CS447: Natural Language Processing

http://courses.engr.illinois.edu/cs447

Lecture 10: Neural language models, CNNs for natural language

Julia Hockenmaier

juliahmr@illinois.edu

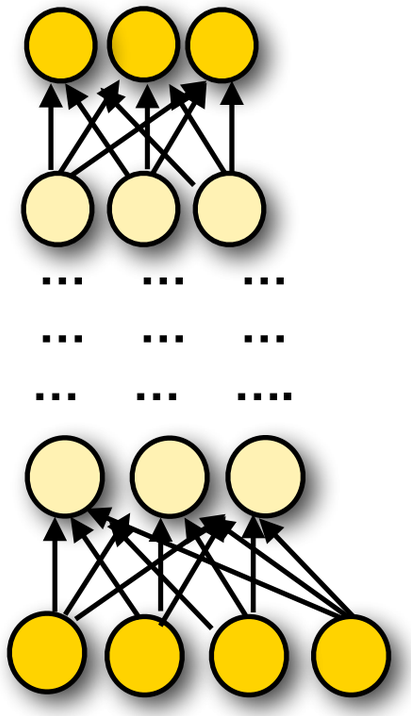
3324 Siebel Center

Recap: What are neural nets?

Fully connected feedforward nets

Three kinds of layers,
arranged in sequence:

- **Input layer**
  (what's fed into the net)
- **Hidden layers**
  (intermediate computations)
- **Output layer**
  (what the net returns)



Output layer: vector **y**

Hidden layer: vector $\mathbf{h_n}$

Hidden layer: vector $\mathbf{h_1}$

Input layer: vector **x**

Each layer consists of a number of **units**.

- Each hidden/output unit computes a ***real-valued* activation**
- In a ***feedforward*** net, each (hidden/output) unit receives inputs from the units in the **immediately *preceding* layer**
- In a ***fully connected*** feedforward net, each unit receives inputs from ***all* units** in the immediately preceding layer

Additional "*Highway connections*" that skip layers can be useful

Feedforward computations

The **activation x_{ij} of unit j in layer i** is computed as

$$x_{ij} = g(\mathbf{w}_{ij} \cdot \mathbf{x}_{i-1} + b_{ij})$$

where

— $\mathbf{w}_{ij} = (w_{ij1}, \ldots, w_{ijK})$ is a (unit-specific) **weight vector**
(K= #units in $(i-1)$-th layer, because each connection into unit j is associated with one real-valued weight for each unit in the preceding layer)

— b_{ij} is a (unit-specific) real-valued **bias term**

— $g()$ is a (layer-specific) **non-linear activation function**

Each **layer** is defined by its number of units, N,
a non-linear activation function $g()$ applied to all units in the layer,
a learned matrix of weights $\mathbf{W}$, and a learned bias vector $\mathbf{b}$.

