Deep learning techniques

Overview

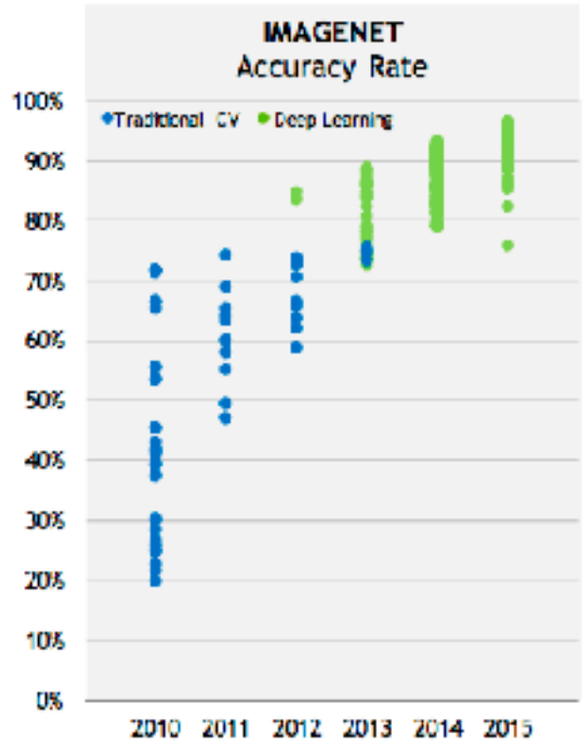
Yann-Aël Le Borgne, Gianluca Bontempi

Machine Learning Group
Université Libre de Bruxelles
Brussels, Belgium

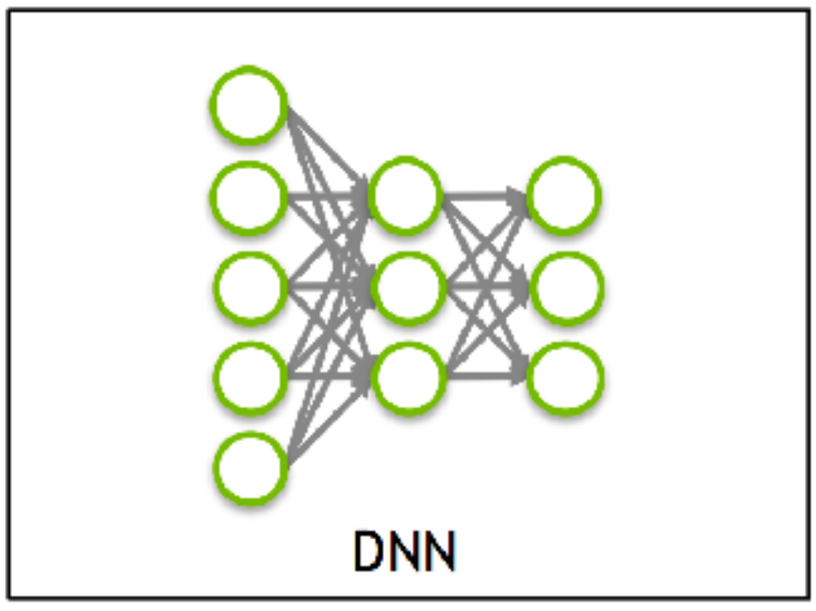
Deep learning = Neural networks. What is new?

- Deep learning are neural networks. Exist since the 70s (and even before).

- However, amazing recent improvements on state of the art ML



- What has changed:



Deep learning: Breakthroughs for tasks ranging from image classification to strategy games to speech recognition

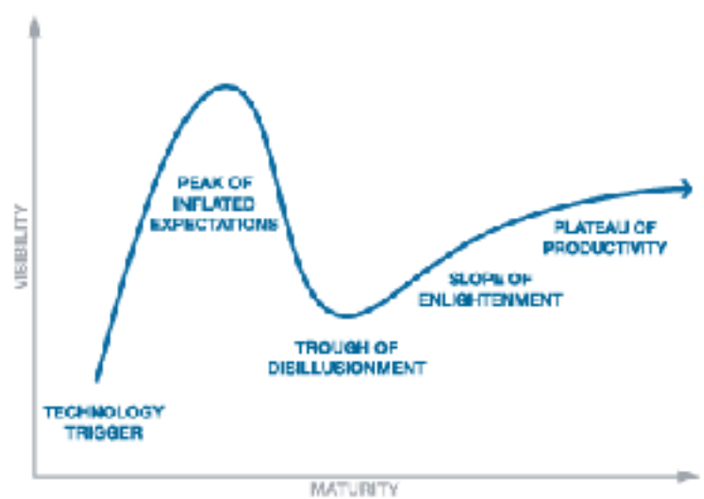
Google trend ‘deep learning’

Some landmarks:

- 2012: ‘AlexNet’ deep network. Provided 10% increase in accuracy on state of the art image classification (ImageNet)
- 2015: Classification accuracy of deep learning for images reported to outperform humans
- 2015: AlphaGo from Google DeepMind outperforms Go world champion
- 2016: Baidu and Microsoft report speech to text translation of deep learning systems outperforms humans
- 2017/2018: Going mainstream

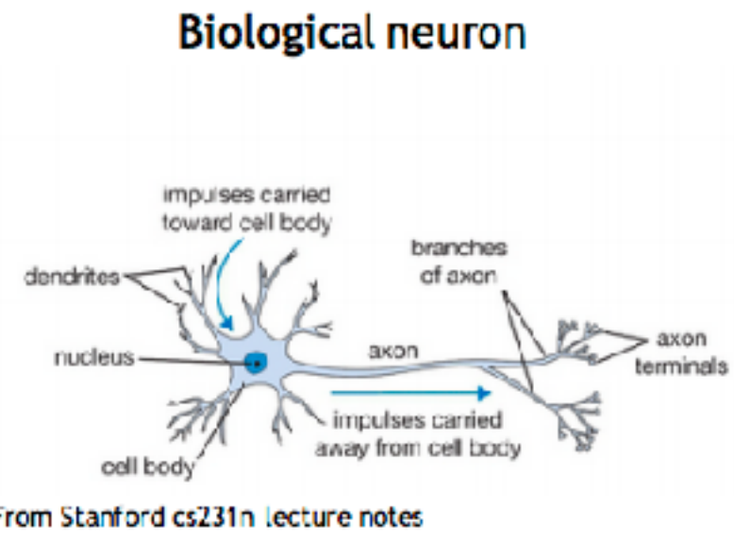
Deep learning everywhere

- Peak of inflated expectations?

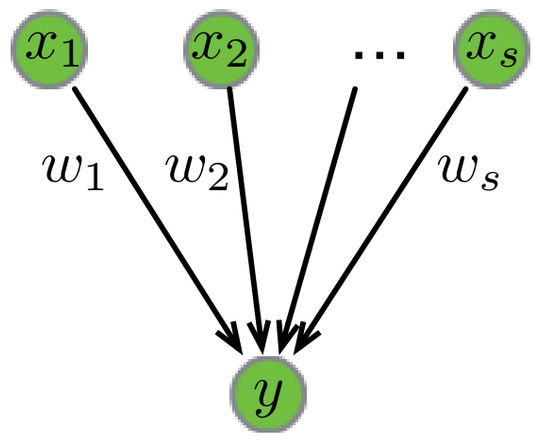


Recall: Perceptron

Basic computing unit: ‘neuron’ Example: Perceptron



Perceptron model (Rosenblatt, 1957)



$$x = (x_1, x_2, \ldots, x_s)^T$$

$$w = (w_1, w_2, \ldots, w_s)$$

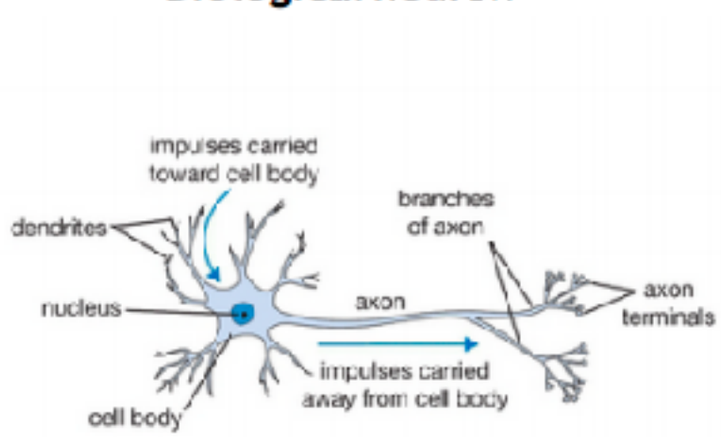
$$y = f(wx)$$

Neuron model

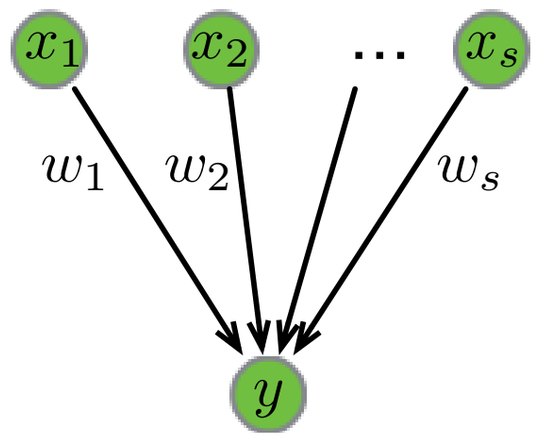
- Integrates activation values from previous layer by means of a scalar product *wx* : **It is a linear separator**
- Activation function *f* : $f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$

(unit step function)

