

Deep Learning I: Convolutional and Recurrent Networks

COMP5318 Machine Learning and Data Mining

Semester 1, 2026, week 8

Chuyang Zhou (slides prepared by Irena Koprinska)

Reference: Witten ch. 10.3 and 10.6, Geron: ch.14 and ch.15





- Convolutional neural networks
- Recurrent neural networks

.



Convolutional Neural Networks (CNN)

- Introduced by Yan LeCun et al. in 1989
<http://yann.lecun.org/exdb/publis/pdf/lecun-89e.pdf>
- Gained popularity in 2012 when AlexNet (Alex Krizhevsky et al.) won the ImageNet competition reducing the error rate from 26% to 15%
<https://proceedings.neurips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>
- A special type of multilayer NN; trained with the backpropagation algorithm
- Designed to recognize visual patterns directly from pixel images with minimal pre-processing
- Can recognize patterns with high variability, e.g. handwritten characters; robust to distortions and geometric transformations such as shifting
- Used in image and speech recognition; have shown excellent performance in hand-written digit classification, face detection and image classification

- CNNs are designed for images
 - (but can also be used with other types of data)
- Images are represented as matrices of pixel values
 - Black and white images: a value from 0 (black) to 255 (white) – 1 matrix
 - Color images: the same but for 3 channels: red, green and blue, so 3 matrices



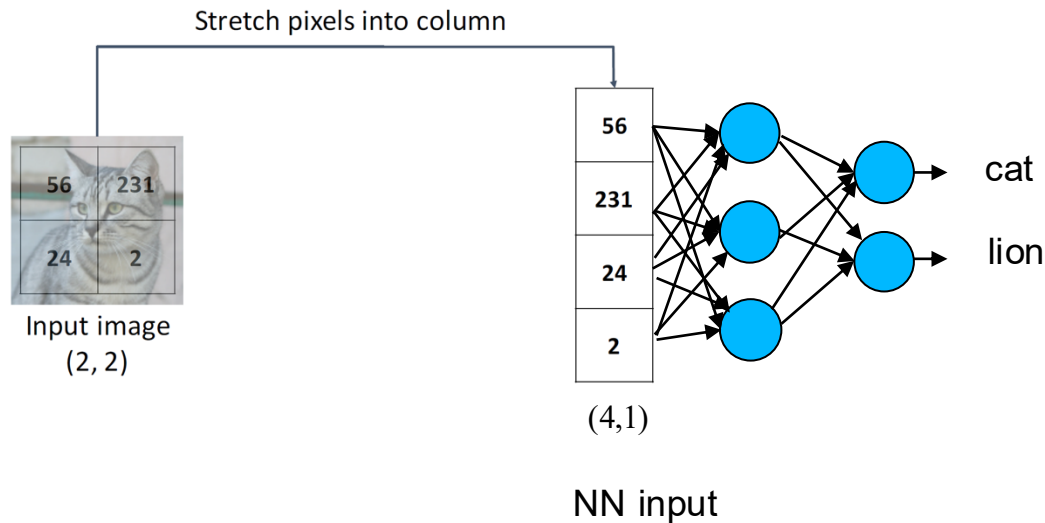
157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	105	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	35	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	105	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	35	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Image ref: <https://ai.stanford.edu/~syyeung/cvweb/tutorial1.html>

Classifying images with feedforward NNs

- Traditional NNs (multilayer feedforward networks) do not consider the **spatial structure of the image**

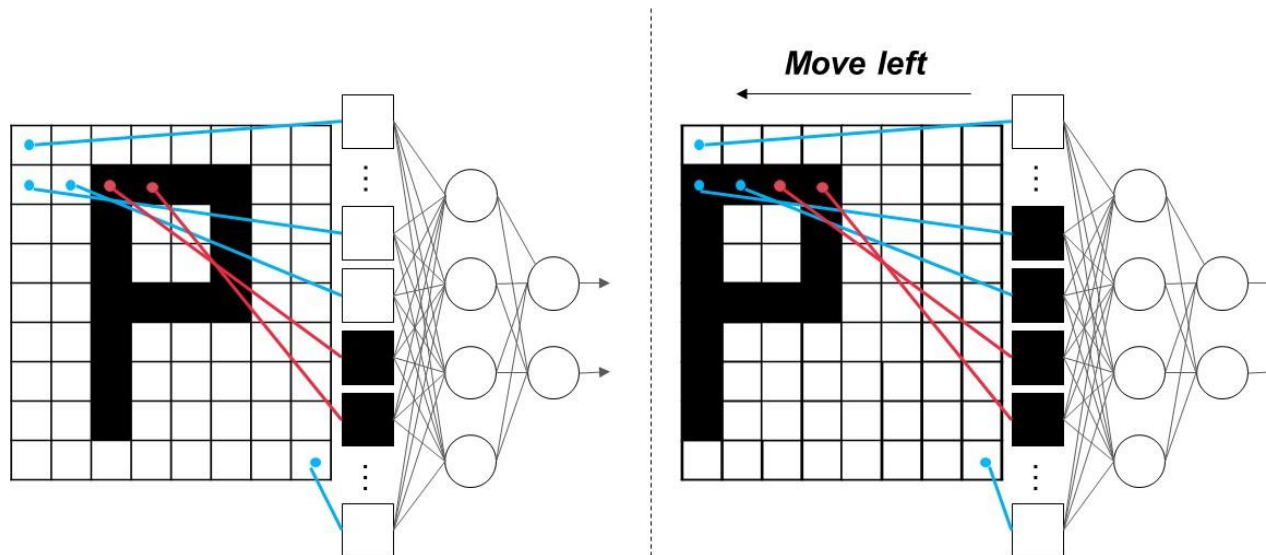


- CNNs do this by using new types of layers - **convolutional** and **pooling**

Adapted from <http://cs231n.stanford.edu/>

Classifying images with feedforward NNs (2)

- Traditional NNs (multilayer feedforward networks) are sensitive to the position of an object in the image; CNN are more robust to shifts/translation
- Robustness to translation is important for image and sound data as translation is one of the main sources of distortion

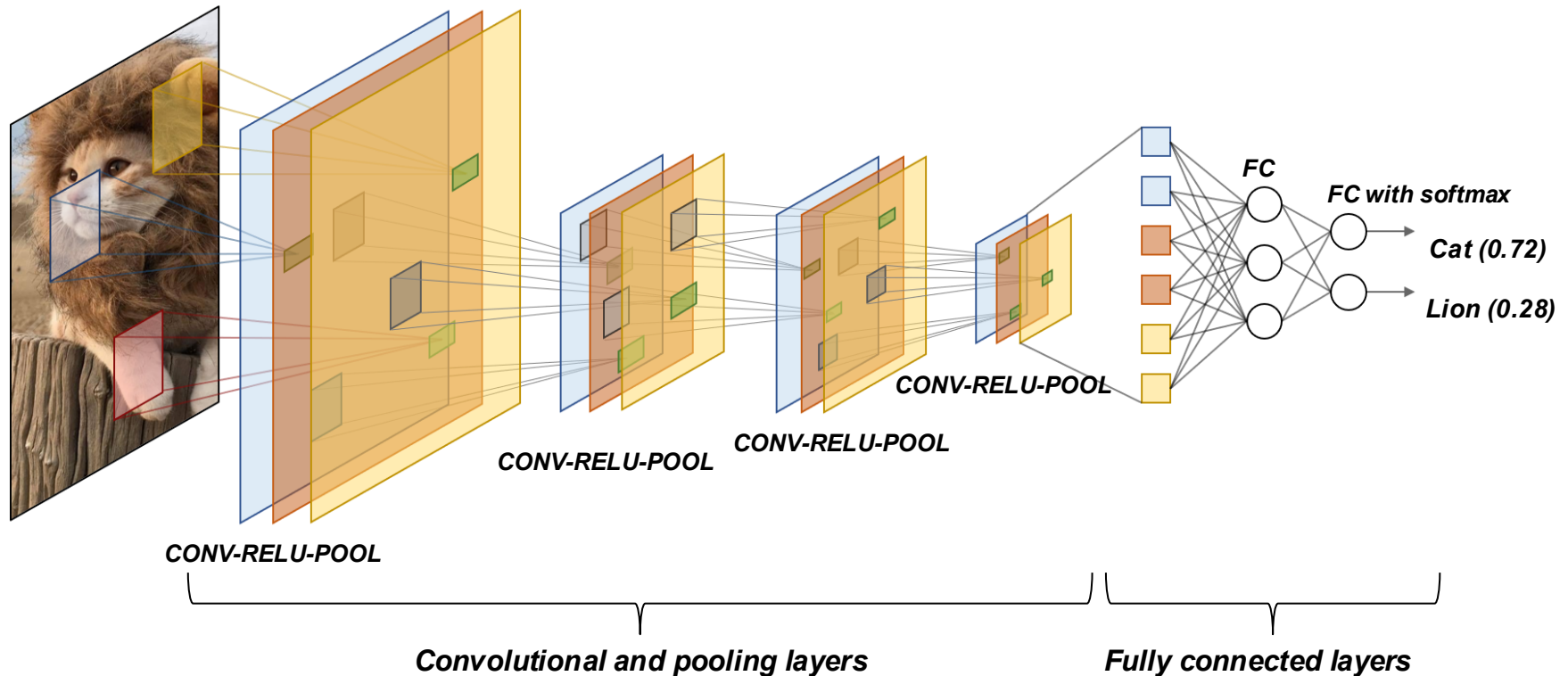


- Moving only 2 pixels but the input values will be very different and the already learned weights will not be very useful

- Input layer for the image values - **INPUT**
- Convolutional layer - **CONV**
- ReLu or LReLu (or other non-linear activation function) - **RELU**
- Pooling layer - **POOL**
- Fully connected layer with softmax activation function at the output - **FC**
- **INPUT-CONV-RELU-POOL-FC**

- The CONV-RELU-POOL block can be repeated several times
- FC can contain one or more FC layers

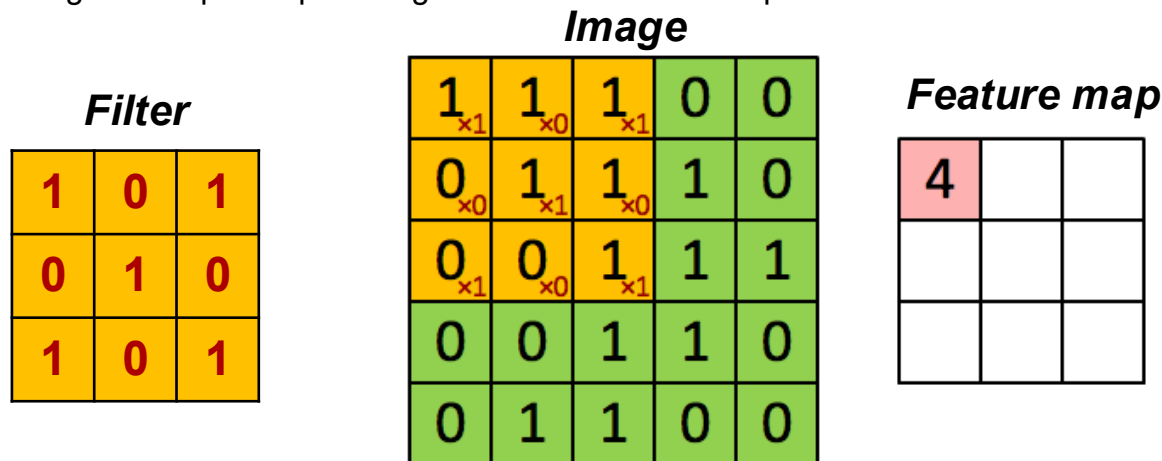
Example: 3 blocks of CONV-RELU-POOL, 2 FC layers



- CNNs are similar to the fully connected NNs we studied last week
- They are made up of neurons that have learnable weights
- Each neuron receives some inputs (e.g. raw pixel values), performs a dot product and follows it with a non-linear activation function
- The network expresses a single differentiable function: from the raw image pixels to the class scores, and is trained with the backpropagation algorithm
- What we studied last week applies here too

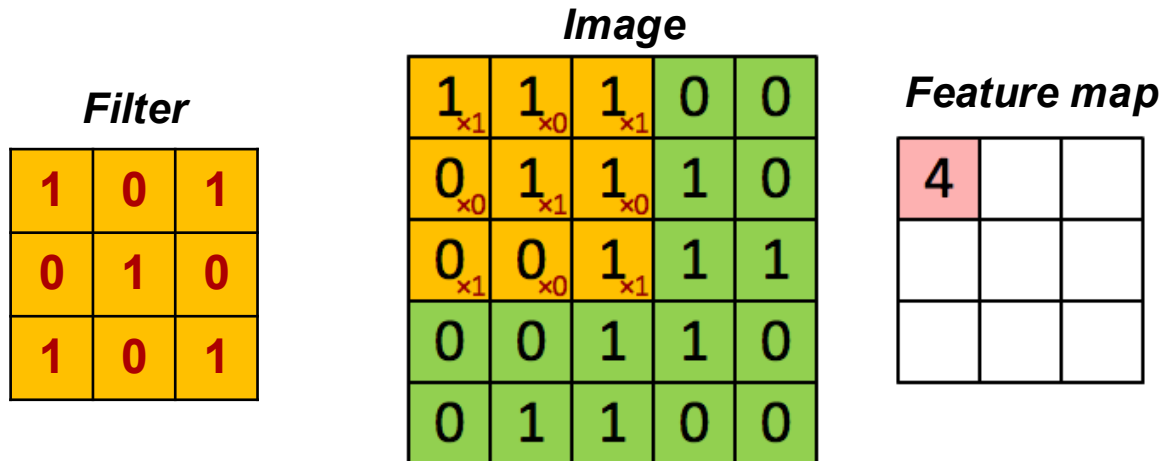
- **Convolution** is like applying a sliding window to a matrix
 - The corresponding elements are multiplied and summed
 - Example: image of black and white values (e.g. 0 is black, 1 is white)
 - 3x3 sliding window (**filter**, kernel) with values (weights) shown in red
 - The filter is shifted from left to right and then down as shown
 - The resulting convolved features formed a **feature map** (or activation map)
 - The first convolved feature is $4 = 1*1+1*0+1*1+0*0+1*1+1*0+0*1+0*0+1*1$

Image ref: <http://deeplearning.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution/>





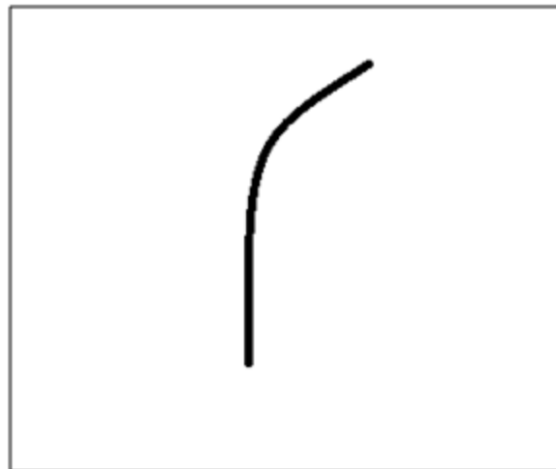
- **Receptive field** of the filter – the image region that is currently being convolved with the filter



- Higher-level perspective – we can think about the filters as **feature detectors**
- Example: a filter for detecting a curve

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter

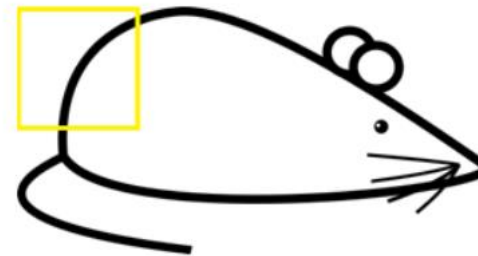
Ref.: <https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>

Understanding convolution (2)

- Let's apply the filter to an image that we want to classify and put the filter at the top left corner



Original image



Visualization of the filter on the image



Visualization of the
receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive
field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Feature
map value:

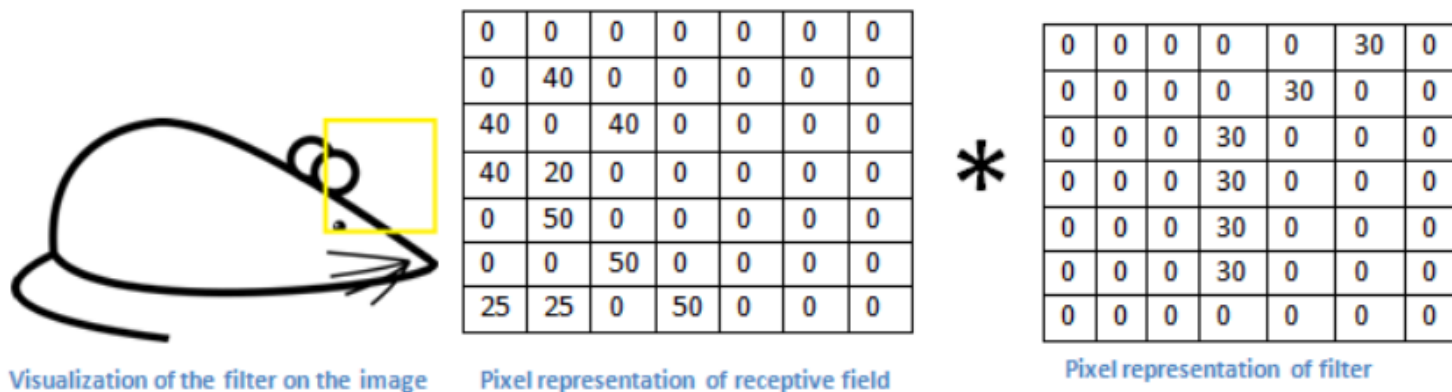
Multiplication and Summation = $(50 \times 30) + (50 \times 30) + (50 \times 30) + (20 \times 30) + (50 \times 30) = 6600$ (A large number!)