Nonlinear Activation Functions $g()$

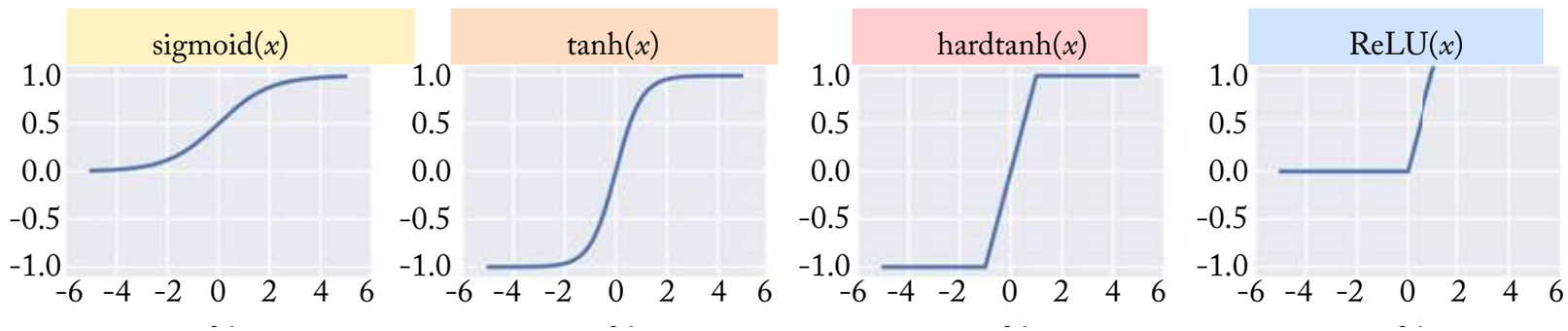


Fig.:Y. Goldberg (2017) Neural Network Methods for Natural Language Processing

Sigmoid (logistic function) $\sigma(x) = \frac{1}{1+e^{-x}}$

Outputs in $[0,1]$ range. Useful for output units (**probabilities**), interpolation

Hyperbolic tangent: $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$

Outputs in $[-1,1]$ range. Useful for **internal** units

Hard tanh $\text{htanh}(x) = -1 \text{ for } x < -1,\ 1 \text{ for } x > 1,\ x \text{ otherwise}$

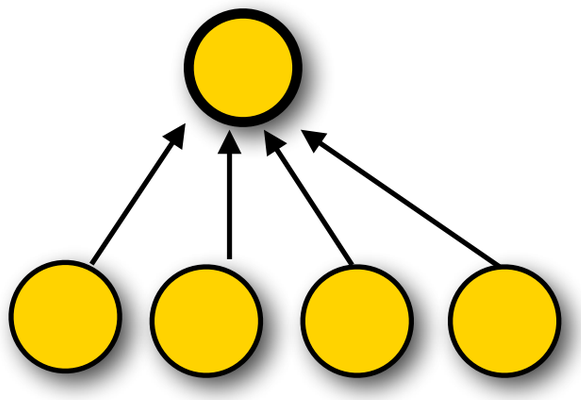
Outputs in $[-1,1]$ range. Approximates $\tanh$

Rectified Linear Unit: $ReLU(x) = \max(0,x)$

Outputs in $[0, +\infty]$. Works very well for **internal** units.

Binary Classification
with a multilayer feedforward net

The **output layer** consists of a **single unit**
with the **sigmoid** activation function

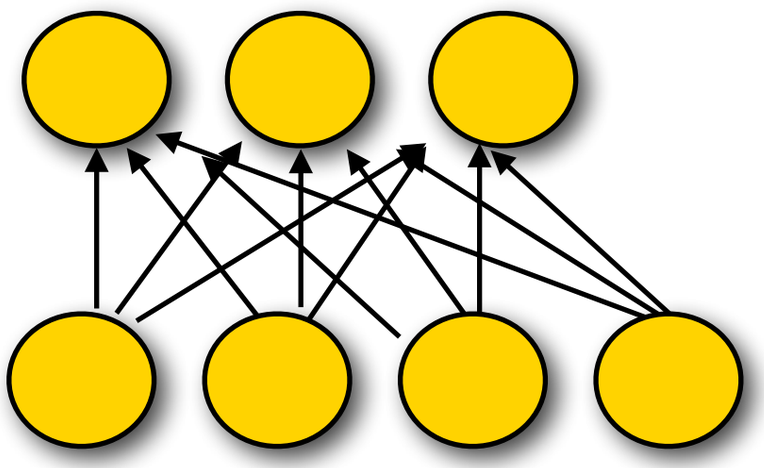


One output unit with
sigmoid activation function

$$y = \sigma(\mathbf{w}\mathbf{x} + b) \in [0...1]$$

Multi-*Class* Classification
with a multilayer feedforward net

With K output classes, the output layer has K **units** with a **softmax** activation function:



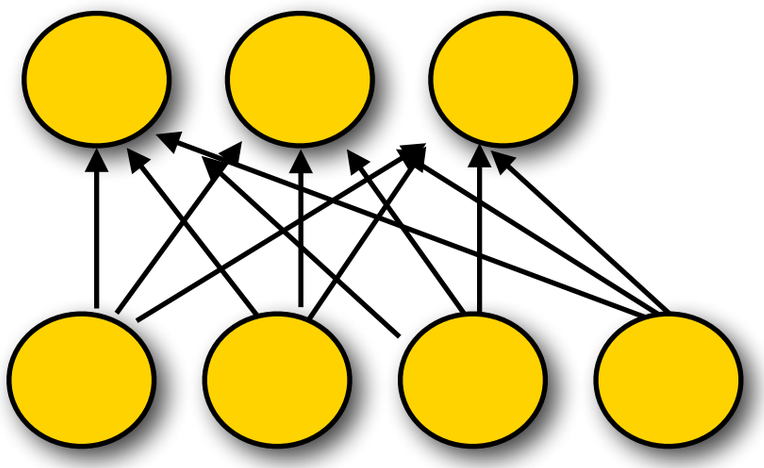
Output layer:
A vector $\mathbf{y} = (y_1, \ldots, y_K)$ where the *i*-th element corresponds to the probability that the input has class *i*:

$$y_i = \mathbf{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{k=1}^{K} \exp(z_k)}$$

such that we get a categorical distribution over all *K* classes

Multi-*Label* Classification
with a multilayer feedforward net

With K output classes, K output units
with K **sigmoid** activation functions:



Output layer:
A vector $\mathbf{y} = (y_1, \ldots, y_K)$ where the *i*-th element corresponds to the probability that the input does (or doesn't) have class *i*:

$$y_i = \mathbf{sigmoid}(\mathbf{w}_i\mathbf{x}_i + b_i)$$

We now have a separate probability for each possible class label.

Part 3: Neural n-gram models

Our first neural net for NLP: A neural n-gram model

Given a fixed-size vocabulary V, an *n*-gram model predicts the probability of the *n*-th word following the preceding *n*–1 words:

$$P(w^{(i)} \mid w^{(i-1)}, w^{(i-2)}, \ldots, w^{i-(n-1)})$$

How can we model this with a neural net?

— **Input layer:** concatenate *n*–1 word vectors

— **Output layer:** a softmax over |V| units

An n-gram model $P(w \mid w_1 \ldots w_k)$ as a feedforward net (naively)

Assumptions:

The **vocabulary** V contains V types (incl. UNK, BOS, EOS)
We want to condition each word on k preceding words

Our (naive) model:

— [Naive]
Each **input word** $w_i \in V$ is a ***V*-dimensional one-hot vector** $v(w)$
→ The **input layer** $\mathbf{x} = [v(w_1),\ldots,v(w_k)]$ has $V \times k$ elements

— We assume **one hidden layer h**

— The **output layer** is a softmax over V elements

$$P(w \mid w_1 \ldots w_k) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$$

An n-gram model $P(w \mid w_1 \ldots w_k)$ as a feedforward net (better)

Assumptions:

The **vocabulary** V contains V types (incl. UNK, BOS, EOS)
We want to condition each word on k preceding words

Our (better) model:

— [Better]
Each **input word** $w_i \in V$ is an ***n*-dimensional dense embedding vector** $v(w)$ (with $n \ll V$)
→ The **input layer** $\mathbf{x} = [v(w_1),\ldots,v(w_k)]$ has $n \times k$ elements

— We assume **one hidden layer h**

— The **output layer** is a softmax over V elements

$$P(w \mid w_1 \ldots w_k) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$$

Our neural n-gram models

Architecture:

Input Layer: $\mathbf{x} = [v(w_1)\ldots.v(w_k)]$

Hidden Layer: $\mathbf{h} = g(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)$

Output Layer: $P(w \mid w_1 \ldots w_k) = \text{softmax}(\mathbf{h}\mathbf{W}^2 + \mathbf{b}^2)$

How many parameters do we need? [# of weights and biases]:

Hidden layer with one-hot inputs: $\mathbf{W}^1 \in \mathbb{R}^{(k \cdot V) \times \dim(\mathbf{h})}$ $\mathbf{b}^1 \in \mathbb{R}^{\dim(\mathbf{h})}$

