Deep Learning
CNN, RNN, and others

Luca Bortolussi

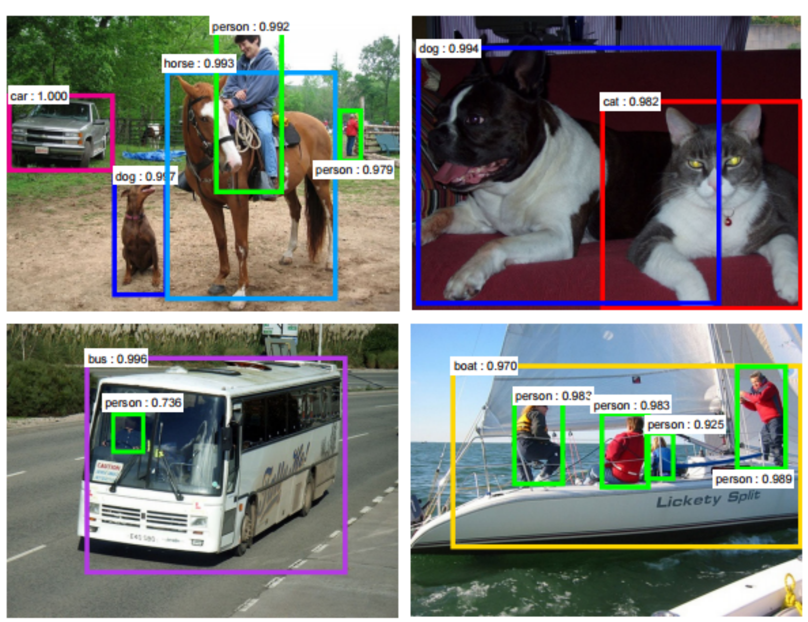
DMG, University of Trieste, IT
Modelling and Simulation, Saarland University, DE

DSSC @ TRIESTE

Convolutional Networks

[Krizhevsky 2012]

[Ciresan et al. 2013]



[Faster R-CNN - Ren 2015]

[NVIDIA dev blog]

Convolution

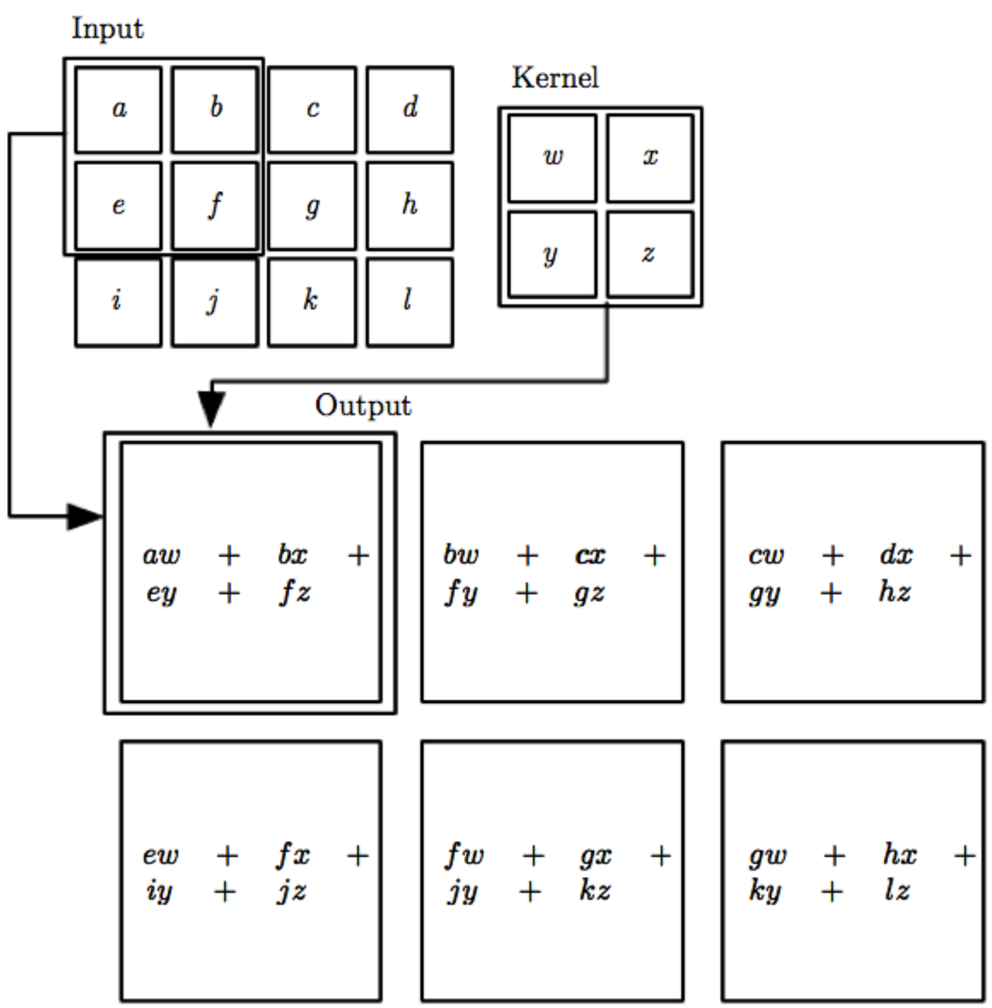
- Given array u_t and w_t, their convolution is a function s

$$s_t = \sum_{a=-\infty}^{+\infty} u_a w_{t-a}$$

- Written as

$$s = (u * w) \quad \text{or} \quad s_t = (u * w)_t$$

- When u_t or w_t is not defined, assumed to be 0



Convolution can be seen as a sort of localised noise filtering (a moving average in 1d).

Convolutional Layers

They are the standard approach for input data distributed in a grid, e.g. images.
They work also for sequence data and 3D data.

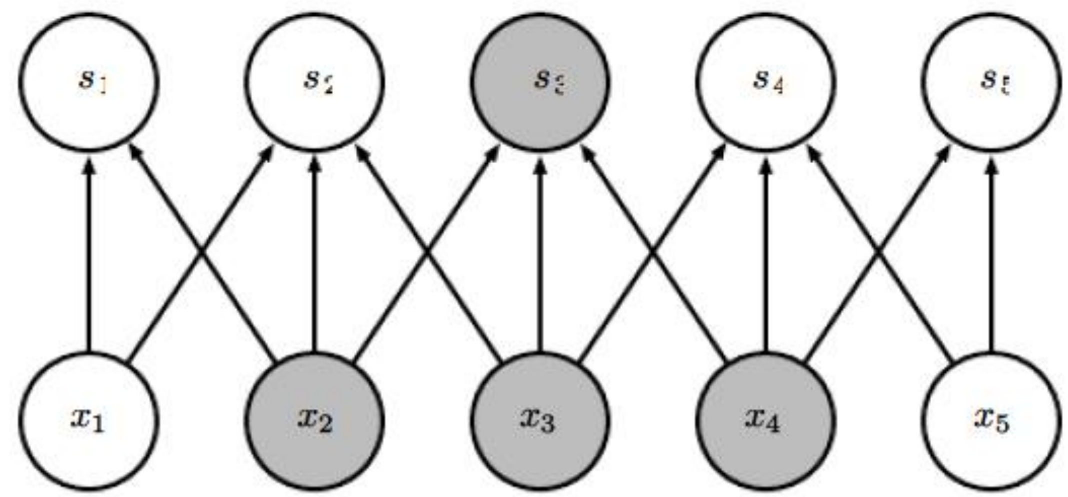
Convolution layers are the core of convolutional networks.

The same convolution is applied to each possible subset of the image.

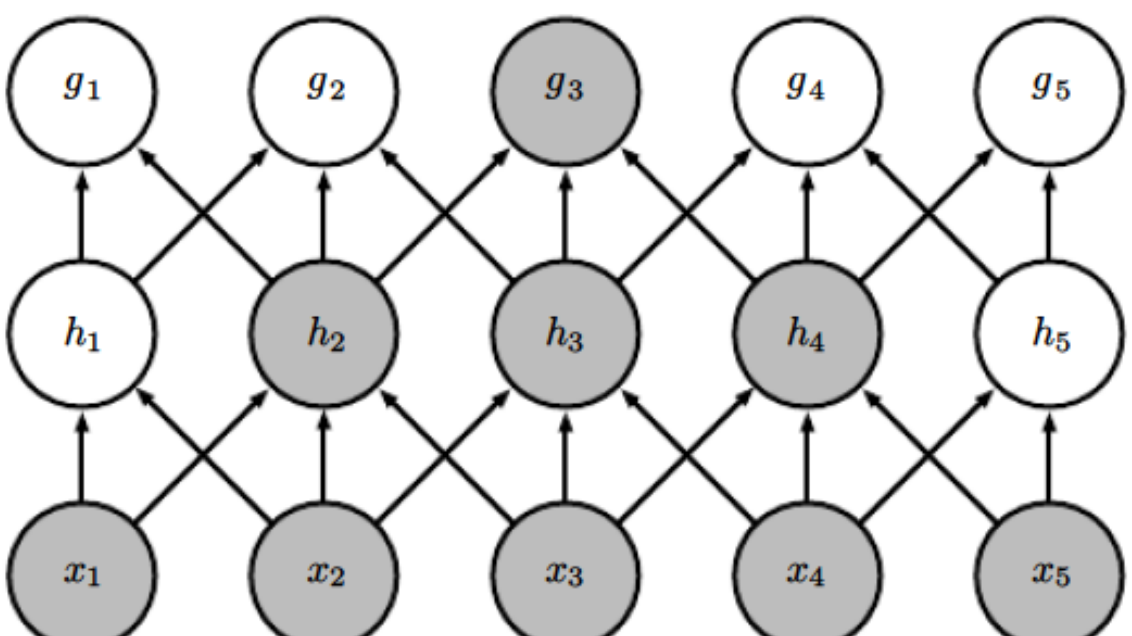
(**Zero padding** may be used at boundaries)

(One can impose a **stride** in each direction)

Convolutional layer, $\leq m \times k$ edges



Multiple convolutional layers: larger receptive field



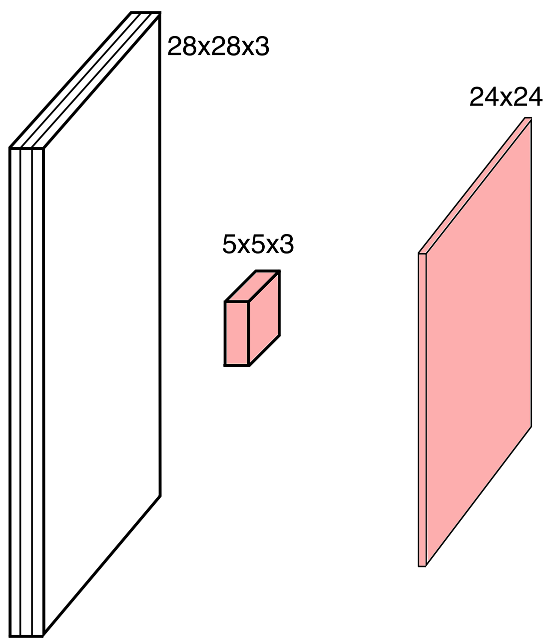
It enforces:

- **sparse connectivity**
- **parameter sharing**
  (kernels are localised and shared)

