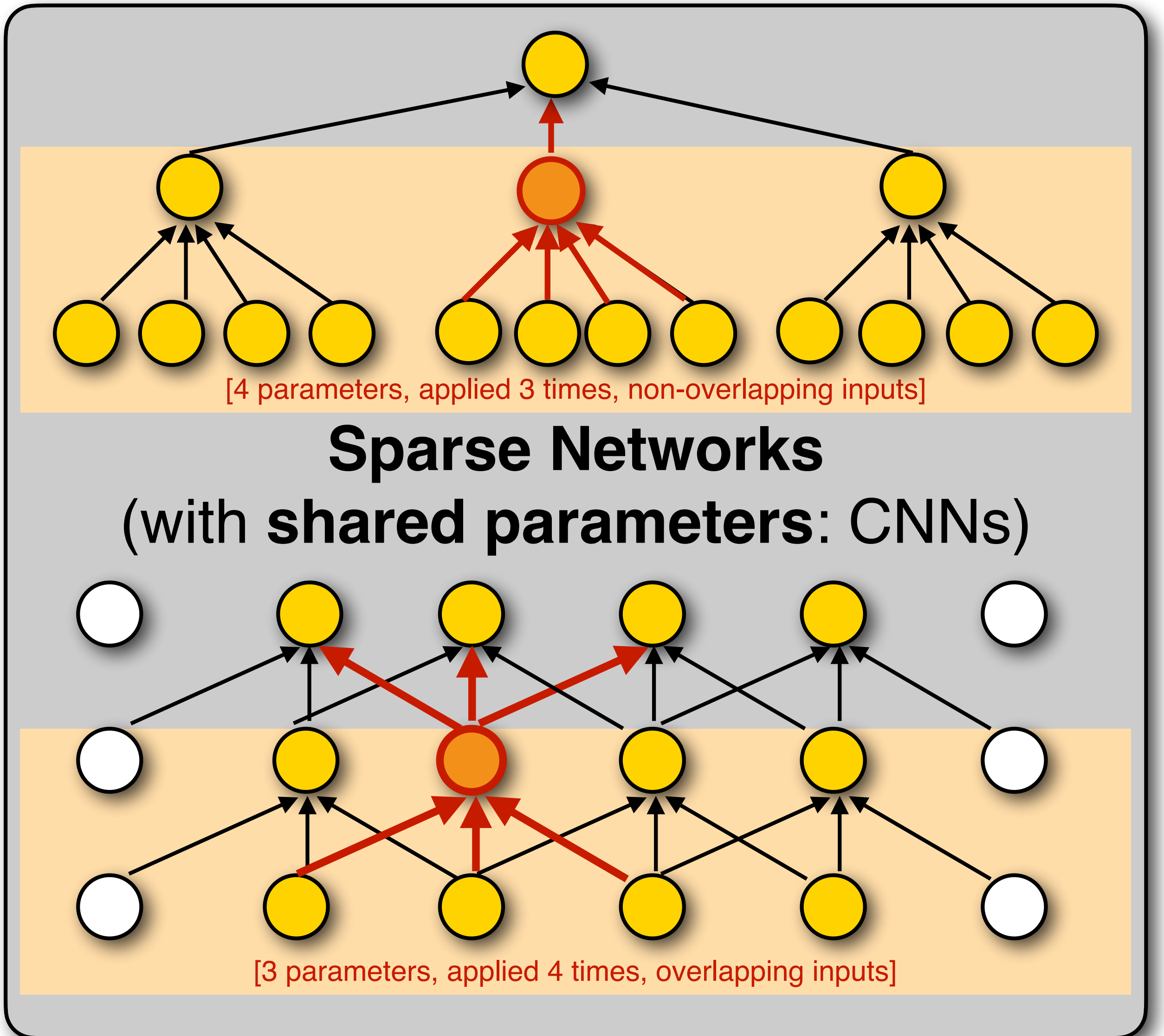
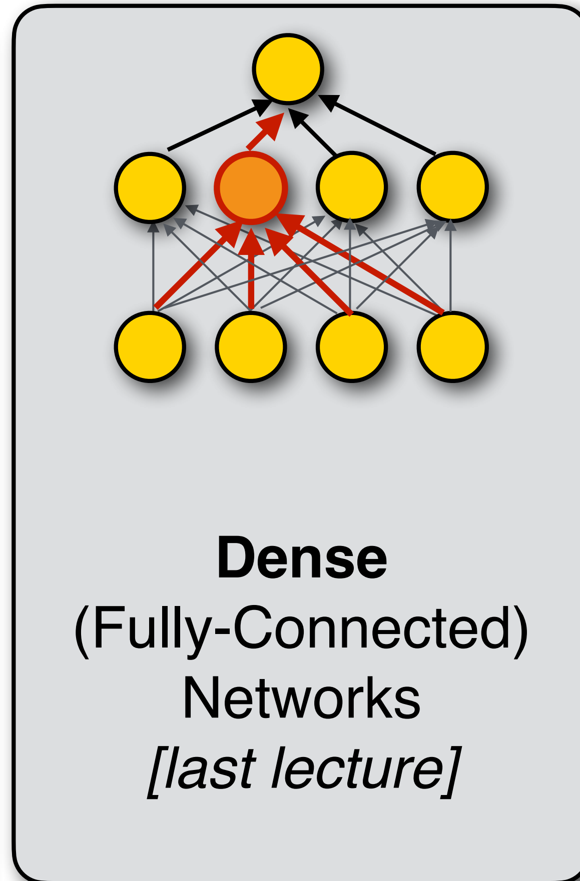
Lecture 8:
Convolutional Neural Nets