The curve is
detected!

Ref.: <https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>

Understanding convolution (3)

- Move the filter to another part of the image
- The value is much lower (0) – the curve is not detected
- There wasn't anything in the image region that responded to the curve detector filter

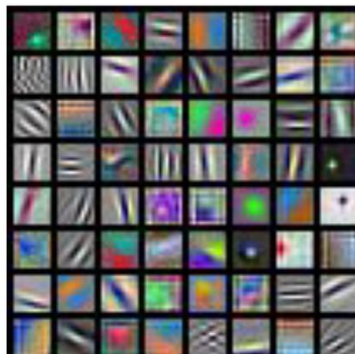


Multiplication and Summation = 0

The curve is not detected!

- The feature map will show the image regions in which there are curves (specific features)
 - High values: there is some curve which activated the filter
 - Low values: there is no curve
- We can use more than one filter – we will detect different features
- The filters convolve with the image parts and “activate” (have high values) when a specific feature is detected

First-layer conv filters: local image templates
(Often learns oriented edges, opposing colors)



AlexNet: 64 filters, each 3x11x11

Image ref.: <http://cs231n.stanford.edu/>

- In CNNs, the filters are not pre-designed to detect a particular type of feature, **their values are learned** during training with the backpropagation algorithm
- A typical convolutional layer includes many filters, each detecting a different feature
 - => Many feature maps: feature map 1 created by filter 1, feature map 2 created by filter 2 etc.

- In our example, we moved the filter along the image 1 pixel at a time;
- When we reached the end of the image, the filter returned to the left but 1 pixel lower
- We say that the **horizontal** and **vertical strides** were both 1: $s_h=1$, $s_v=1$
- Different strides can be used – illustration of $s_h=2$, $s_v=2$:

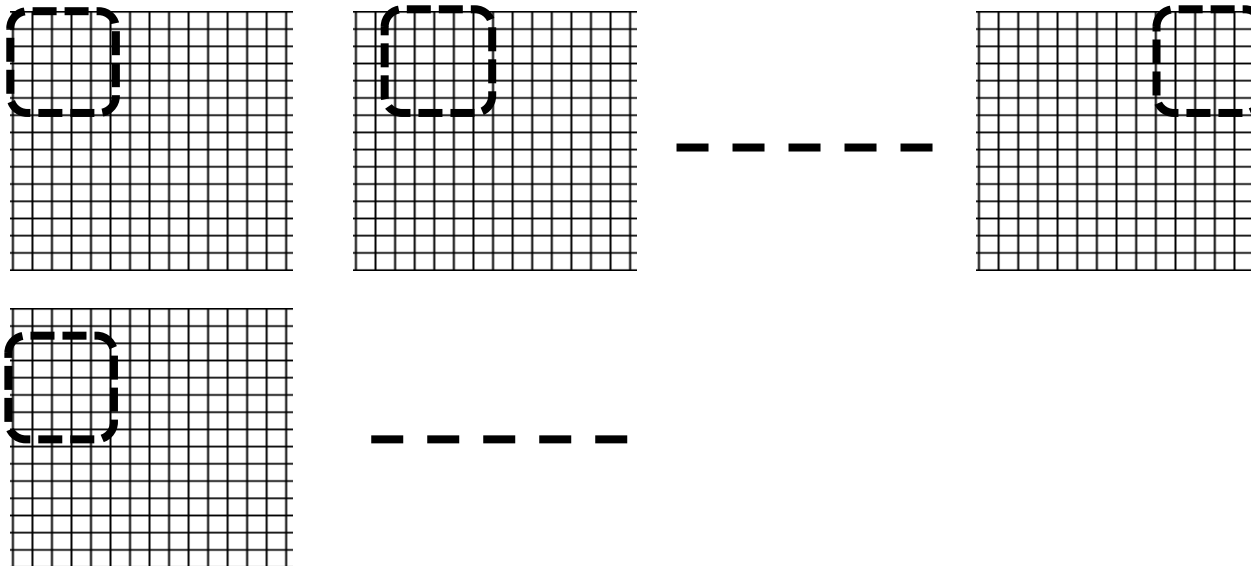


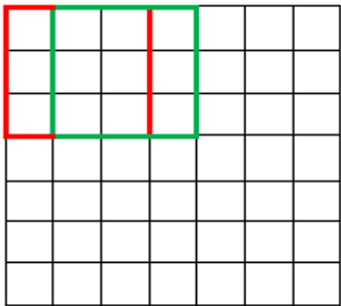
Image ref.: M. Kubat, Introduction to ML, Springer

Strides – another example

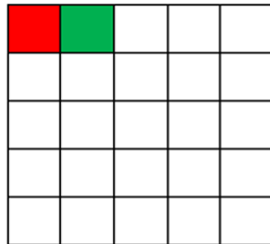
- Bigger strides produce smaller feature maps

Stride = 1 (filter = 3 x 3)

7 x 7 image

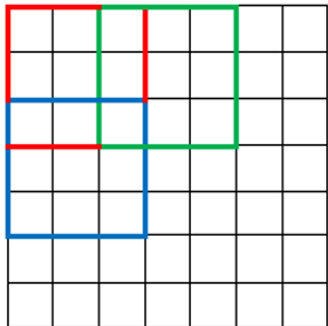


5 x 5 feature map

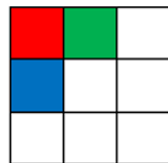


Stride = 2 (filter = 3 x 3)

7 x 7 image



3 x 3 feature map

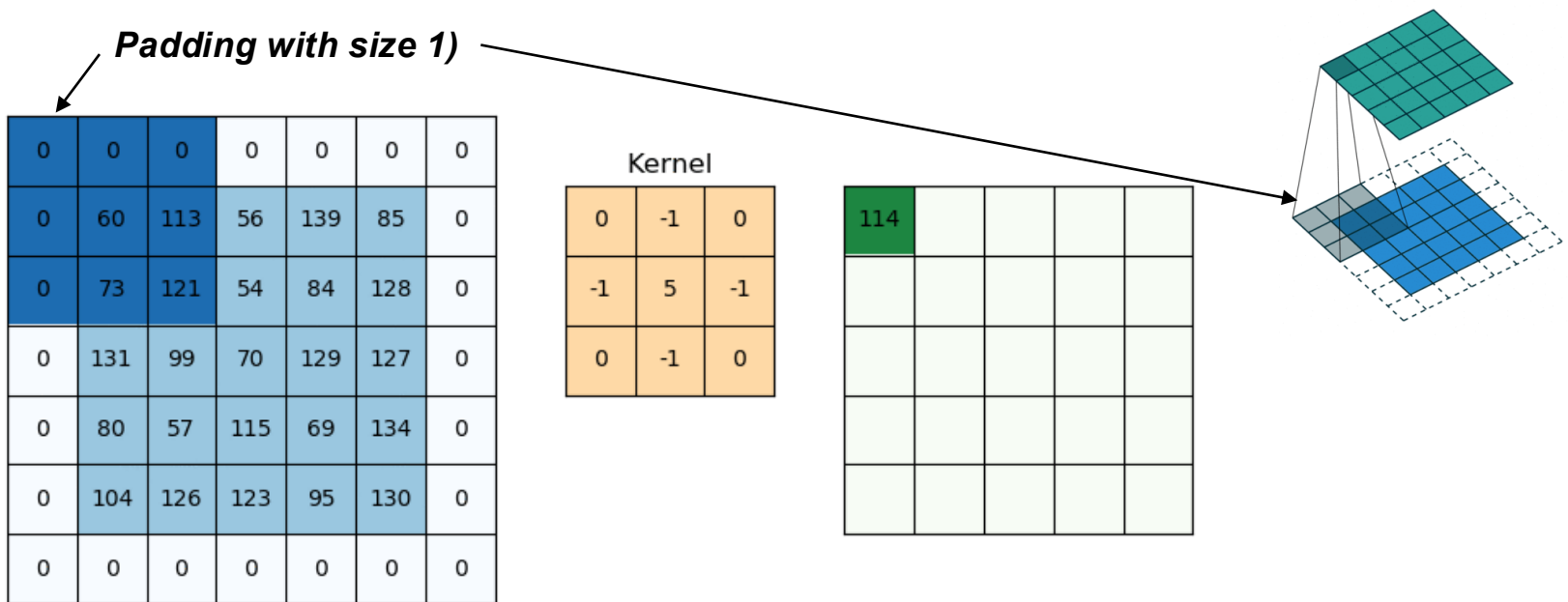


- Motivation: at the end of the row (or column), the filter may “reach beyond” the original image
- If the next stride would make the filter reach beyond the image, fill the fields outside the image with zeros
 - This means adding zeros around the border
 - The size of the padding is a hyperparameter
- Padding allows for better coverage of the image around the image and feature extraction



Padding - example

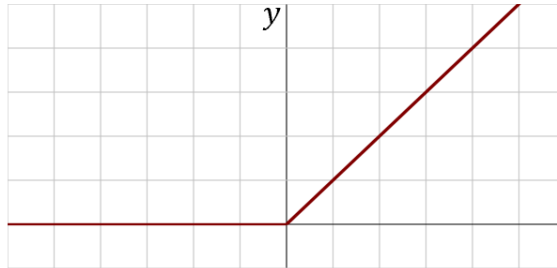
- An example of padding with size 1



- We apply an activation function to the feature maps, typically ReLU
- ReLu is simpler than sigmoid or tanh => faster computation
- Simplifies the gradient-descent calculation
- No upper bound, no saturation of the output, less vanishing gradient

ReLU

$$y = \max(0, x)$$



- The exact location of a feature is not so important; we just want to know where approximately it is and how it relates to other features
- We can use a **pooling** layer (also called sub-sampling layer)
 - “Summarizes” the input from the previous layer
 - Reduces the number of parameters (by 75% in the example below) and helps prevent overfitting
 - Improves robustness to shifts in the receptive field

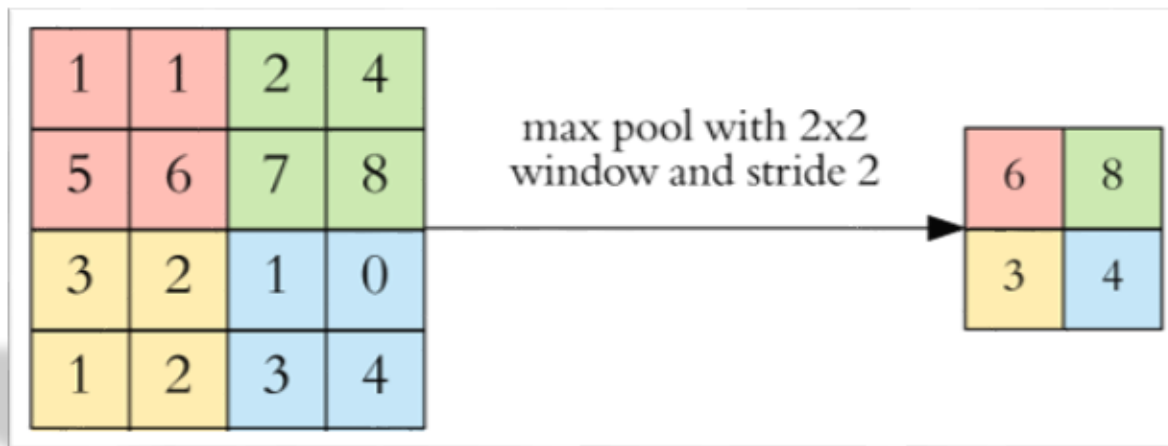
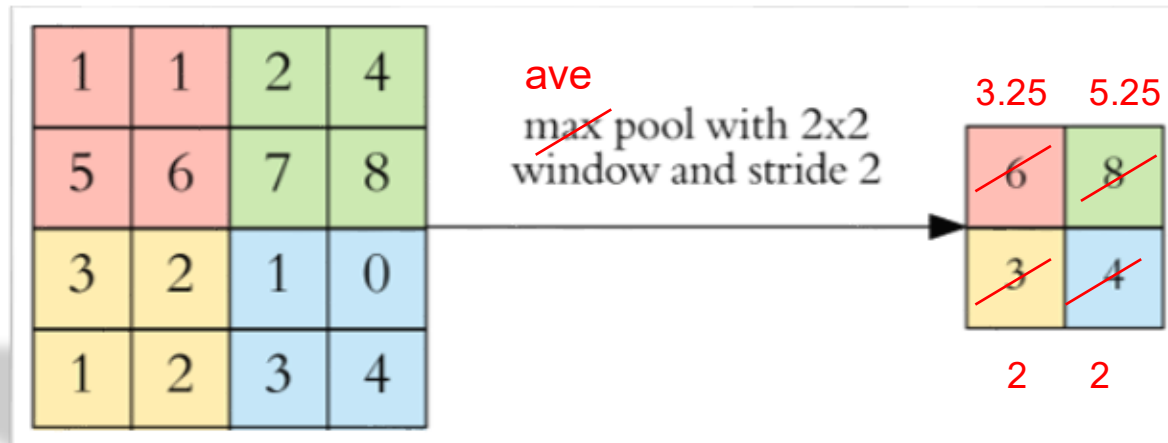


Image ref.: M. Kubat, Introduction to ML, Springer

Two versions of pooling

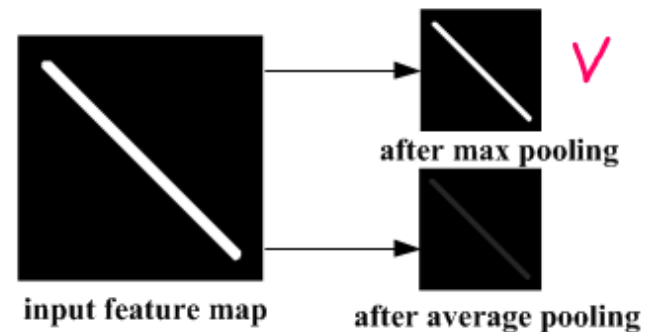
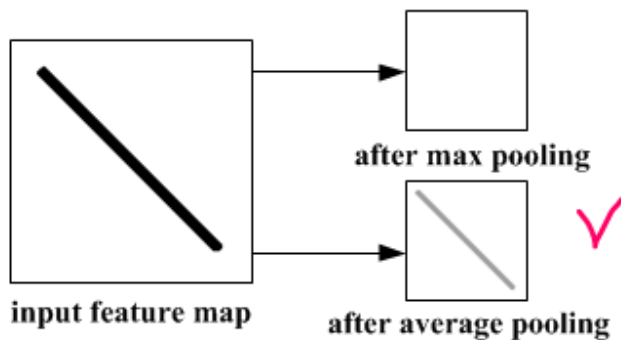
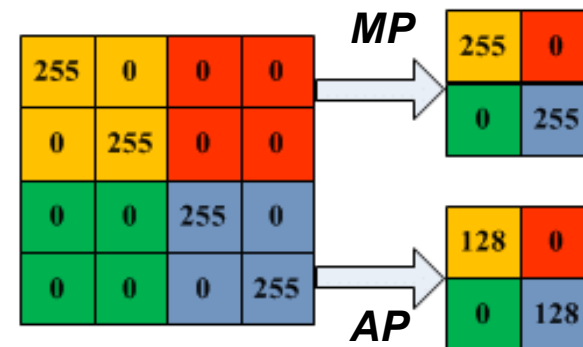
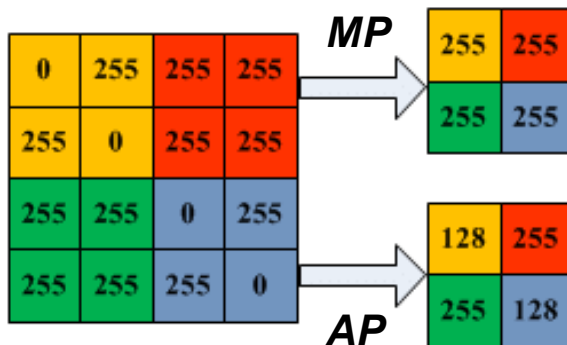
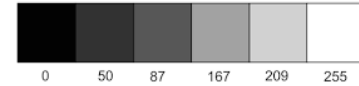
- **Max pooling** (max pool) – take the maximum value of the region
- **Average pooling** (ave pool) – take the average value of the image region



Which type of pooling will be better?