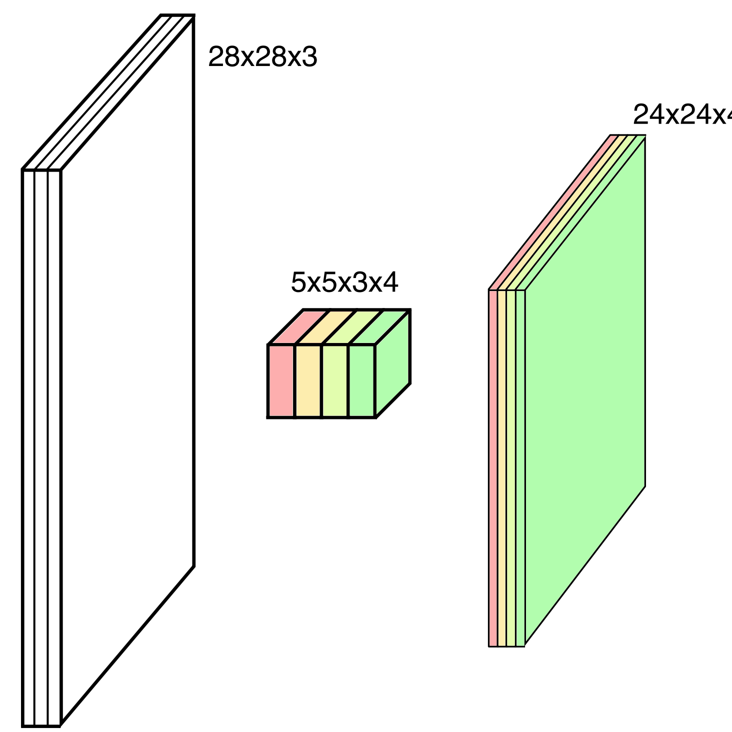
Convolutional Layers

Colored image = tensor of shape `(height, width, channels)`

Convolutions are usually computed for each channel and summed:



$$(k \star im^{color}) = \sum_{c=0}^{2} k^c \star im^c$$

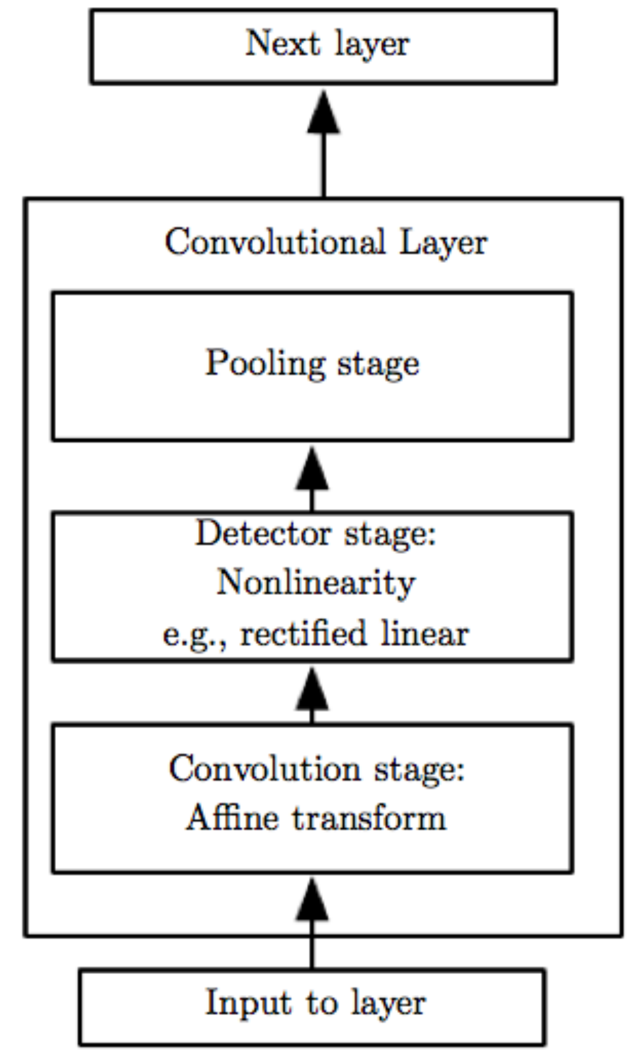


Typically, **multiple** convolutions are applied in parallel

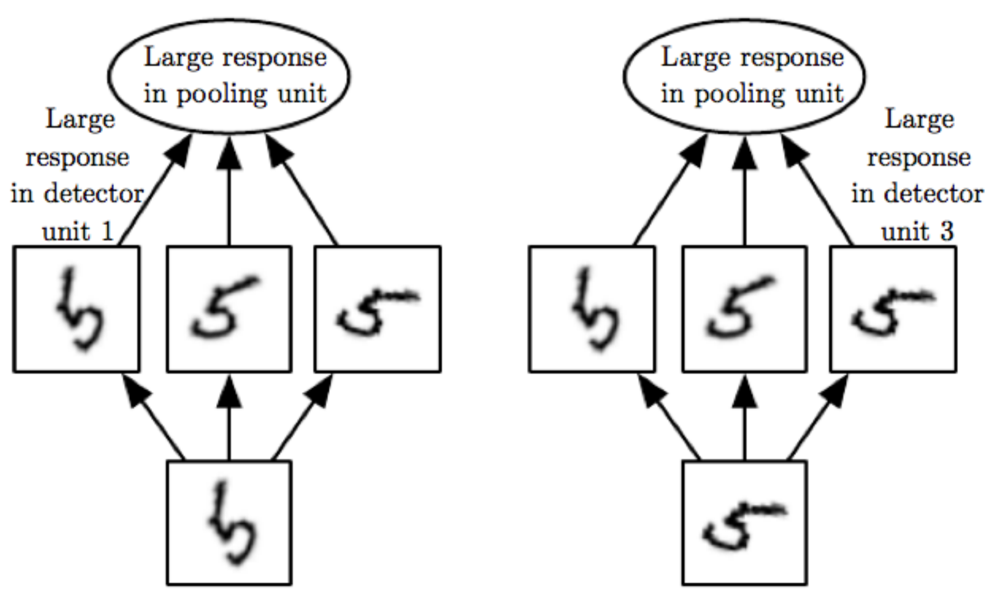
CNN - Detector Stage and Pooling

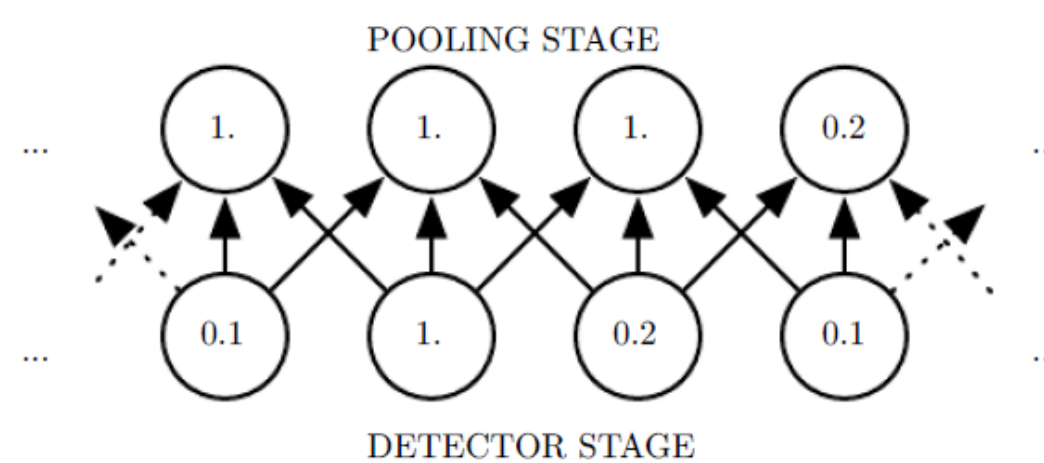
Each convolution layer applies three operations (or can be seen as three layers, like in Keras):

1. **convolution** with a local kernel (**linear filter**)
2. application of a **non-linear activation function** (**detector stage**)
3. **pooling** the values in a neighbourhood of pixels



Pooling (e.g. max or averaging in a neighbourhood) enforces **invariance to small translations** of the input. Useful also to deal with images of different sizes.





CNN – An example of architecture

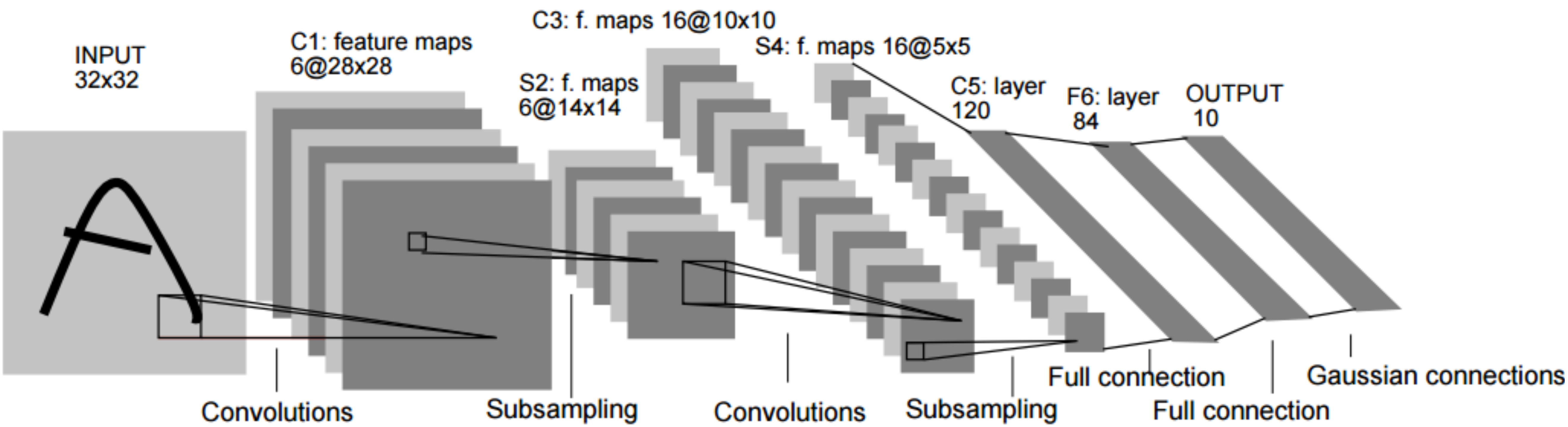
LeNet-5

- Proposed in *"Gradient-based learning applied to document recognition"*, *by Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, in Proceedings of the IEEE, 1998*

