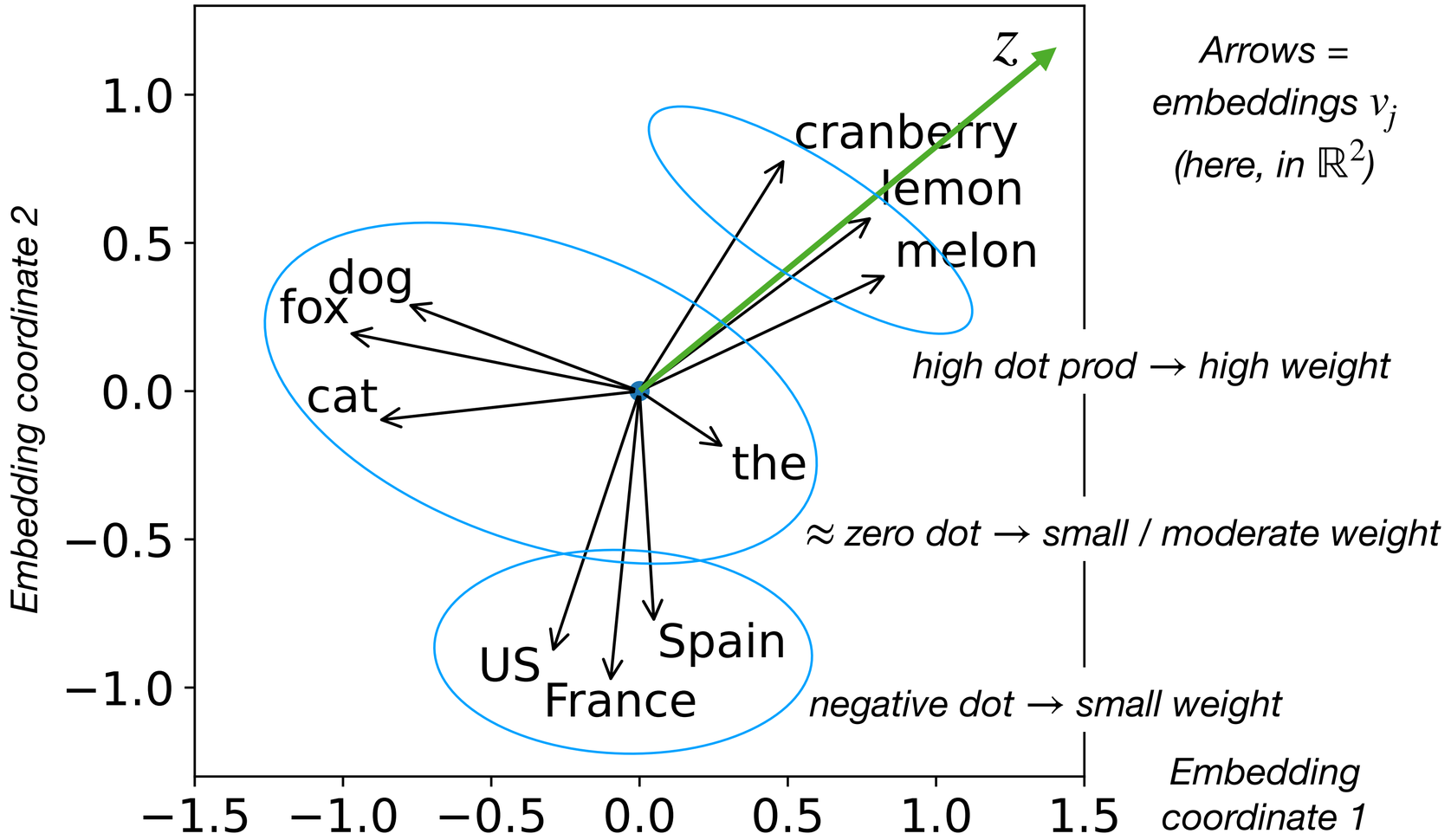
- Dot product $v_j \cdot z$ is highest for token embeddings that are both *high norm* and *similar direction* to z
- The $\exp$ in softmax further emphasizes highest values

Second layer



- Dot product $v_j \cdot z$ is highest for token embeddings that are both *high norm* and *similar direction* to z
- The $\exp$ in softmax further emphasizes highest values