Basic computing unit: ‘neuron’ Example: Perceptron



Perceptron model (Rosenblatt, 1957)



$$x = (x_1, x_2, \ldots, x_s)^T$$

$$w = (w_1, w_2, \ldots, w_s)$$

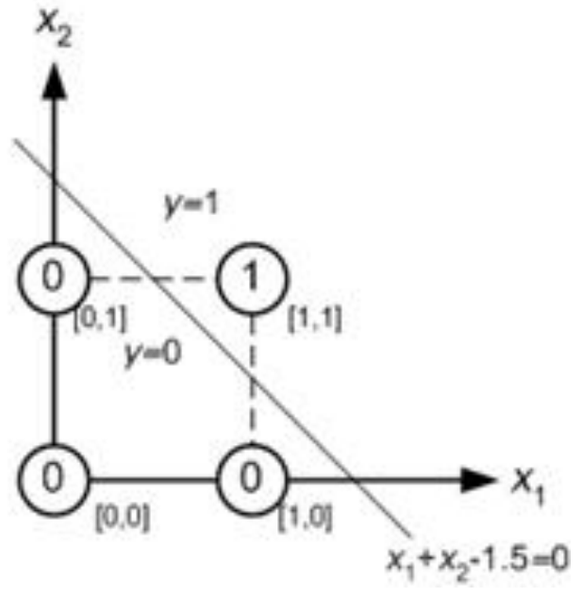
$$y = f(wx)$$

Neuron model

- Integrates activation values from previous layer by means of a scalar product *wx* : **It is a linear separator**
- Activation function *f* : $f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$

(unit step function)

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



Activation functions

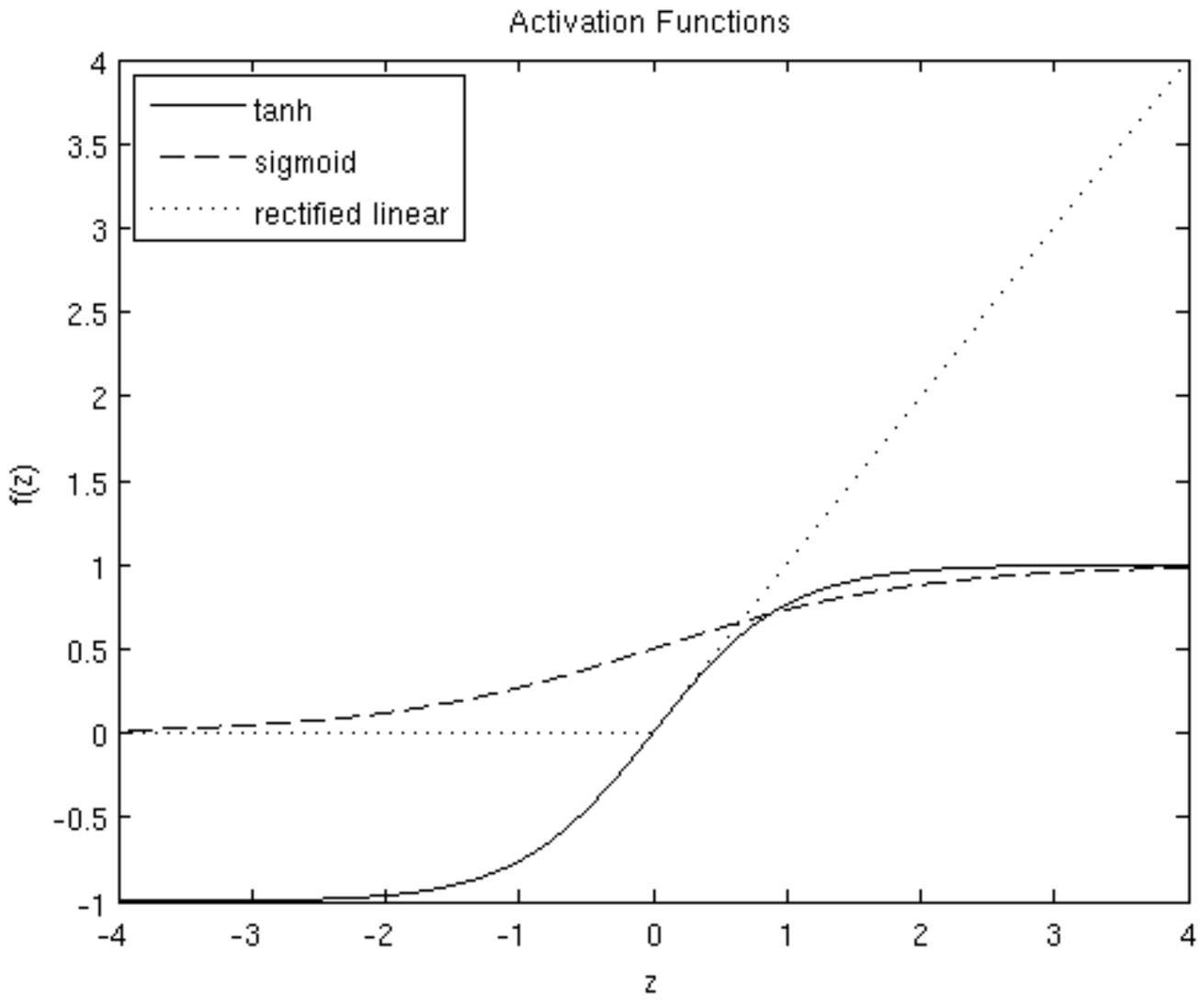
- Activation functions *f* : introduces nonlinearity.

- Examples of common (‘differentiable’) activation functions:

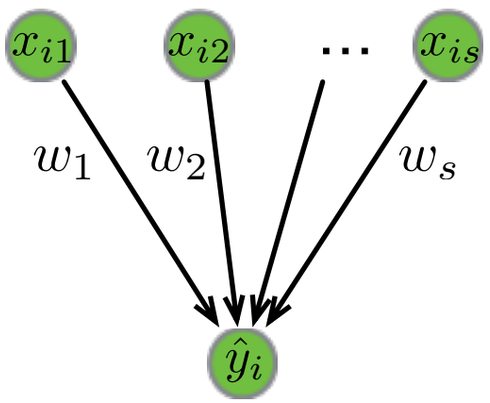
  - tanh,

  - logistic,

  - rectified linear (Relu)



Learning:
Define training set and loss function



$$x_i = (x_{i1}, x_{i2}, ..., x_{is})^T$$

$$w = (w_1, w_2, \ldots, w_s)$$

$$\hat{y}_i = f(wx_i)$$

- Training set

$$D_N = \{x_i, y_i\}_{1 \leq i \leq N}$$

N examples,
input x_i, output y_i

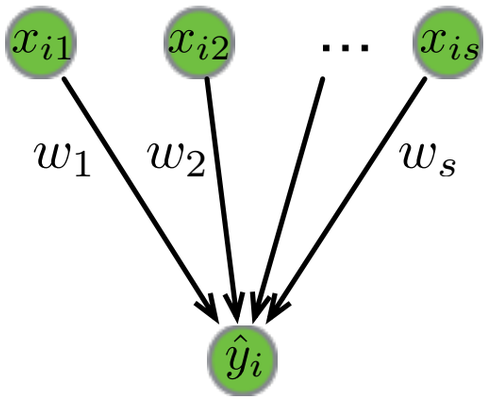
- Loss function

$$J(w, x_i, y_i)$$

Example: squared loss for regression

$$J(w, x_i, y_i) = \frac{1}{2} \sum_i (f(wx_i) - y_i)^2$$

Training: Gradient descent
Forward and backward pass



$$x_i = (x_{i1}, x_{i2}, ..., x_{is})^T$$

$$w = (w_1, w_2, \ldots, w_s)$$

$$\hat{y}_i = f(wx_i)$$

Repeat

$\forall (x_i, y_i) \in D_N$

- Forward pass:
  Compute output

- Backward pass
  Compute gradient

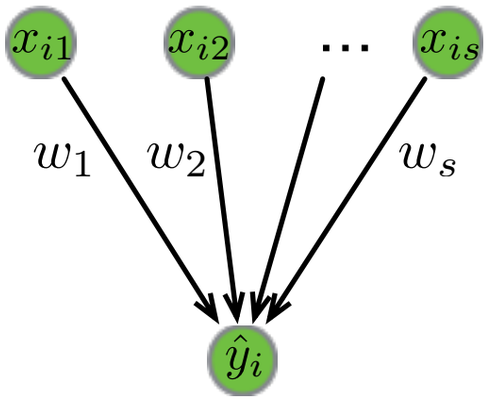
  $\Delta w = \nabla_w J(w, x_i, y_i)$

- Update parameters

  $w = w - \alpha \Delta w$

  (α : learning rate)

Training: Gradient descent
Forward and backward pass



$$x_i = (x_{i1}, x_{i2}, ..., x_{is})^T$$

$$w = (w_1, w_2, \ldots, w_s)$$

$$\hat{y}_i = f(wx_i)$$

For perceptron, gradient update (with squared loss) is:

$$\Delta w = (\hat{y} - y_i)x_i$$

- Demo: Tensorflow playground

Repeat

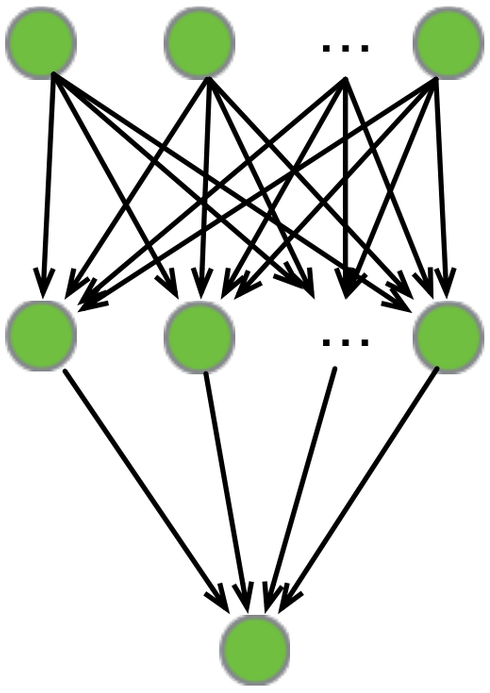
$\forall (x_i, y_i) \in D_N$

- Forward pass:
  Compute output
- Backward pass
  Compute gradient
  $\Delta w = \nabla_w J(w, x_i, y_i)$
- Update parameters
  $w = w - \alpha \Delta w$
  (α : learning rate)

Recall:

Multilayer perceptron

Multilayer perceptron (Fully connected layers)



$$x_i \in \mathbb{R}^{s1}$$

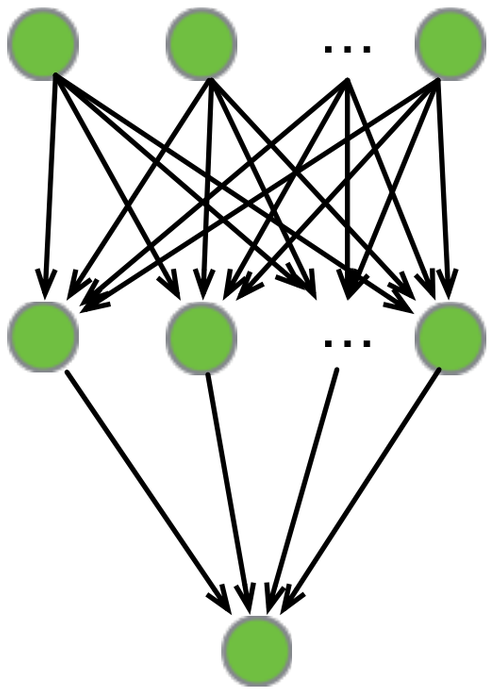
$$W^{(1)} \in \mathbb{R}^{s_2 \times s_1}$$

$$a_i = f(W^{(1)} x_i) \in \mathbb{R}^{s2}$$

$$W^{(2)} \in \mathbb{R}^{1 \times s_2}$$

$$\hat{y}_i = f(W^{(2)} a_i) \in \mathbb{R}$$

Fully connected layers
Forward and backward pass



$x_i \in \mathbb{R}^{s1}$

$W^{(1)} \in \mathbb{R}^{s_2 \times s_1}$

$a_i = f(W^{(1)} x_i) \in \mathbb{R}^{s2}$

$W^{(2)} \in \mathbb{R}^{1 \times s_2}$

$\hat{y}_i = f(W^{(2)} a_i) \in \mathbb{R}$

Repeat

$\forall (x_i, y_i) \in D_N$

- Forward pass:
  Compute output

- Backward pass
  Compute gradient

  $\Delta W = \nabla_W J(W, x_i, y_i)$

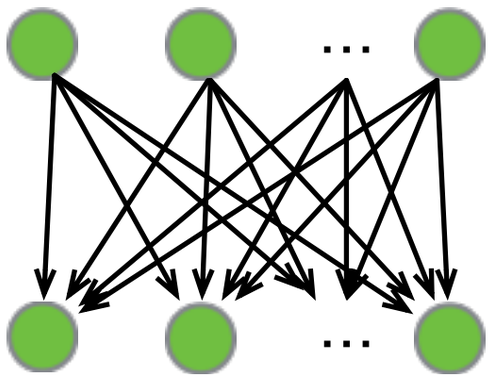
- Update parameters

  $W = W - \alpha \Delta W$

  (α : learning rate)

Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986). "Learning representations by back-propagating errors". Nature. 323 (6088): 533–536.

Fully connected layers act as space partitioners

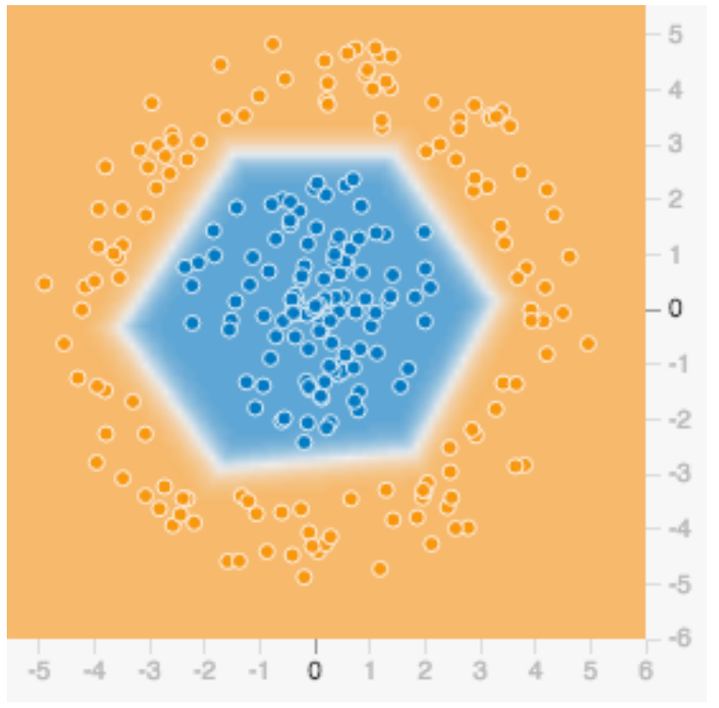


$$a^{(l)} \in \mathbb{R}^{s_l}$$

$$W^{(l)} \in \mathbb{R}^{s_{l+1} \times s_l}$$

$$a^{(l+1)} = f(W^{(l)} a^{(l)}) \in \mathbb{R}^{s_{l+1}}$$

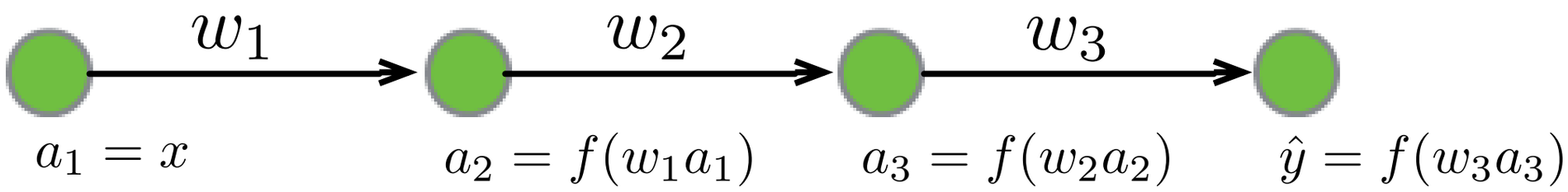
- Each output of a fully connected layer is a linear partitioner (product *Wa*) of the input space.
- Demo: Tensorflow playground



Summary

- Fully connected layers are space partitioners
- Gradient descent is used to parametrise weights W through learning (backpropagation)
- The learning rate determines how fast the descent is performed
- Open issues: Network structure, vanishing/ exploding gradient.

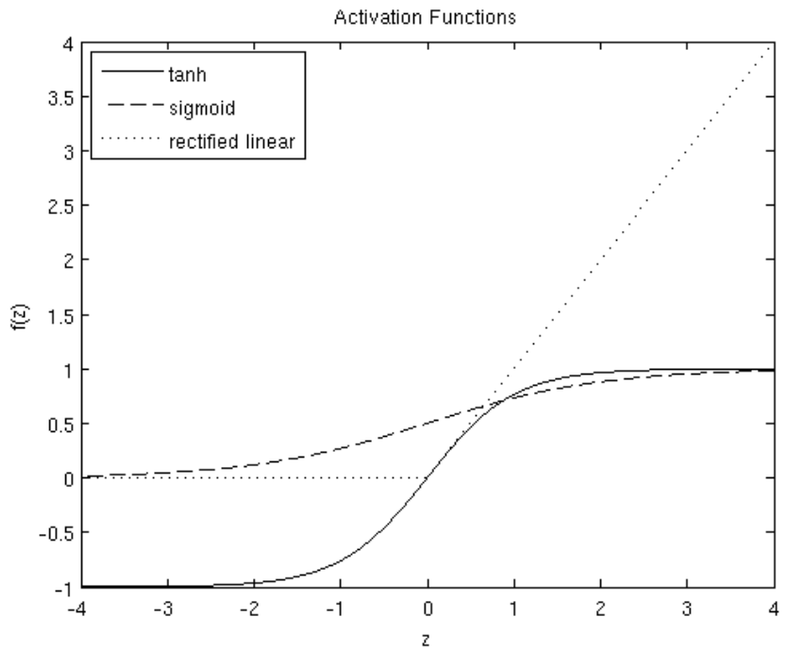
Gradient vanishing/exploding issue



$$\frac{\delta(J(W,x,y))}{\delta w_1} = \frac{\delta(J(W,x,y))}{\delta f(w_3 a_3)} \frac{\delta f(w_3 a_3)}{\delta w_3 a_3} \frac{\delta w_3 a_3}{\delta f(w_2 a_2)} \frac{\delta f(w_2 a_2)}{\delta w_2 a_2} \frac{\delta w_2 a_2}{\delta f(w_1 a_1)} \frac{\delta f(w_1 a_1)}{\delta w_1 a_1} \frac{\delta w_1 a_1}{\delta w_1}$$