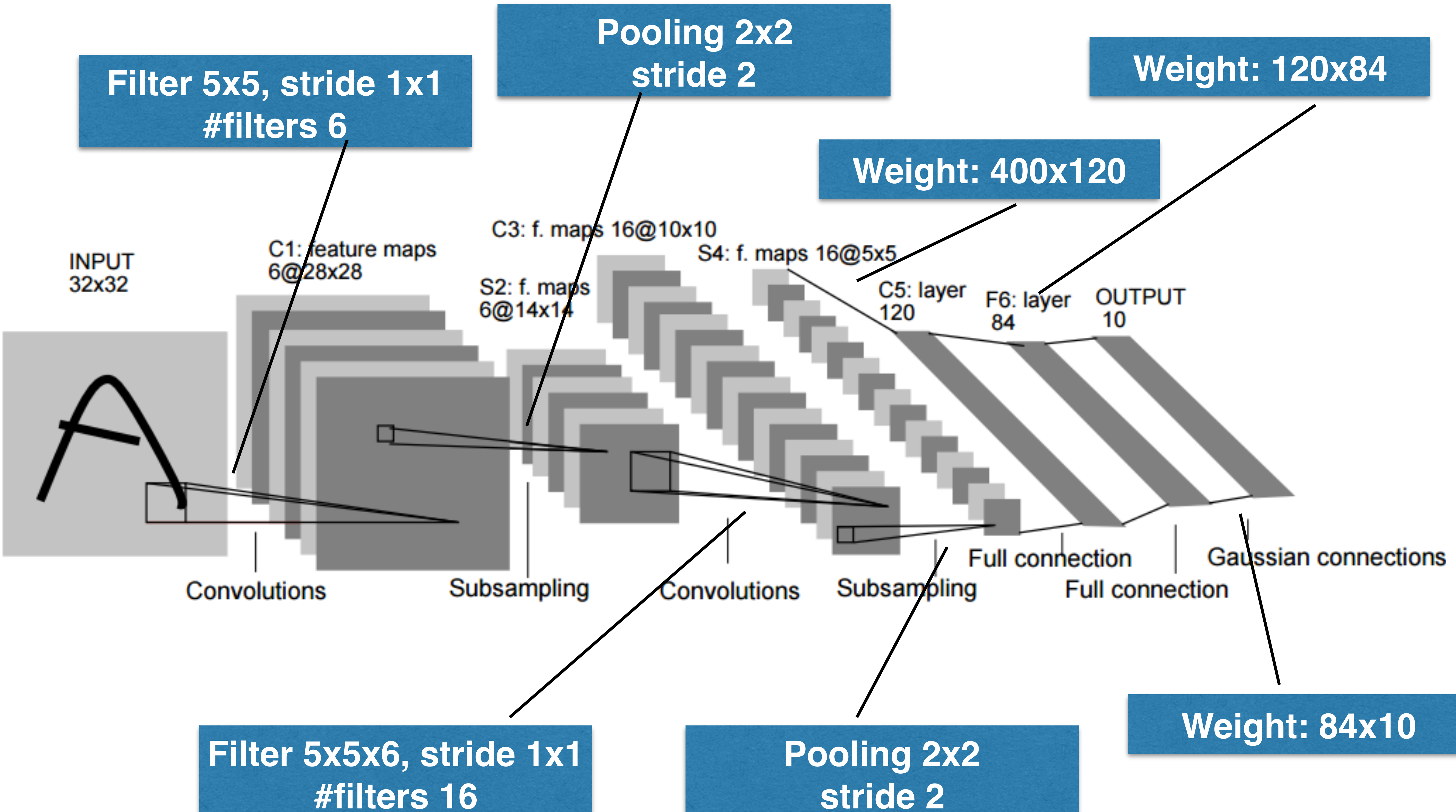
- Apply convolution on 2D images (MNIST) and use backpropagation

- Structure: 2 convolutional layers (with pooling) + 3 fully connected layers
  - Input size: 32x32x1
  - Convolution kernel size: 5x5
  - Pooling: 2x2

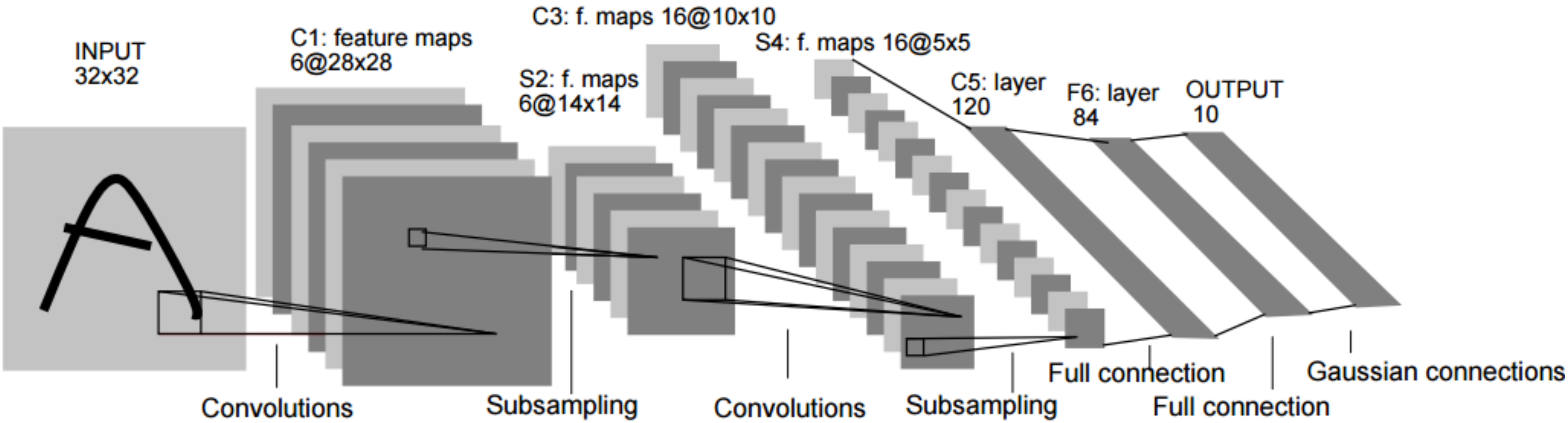
CNN - An example of architecture



CNN - An example of architecture

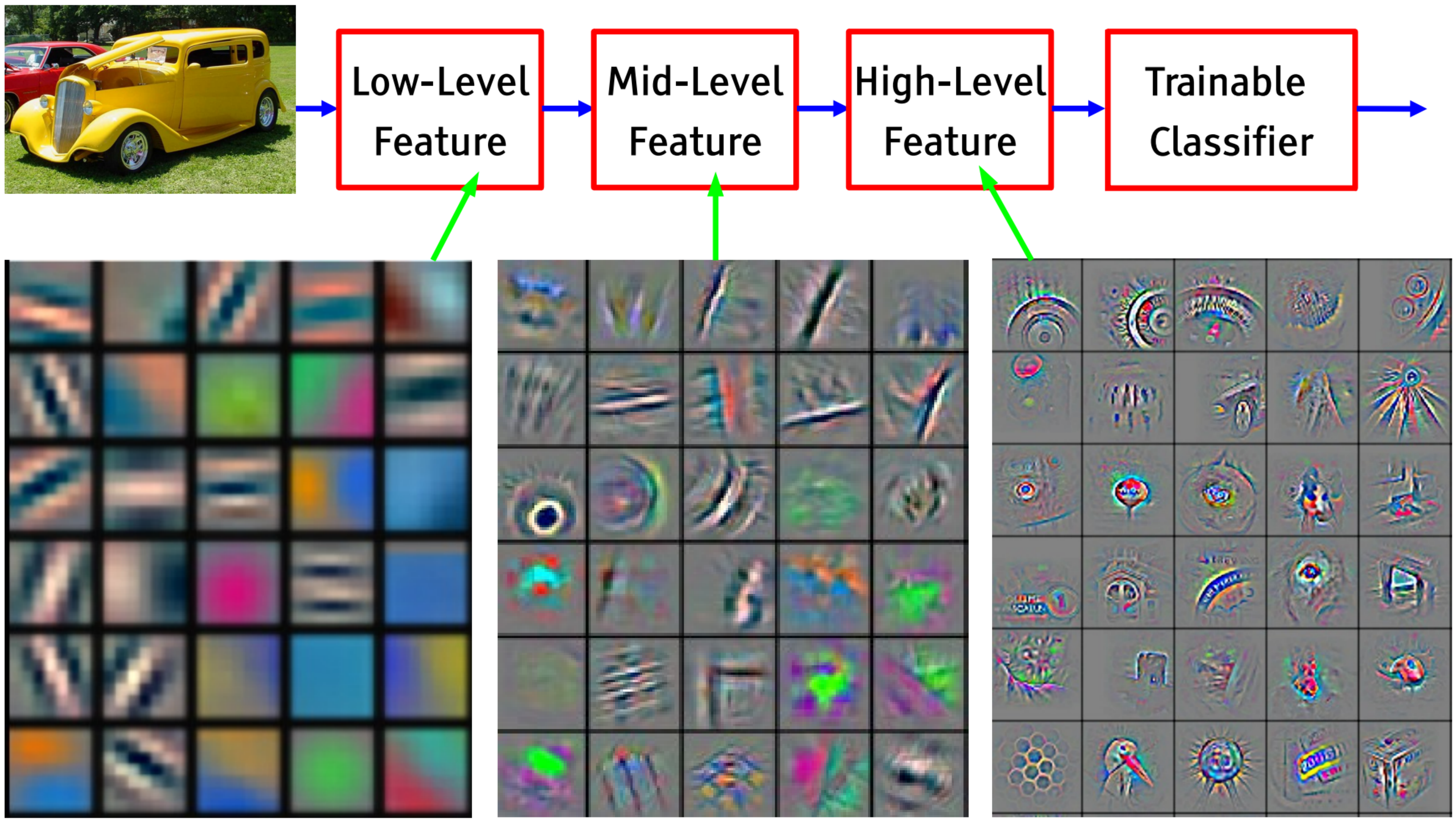


CNN - An example of architecture



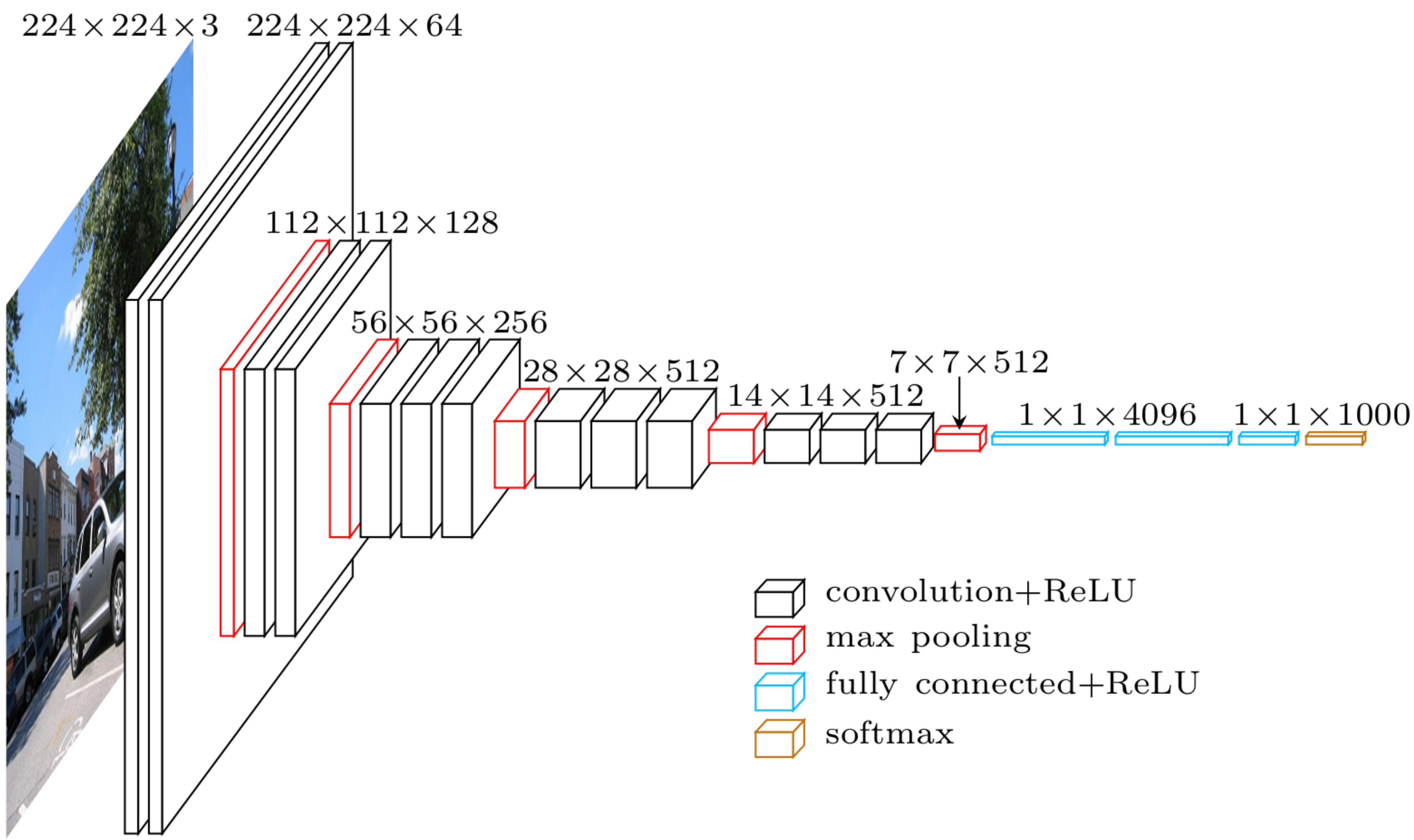

```
input_image = Input(shape=(28, 28, 1))
x = Conv2D(32, 5, activation='relu')(input_image)
x = MaxPool2D(2, strides=2)(x)
x = Conv2D(64, 3, activation='relu')(x)
x = MaxPool2D(2, strides=2)(x)
x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dense(10, activation='softmax')(x)
convnet = Model(inputs=input_image, outputs=x)
```