Convolutional
Neural Nets

Convolutional Neural Nets (ConvNets, CNNs)





Convolutional Neural Nets

2D CNNs are a standard architecture for **image** data.

Neocognitron (Fukushima, 1980):
CNN with convolutional and downsampling (pooling) layers

CNNs are inspired by **receptive fields** in the **visual cortex:** Individual neurons respond to small regions (patches) of the visual field.

Neurons in deeper layers respond to larger regions.

Neurons in the same layer **share the same weights**.

This **parameter tying** allows CNNs to handle **variable size inputs** with a **fixed number of parameters**.

CNNs can be used as input to fully connected nets.

In NLP, CNNs are mainly used for **classification**.

A toy example

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix}$$

A 3x4 black-and-white image is a 3x4 matrix of pixels.

Applying a 2x2 filter

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \qquad \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

$$\begin{bmatrix} aw + bx + ey + fz & bw + cx + fy + gz & cw + dx + gy + hz \\ ew + fx + iy + jz & fw + gx + jy + kz & gw + hx + ky + lz \end{bmatrix}$$

A $N \times N$ filter is an $N \times N$-size matrix that can be applied to $N \times N$-size patches of the input image.

This operation is called convolution, but it works just like a dot product of vectors.

Applying a 2x2 filter

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \qquad \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

$$\begin{bmatrix} aw + bx + ey + fz & bw + cx + fy + gz & cw + dx + gy + hz \\ ew + fx + iy + jz & fw + gx + jy + kz & gw + hx + ky + lz \end{bmatrix}$$

We can apply the *same* $N \times N$ filter to *all* $N \times N$-size patches of the input image.

We obtain another matrix (the **next layer** in our network).

The **elements of the filter** are the **parameters** of this layer.

Applying a 2x2 filter

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \qquad \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

$$\begin{bmatrix} aw+bx+ey+fz & bw+cx+fy+gz & cw+dx+gy+hz \\ ew+fx+iy+jz & fw+gx+jy+kz & gw+hx+ky+lz \end{bmatrix}$$

We can apply the *same* $N \times N$ filter to *all* $N \times N$-size patches of the input image.

We obtain another matrix (the **next layer** in our network).

The **elements of the filter** are the **parameters** of this layer.

Applying a 2x2 filter

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \qquad \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

$$\begin{bmatrix} aw+bx+ey+fz & bw+cx+fy+gz & cw+dx+gy+hz \\ ew+fx+iy+jz & fw+gx+jy+kz & gw+hx+ky+lz \end{bmatrix}$$

We can apply the *same* $N \times N$ filter to *all* $N \times N$-size patches of the input image.