$$\downarrow \qquad \downarrow \qquad \downarrow$$

$$f'(w_3 a_3) \qquad f'(w_2 a_2) \qquad f'(w_1 a_1)$$

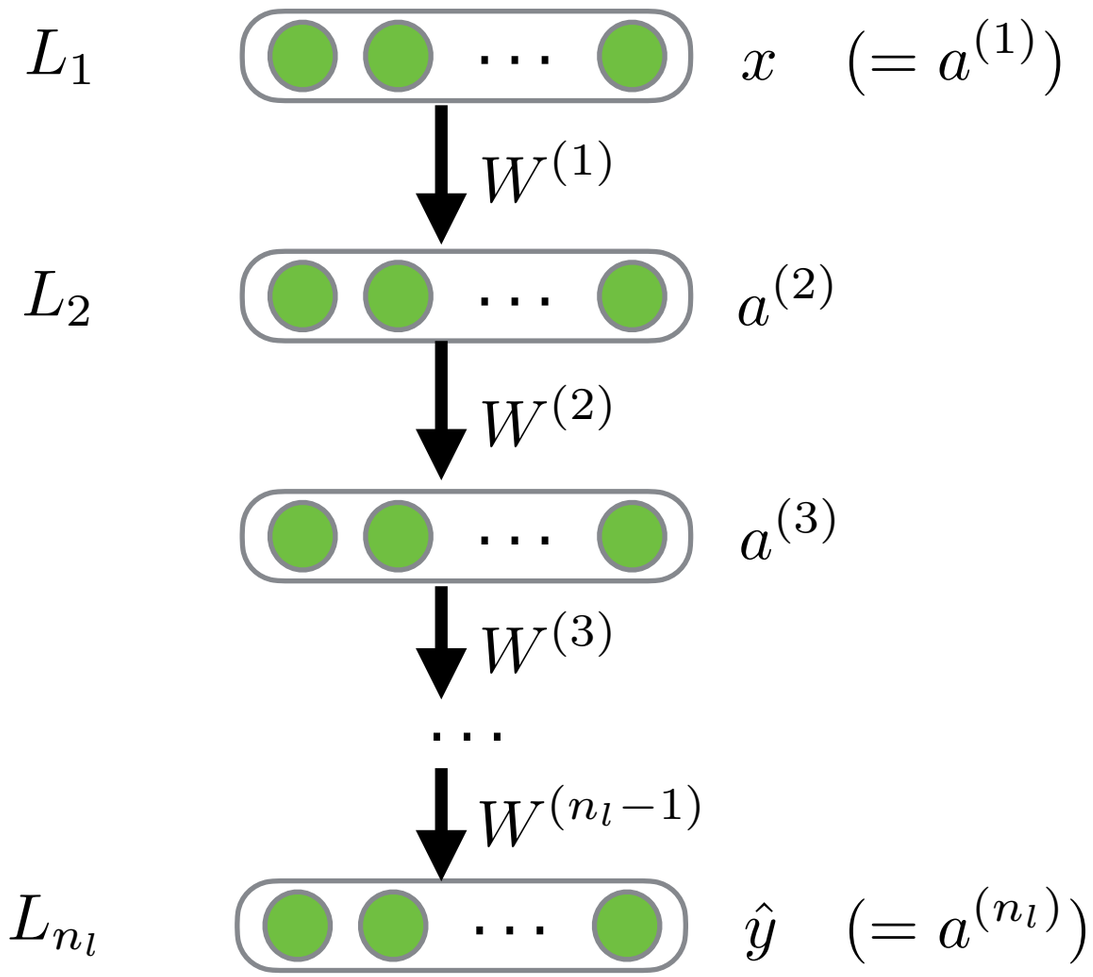


Issue caused by products of derivatives

RELU mitigates effect since its derivative is 1 if its input is positive (but 0 if input negative - ‘dead neuron’ issue)

Deep networks: Generalisation

Deep networks - DNN



$$\hat{y} = DNN(W, x)$$

- An input layer (x)
- Layers of 'hidden' computing units ('neurons'), parametrised (W)
- Each layer computes $a^{(l)} = h^{(l)}(W^{(l-1)}, a^{(l-1)})$
- An output layer (*y*)
- Overall, y=*DNN(W,x)*

Notes

- In DNN, layers may achieve a wide range a different processing tasks (partitioning, convolutional, pooling, memory units, …).
- The processing task (or function) is denoted *h*, and takes as parameters *W*
- The number of layers can be very high (hundreds)
- New architectures aim at mitigating the vanishing/ exploding gradient problem

Convolutional neural networks (CNN)

CNN
Main purpose

- Automatically extract high-level and relevant spatial/ temporal features

- Convolutional neural networks are state of the art for image classification, and are applied successfully to many domains (text mining, time series analysis, genomics, …)

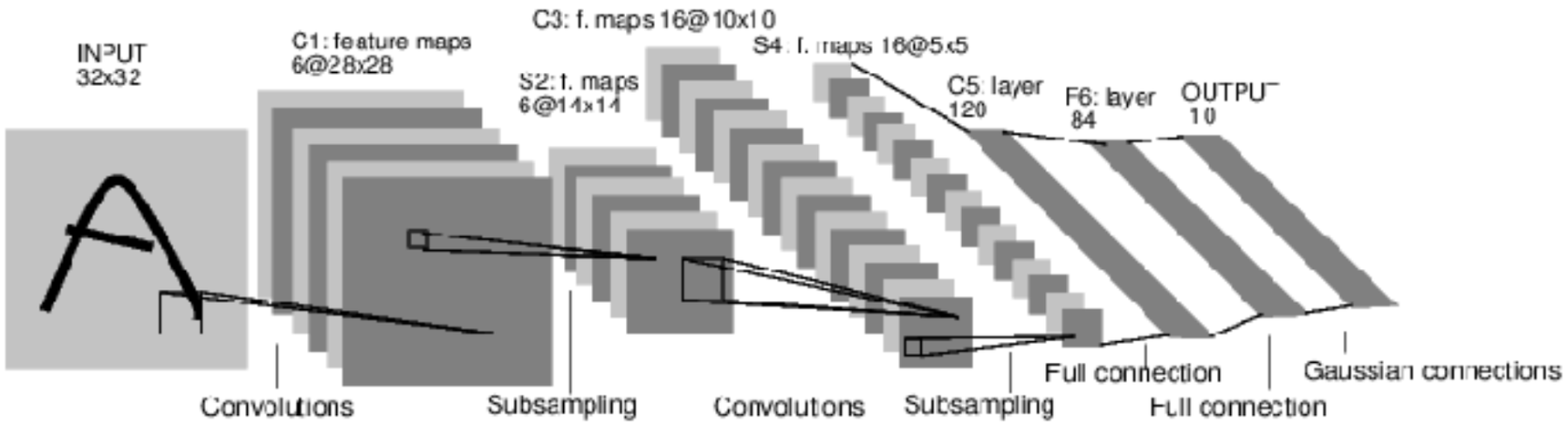
- Main types of layers: Convolution, pooling, and fully connected

- For the sake of clarity, we will use 2D images (spatial data) as examples of inputs in this section

CNN examples

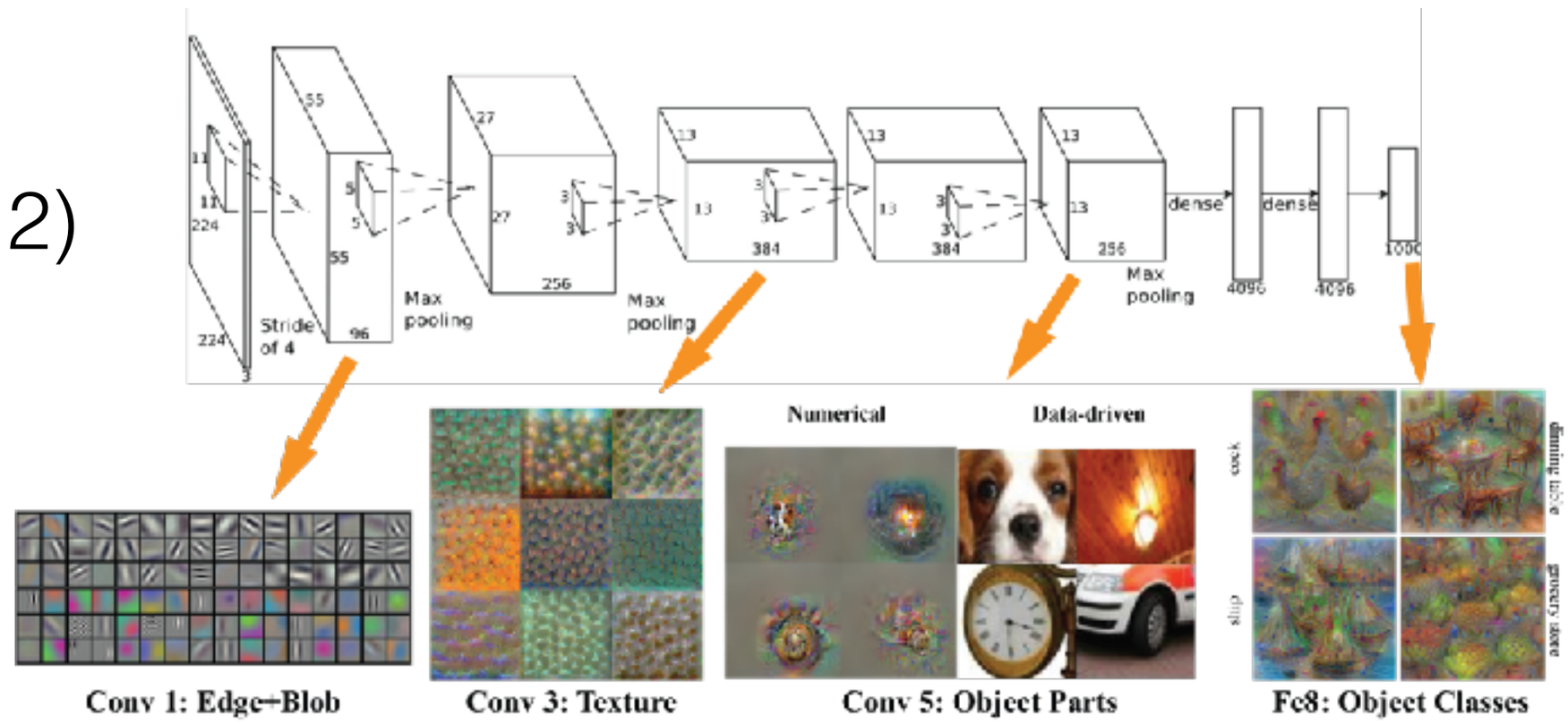
LeNet (1998)

LeCun et al.



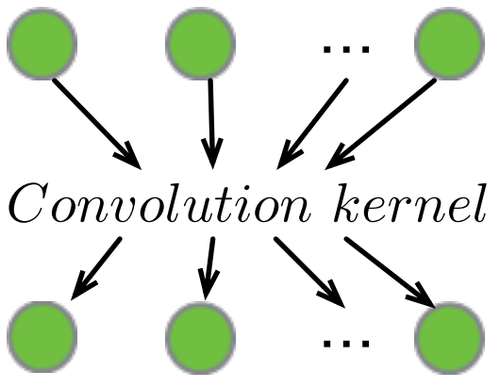
AlexNet (2012)

Krizhesvky et al.



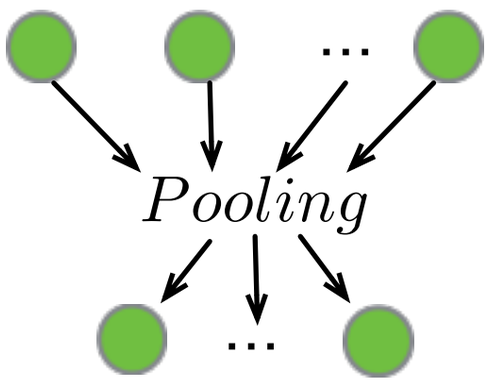
Main types of layers and their functions

Convolutional



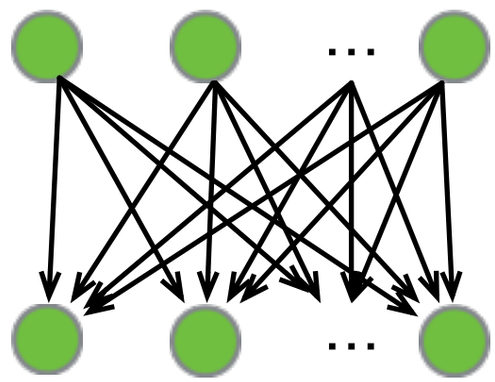
Filter

Pooling



Downsampler

Fully connected



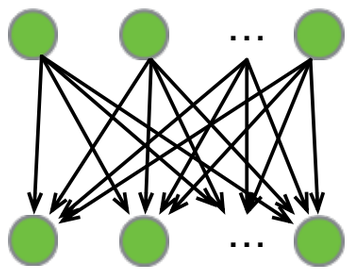
Partitioner

Note: The size of input and output layers usually differ.
Pooling always leads to an output layer of smaller size.