Hierarchical representation



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Architecture: VGG-16



Simonyan, Karen, and Zisserman. "Very deep convolutional networks for large-scale image recognition." (2014)

Architecture: VGG-16 in Keras

```
model.add(Convolution2D(64, 3, 3, activation='relu',input_shape=(3,224,224)))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Convolution2D(128, 3, 3, activation='relu'))
model.add(Convolution2D(128, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(Convolution2D(256, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(Convolution2D(512, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1000, activation='softmax'))
```

Architecture: VGG-16 - parameters

```
              Activation maps              Parameters
INPUT:     [224x224x3]   = 150K         0
CONV3-64:  [224x224x64]  = 3.2M         (3x3x3)x64     =       1,728
CONV3-64:  [224x224x64]  = 3.2M         (3x3x64)x64    =      36,864
POOL2:     [112x112x64]  = 800K         0
CONV3-128: [112x112x128] = 1.6M         (3x3x64)x128   =      73,728
CONV3-128: [112x112x128] = 1.6M         (3x3x128)x128  =     147,456
POOL2:     [56x56x128]   = 400K         0
CONV3-256: [56x56x256]   = 800K         (3x3x128)x256  =     294,912
CONV3-256: [56x56x256]   = 800K         (3x3x256)x256  =     589,824
CONV3-256: [56x56x256]   = 800K         (3x3x256)x256  =     589,824
POOL2:     [28x28x256]   = 200K         0
CONV3-512: [28x28x512]   = 400K         (3x3x256)x512  =   1,179,648
CONV3-512: [28x28x512]   = 400K         (3x3x512)x512  =   2,359,296
CONV3-512: [28x28x512]   = 400K         (3x3x512)x512  =   2,359,296
POOL2:     [14x14x512]   = 100K         0
CONV3-512: [14x14x512]   = 100K         (3x3x512)x512  =   2,359,296
CONV3-512: [14x14x512]   = 100K         (3x3x512)x512  =   2,359,296
CONV3-512: [14x14x512]   = 100K         (3x3x512)x512  =   2,359,296
POOL2:     [7x7x512]     =  25K         0
FC:        [1x1x4096]    = 4096         7x7x512x4096   = 102,760,448
FC:        [1x1x4096]    = 4096         4096x4096      =  16,777,216
FC:        [1x1x1000]    = 1000         4096x1000      =   4,096,000

TOTAL activations: 24M x 4 bytes ~=  93MB / image (x2 for backward)
TOTAL parameters: 138M x 4 bytes ~= 552MB (x2 for plain SGD, x4 for Adam)
```

Architecture: ResNet

Even deeper models:

34, 50, 101, 152 layers

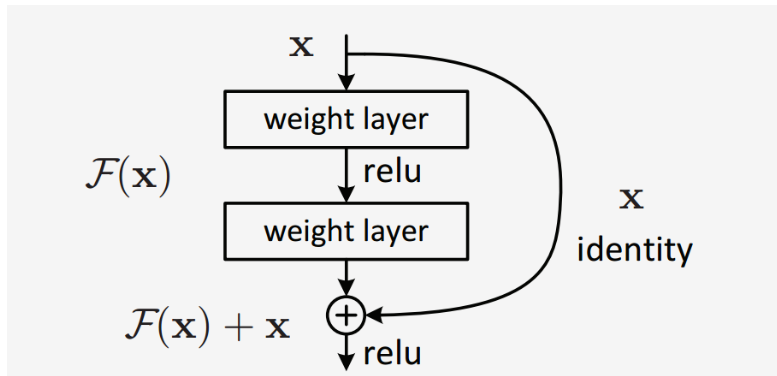


Figure 2. Residual learning: a building block.

ResNet50 Compared to VGG:

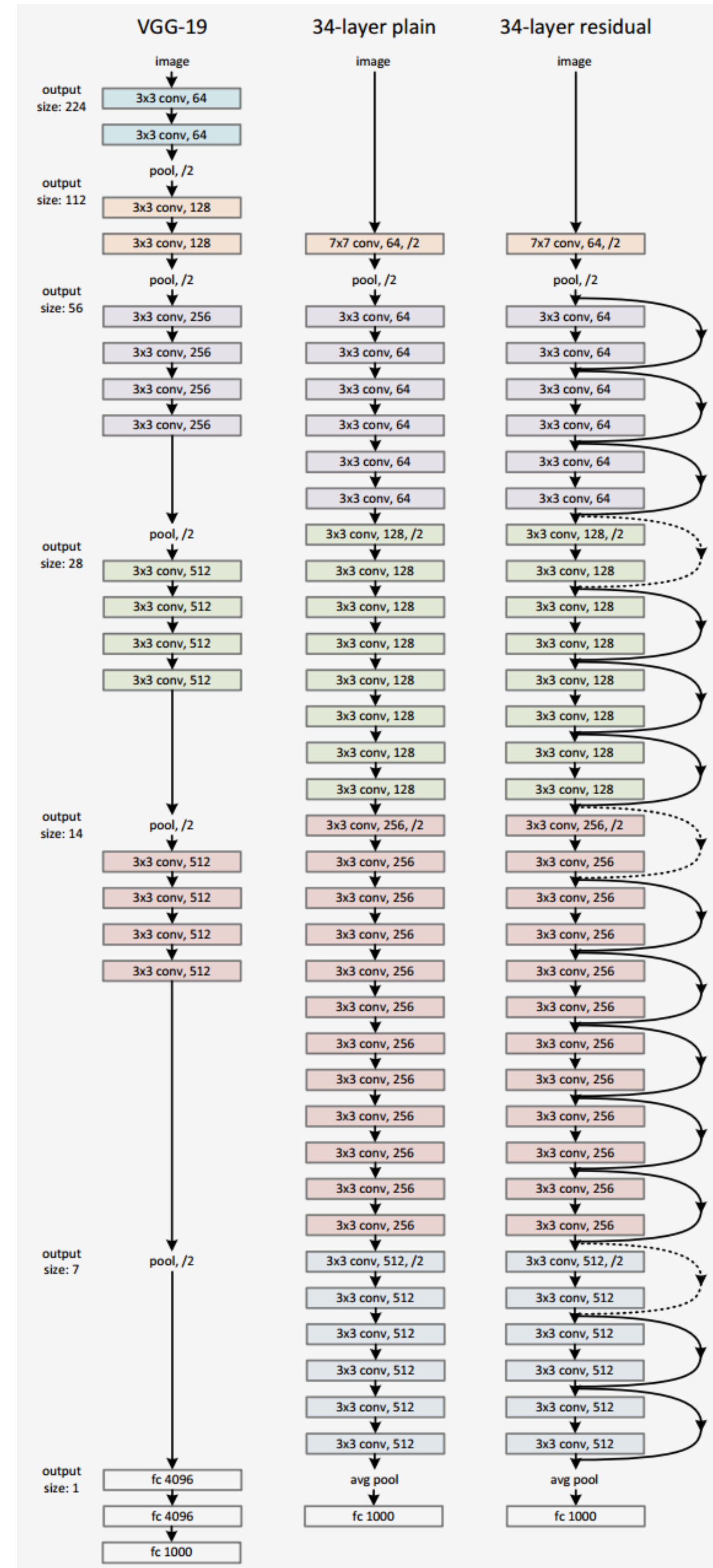
Superior accuracy in all vision tasks
5.25% top-5 error vs 7.1%

Less parameters
25M vs 138M

Computational complexity
3.8B Flops vs 15.3B Flops

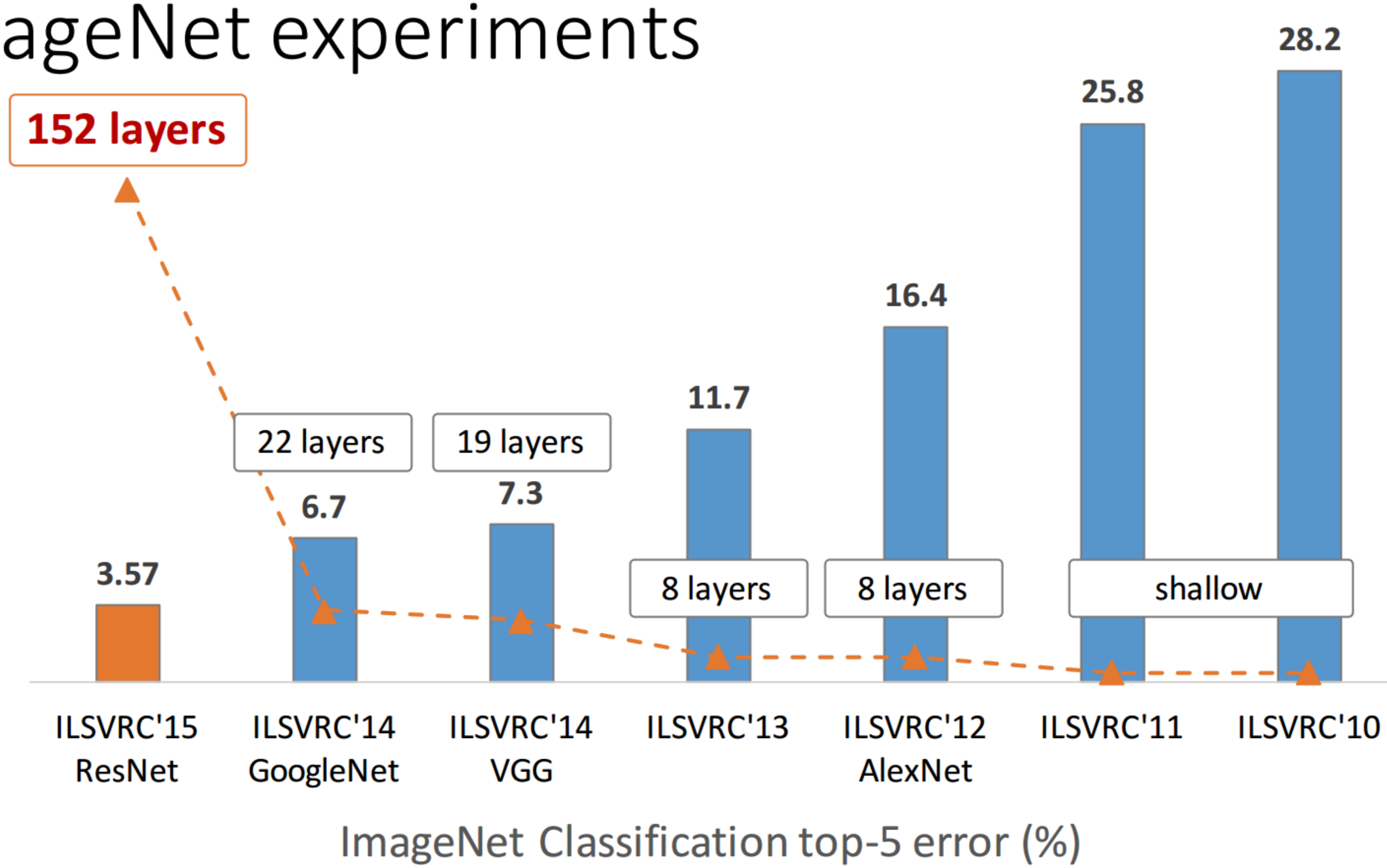
Fully Convolutional until the last layer

He, Kaiming, et al. "Deep residual learning for image recognition." CVPR. 2016.