Hidden layer with dense inputs: $\mathbf{W}^1 \in \mathbb{R}^{(k \cdot n) \times \dim(\mathbf{h})}$ $\mathbf{b}^1 \in \mathbb{R}^{\dim(\mathbf{h})}$

Output layer (any inputs): $\mathbf{W}^2 \in \mathbb{R}^{\dim(\mathbf{h}) \times V}$ $\mathbf{b}^2 \in \mathbb{R}^{V}$

With V = 10K, n = 300 (word2vec), dim(h) = 300

k = 2 (trigram): $\mathbf{W}^1 \in \mathbb{R}^{20{,}000 \times 300}$ or $\mathbf{W}^1 \in \mathbb{R}^{600 \times 300}$ and $\mathbf{b}_1 \in \mathbb{R}^{300}$

k = 5 (six-gram): $\mathbf{W}^1 \in \mathbb{R}^{50{,}000 \times 300}$ or $\mathbf{W}^1 \in \mathbb{R}^{1500 \times 300}$ and $\mathbf{b}_1 \in \mathbb{R}^{300}$

$\mathbf{W}^2 \in \mathbb{R}^{300 \times 10{,}000}$ $\mathbf{b}^2 \in \mathbb{R}^{10{,}000}$

Six-gram model with one-hot inputs: 27,000,460,000 parameters,
with dense inputs: 3,460,000 parameters

Traditional six-gram model: $10^{4 \times 6} = 10^{24}$ parameters

Naive (one-hot input) neural n-gram model

Advantage over non-neural n-gram model:

— The hidden layer captures **interactions** among context words

— **Increasing the order** of the n-gram requires only a small **linear increase** in the number of parameters.

dim($\mathbf{W}^1$) goes from $(k \cdot \dim(V)) \cdot \dim(\mathbf{h})$ to $((k+1) \cdot \dim(V)) \cdot \dim(\mathbf{h})$

— **Increasing the vocabulary** also leads only to a linear increase in the number of parameters

But: With a one-hot encoding and dim(V) ≈ 10K or so, this model still requires a LOT of parameters to learn.

And: The Markov assumption still holds

Better (dense embeddings input) neural n-gram model

Advantage over non-neural n-gram model:

— Same as naive neural model, plus:

Advantages over naive neural n-gram model:

— We have far **fewer parameters** to learn

— **Better generalizations:** If similar input words have similar embeddings, the model will predict similar probabilities in similar contexts:

$$P(w \mid \text{the doctor saw the}) \approx P(w \mid \text{a nurse sees her})$$

But: This generalization only works if the contexts have similar words in the *same* position.

And: The Markov assumption still holds.

Neural n-gram models

Naive neural n-gram models (one-hot inputs) have similar shortcomings to standard n-gram models

- Models get very large (and sparse) as n increases
- We can't generalize across similar contexts
- Markov (independence) assumptions are too strict

Better neural n-gram models can be obtained with dense word embeddings:

- Models remain much smaller
- Embeddings may provide some (limited) generalization across similar contexts

Future lectures: CBOW neural nets as a language model, recurrent language models

Part 3: Word2vec as Language model

Word2Vec as language model

Instead of training a binary classifier for pairs of words, **predict context words from the target** (**Skipgram**) or **the target word from context words** (**CBOW**)