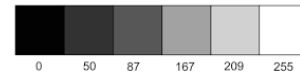
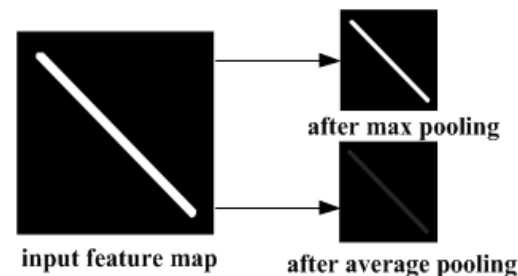
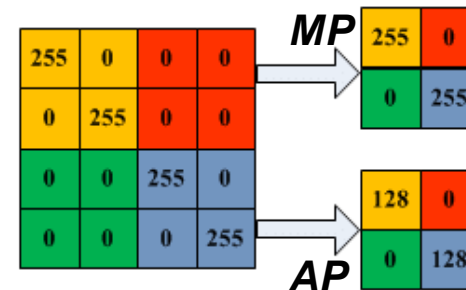
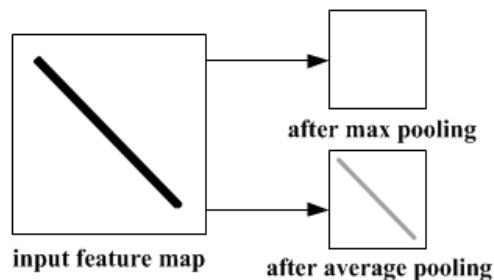
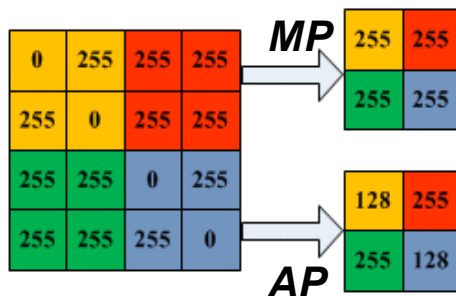
Max Pooling (MP)

Average Pooling (AP)

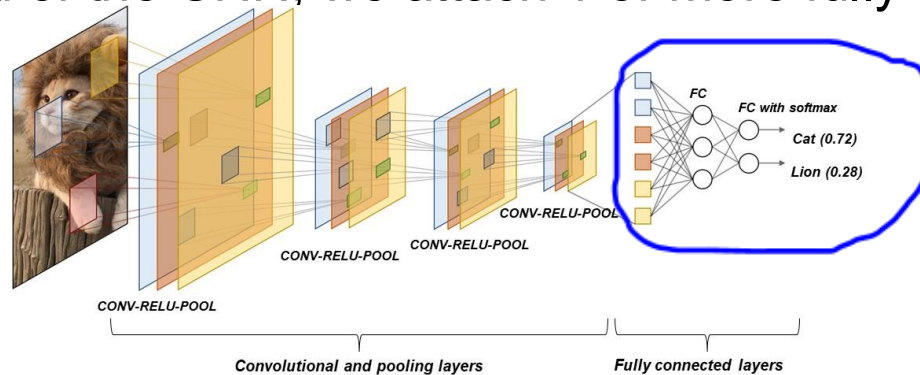


Which type of pooling will be better? (2)

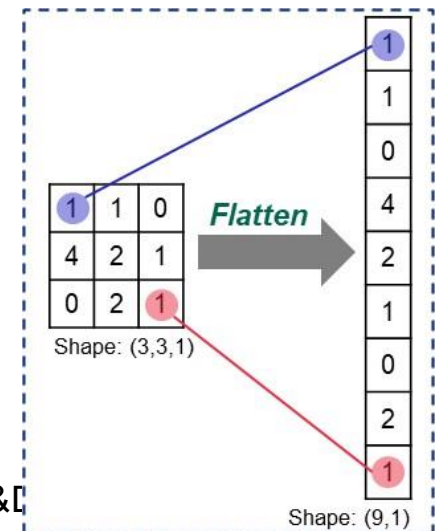
- MP or AP - it depends on the task
- MP selects the max value (brightest pixel) -> useful for images with dark background, when we are interested in the lighter pixels, e.g. MNIST dataset – white digits on a dark background
- AP smooths the image



- At the end of the CNN, we attach 1 or more fully connected (FC) layers



- The first FC layer takes the output of the last layer which is either convolutional or pooling (conv are not always followed by pool)
- The first thing to do is to “flatten” the output -
- transform the multi-dimensional matrix of features into a vector that can be fed in the FC layer

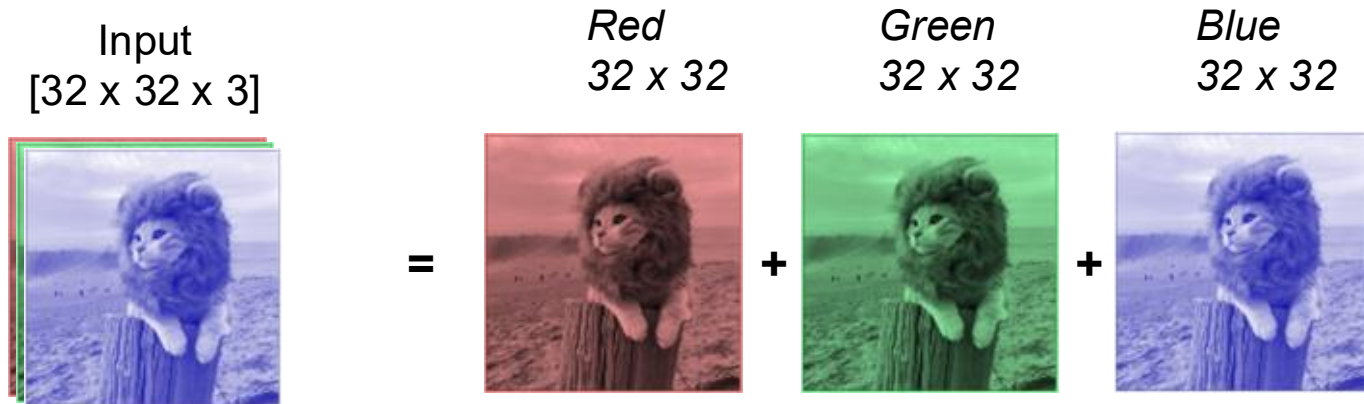


- The last FC layer outputs a n -dimensional vector where n is the number of classes. It typically uses a softmax function to output probabilities for each class.
 - E.g. if the task is digit classification and the resulting output is $[0.1, 0, 0, 0.1, 0, 0, 0.6, 0, 0, 0.2]$, this means 10% probability that the image is “1”, 10% that it is “3”, 60% that it is “6” and 20% that it is ‘9’
 - The probabilities sum up to 1
- Essentially, the FC layers looks at the features extracted by the convolutional layers, combine them and output the probabilities for each class



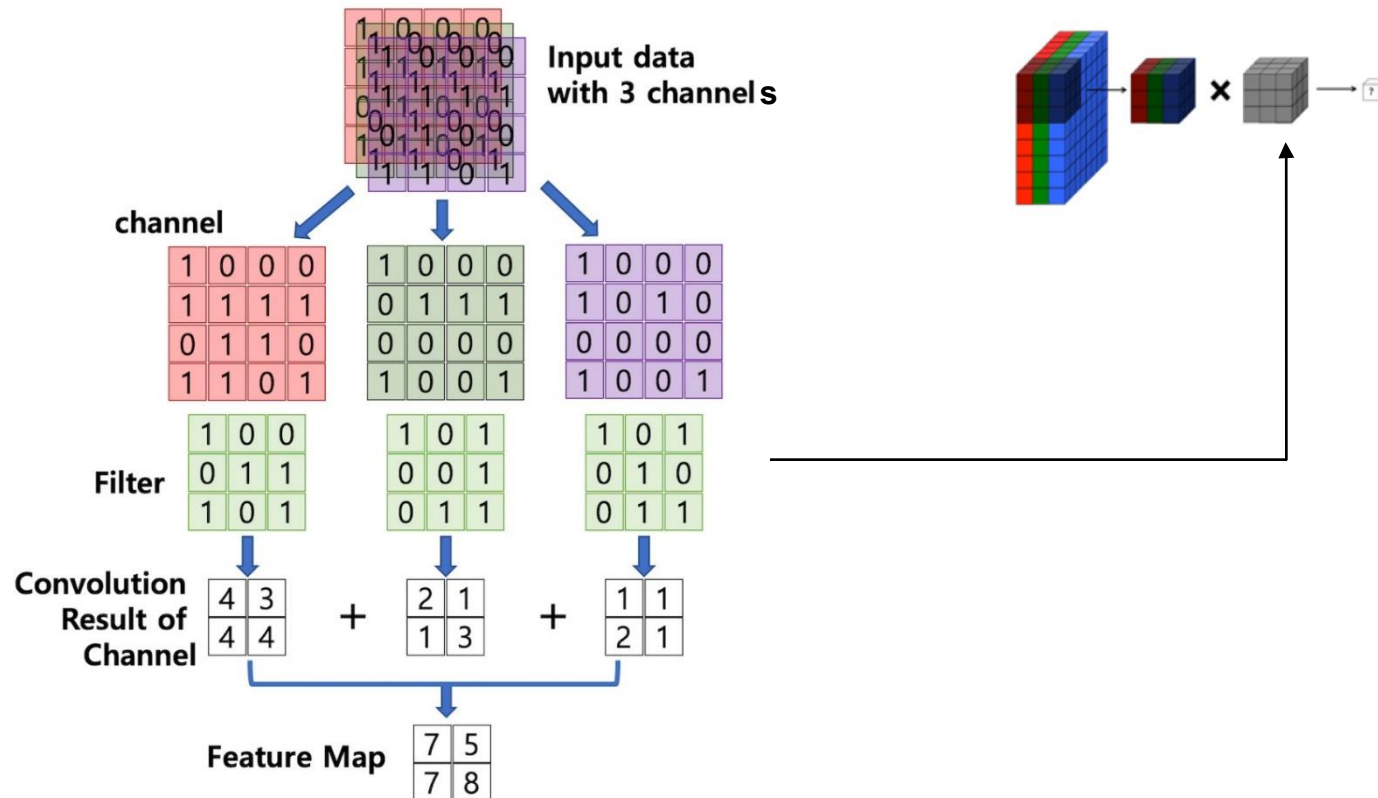
Extending CNNs to multi-channel input

- Color images have 3 channels: **Red**, **Green** and **Blue**



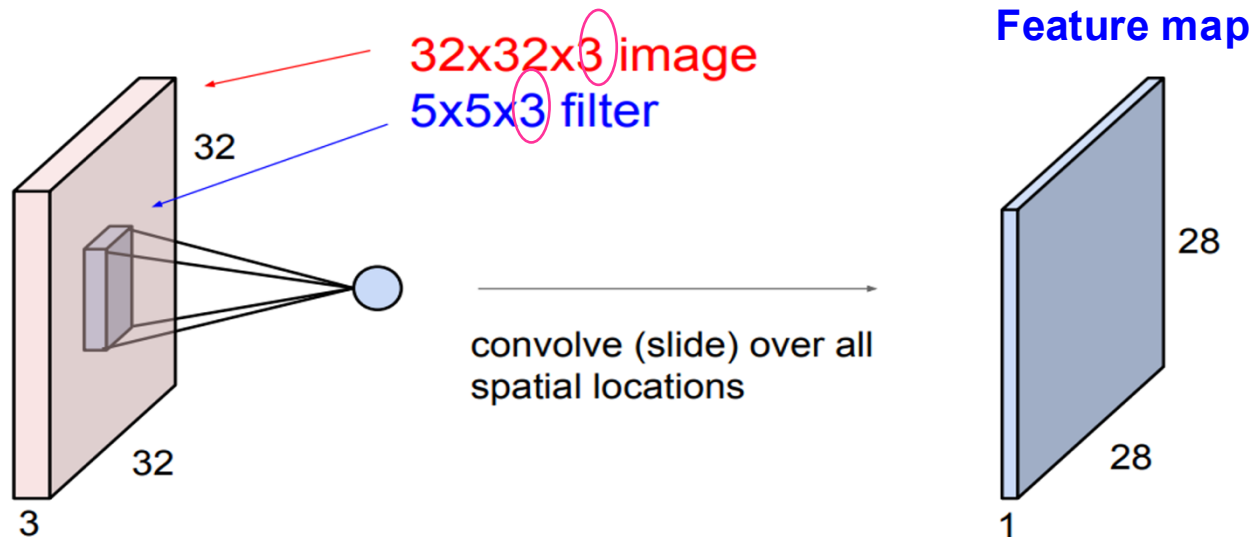
Extending CNNs to multi-channel input (2)

- We can modify the CNN architecture to work with multiple channels



Extending CNNs to multi-channel input (3)

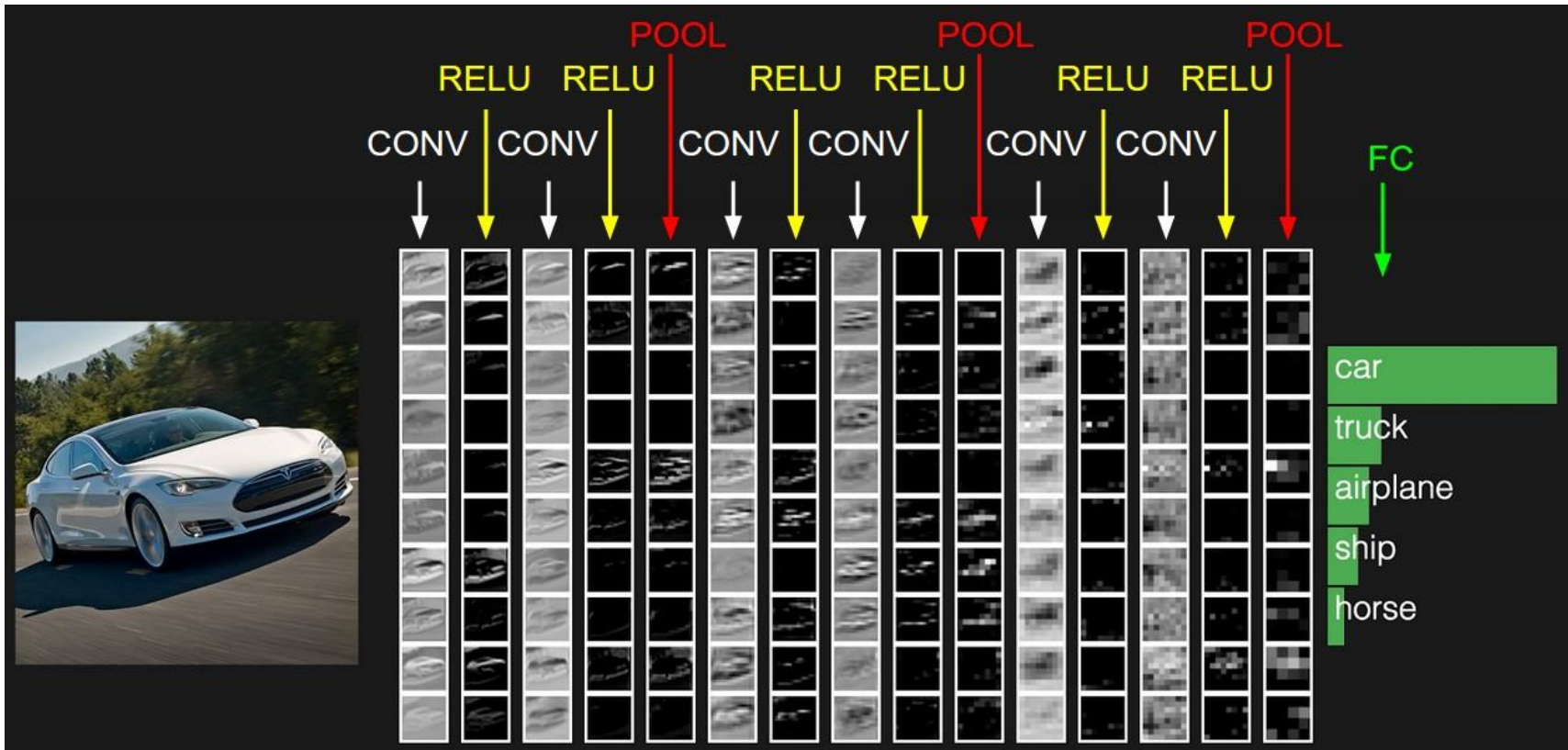
- Using multi-channel data increases the dimension of the filters too - the third dimension should be the same
- Example: 3-dim data (3 color channels) $\rightarrow$ 3-dim filter; the third dimension should be the same – 3 in this case





CNN architecture - example

INPUT – CONV – RELU – POOL – FC



Web-based demo: <http://cs231n.stanford.edu/>

- What is the right way to combine the CNN components?
- How many convolution layers?
- How many filters in each of these layers?
- What should be the size of these filters (e.g., 3-by-3 or 5-by-5)?
- How frequent should be the pooling layers? After each convolution layer? Or less frequent?
- What should be the size of the pooling layers? 2-by-2 or 3-by-3? Other dimensions? Should they perform max-pooling or ave-pooling?