Deeper is better

ImageNet experiments



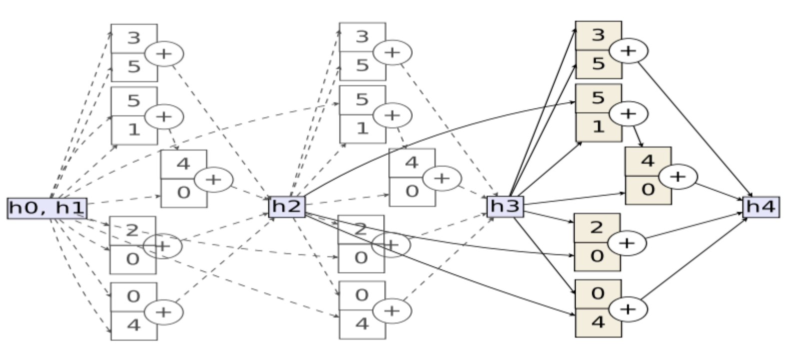
The right architecture

- Finding right architectures: Active area or research

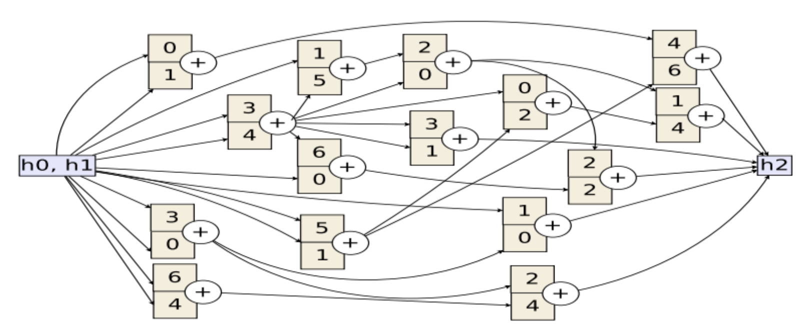
Model	Params	×+	1/5-Acc (%)
Inception V3	23.8M	5.72B	78.0 / 93.9
Xception	22.8M	8.37B	79.0 / 94.5
Inception ResNet V2	55.8M	13.2B	80.4 / 95.3
ResNeXt-101 (64x4d)	83.6M	31.5B	80.9 / 95.6
PolyNet	92.0M	34.7B	81.3 / 95.8
Dual-Path-Net-131	79.5M	32.0B	81.5 / 95.8
Squeeze-Excite-Net	145.8M	42.3B	82.7 / **96.2**
GeNet-2	156M	–	72.1 / 90.4
Block-QNN-B, N=3	–	–	75.7 / 92.6
Hierarchical (2, 64)	64M	–	79.7 / 94.8
PNASNet-5 (4, 216)	86.1M	25.0B	**82.9** / 96.1
NASNet-A (6, 168)	88.9M	23.8B	82.7 / **96.2**
AmoebaNet-B (6, 190)	84.0M	22.3B	82.3 / 96.1
AmoebaNet-C (6, 168)	85.5M	22.5B	82.7 / 96.1
AmoebaNet-A (6, 190)	86.7M	23.1B	82.8 / 96.1
AmoebaNet-A (6, 204)	99.6M	26.2B	82.8 / **96.2**

Automated Architecture search:

- reinforcement learning
- evolutionary algorithms



0 = sep. 3x3
1 = sep. 5x5
2 = sep. 7X7
3 = none
4 = avg. pool
5 = max pool
6 = dil. 3x3
7 = 1x7+7x1



Pre-trained models

Training a model on ImageNet from scratch takes **days or weeks**.

Many models trained on ImageNet and their weights are publicly available!

Transfer learning

- Use pre-trained weights, remove last layers to compute representations of images
- Train a classification model from these features on a new classification task
- The network is used as a generic feature extractor
- Better than handcrafted feature extraction on natural images

Fine-tuning

Retraining the (some) parameters of the network (given enough data)

- Truncate the last layer(s) of the pre-trained network
- Freeze the remaining layers weights
- Add a (linear) classifier on top and train it for a few epochs
- Then fine-tune the whole network or the few deepest layers
- Use a smaller learning rate when fine tuning

Data Augmentation

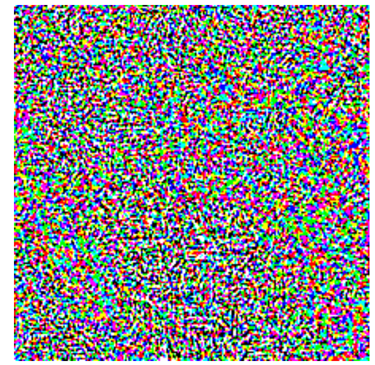
```python
from keras.preprocessing.image import ImageDataGenerator

image_gen = ImageDataGenerator(
    rescale=1. / 255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    channel_shift_range=9,
    fill_mode='nearest'
)

train_flow = image_gen.flow_from_directory(train_folder)
model.fit_generator(train_flow, train_flow.n)
```

Adversarial Examples

$+\ .007\ \times$



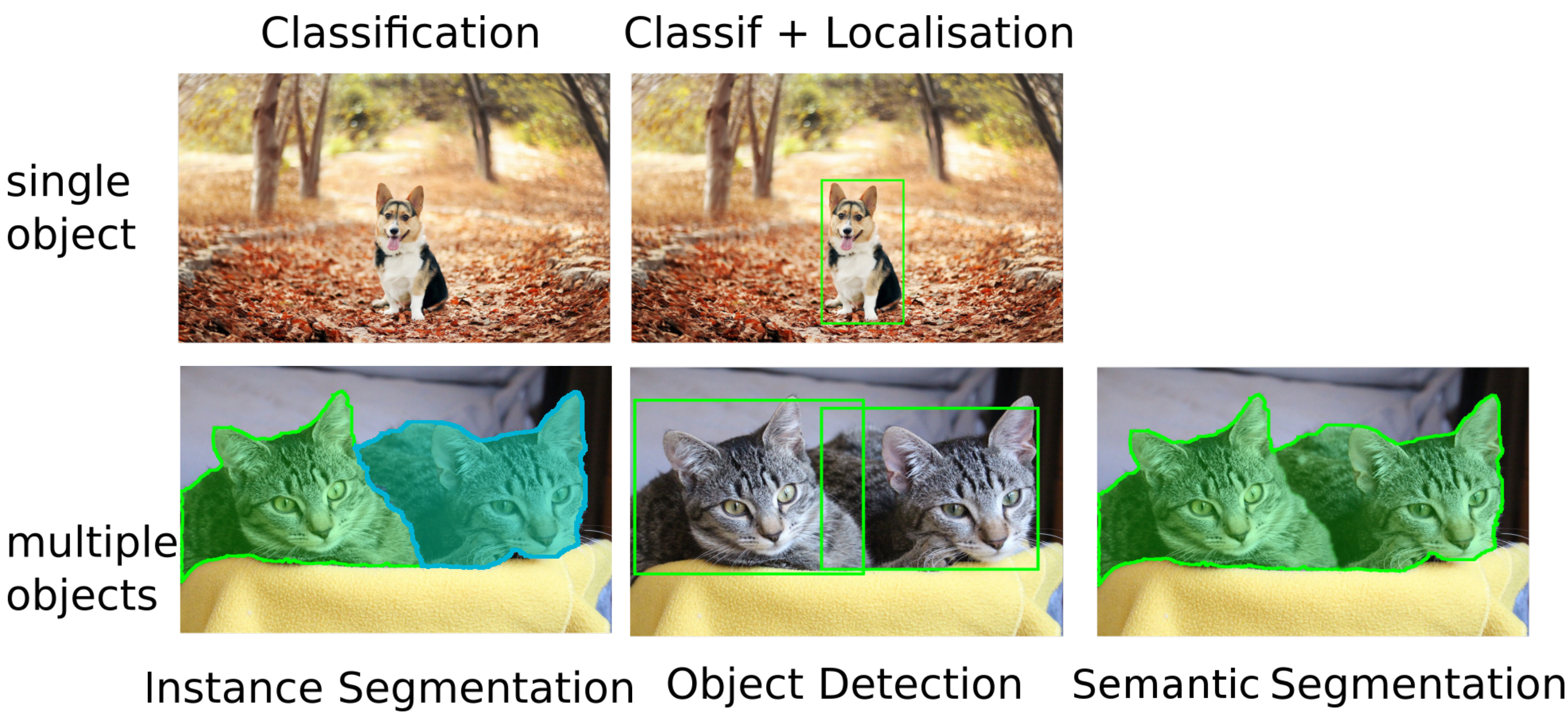
$=$

$\boldsymbol{x}$	$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$	$\boldsymbol{x} + \epsilon\,\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
y =“panda” w/ 57.7% confidence	“nematode” w/ 8.2% confidence	“gibbon” w/ 99.3 % confidence