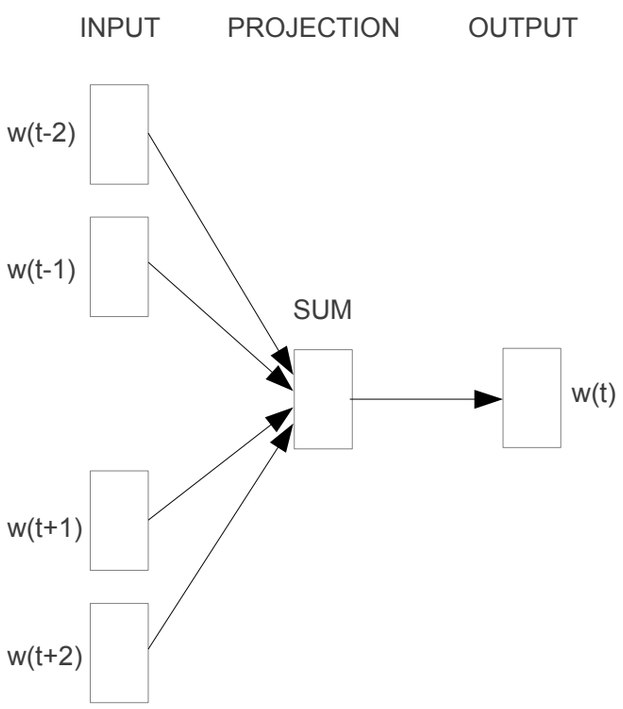
The output of this model is a distribution over words.

(Mikolov et al. use a “hierarchical” softmax, based on a Huffman (binary tree) encoding of the (output) vocabulary, where the most common words have the shortest bit vector to be predicted.

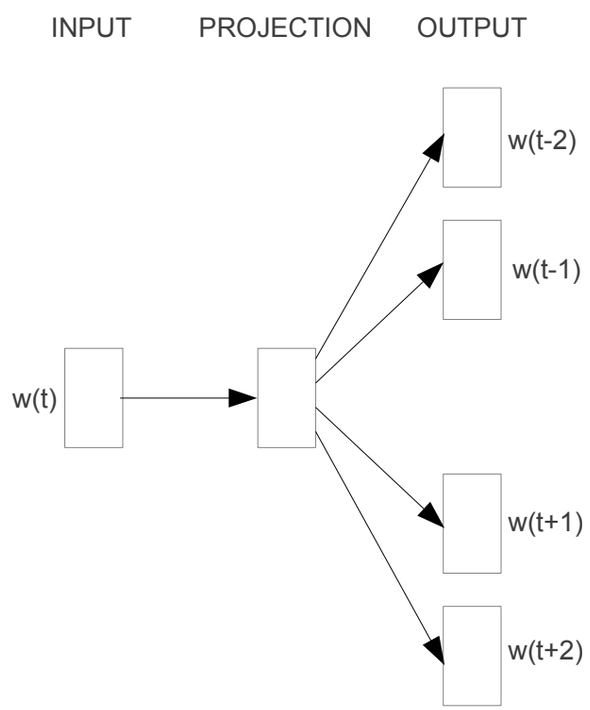
CBOW vs Skipgram

CBOW (continuous bag of words):
predict target word from surrounding context

Skipgram: Predict context words from target word



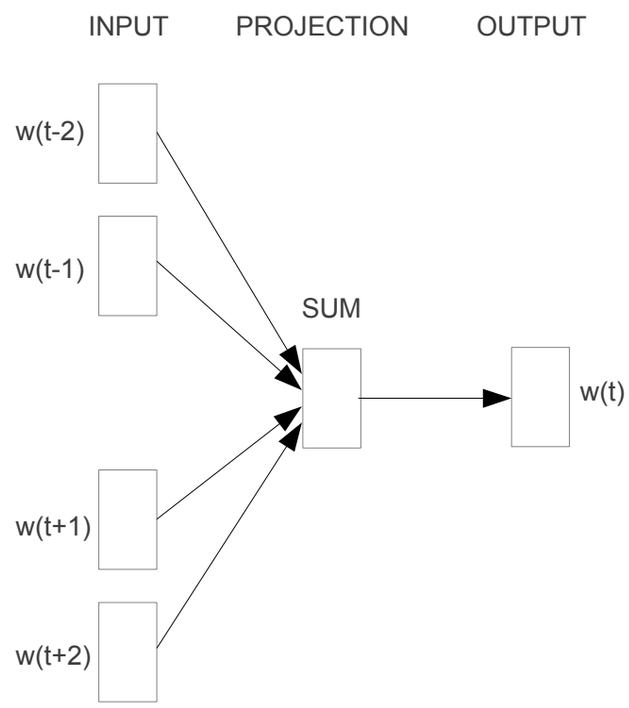
CBOW



Skip-gram

CBOW (as neural LM)