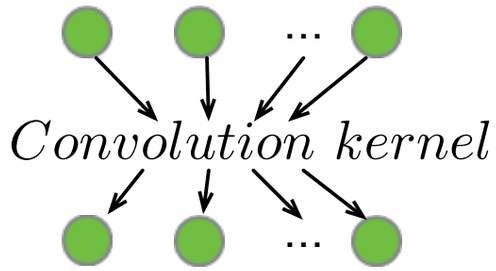
Convolution layer (Conv)

- Whereas a FC layer acts as a space partitioner, a conv layer act as a **pattern filter**

- As the FC layer, a conv layer has inputs neurons, a set of weights, and output neurons.



Fully connected



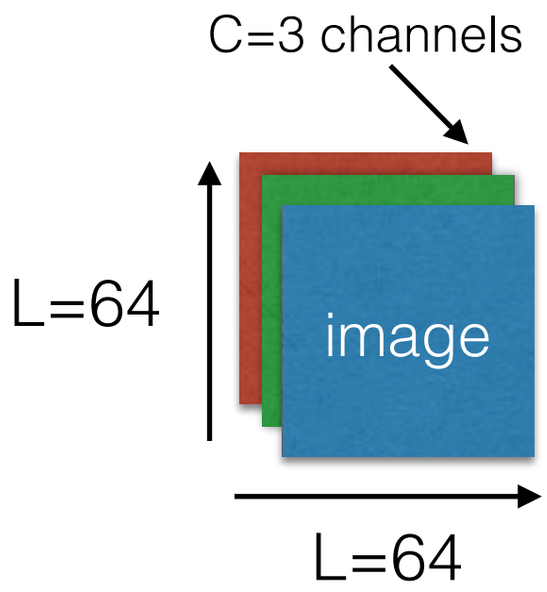
Convolutional

- However, the weights (called kernel) are not tied to specific neurons.

Convolution layer

- A different representation and terminology is therefore often preferred to better reflect the convolution layer purpose

- The input and output neurons are arranged in volumes to reflect the spatial dependencies of images. Inputs are represented as **‘stacks’ of 2D data, called channels**



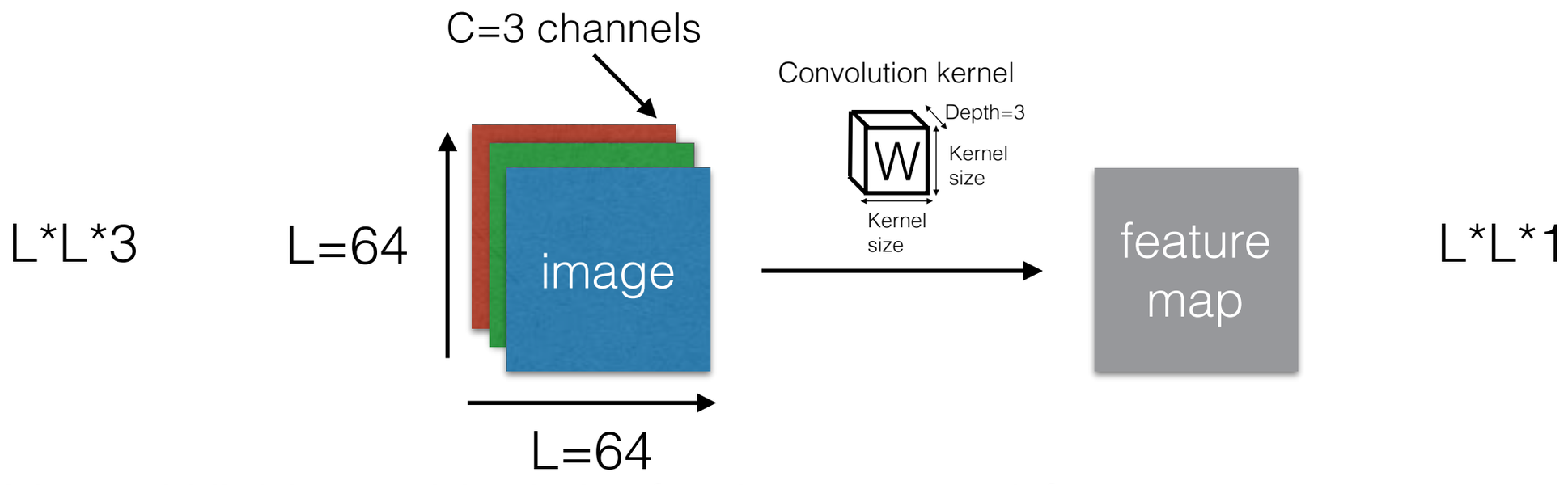
An input block has size L*L*C (‘neurons’)

Example of input image of size 64*64 with 3 channels (e.g., RGB)

Convolution layer

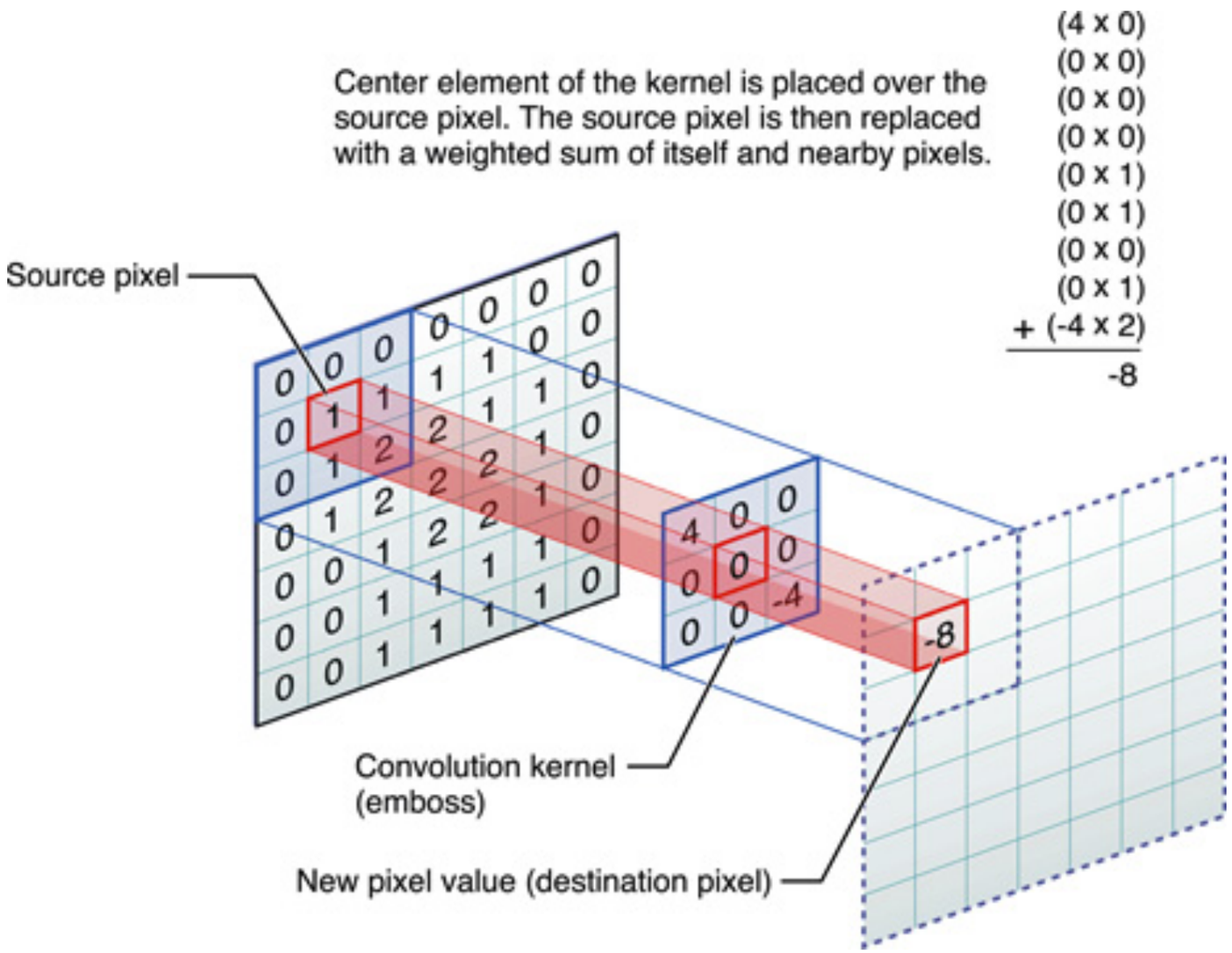
- We still have the parameters (W), which are represented by another array of size K*K*C, called **kernel**

- The convolution of the input array (L*L*C) by the kernel (K*K*C) results in an array of size L*L*1, called **feature map (FM)**



Note: Width and height for inputs and feature maps may differ. Feature map may have a size smaller than L*L*1. More on that later.

How does convolution work?