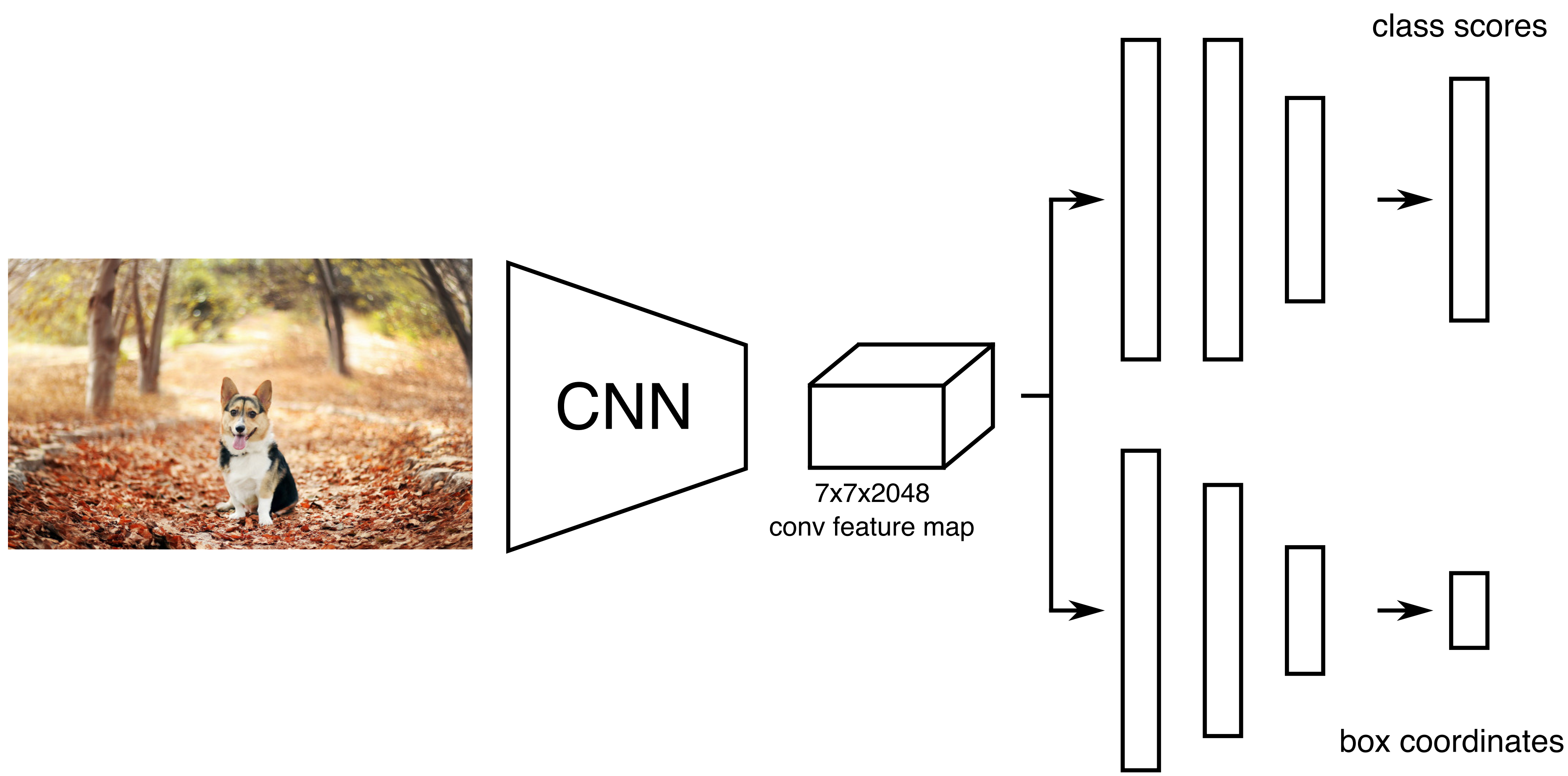
Adversarial examples are often generated from white-box models, following the gradient at a given image to maximise the loss.

Training on adversarial examples is mostly intended to improve security, but can sometimes provide generic regularisation.

Computer Vision with CNN



Example: classification + localisation



- Use a pre-trained CNN on ImageNet (ex. ResNet)
- The "localisation head" is trained seperately with regression
- Possible end-to-end finetuning of both tasks
- At test time, use both heads

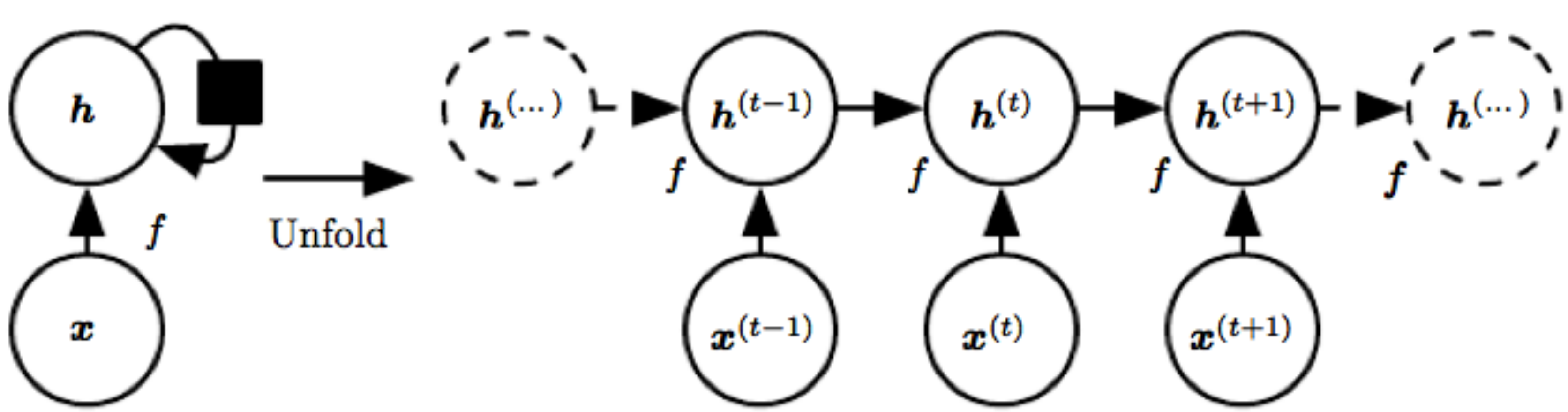
Recurrent Neural Networks

They are the mainstream approach for **time series** data, or **sequence** data (like sentences in natural language).
We can observe input/output pairs $\mathbf{x}^{(t)},\mathbf{y}^{(t)}$ at each time step t.

The basic idea is that of keeping a form of **memory** depending on the sequence of symbols/ inputs $\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(t-1)}$ seen up to time t, in the form of an hidden state $\mathbf{h}^{(t-1)}$, which is then combined with the input $\mathbf{x}^{(t)}$ at time t to compute a new hidden state, and from it the output $\mathbf{o}^{(t)}$.

Formally, we define a dynamical system by a **recurrent equation**

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$



Recurrent Neural Networks

Recurrent NN produce an output for each time-step, and then compute a loss from an observed output.
Networks are trained by unfolding the graph in time and evaluating the gradient with backpropagation on the unfolded graph: this is called **backpropagation through time**.

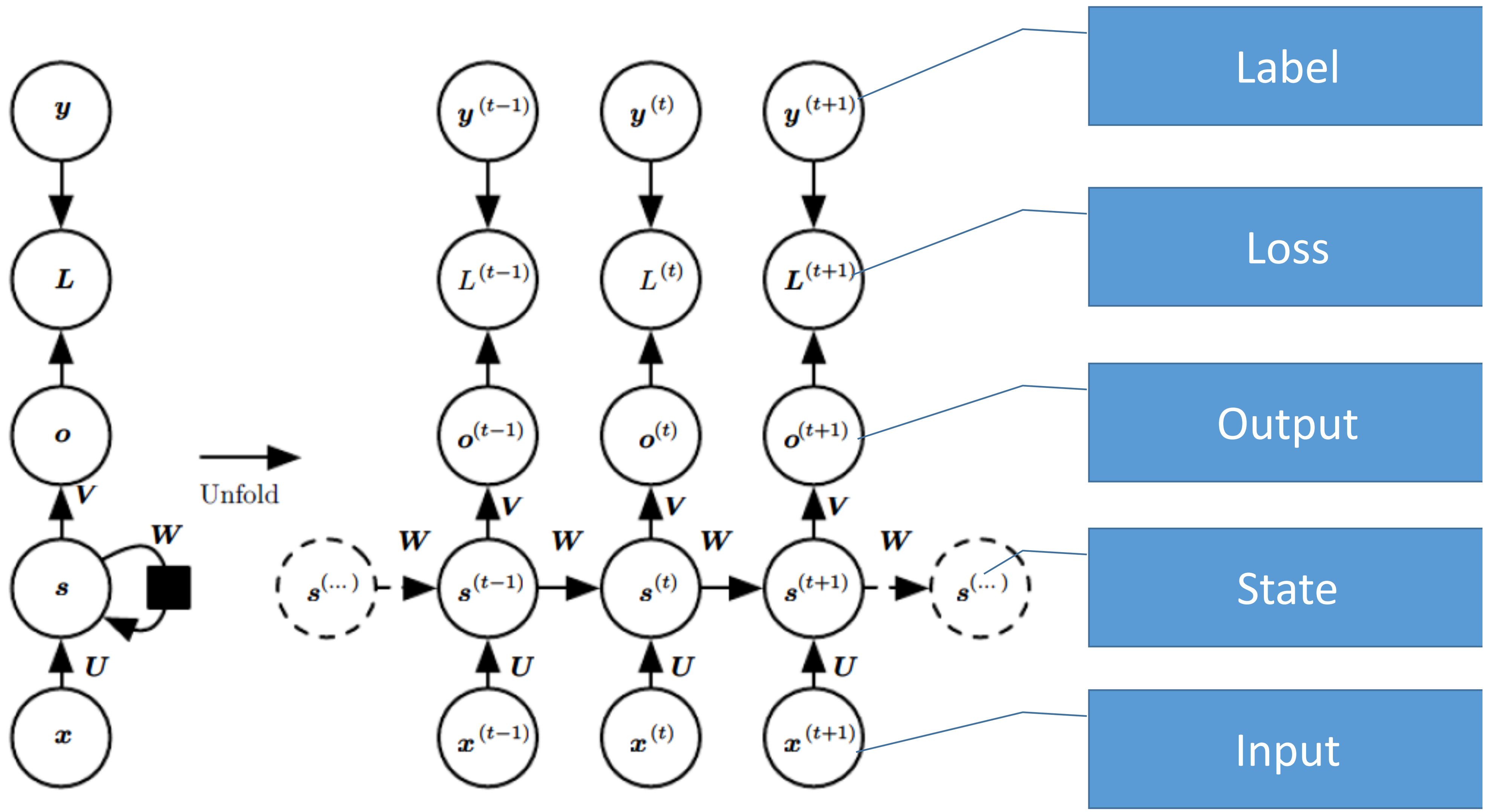
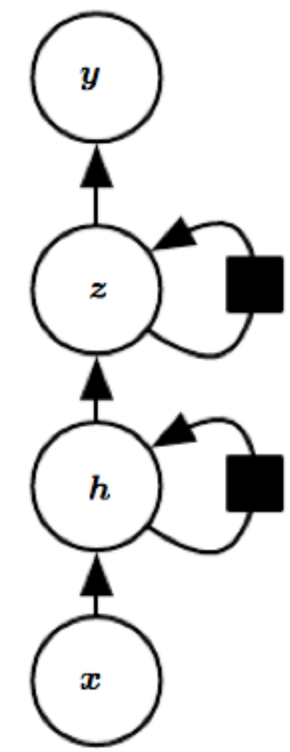


Figure from *Deep Learning*, by Goodfellow, Bengio and Courville

Recurrent Neural Networks

Deep RNN are commonly used to improve model capacity.



Advantages

- Hidden state keeps info about the past
- Shared functions and params across time: reduce model capacity, good for **generalization**.
- Still powerful: RNN of finite size are **Turing complete** (they can emulate any Turing Machine.

Downsides

- long-term dependencies tend to be forgotten in $\mathbf{h}^{(t)}$ exponentially fast.
- Tend to have very small (or very large) gradients.
- **Gradient clipping** is often used.