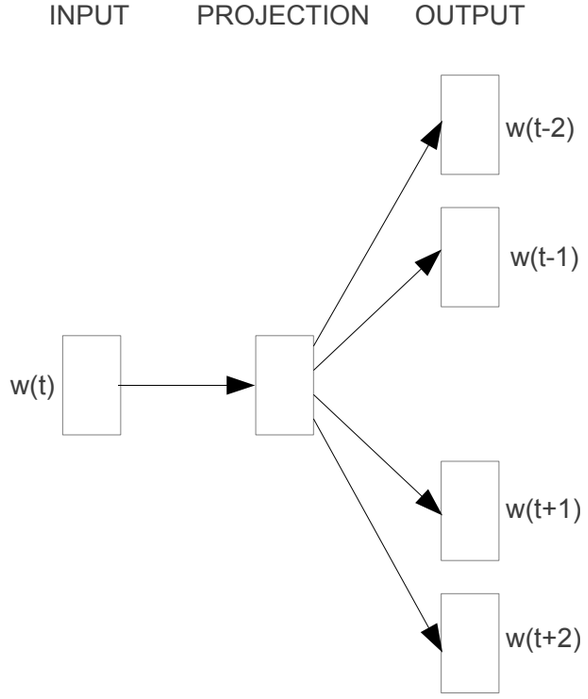
- Learn D-dimensional embeddings for each (context) word
- Predict target word based on the sum (average) of the embeddings of the words in its context with a standard neural language model
- Return trained context embeddings