We obtain another matrix (the **next layer** in our network).

The **elements of the filter** are the **parameters** of this layer.

Applying a 2x2 filter

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \qquad \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

$$\begin{bmatrix} aw+bx+ey+fz & bw+cx+fy+gz & cw+dx+gy+hz \\ ew+fx+iy+jz & fw+gx+jy+kz & gw+hx+ky+lz \end{bmatrix}$$

We can apply the *same* $N \times N$ filter to *all* $N \times N$-size patches of the input image.

We obtain another matrix (the **next layer** in our network).

The **elements of the filter** are the **parameters** of this layer.

Applying a 2x2 filter

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \qquad \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

$$\begin{bmatrix} aw+bx+ey+fz & bw+cx+fy+gz & cw+dx+gy+hz \\ ew+fx+iy+jz & fw+gx+jy+kz & gw+hx+ky+lz \end{bmatrix}$$

We can apply the *same* $N \times N$ filter to *all* $N \times N$-size patches of the input image.

We obtain another matrix (the **next layer** in our network).

The **elements of the filter** are the **parameters** of this layer.

Applying a 2x2 filter

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \qquad \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

$$\begin{bmatrix} aw + bx + ey + fz & bw + cx + fy + gz & cw + dx + gy + hz \\ ew + fx + iy + jz & fw + gx + jy + kz & gw + hx + ky + lz \end{bmatrix}$$

We've turned a **3x4 matrix** into a **2x3 matrix**, so our image has shrunk.

Can we preserve the size of the input?

Zero padding

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & a & b & c & d \\ 0 & e & f & g & h \\ 0 & i & j & k & l \end{bmatrix} \qquad \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0w+0x+0y+az & 0w+0x+ay+bz & 0w+0x+by+cz & 0w+0x+cy+dz \\ 0 & 0w+ax+0y+ez & aw+bx+ey+fz & bw+cx+fy+gz & cw+dx+gy+hz \\ 0 & 0w+ex+0y+iz & ew+fx+iy+jz & fw+gx+jy+kz & gw+hx+ky+lz \end{bmatrix}$$

If we pad each matrix with 0s, we can maintain the same size throughout the network