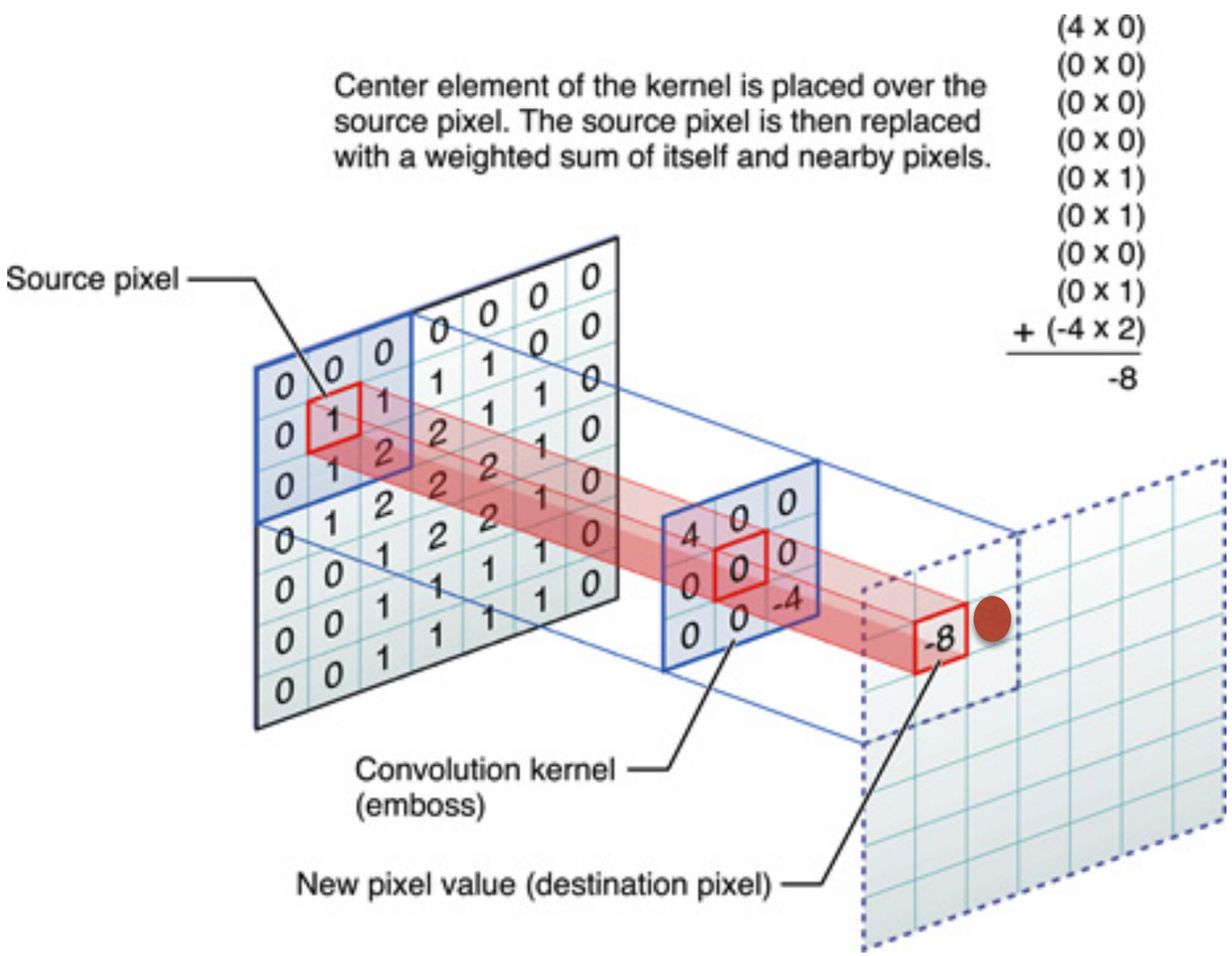


Scalar product of kernel with each position in the input array

Compute $\sigma(Wx(r,c)+b)$ for all valid (r,c)

Credit: Apple

How does convolution work?



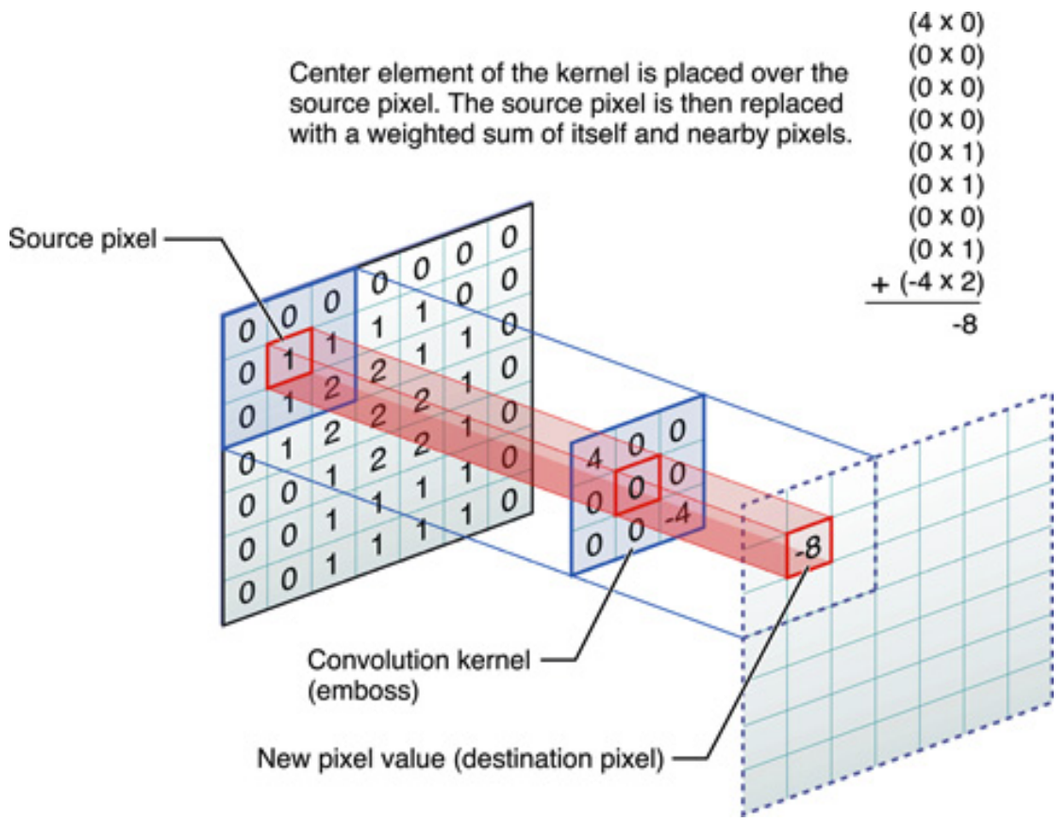
Scalar product of kernel with each position in the input array

Compute $\sigma(Wx(r,c)+b)$ for all valid (r,c)

Credit: Apple

- What would be the value on the red dot?
- For which input would the convolution be maximum?

Convolution kernels are pattern filters



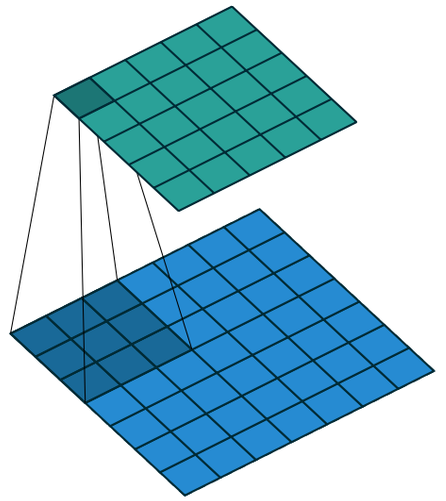
Convolution will give highest scores on the feature map for regions in the input image that match the kernel

- After learning, kernels become edge detectors, texture detectors, and in the last layers, very high levels (cat, face, …) detectors

- Demo: Image kernel

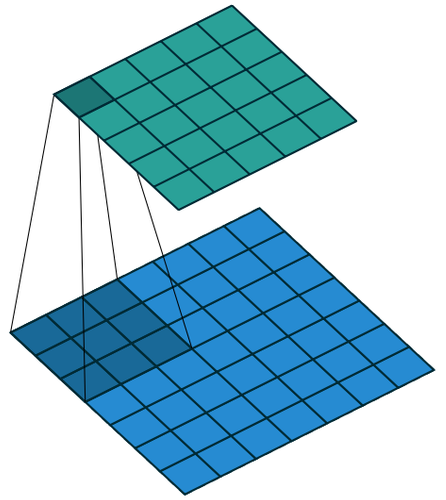
Padding

- If input FM is 7*7 and kernel 3*3, what is the size of the output FM?



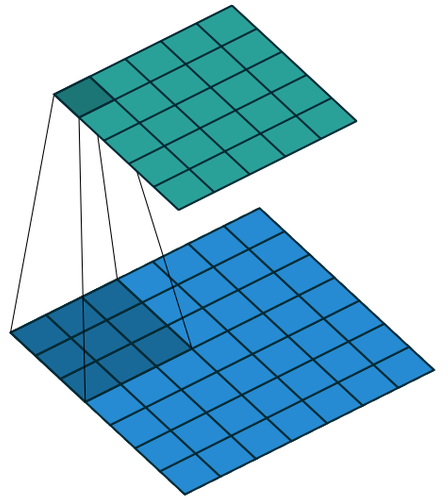
Padding

- If input FM is 7*7 and kernel 3*3, what is the size of the output FM?
- Any way to make output size the same as input size?

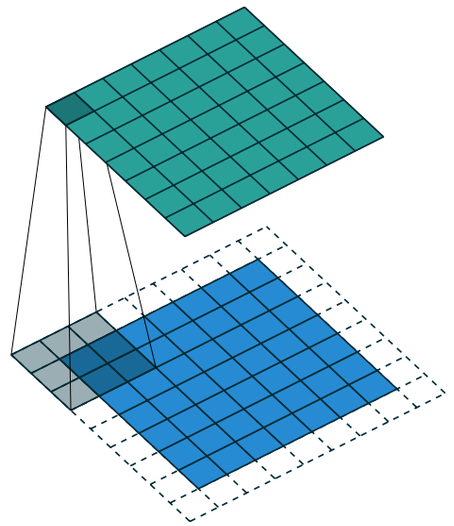


Padding

- If input FM is 7*7 and kernel 3*3, what is the size of the output FM?
- Any way to make output size the same as input size?