CNN architectures

- Task: Handwritten digit recognition
 - LeCun, L. Bottou, Y. Bengio and P. Haffner (1998), Gradient-based learning applied to document recognition, Proceedings of the IEEE.
 - <http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf>
- Architecture: 2 convolution, 2 max-pooling, 3 fully connected

PROC. OF THE IEEE, NOVEMBER 1998

7

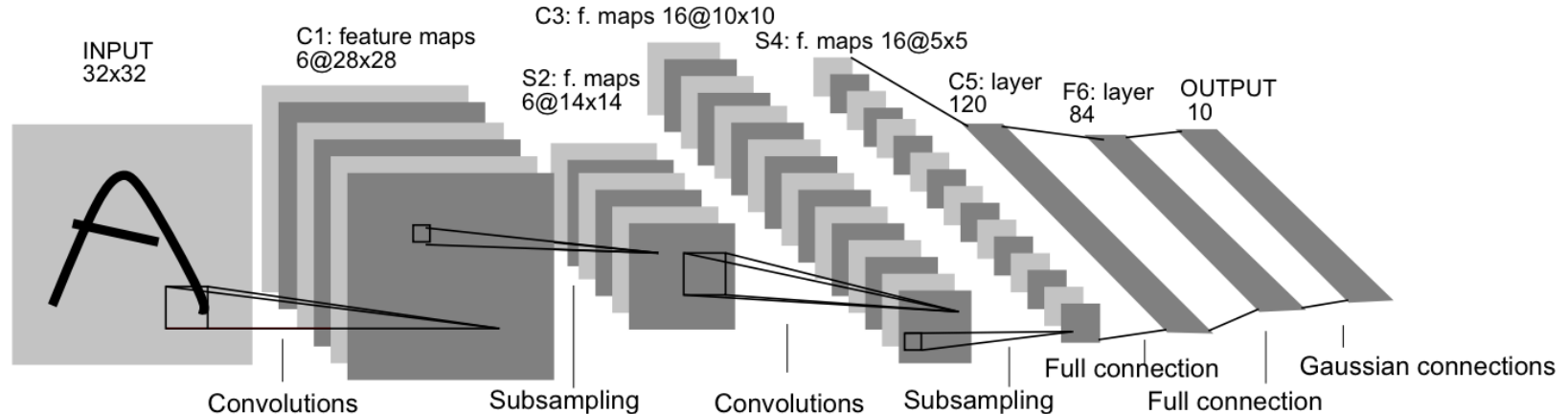


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

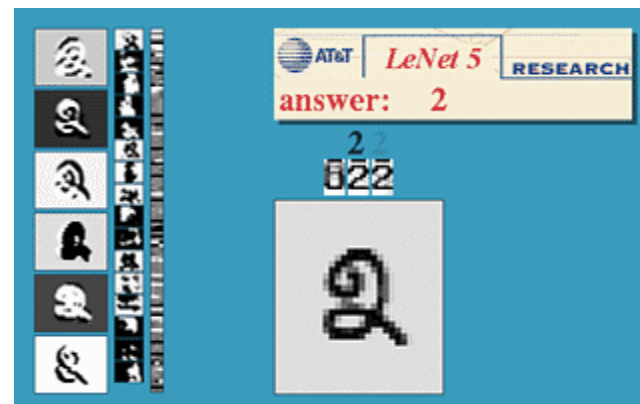
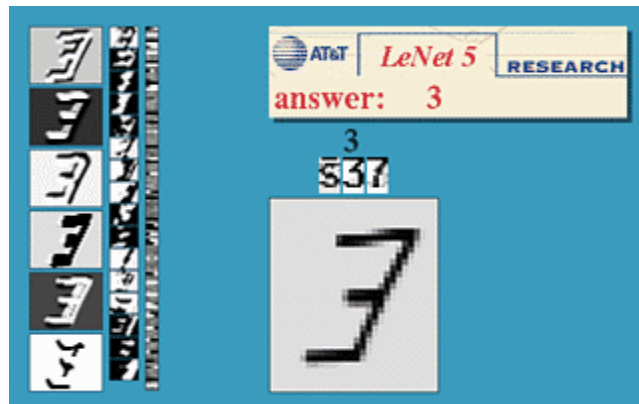
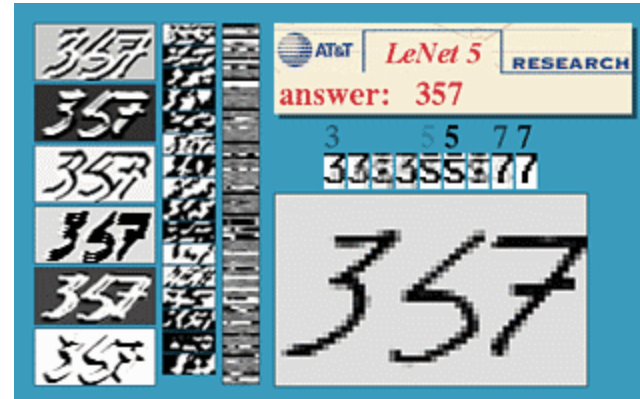
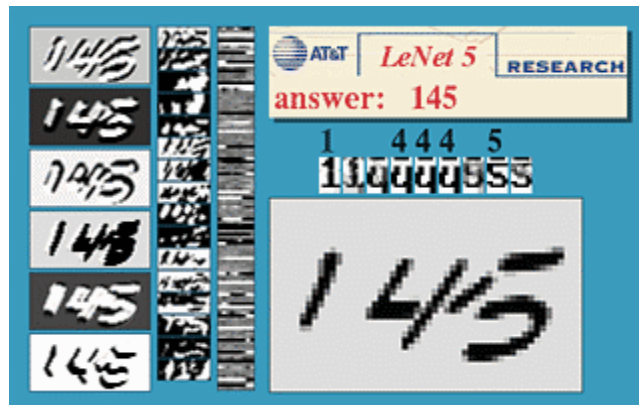


Image from <http://yann.lecun.com/exdb/lenet/multiples.html>

- Task: Classification of images into different categories
- ImageNet and ImageNet Large Scale Visual Recognition Challenge - facilitated important advances in computer vision
 - A. Krizhevsky, I. Sutskever, G. Hinton (2012), ImageNet classification with deep convolutional neural networks, Proceedings of NIPS
 - <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>
- Training data: 1.2 million images from ImageNet dataset labelled in 1000 classes



Image from http://vision.stanford.edu/resources_links.html

- Input: 227 x 227 images
- 5 convolutional layers, 3 max pooling layers, 3 fully-connected layers

Architecture:

CONV1

MAX POOL1

NORM1

CONV2

MAX POOL2

NORM2

CONV3

CONV4

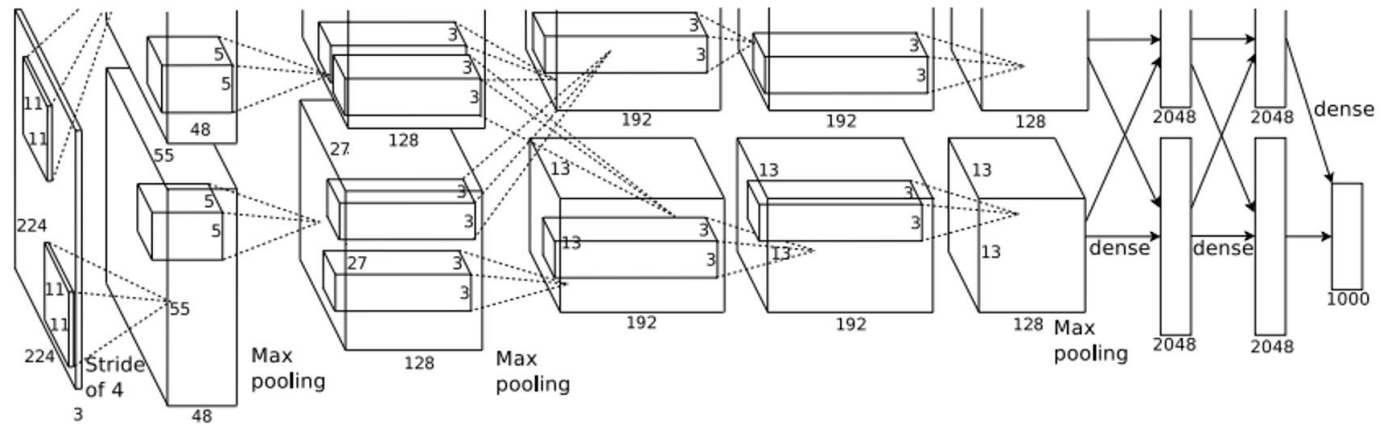
CONV5

Max POOL3

FC6

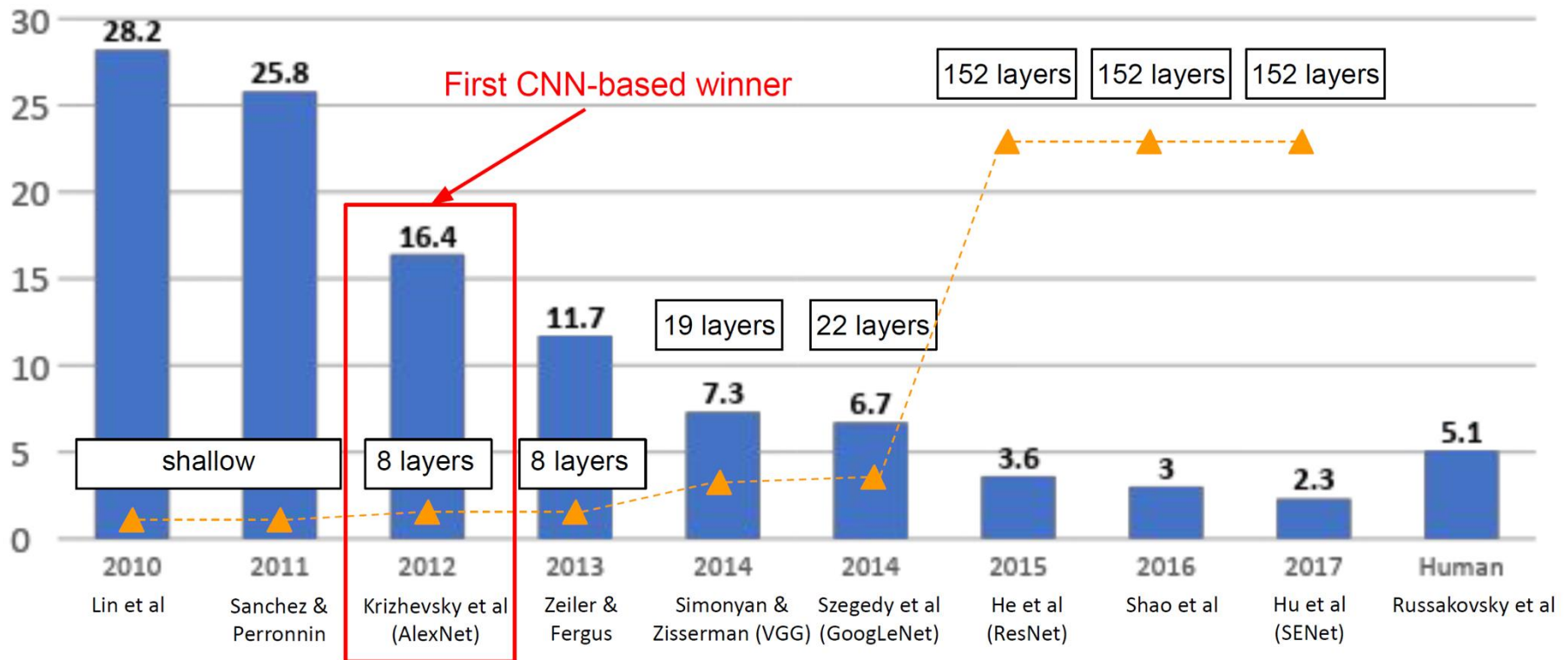
FC7

FC8



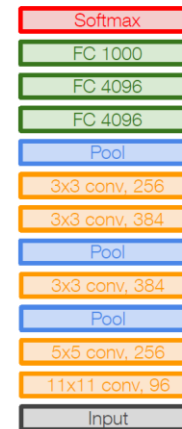
<http://cs231n.stanford.edu/>

- ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



<http://cs231n.stanford.edu/>

- Simonyan and Zisserman, 2014; <https://arxiv.org/abs/1409.1556>
- An improved version of AlexNet
- Deeper network, simpler design
- Convolutions: 3x3 filters, stride=1, padding=1
- Max pooling: 2x2, stride=2



AlexNet

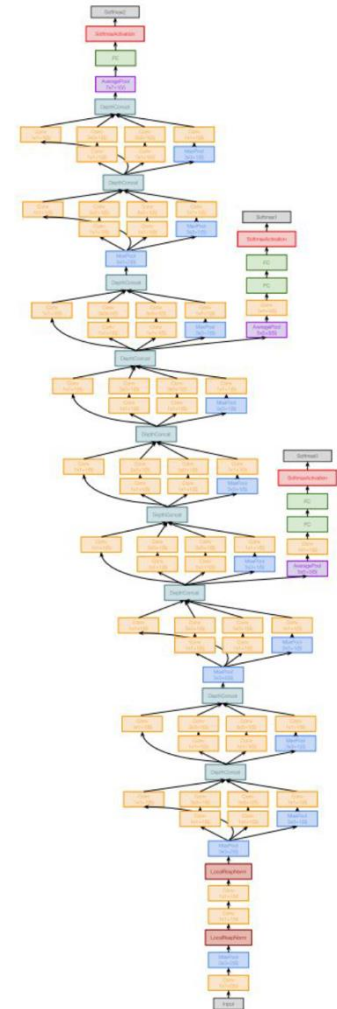
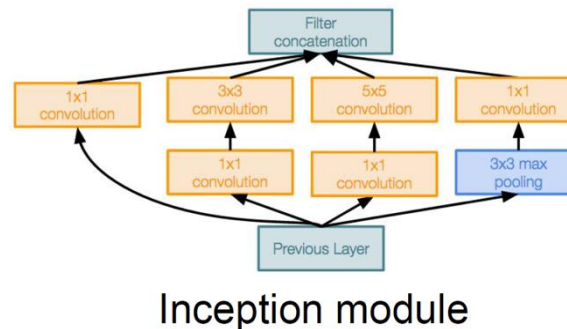