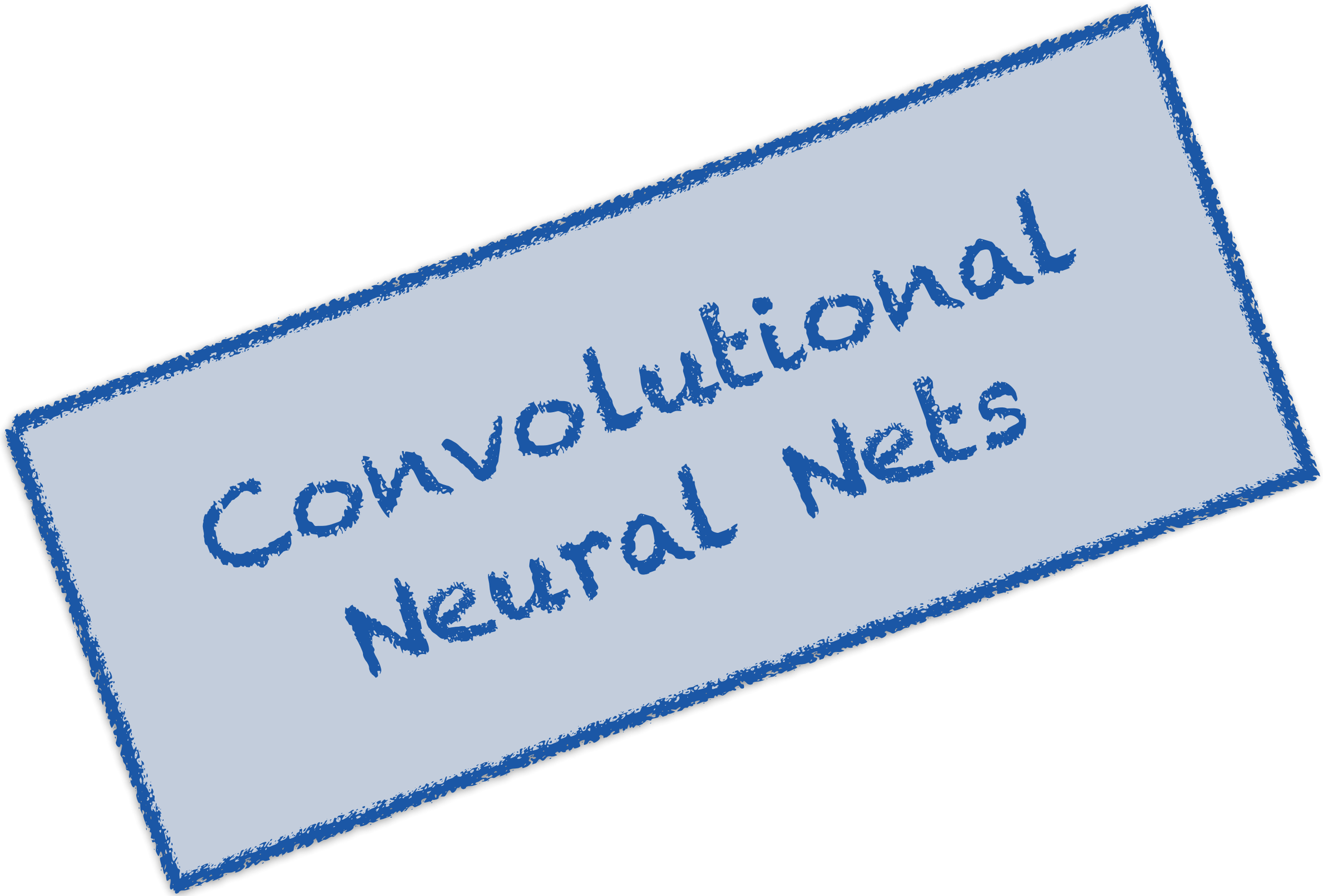
CBOW

Skipgram (as neural LM)

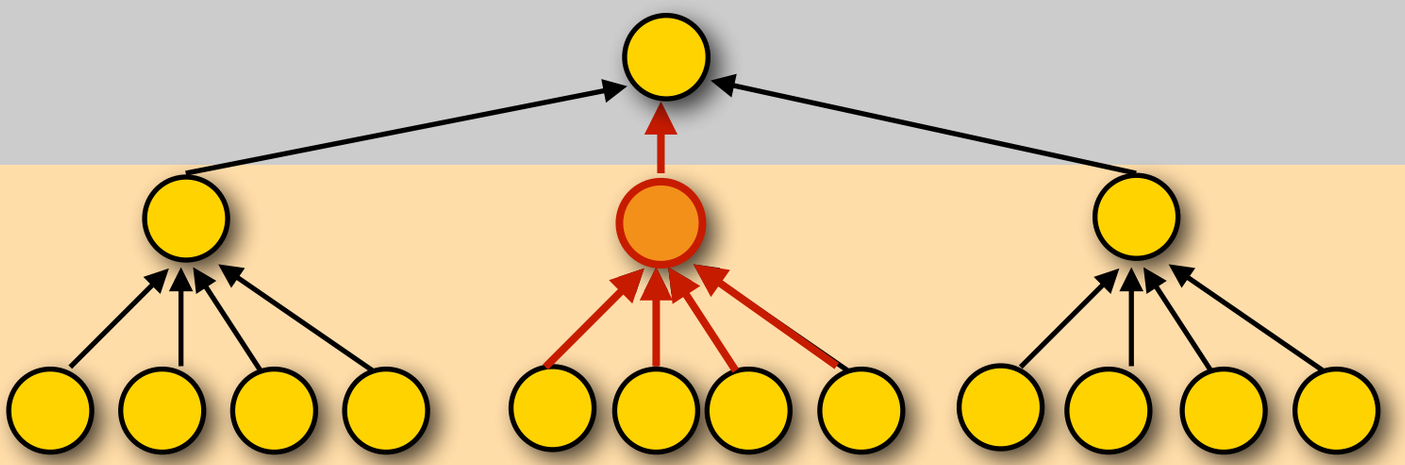
- Learn D-dimensional embeddings for each target word
- Predict all context words based on the target word embedding
- Return trained target embeddings