RNN Variants

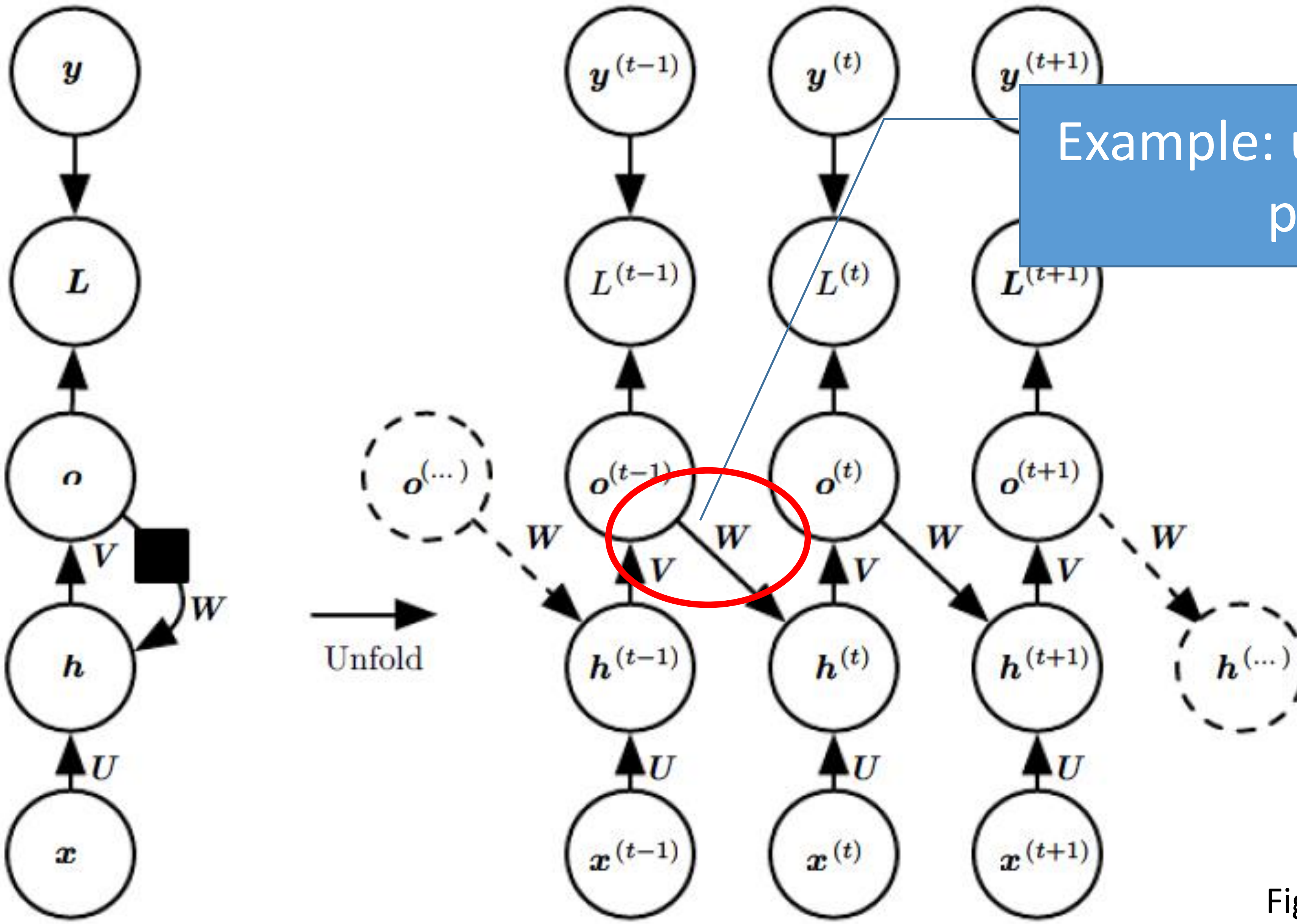


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

RNN Variants

Example: only output at the end

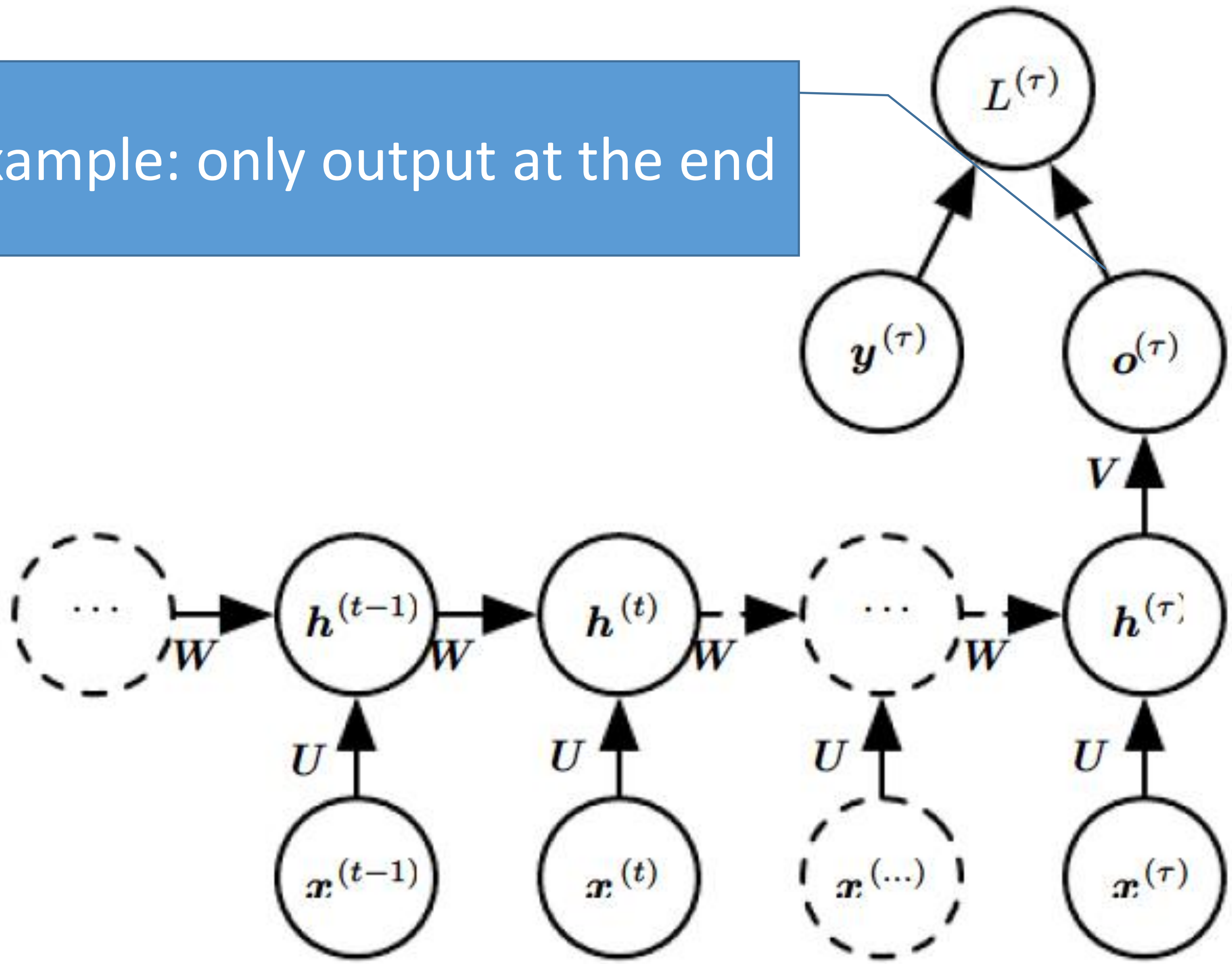


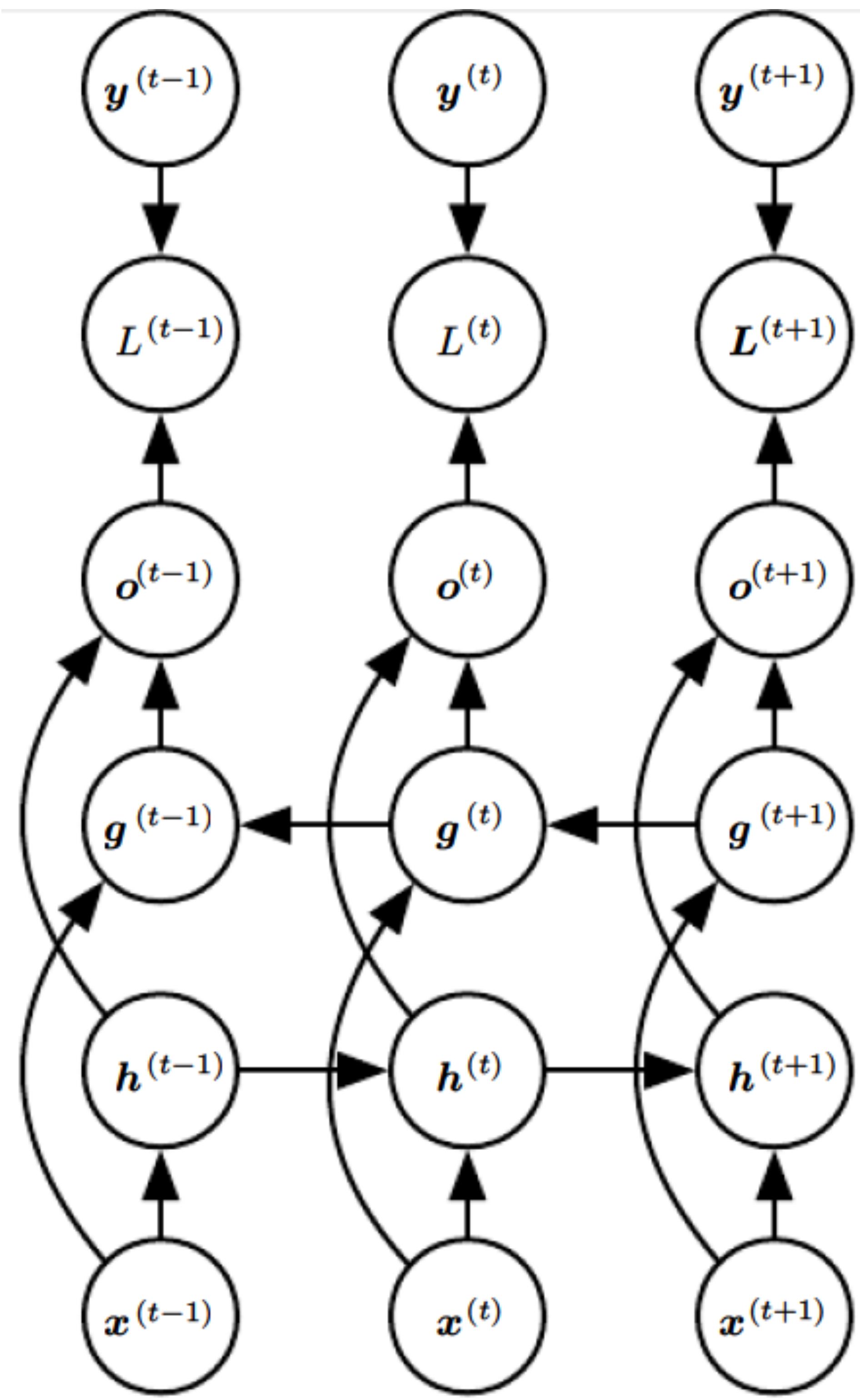
Figure from *Deep Learning*, Goodfellow, Bengio and Courville

RNN Variants

Bidirectional RNN tackle the problem of output dependency on the whole input sequence, like for speech recognition.

They have two recurrent equations, one going forward and one backward in time.

The graph unfolded in time is still acyclic, hence back propagation in time still works.