VGG16

VGG19

<http://cs231n.stanford.edu/>

- Szegedy et al., 2014; <https://arxiv.org/abs/1409.4842>
- Deeper networks, with computational efficiency
- - ILSVRC'14 classification winner - 6.7% error
- 22 layers
- Only 5 million parameters!
 - 12x less than AlexNet and 27x less than VGG-16
- Efficient “Inception” module
- No FC layers



<http://cs231n.stanford.edu/>

- CNNs are designed for image data but can be applied to other data types
- Key elements: convolution, pooling and fully connected layer
- The convolutional layers extract features
- The pooling layers summarize the information from the convolutional layers, reduce the number of parameters to prevent overfitting and improve the robustness to shifts in the data
- The fully connected layer combines the extracted features and outputs the probabilities for each class
- CNNs have been very successful in image processing and computer vision tasks

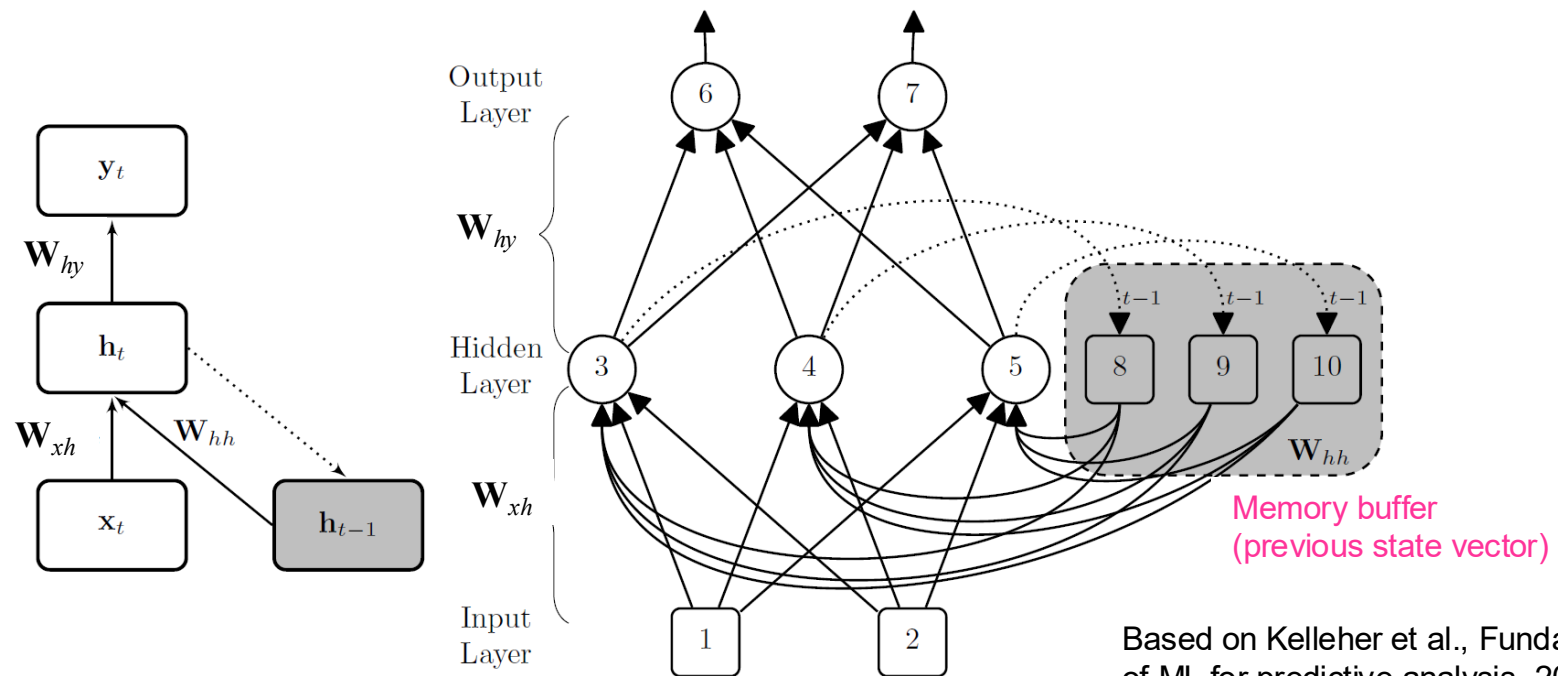


Recurrent Neural Networks

- Recurrent Neural Networks (RNN) are used for **sequential data**
- Text (natural language) data is an example of sequential data
 - Sentences are sequences of words – 1 word follows the other
 - Each sentence may have a different number of words (varying length)
 - Each sentence may contain long-distance dependencies between the words
 - The **dog** in that house **is** aggressive
 - The **dogs** in that house **are** aggressive
- Processing this type of data requires a model that can **remember** relevant information from the earlier in the sequence (needs **memory**)
- RNNs can do this

- When processing the sequence, a RNN takes 1 element from the sequence at each time point, e.g. 1 word:
 - The dog in the house is aggressive
 - at time t : the, $t+1$: dog, $t+2$: in, $t+3$: the, $t+4$: house, $t+5$: is, $t+6$: aggressive
- It is called **recurrent** because it contains feedback connections (e.g. from the hidden layer to the input layer)
 - $\Rightarrow$ feedforward NN is a directed **acyclic** graph; RNN is a directed **cyclic** graph
- Specifically, the output of a neuron at one time point is fed back into the same neuron at the next time point (or another time point)
- Consequence: RNN has a memory over past activations (and hence past inputs which contributed to these activations)
- $\Rightarrow$ RNN can capture long-distance dependences $\Rightarrow$ useful for sequences
- There are different architectures; we will cover:
 - Simple RNN (also called Elman network)
 - Long Short-Term Memory (LSTM) network

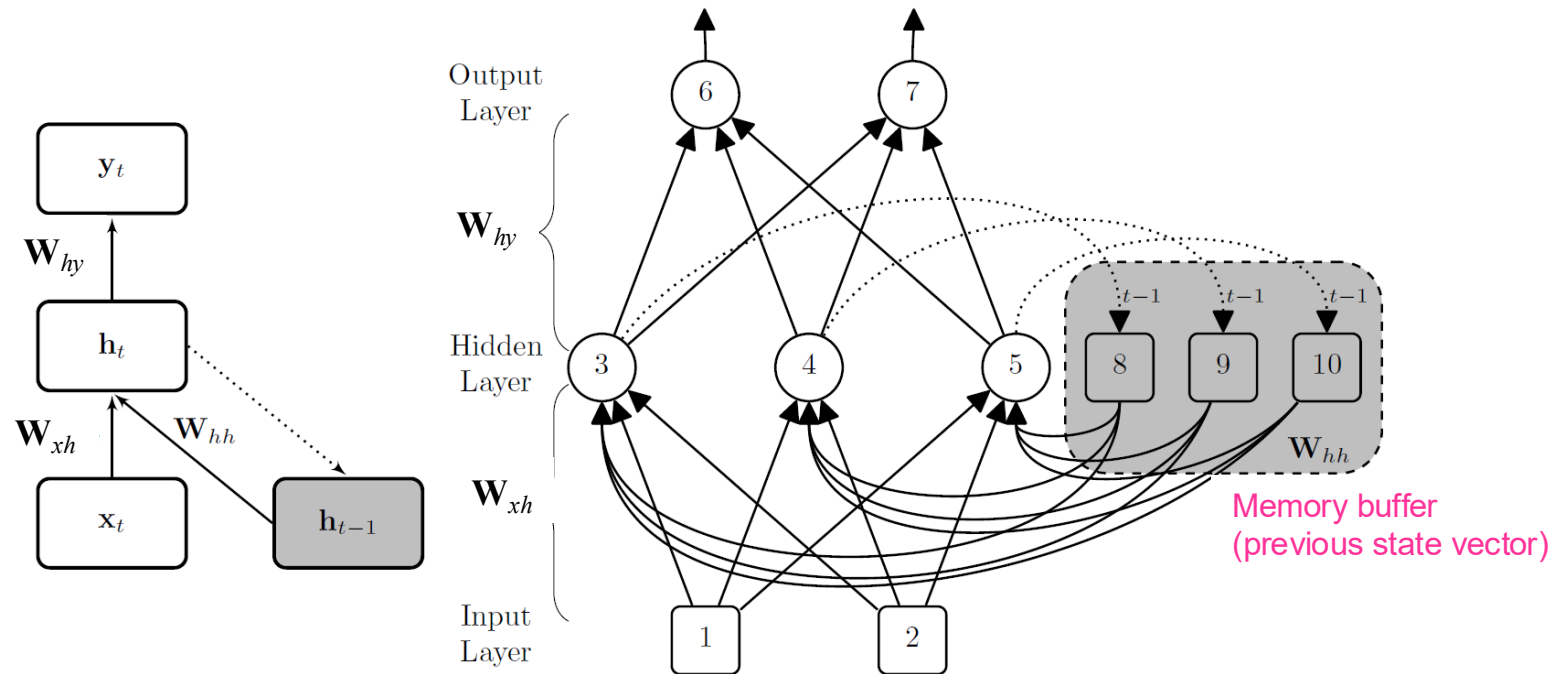
- A feedforward architecture with 1 hidden layer
- The hidden layer is extended with a **memory buffer** which stores the **previous state** of the hidden layer (from the previous time-step)
- At each time-step, the information from the memory is concatenated with the next input to each neuron



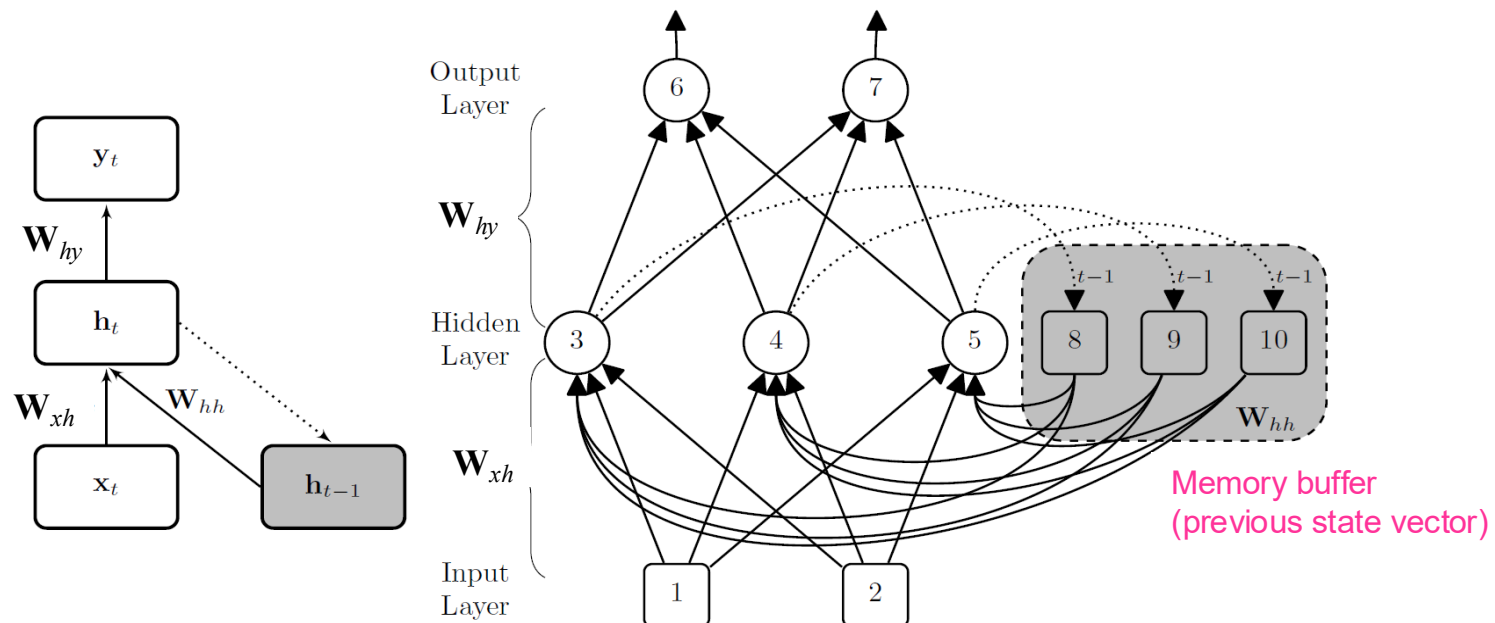
Based on Kelleher et al., Fundamentals of ML for predictive analysis, 2020, MIT.

Input, hidden and output layers

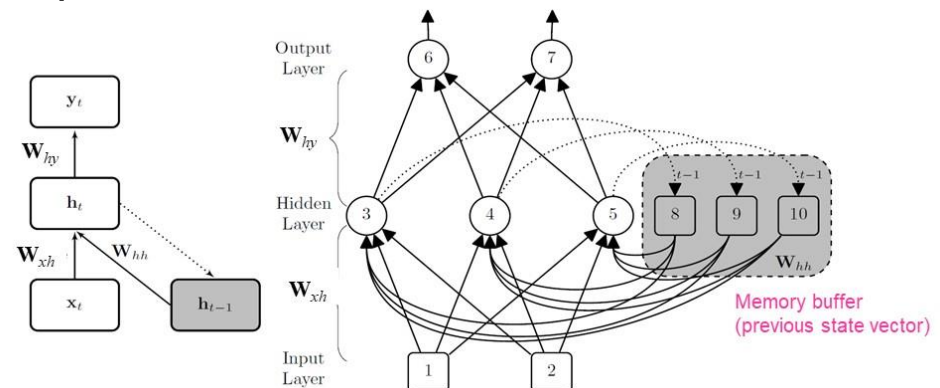
- x_t – input vector at time t
- h_t - hidden layer activations at time t
- y_t – output vector at time t
- E.g. given word at t (x_t), predict the next word – the word at $t+1$ (y_t)



- 3 weight matrices:
 - W_{xh} - weights between the input and hidden layer
 - W_{hy} - weights between the hidden and output layer
 - W_{hh} - weights between the memory buffer and the hidden layer (hh because these are weights of the recurrent connections from the hidden layer back to the hidden layer)



- Let's consider a specific example:
 - 2 neurons in the input layer, 3 in the hidden and 2 in the output layer
- At each time-step a new input vector is presented
- Each hidden neuron
 - 1) Receives both the input vector and the previous state information from the memory buffer
 - 2) Processes this information and generates activations that are passed to the output layer and also written to the memory buffer
- At the next time-step, the information from the memory buffer is fed back to the hidden neurons, together with the new input



Forward pass - calculating output

New state Old state Input vector

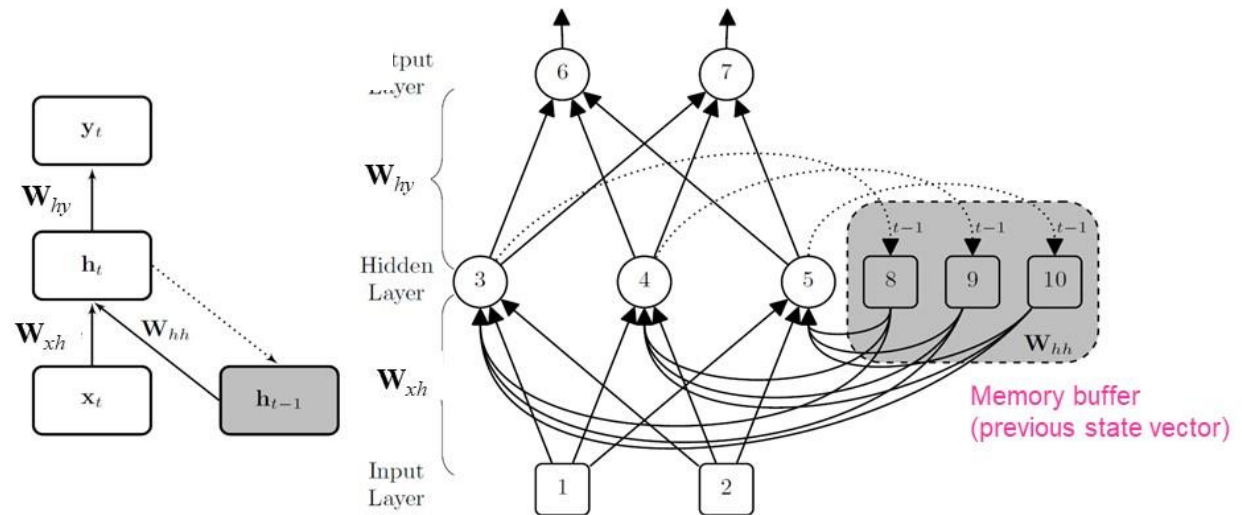
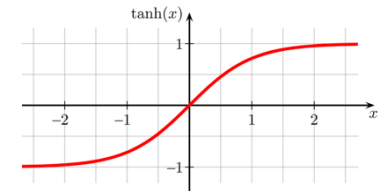
$$h_t = f_W(h_{t-1}, x_t)$$