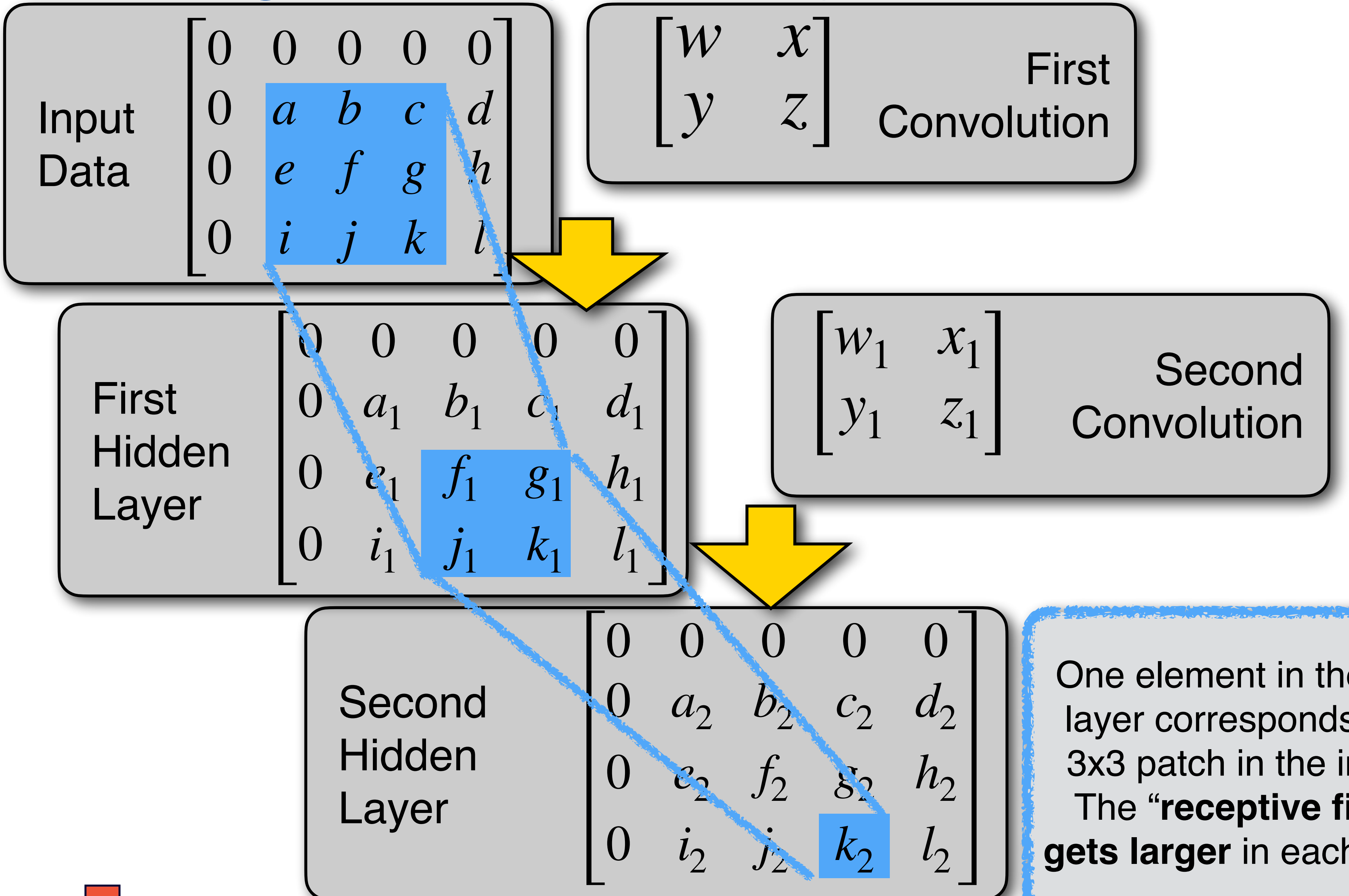
After the nonlinear activation function

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & a & b & c & d \\ 0 & e & f & g & h \\ 0 & i & j & k & l \end{bmatrix} \qquad \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & g(az) & g(ay+bz) & g(by+cz) & g(cy+dz) \\ 0 & g(ax+ez) & g(aw+bx+ey+fz) & g(bw+cx+fy+gz) & g(cw+dx+gy+hz) \\ 0 & g(ex+iz) & g(ew+fx+iy+jz) & g(fw+gx+jy+kz) & g(gw+hx+ky+lz) \end{bmatrix}$$

NB: Convolutional layers are typically followed by ReLUs.

Going from layer to layer…



One element in the 2nd layer corresponds to a 3x3 patch in the input: The "**receptive field**" **gets larger** in each layer

Changing the stride

Stride = the step size for sliding across the image

Stride = 1: Consider all patches [see previous example]

Stride = 2: Skip one element between patches

Stride = 3: Skip two elements between patches,…

A larger stride size yields a smaller output image.

Input: $\begin{bmatrix} 0 & 0 & 0 & 0 \\ a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix}$ Filter: $\begin{bmatrix} w & x \\ y & z \end{bmatrix}$

[Note that different zero-padding may be required with a different stride]

Stride = 2: $\begin{bmatrix} 0w + 0x + ay + bz & 0w + 0x + cy + dz \\ ew + fx + iy + jz & gw + hx + ky + lz \end{bmatrix}$

Handling color images: channels

Color images have a number of color channels:

Each pixel in an RGB image is a *(red, green, blue)* triplet: ■ =(255, 0, 0) or ■ =(120, 5, 155)

An $N \times M$ RGB image is a $N \times M \times 3$ **tensor**

height × *width* × *depth*

#channels = depth of the image

Convolutional filters are applied to **all channels** of the input

We still specify filter size in terms of the image patch, because the #channels is a function of the data (not a parameter we control)

We still talk about 2×2 or 3×3 etc. filters, although with C channels, they apply to a $N \times N \times C$ region (and have $N \times N \times C$ weights)

Channels in internal layers

So far, we have just applied a single $N \times N$ filter to get to the next layer.

But we could run K different $N \times N$ filters (with different weights) to define a layer with K channels.