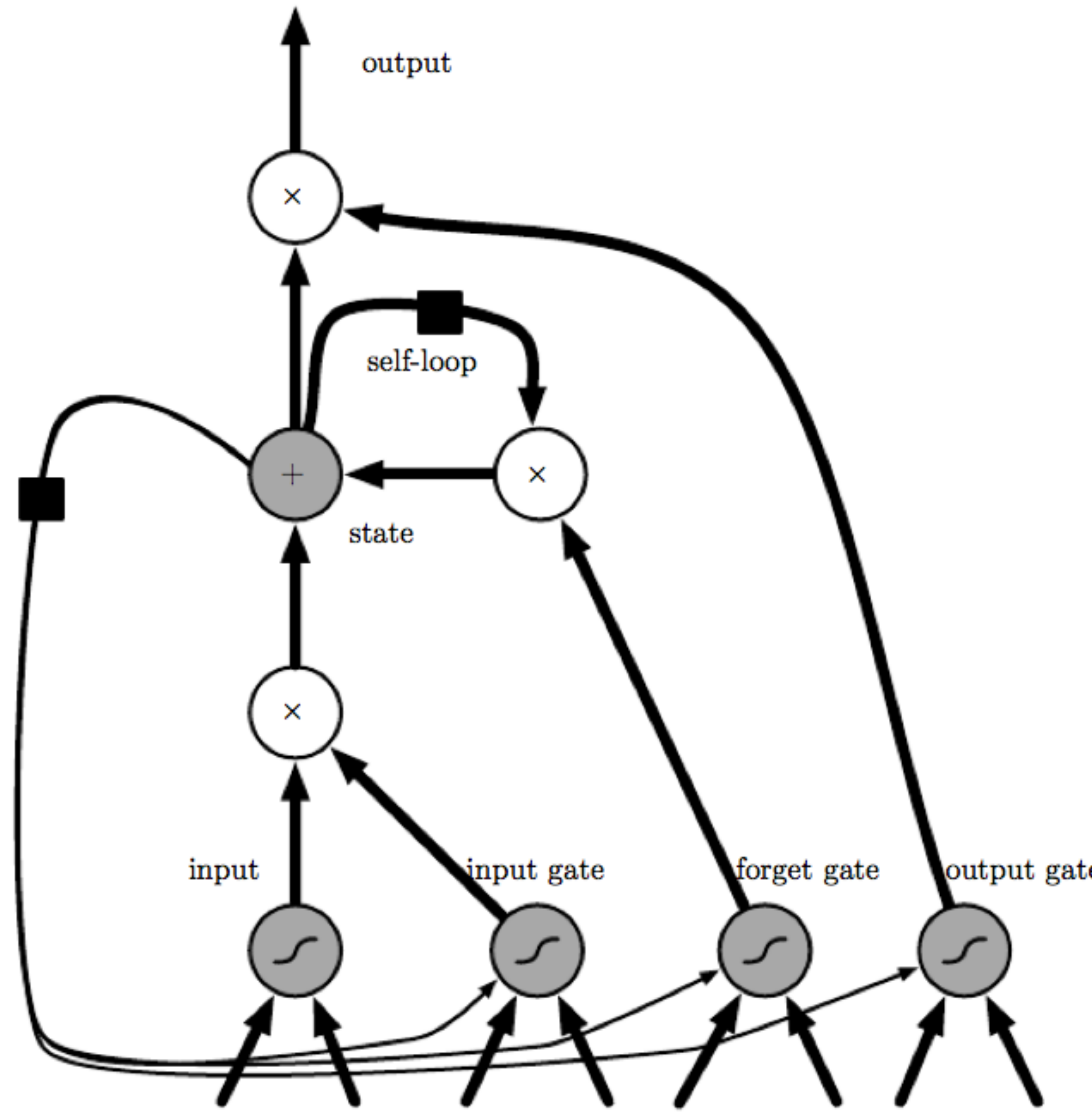
Gated Recurrent Neural Networks

A way to improve the ability of RNNs to keep a long term memory is to use **gated RNNs**. Gated units control how information is accumulated or forgotten, in an input dependent way.

The most common gRNN is the **Long-Short Term Memory** (LSTM) NN.

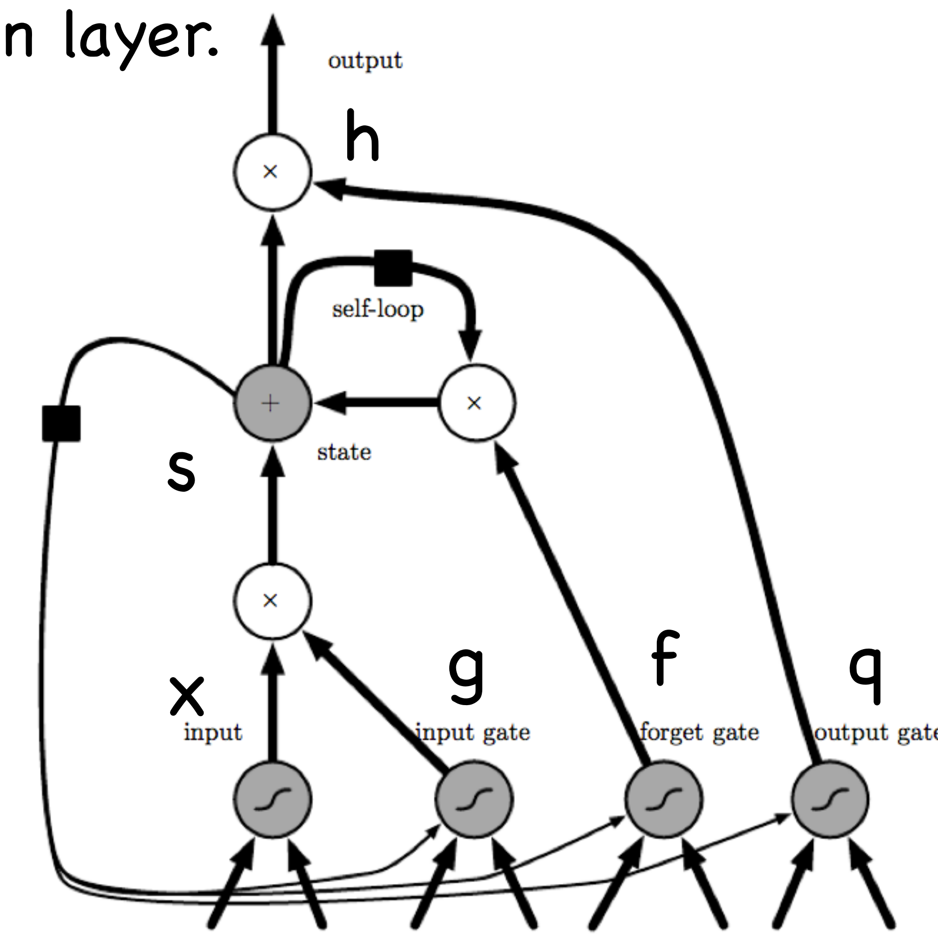
The core unit is a **leaky unit**, namely a node that accumulate information linearly, with an exponential decaying factor close to one:

$$\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1-\alpha)v^{(t)}$$



Gated Recurrent Neural Networks

In LSTM networks, leaky units have a decay rate controlled by a **forget gate f**, and modulated by the input and the hidden states. There are also **input gates g** and **output gates q** controlling the state and the hidden layer.



$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right)$$

$$f_i^{(t)} = \sigma \left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} \right)$$

$$g_i^{(t)} = \sigma \left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)} \right)$$

$$h_i^{(t)} = \tanh \left(s_i^{(t)} \right) q_i^{(t)}$$

$$q_i^{(t)} = \sigma \left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)} \right)$$

s - state of the LSTM cell
h - output of the LSTM cell
f - forget gate
g - input gate
q - output gate

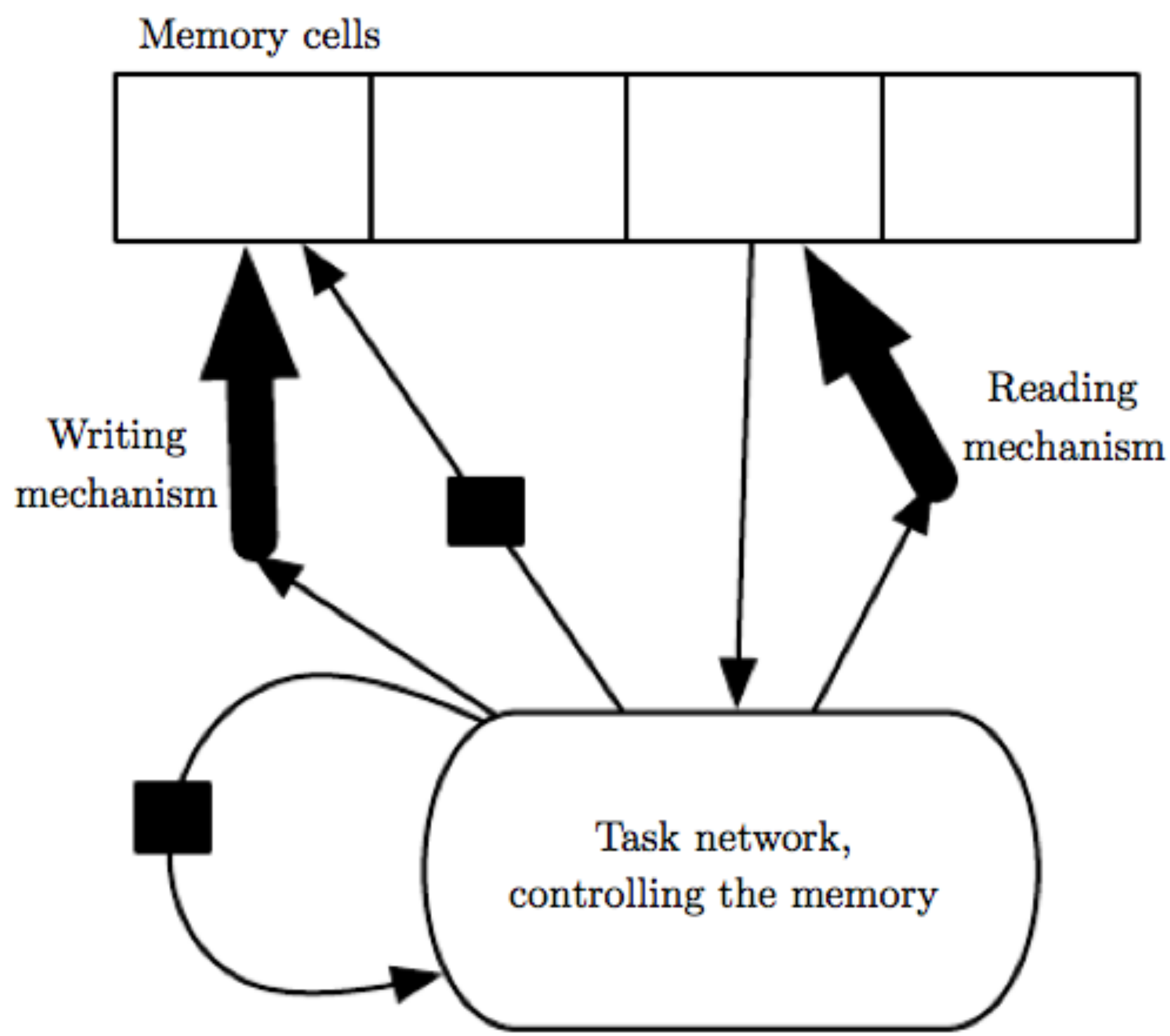
Neural Turing Machines

Another way to keep track of long term effects is to have an **explicit memory**, which can be read or written.

Neural Turing Machines extend a NN with an array of memory cells, and with mechanisms to read and write on them.

Reading and writing are done via **soft addressing**, namely each cell is read with a certain weight, or probability, which can be a function of the cell content (content-based addressing).

Soft read and write rules can be learned during training using a SGD approach.



Variants of such memory are heavily used for sequence modelling, under the umbrella of **attention mechanisms**.