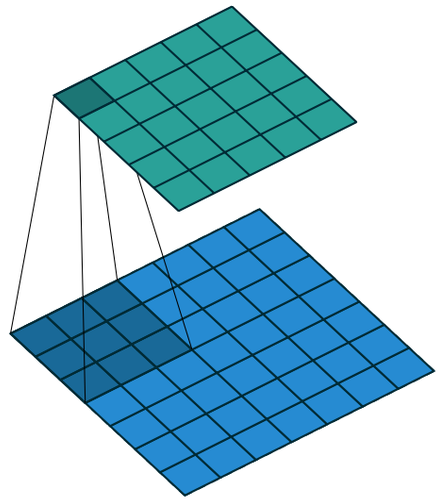


- **Padding** consists in adding extra zeros around input FM so that the dimension of the input FM and output FM is the same

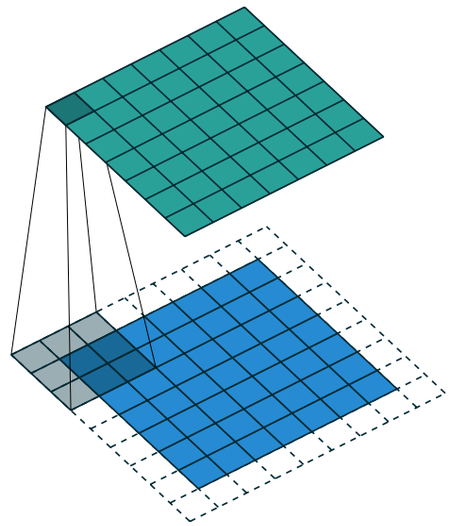


Padding

- If input FM is 7*7 and kernel 3*3, what is the size of the output FM?
- Any way to make output size the same as input size?



- **Padding** consists in adding extra zeros around input FM so that the dimension of the input FM and output FM is the same

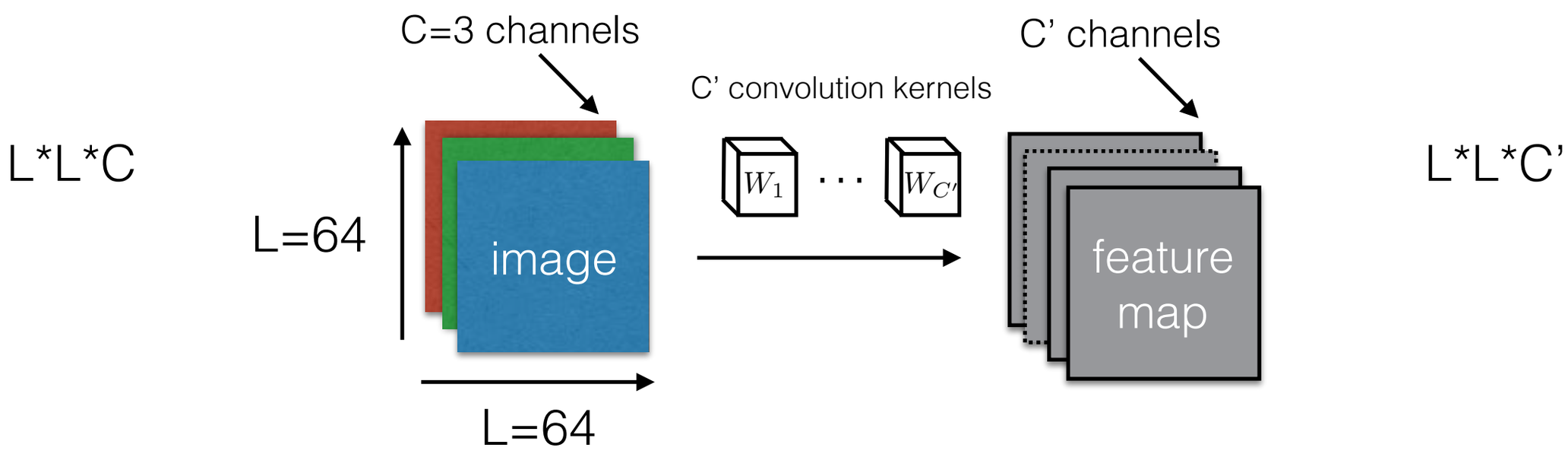


If kernel is k*k, what should be the padding width?

Input and output channels

- If the input has C channels, the kernel has depth C. Scalar product is extended to the ‘depth’ of the input

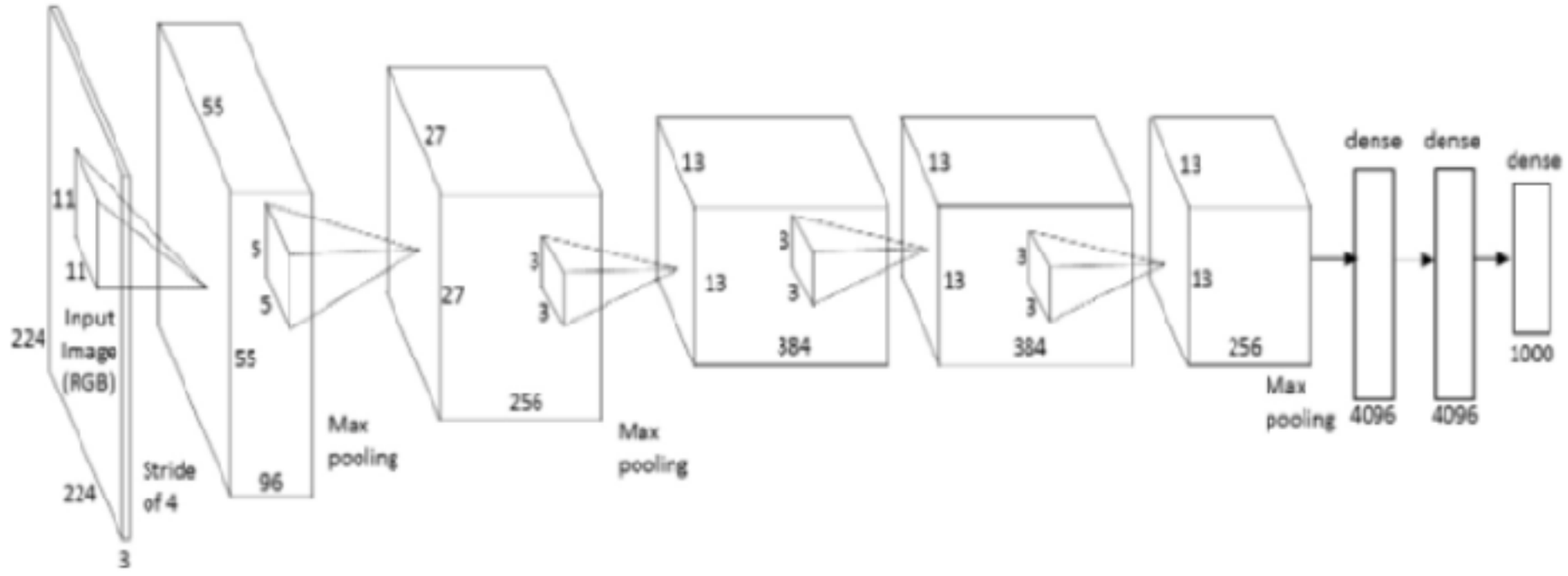
- The number of kernels determine the number of feature maps (output channels)



Parameters

- There are typically tens or hundreds of kernels per convolution layers, leading to many output feature maps

Example:
AlexNet (2012)
Krizhesvky et al.



- Number of parameters
  - Convolution to convolution layer : K*K*C*C’
  - Convolution to fully connected layer: L*L*C*S_I (may be very high!)

Programming example (Keras)

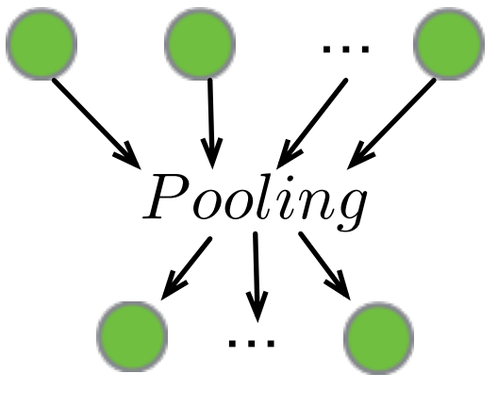
- Recent DL toolboxes make it very easy to implement DNN layers such as convolution layers. Example in Keras:

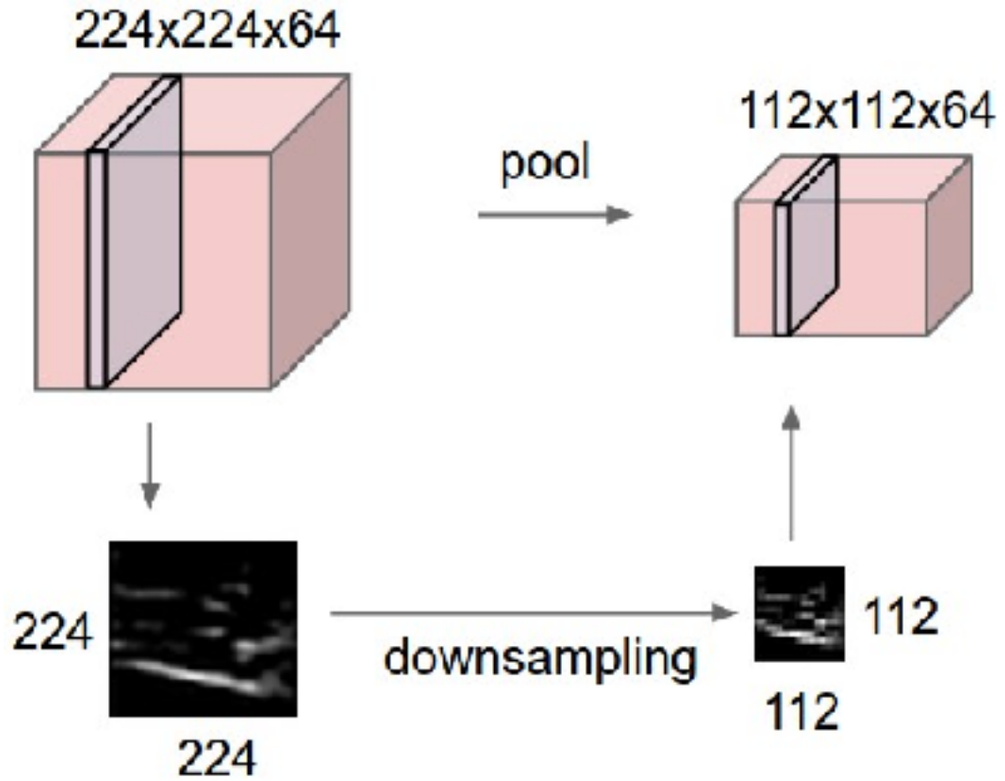
```
# apply a 3x3 convolution with 64 output filters on a 256x256 image:
model = Sequential()
model.add(Convolution2D(64, 3, 3, border_mode='same', input_shape=(3, 256, 256)))
# now model.output_shape == (None, 64, 256, 256)

# add a 3x3 convolution on top, with 32 output filters:
model.add(Convolution2D(32, 3, 3, border_mode='same'))
# now model.output_shape == (None, 32, 256, 256)
```

- Note: ‘border_mode’ adds padding so the output size is the same as input size

Pooling layer
acts as a downsamplers





- Allows to reduce dimensionality

Credit: cs231n

- Pooling typically reduce the size of the input block by 4, using the max or mean operators.