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

f_W – function with parameters W ;
tanh is typically used: output
between 1 and -1



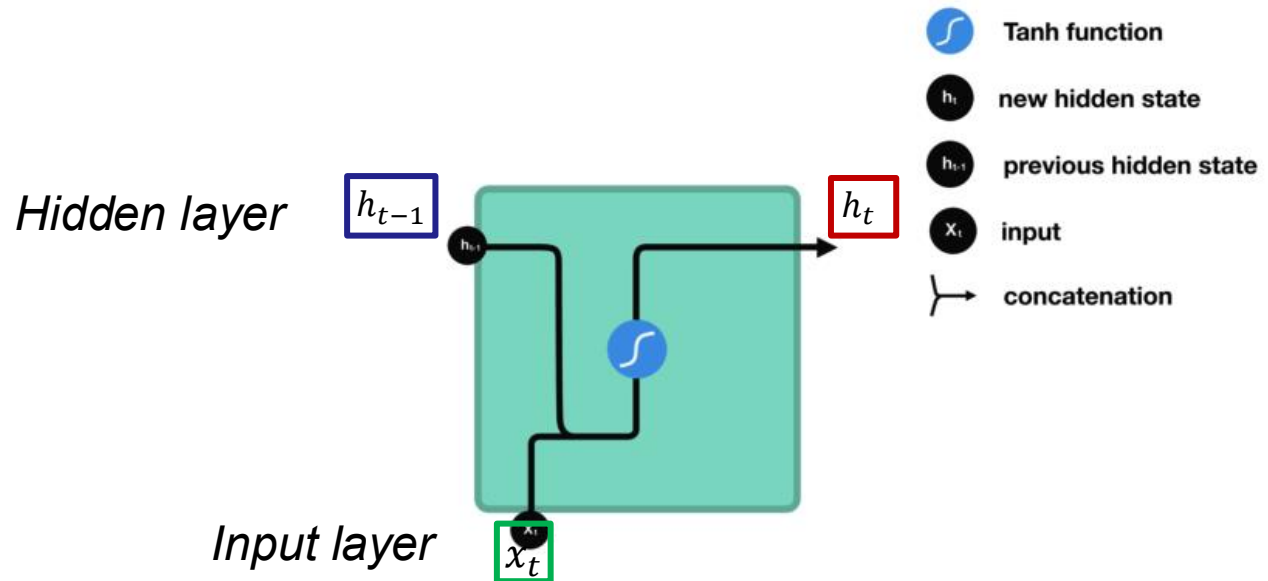
- Animation :-)

New
state

Old
state

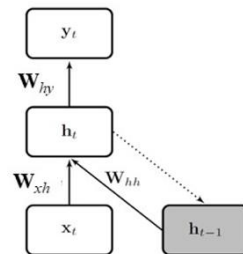
Input
vector

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$



<https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9>

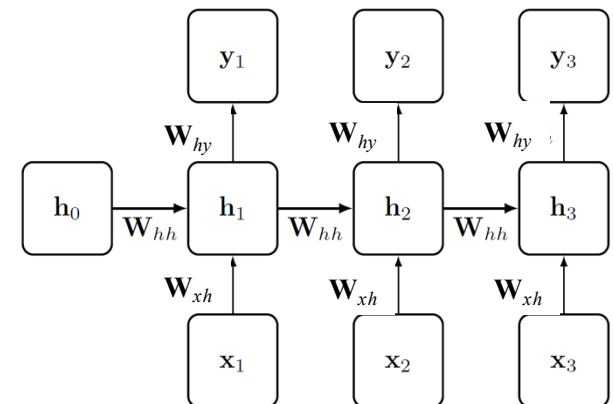
- As we saw, RNNs have cyclic connections - the output of the hidden neuron at time $t-1$ is fed back as input to the hidden neuron at time t



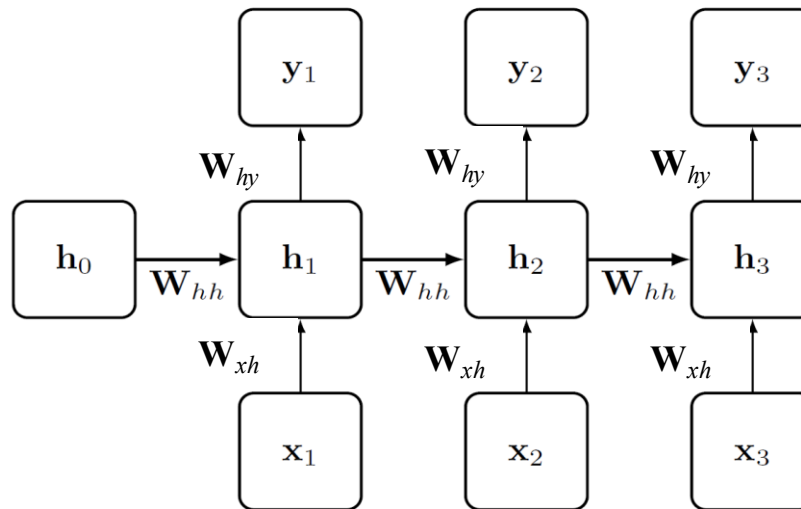
New state Old state Input vector

$$h_t = f_W(h_{t-1}, x_t)$$

- These cycles make it difficult to understand the information flow in RNN over many time steps
- Instead, we can visualize the information flow in another way – by “unrolling” (or “unfolding”) the RNN along the time steps
 - There will be no cycles in the graph
 - Important: the RNN is the same between the unfolded time steps – the weights do not change



- Suppose there are 3 time-steps – we can “unroll” the RNN through these 3 steps to calculate the output:



- h_0 is the initial state of the memory buffer when RNN is initialized
- Note: RNN has only 3 weight matrices, not $3 \times 3 = 9$ (3×3 time steps)
- Again, “unroll over time” means following the sequence of steps that are performed

Training RNN: backpropagation through time

- We can see that a RNN is an augmented feedforward NN (feedforward NN + memory). It has an additional matrix: W_{hh} .
- It can be trained with a variant of the backpropagation algorithm, called **backpropagation through time**
- The error at time t is backpropagated to all parameters that contributed to it, this means it is propagated through the previous states of the network. Hence the name - we are “going back in time”

Algorithm 3 The Backpropagation Through Time Algorithm

Require: h_0 initialized hidden state

Require: $\mathbf{x}$ a sequence of inputs

Require: $\mathbf{y}$ a sequence of target outputs

Require: n length of the input sequence

Require: Initialized weight matrices (with associated biases)

Require: $\Delta \mathbf{w}$ a data structure to accumulate the summed weight updates for each weight across time-steps

```
1: for  $t = 1$  to  $n$  do
2:    $Inputs = [x_0, \dots, x_t]$ 
3:    $h_{tmp} = h_0$ 
4:   for  $i = 0$  to  $t$  do
5:      $h_{tmp} = ForwardPropagate(Inputs[i], h_{tmp})$ 
6:   end for
7:    $\hat{y}_t = OutputLayer(h_{tmp})$ 
8:    $\mathcal{E}_t = \mathbf{y}[t] - \hat{y}_t$ 
9:    $Backpropagate(\mathcal{E}_t)$ 
10:  For each weight, sum the weight updates across the unrolled network and update  $\Delta \mathbf{w}$ 
11: end for
12: Update the network weights using  $\Delta \mathbf{w}$ 
```

▷ Unroll the network through t steps

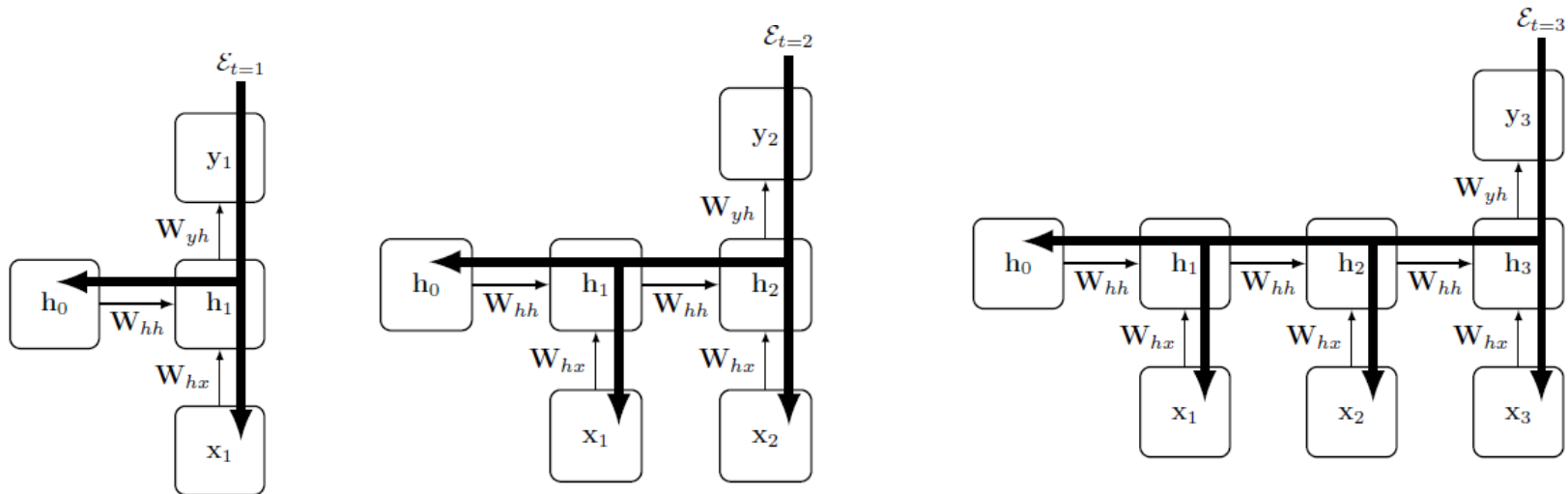
▷ Generate the output for time-step t

▷ Calculate the error at time-step t

▷ Backpropagate $\mathcal{E}_t$ through t steps

Kelleher et al., Fundamentals of ML for predictive analysis, 2020, MIT.

- The error at time t is backpropagated to all parameters that contributed to it
- Propagating the error for outputs y_1 , y_2 and y_3 :



Ref.: Kelleher et al., Fundamentals of ML for predictive analysis, 2020, MIT.

RNN example: predicting the next character

- Given a sequence of previous characters, predict the next one, i.e. given a sequence of characters at times 1, 2, ..., $t-1$, predict the character at time t
- Application: generating new text, 1 character at a time
- Simple example: assume vocabulary of 4 letters only = {h, e, l, o}
- Let's train RNN on the sequence "hello"!
- This means 4 training examples:
 - given "h", predict "e", given "he", predict "l"
 - given "hel", predict "l", given "hell", predict "o"
- Encode each character using 1-hot encoding, e.g. "h" is 1000, "e" is 0100
- => Input vector is 4-dim
- Output vector is 4-dim – 1 value for "h", 1 for "e", 1 for "l" and 1 for "o"; the highest activation determines the predicted letter
- Let's use 3 hidden neurons
- We feed the characters 1 at a time

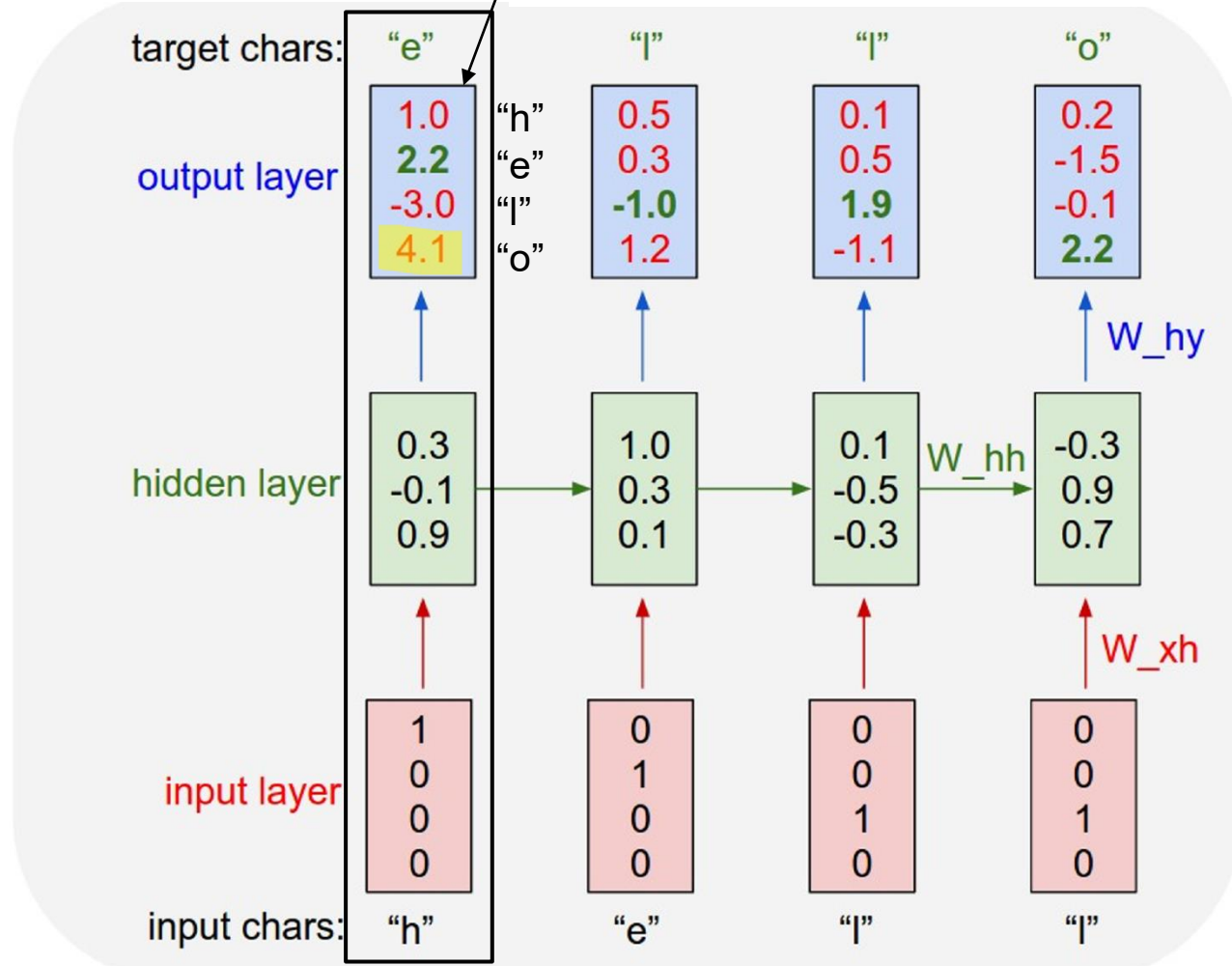
Example and images from:

<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

<http://cs231n.stanford.edu/>

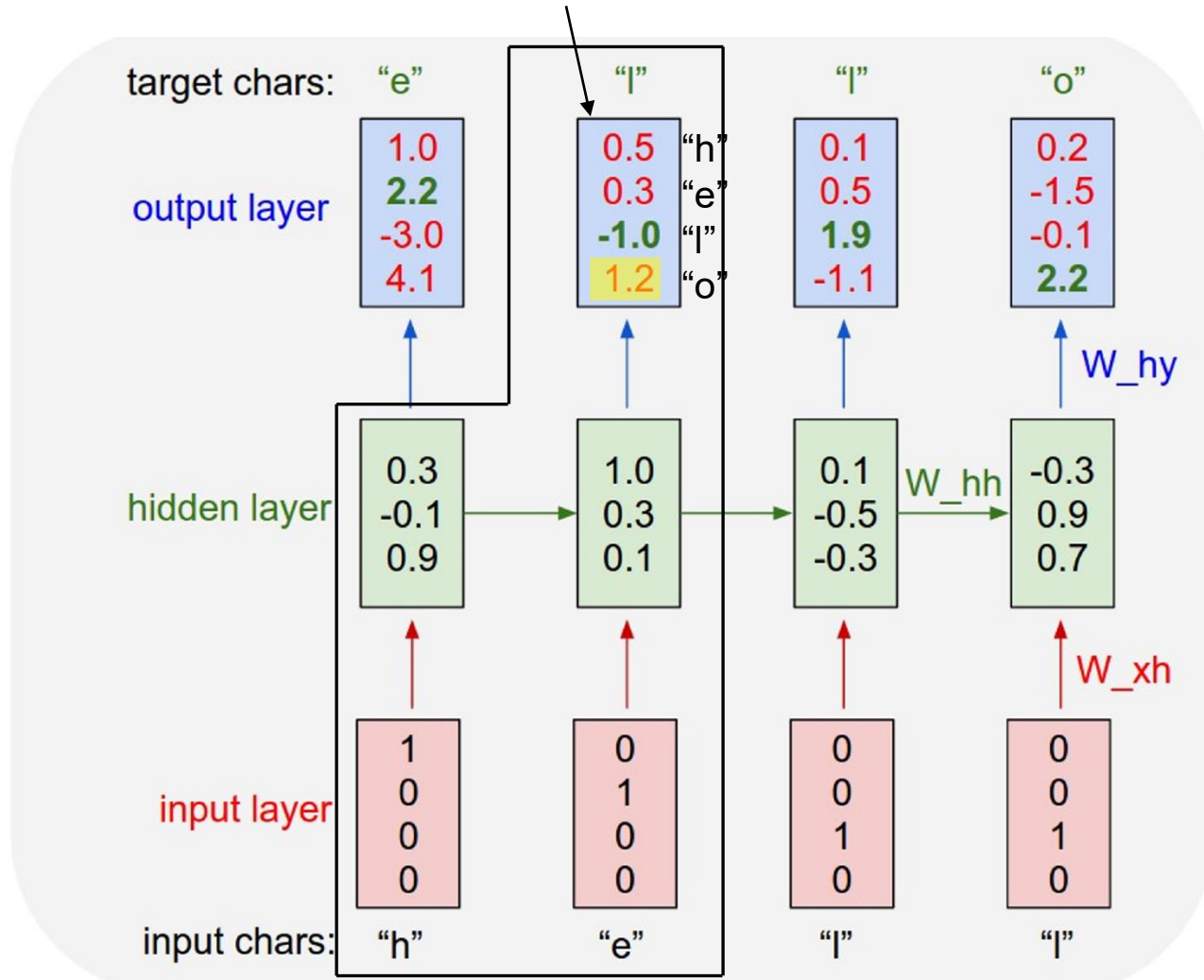
- Given “h”, predict “e”

Probability for each class; in green – for the correct class, in red – for the incorrect. Highest value is 4.1 for “o”; RNN predicts “o” for input “h” – incorrect.



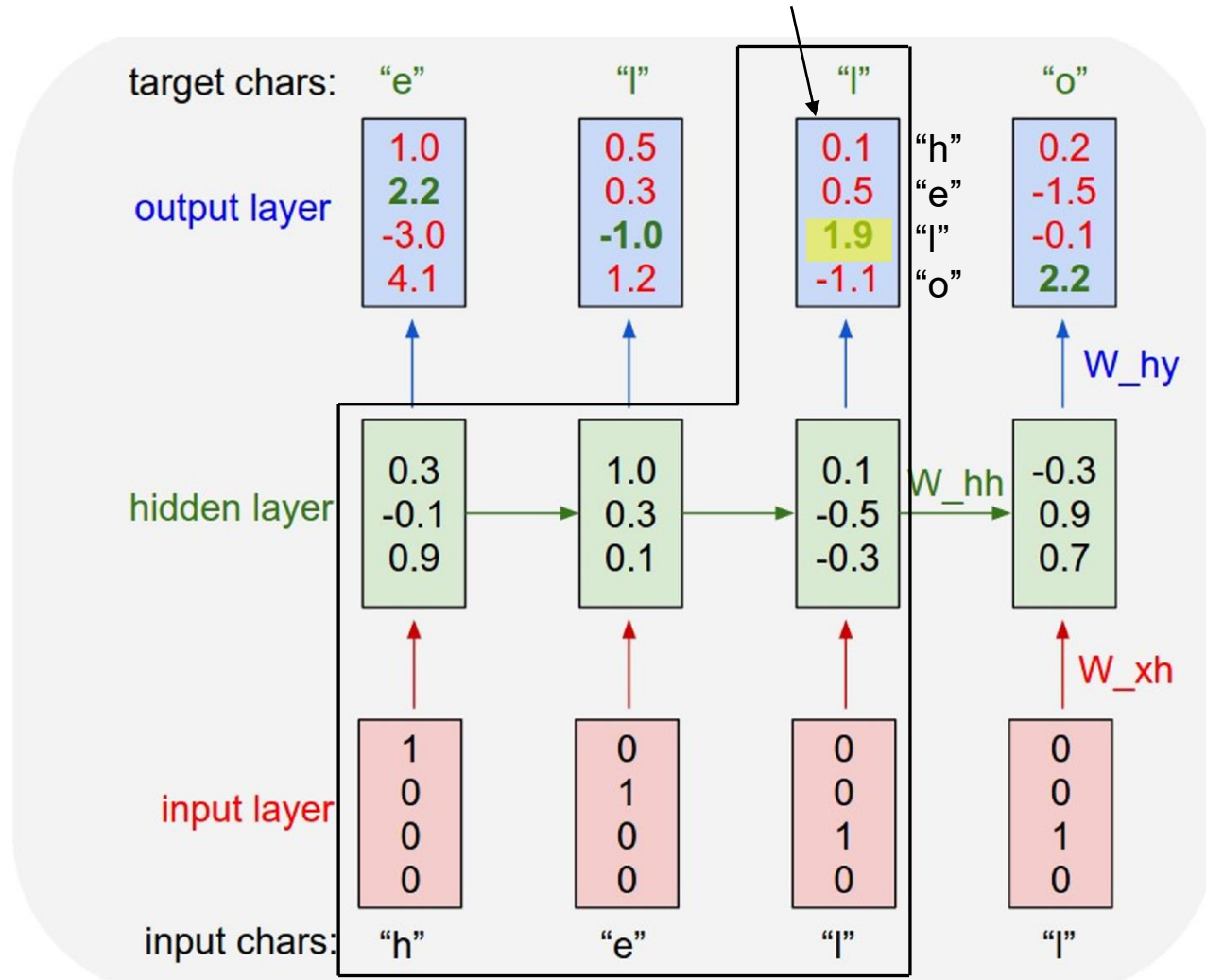
- Given “he”, predict “l”

Highest value is 1.2 for “o”. RNN predicts “o” for input “he” – incorrect.



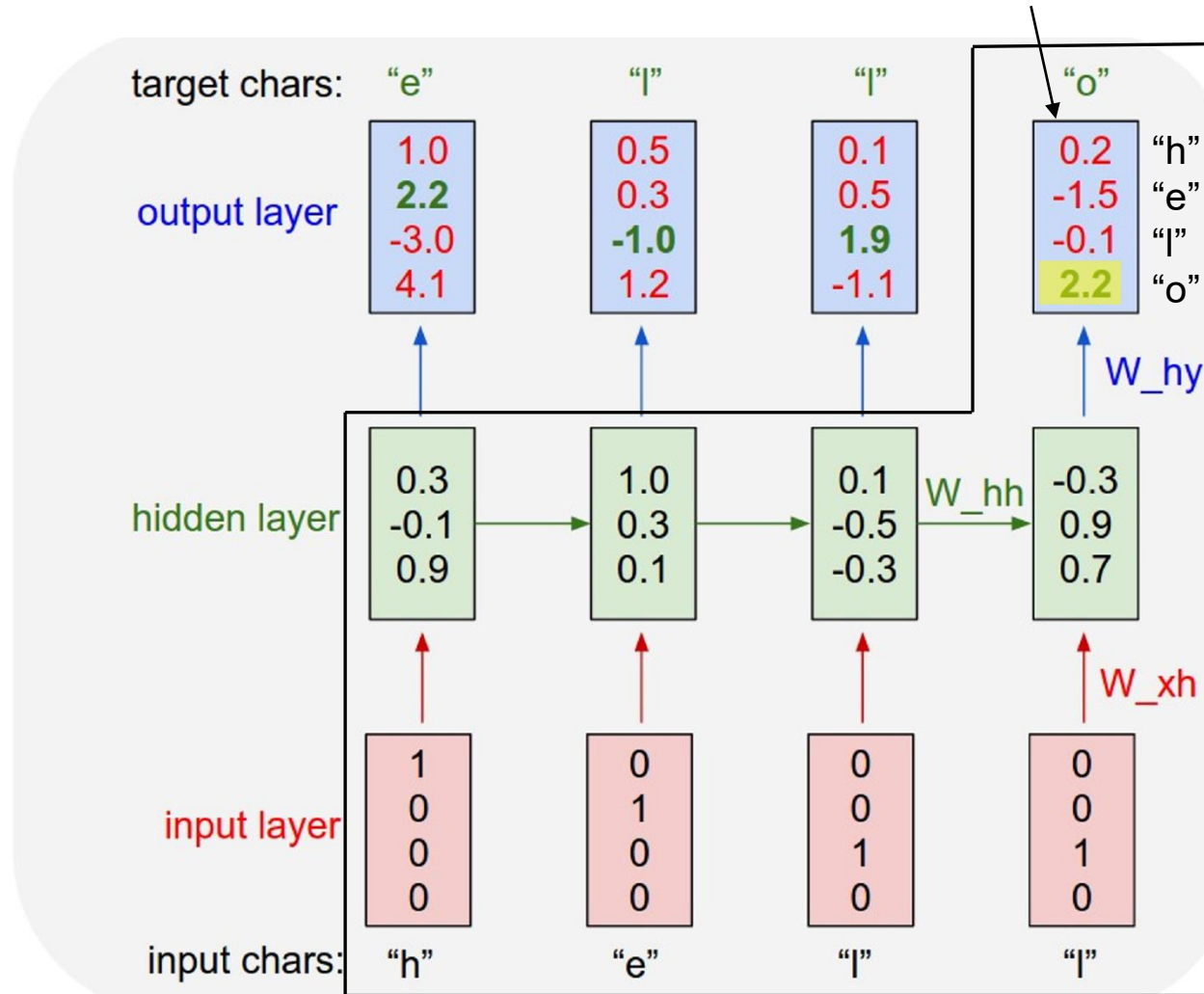
- Given “hel”, predict “l”

Highest value is 1.9 for “l”. RNN predicts “l” for input “hel” – correct.



- Given “hell”, predict “o”

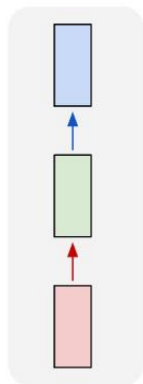
Highest value is 2.2 for “o”. RNN predicts “o” for input “hell” – correct.



- Backward pass – weight update:
 - After each character (element of the sequence) is applied, a weight update is calculated for each location of the network that was used
 - At the end of the epoch, the weight updates for the same locations are summed and the whole network is updated using the summed weight update
- The backpropagation algorithm is applied many times, until convergence, e.g. until all characters are classified correctly
- After the training is completed, we can use the RNN to generate text, character after character (words, paragraphs, etc.)
- Feed a character to RNN and predict the next one
- Feed the predicted character and predict the next one
- ...

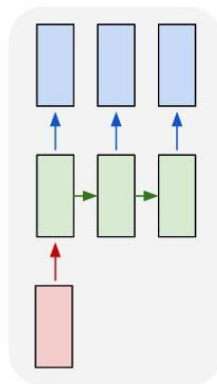
- RNN can be deployed in different scenarios

one to one



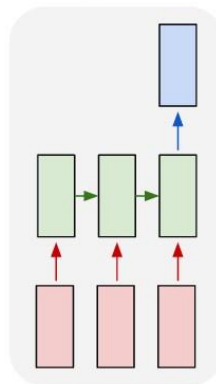
No RNN, no sequence
Ex: image classification
Input: image
Output: image class

one to many



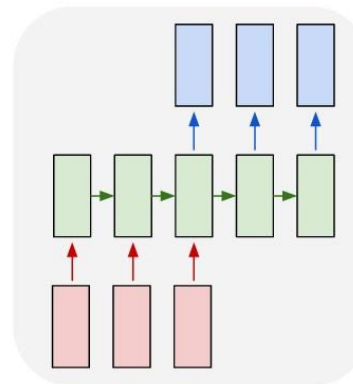
Sequence output
Ex: image captioning
Input: image
Output: sequence of words

many to one



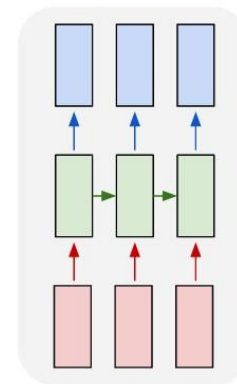
Sequence input
Ex: sentiment analysis
Input: text document
Output: positive or negative sentiment

many to many



Sequence input and sequence output
Ex: machine translation:
Input: a sentence in English, output: a sentence in Italian

many to many



Synced sequence input and sequence output
Ex: video classification where we want to label each video frame with a class

<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

- They don't need to generate an output after each element (e.g. after each word) as in our example
- In some tasks, e.g. sentiment analysis, RNN outputs a single value (positive or negative sentiment) after the whole sentence is processed
- We have a sentiment analysis exercise at the tutorial

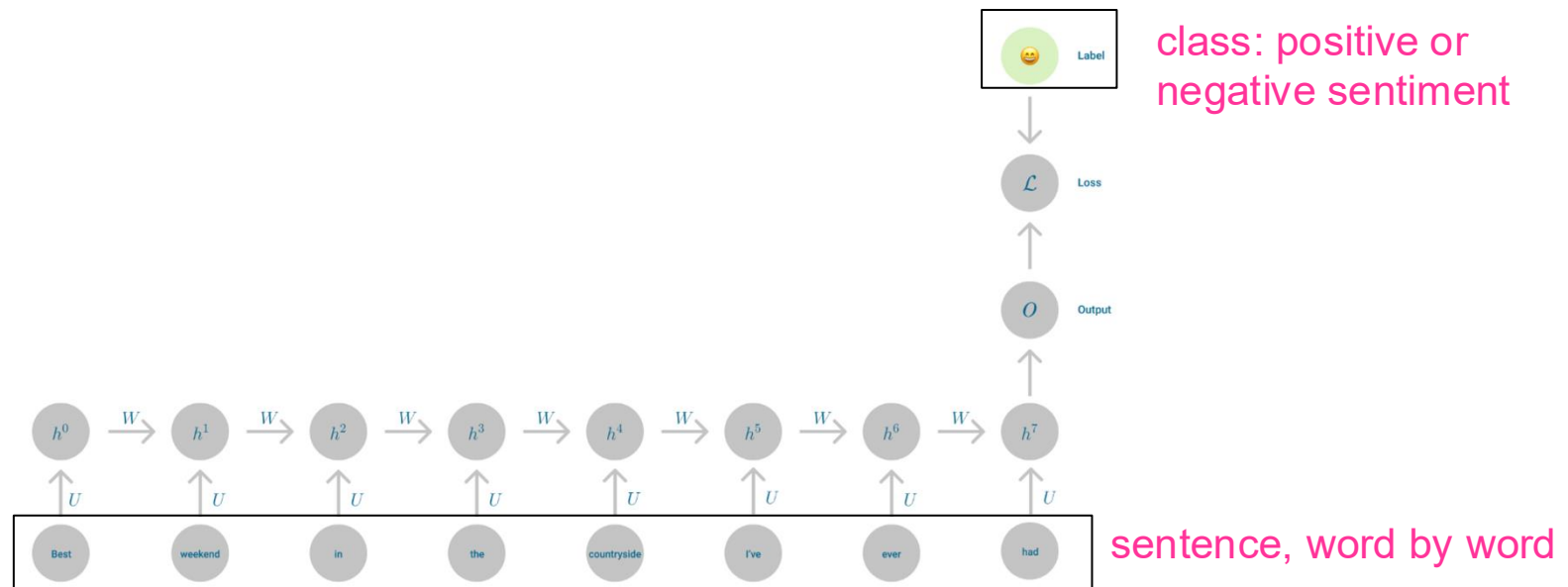
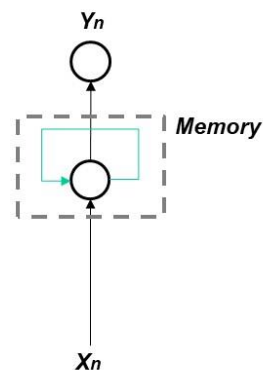