(If we initialize their weights randomly, they will learn different properties of the input)

The **hidden layers** of CNNs have often a large number of channels.

(Useful trick: 1x1 convolutions increase or decrease the nr. of channels without affecting the size of the visual field)

Pooling Layers

Pooling layers reduce the size of the representation, and are often used following a pair of conv+ReLU layers

Each **pooling layer** returns a 3D tensor of the same depth as its input (but with smaller height & width) and is defined by

— a **filter** size (what region gets reduced to a single value)

— a **stride** (step size for sliding the window across the input)

— a **pooling function** (**max pooling**, avg pooling, min pooling, …)

Pooling units don't have weights, but simply return the maximum/minimum/average value of their inputs

Typically, pooling layers only receive input from a single channel.
So they don't reduce the depth (#channels).

Max-pooling

Max-pooling in our example
with a **2x2 filter** and **stride=2**:

Input: $\begin{bmatrix} 0 & 0 & 0 & 0 \\ a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix}$ 2x2 MaxPooling

Stride = 2: $\begin{bmatrix} \max(0,0,a,b) & \max(0,0,c,d) \\ \max(e,f,i,j) & \max(g,h,k,l) \end{bmatrix}$

(2D) CNNs

An image is a 2D (width × height) matrix of pixels (e.g. RGB values)

=> it is a 3D tensor: color channels (“depth”) × width × height

Each **convolutional layer** returns a 3d tensor, and is defined by:

- — the **depth** (#filters) of its output
- — a **filter size** (the square size of the input regions for each filter),
- — a **stride** (the step size for how to slide filters across the input)
- — **zero padding** (how many 0s are added around edges of input)

=> Filter size, stride, zero padding define the width/height of the output

Each unit in a convolutional layer

— receives input from a square region/patch (across w×h) in the preceding layer (across all depth channels)

— returns the **dot product** of the **input** activations and its **weights**

Within a layer, all units at the same depth use the same weights

Convolutional layers are often followed by ReLU activations

http://cs231n.github.io/convolutional-networks/

1D CNNs for text

Text is a (variable-length) **sequence** of words (word vectors)

[#channels = dimensionality of word vectors]

We can use a **1D CNN** to slide a window of *n* tokens across:

— Filter size n = 3, stride = 1, no padding

The quick brown fox jumps over the lazy dog

The **quick brown fox** jumps over the lazy dog

The quick **brown fox jumps** over the lazy dog

The quick brown **fox jumps over** the lazy dog

The quick brown fox **jumps over the** lazy dog

The quick brown fox jumps **over the lazy** dog

— Filter size n = 2, stride = 2, no padding:

The quick brown fox jumps over the lazy dog

The quick **brown fox** jumps over the lazy dog

The quick brown fox **jumps over** the lazy dog

The quick brown fox jumps over **the lazy** dog

1D CNNs for text classification

Input: a variable length sequence of word vectors
(#channels/depth = dimensionality of word vectors)

Zero padding: Add zero vectors (or to BOS/EOS)
to beginning and/or end of sentence (and/or hidden layers)

Filters: N-dimensional vectors (sliding windows of N-grams)

Filter size N in the first layer: size of the N-grams we consider

Conv. layers typically have a **ReLU** (or tanh) activation

Maxpooling layers reduce the dimensionality.

CNN depth: how many layers do we use?

The **last CNN layer** (a $H \times W \times D$ **tensor**) needs to be **reshaped** (flattened) into a $(H \times W \times D)$-dimensional **vector** to be fed into a dense **feedforward net** for classification

Understanding CNNs for text classification

Jacovi et al.'18 https://www.aclweb.org/anthology/W18-5408/

— Different **filters detect (suppress)** different **types of ngrams**

— **Max-pooling** removes irrelevant n-grams

— In a **single-layer CNN with max-pooling**, each filter output can be traced back to a **single input ngram**

— Each filter can also be associated with a **class it predicts**

— The **positions in a filter** check whether specific **types of words** are present or absent in the input

— Filters can produce erroneous output (abnormally high activations) on artificial input

Readings and nice illustrations

https://www.deeplearningbook.org/contents/convnets.html

https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md