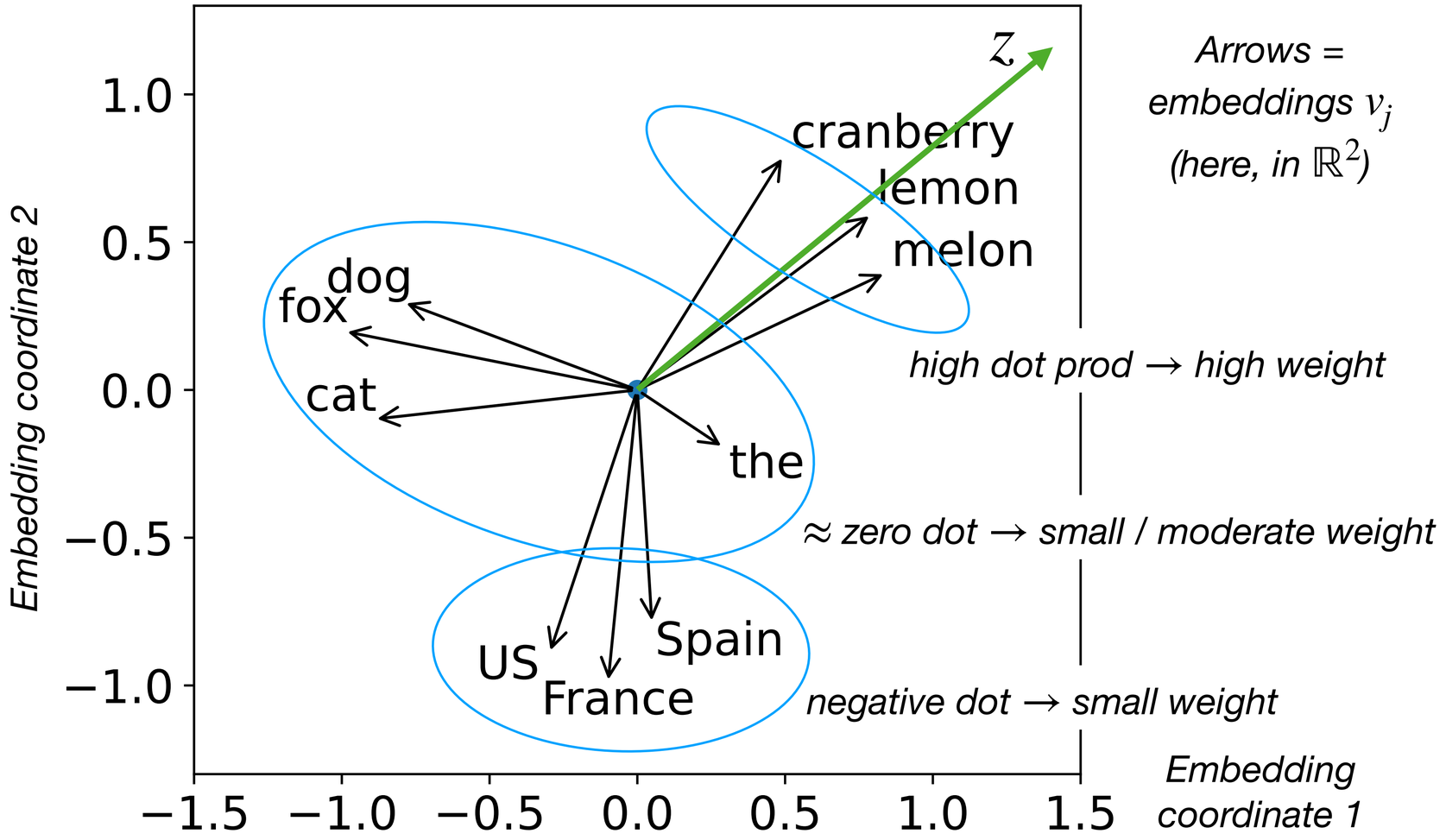
Second layer



- Dot product $v_j \cdot z$ is highest for token embeddings that are both *high norm* and *similar direction* to z
- The $\exp$ in softmax further emphasizes highest values

Second layer

This use of softmax (to find keys that have high dot product w/ query) is often called ***attention***



- Dot product $v_j \cdot z$ is highest for token embeddings that are both *high norm* and *similar direction* to z
- The $\exp$ in softmax further emphasizes highest values

SGD step

- Each optimization step will try to
  - shift embedding of (words in the) query to better retrieve correct masked word (put more attention weight on it)
  - shift the correct masked word's embedding to better match query (have higher dot product with it)
  - shift all other words' embeddings to have lower dot products with the query

Using embeddings

- After we train word2vec, save the embedding matrix
- Can use the learned vectors for tasks like document classification (e.g., words in review → review sentiment)

- Masked token prediction is a ***surrogate*** or ***proxy*** *task:*
  - we don't actually care about performance on it
  - instead, learned model (or some of its parameters) are useful for a different task like classification
  - proxy task is useful because it's easy to get lots of training data or easy to train model