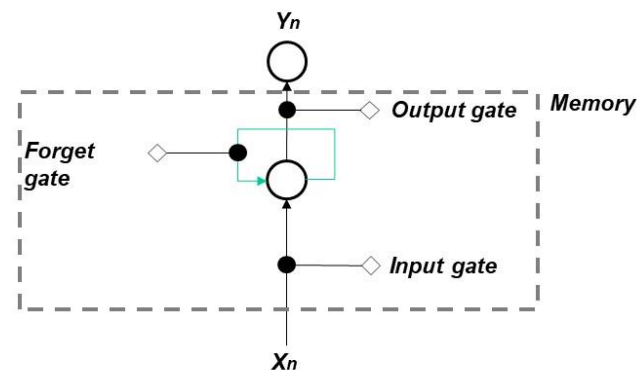
Image ref: <https://towardsdatascience.com/recurrent-neural-networks-explained-with-a-real-life-example-and-python-code-e8403a45f5de>

- RNN are susceptible to the vanishing gradient problem
- Reason: during the backward pass, the error gradients will be multiplied by the W_{hh} matrix multiple times - once for each time-step
- This repeated multiplication may cause the gradient to vanish if the weights are too small
- =>Although in principle RNNs have the ability to learn long-term dependencies, in practice this is limited
- Long-term dependencies – dependencies over long distances in a sequence
- Ex.: predicting the next word based on the previous; predicting the last word:
 - The clouds are in the sky
 - Given the previous words, it is obvious that the next word will be “sky” – the gap between the relevant information and the point where it is needed is small (short distance)
 - I grew up in Italy...I speak fluent Italian
 - Recent information suggests we need the name of a language but which one? We need the context of Italy, from further back (long distance)
 - Another example: Translating a long sentence; by the time you have finished it, you have forgotten how it started ☺

- Long Short Term Memory (LSTM) networks are a type RNN, specifically designed to improve the ability of RNN to model long dependencies
- Introduced by Hochreiter and Schmidhuber in 1997
<https://doi.org/10.1162/neco.1997.9.8.1735>
- The key idea is the network can learn what to store, what to throw away and what to read from
- An LSTM cell includes 3 gates: forget, input and output

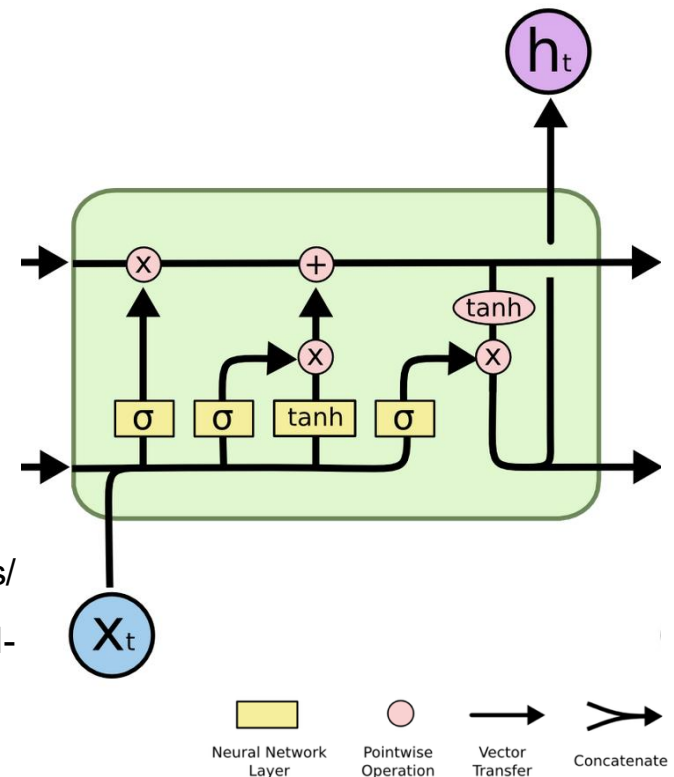


Standard RNN

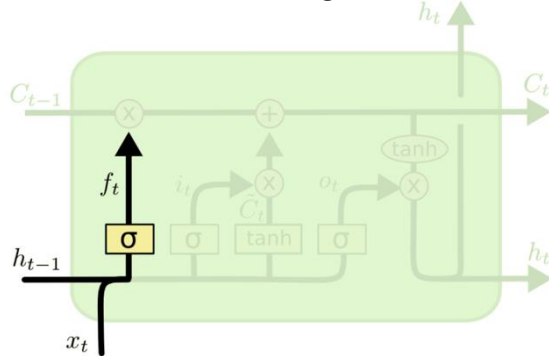


LSTM

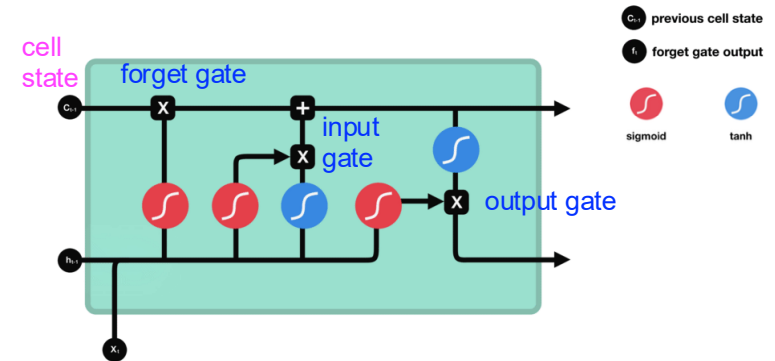
- The current input vector x_t and the previous short-term state h_{t-1} are fed to 4 different fully connected layers (not 1 as in the simple RNN)
- The 3 new layers have sigmoid activation function denoted with $\sigma \Rightarrow$ their outputs range from 0 to 1
- Then, their outputs are fed to element-wise multiplication 
 - If they output 0, they “close” the gate
 - If they output 1s, they “open” the gate
- These and next slides are based on <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- **Animations:** <https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21>



- Decides **what information** from the long-term state C_{i-1} **should be thrown away** or kept
- Uses information from the previous hidden state h_{t-1} and information from the current input x_t , passed through a sigmoid function, which outputs values between 0 and 1 for each number in the cell state C_{i-1}
- Values closer to 0 mean “forget” this and closer to 1 mean “keep this”

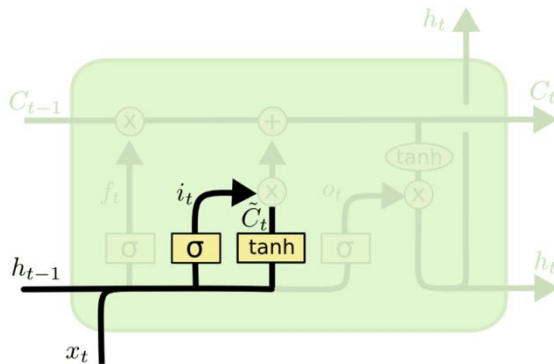


$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$



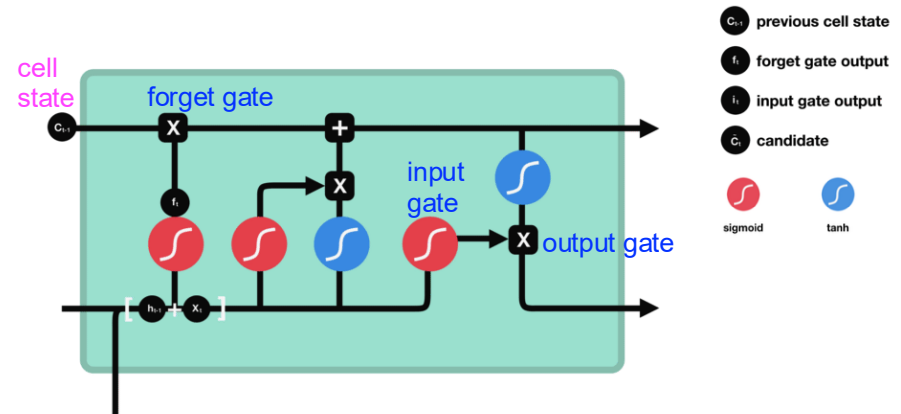
- For our example (predicting the next word based on the previous): the cell state may include the gender of the present subject, so that the correct pronoun (he/she) can be used. When we see a new subject, we want to forget the gender of the old subject.

- Decides **what new information should be stored** in the cell state
- 1) A sigmoid layer (input gate layer) decides which values to update
- 2) A tanh layer (as in the simple RNN) creates a vector of new candidate state values $\tilde{C}_t$ that could be added to the state
- 3) Combine them using point-wise multiplication



$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

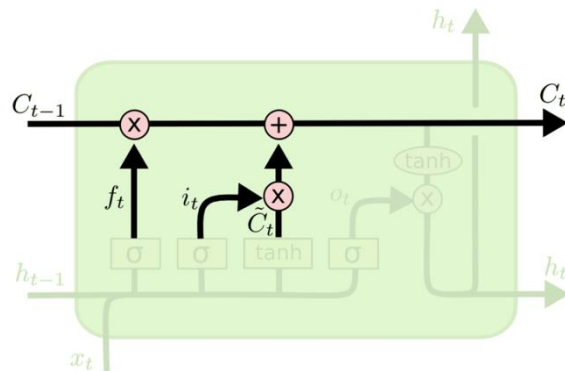
$$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$$



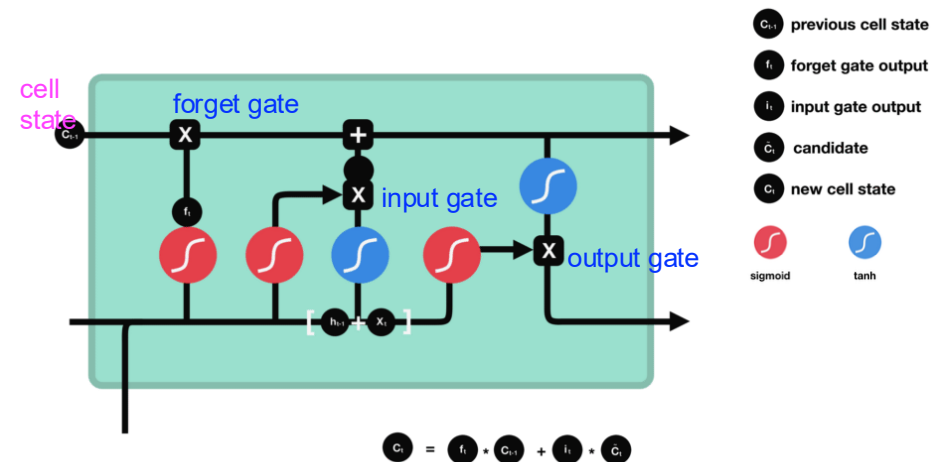
- For our example: add the gender of the new subject to the cell state, to replace the old one we are forgetting

Update the cell state

- Update the old cell state C_{t-1} to produce the new state C_t by combining the previous 2 results (the values of the forget and input layers):
- 1) Multiply the old state C_{t-1} by f_t = forgetting the things we decided to forget earlier
- 2) Add $i_t * \tilde{C}_t$ - the new candidate cell values, scaled by how much we decided to update each of them

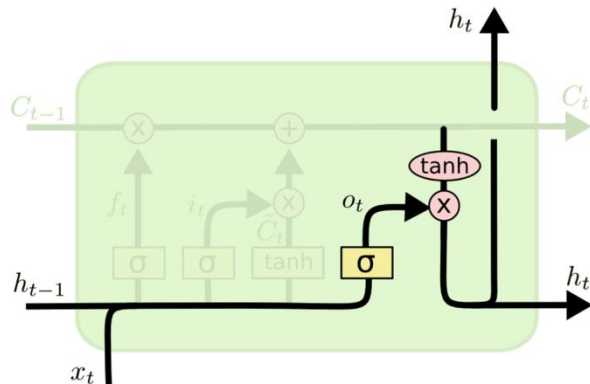


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



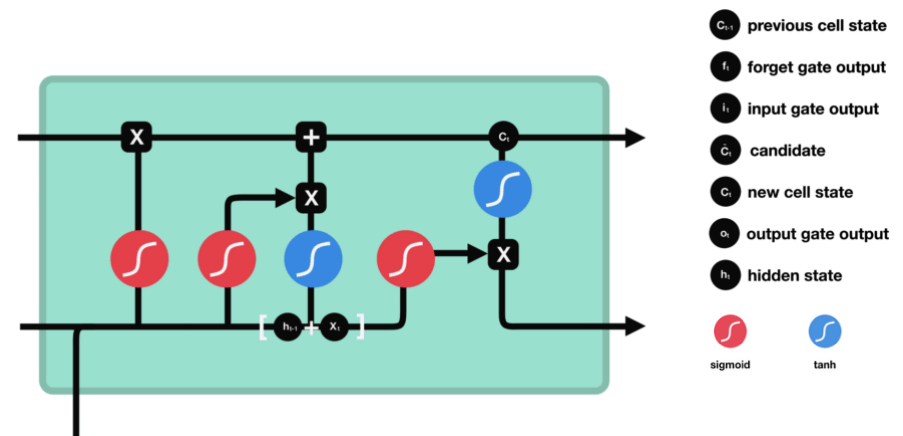
- For our example: drop the information about the old subject's gender and add the new information, as decided in the previous steps

- Decides **what exactly to output** - the output will be based on the cell state, but it will be a filtered version
- 1) A sigmoid layer computes o_t which determines the parts of the cell state we will output
- 2) The previously computed cell state C_t goes through tanh (to push the values to -1 and 1) and is then multiplied by o_t , so that we only output the parts we decided to



$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$



- For our example: the network just saw a subject, we can output information about the verb, in case that is what is coming next – e.g. if the subject is singular or plural, so that we know what form of the verb should be used next

- Many other RNN architectures have been proposed, e.g.:
- Gated Recurrent Unit (GRU)
Cho et al. (2014), *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*
<https://arxiv.org/abs/1406.1078>
- IRNN
Le et al. (2015), *A Simple Way to Initialize Recurrent Networks of Rectified Linear Units*
<https://arxiv.org/abs/1504.00941>

- RNNs have cyclic connections (e.g. from the hidden layer output back to the hidden layer as input)
- They have memory which stores the previous state of the hidden layer – the past activations and the inputs which contributed to these activations
- They are used on sequential data, where each element of the sequence is applied one-by-one
- RNNs have flexible architecture and can be applied for various types of tasks: one-to-many, many-to-one, etc.
- We studied 2 types of RNNs: simple RNN and LSTM
- Simple RNNs work well with sequences which include dependent features that are short distances apart. They don't work well for long distances because of the vanishing gradient problem.

- LSTM are specifically designed to model dependencies between features that are long distances apart in the input sequence
- They have a complex internal structure which includes 4 fully connected layers and 3 gates which control the information flow: forget, input and output
- An LSTM cell can recognize an important input (via the input gate), store it in the long-term state and keep it as long as needed (via the forget gate) and extract it when needed (via the output gate)
- As RNN, LSTMs are trained with the “backpropagation through time” algorithm
- However, there is no repeated multiplication by W_{hh} as we backpropagate the error – no vanishing gradient problem
- LSTMs have a complex architecture but have produced state-of-the-art results for many tasks, e.g. machine translation, speech recognition, handwriting recognition, speech synthesis and video analysis