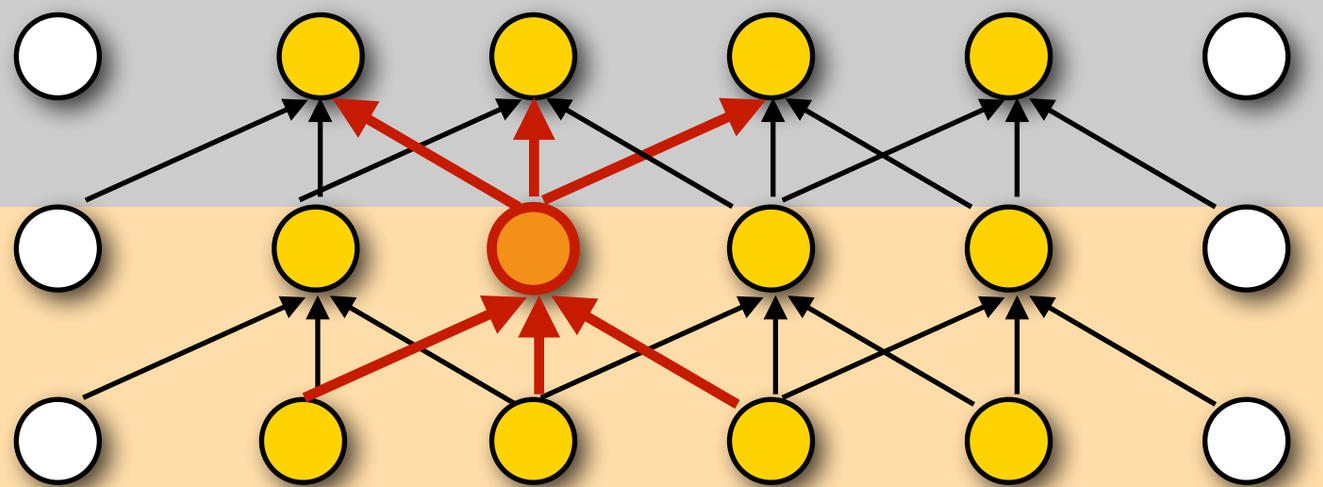
Skip-gram

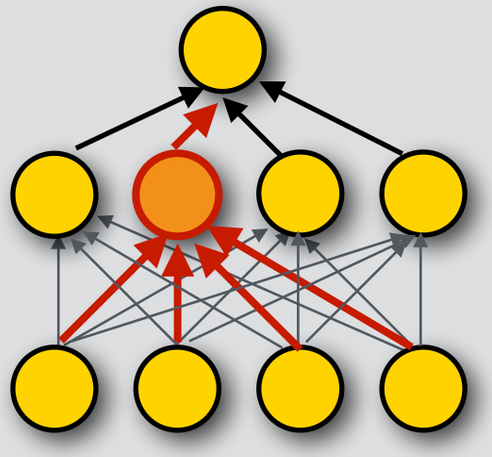

Convolutional
Neural Nets

Convolutional Neural Nets (ConvNets, CNNs)



Sparse Networks
(with **shared parameters**: CNNs)





Dense
(Fully-Connected)
Networks
[last lecture]

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

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

$$\begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

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

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…

Input Data

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

$$\begin{bmatrix} w & x \\ y & z \end{bmatrix}$$ First Convolution

First Hidden Layer

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & a_1 & b_1 & c_1 & d_1 \\ 0 & e_1 & f_1 & g_1 & h_1 \\ 0 & i_1 & j_1 & k_1 & l_1 \end{bmatrix}$$

$$\begin{bmatrix} w_1 & x_1 \\ y_1 & z_1 \end{bmatrix}$$ Second Convolution

Second Hidden Layer

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & a_2 & b_2 & c_2 & d_2 \\ 0 & e_2 & f_2 & g_2 & h_2 \\ 0 & i_2 & j_2 & k_2 & l_2 \end{bmatrix}$$

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:

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

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

Filter: $\begin{bmatrix} w & x \\ y & z \end{bmatrix}$

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