CNNs: Summary

- In their most simple forms, CNNs are a set of convolutional and pooling layers, followed by a couple of fully connected layers

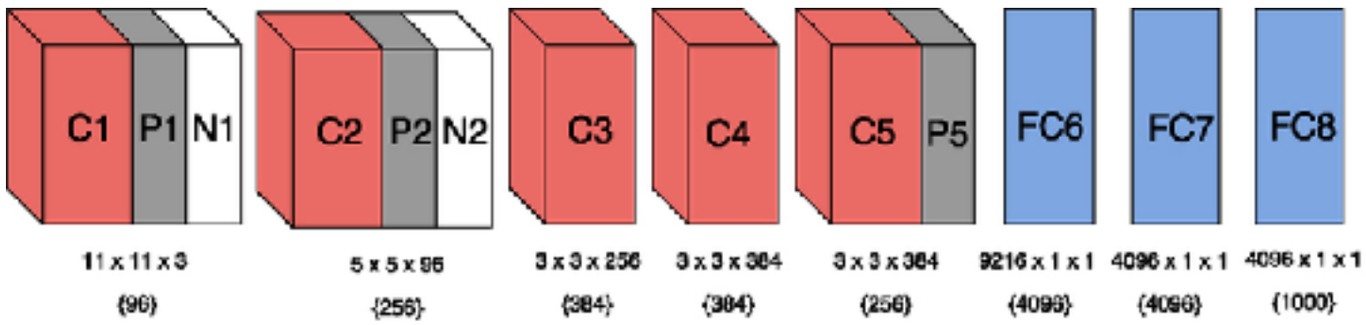
- A convolution layer is parametrised by a set of kernels

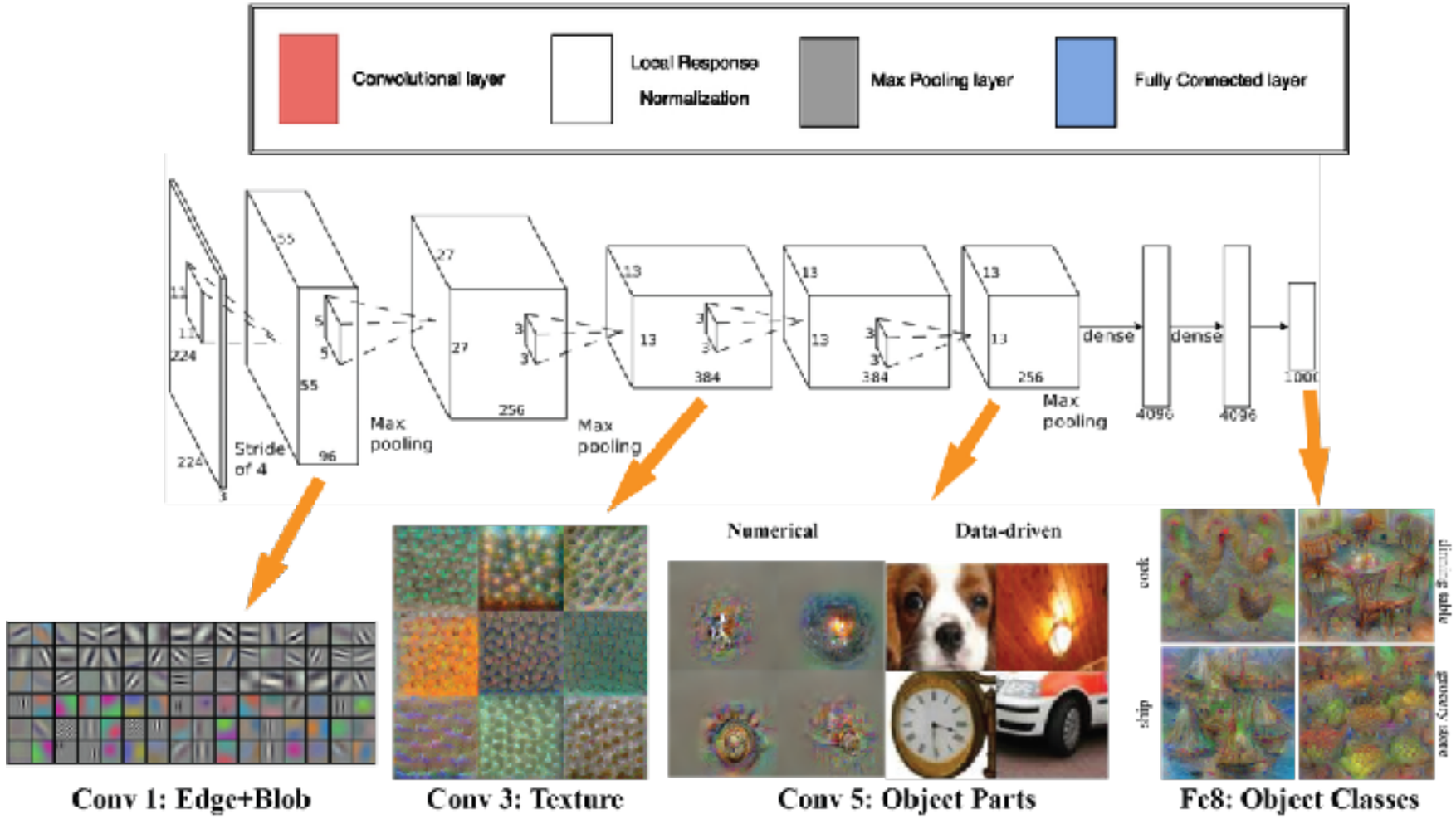
- Padding allows to keep the output FM the same size as the input FM.

- Pooling allows to reduce the dimensionality of a FM.

- Demo: Digits classification with ConvNetJS

Example revisited: AlexNet





Application components

- Task objective: image classification
- Training data
  1.28M images,
  1000 classes
- Network architecture
  5 conv, 3FC
  650m neurons, 60M parameters
- Training time
  ~few days on a GPU

Demo

Deep Visualization Toolbox

yosinski.com/deepvis

#deepvis


Jason Yosinski


Jeff Clune


Anh Nguyen


Thomas Fuchs


Hod Lipson

Cornell University

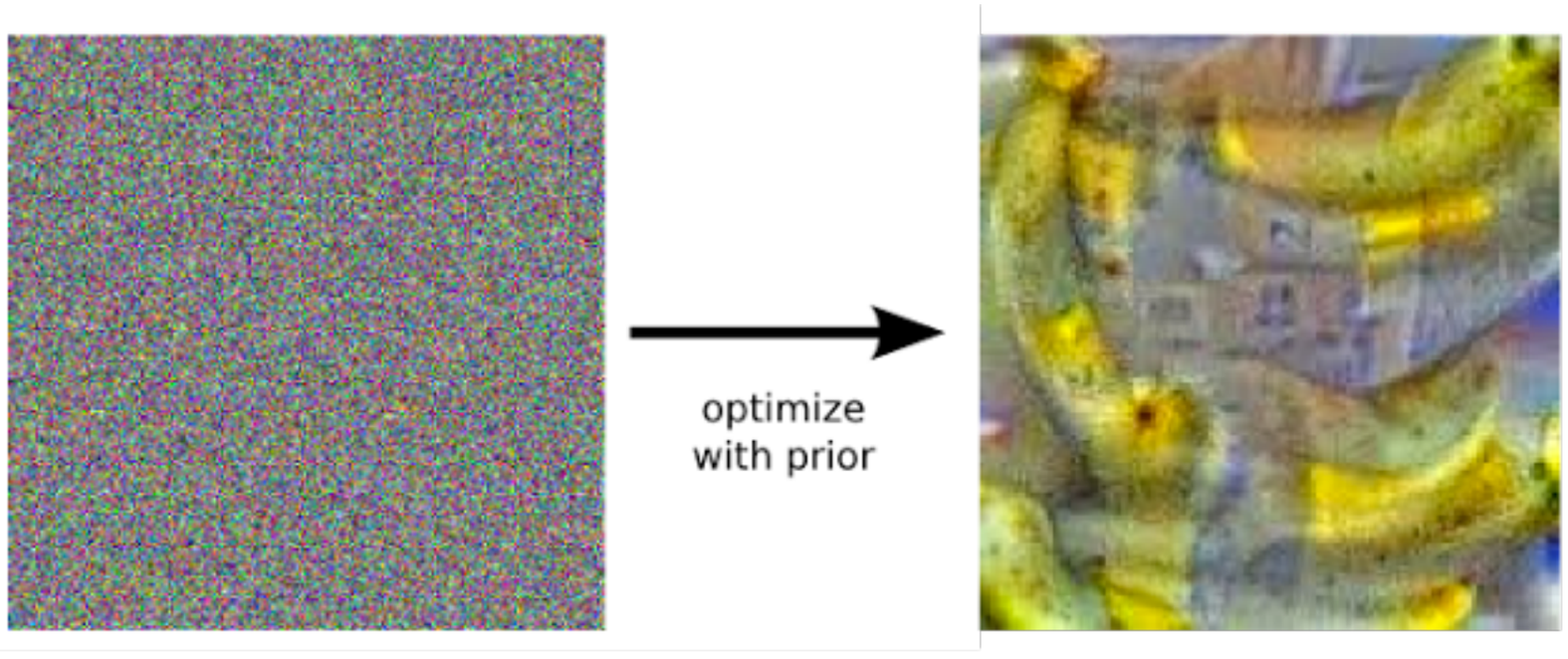
University of Wyoming

NASA Jet Propulsion Laboratory
California Institute of Technology

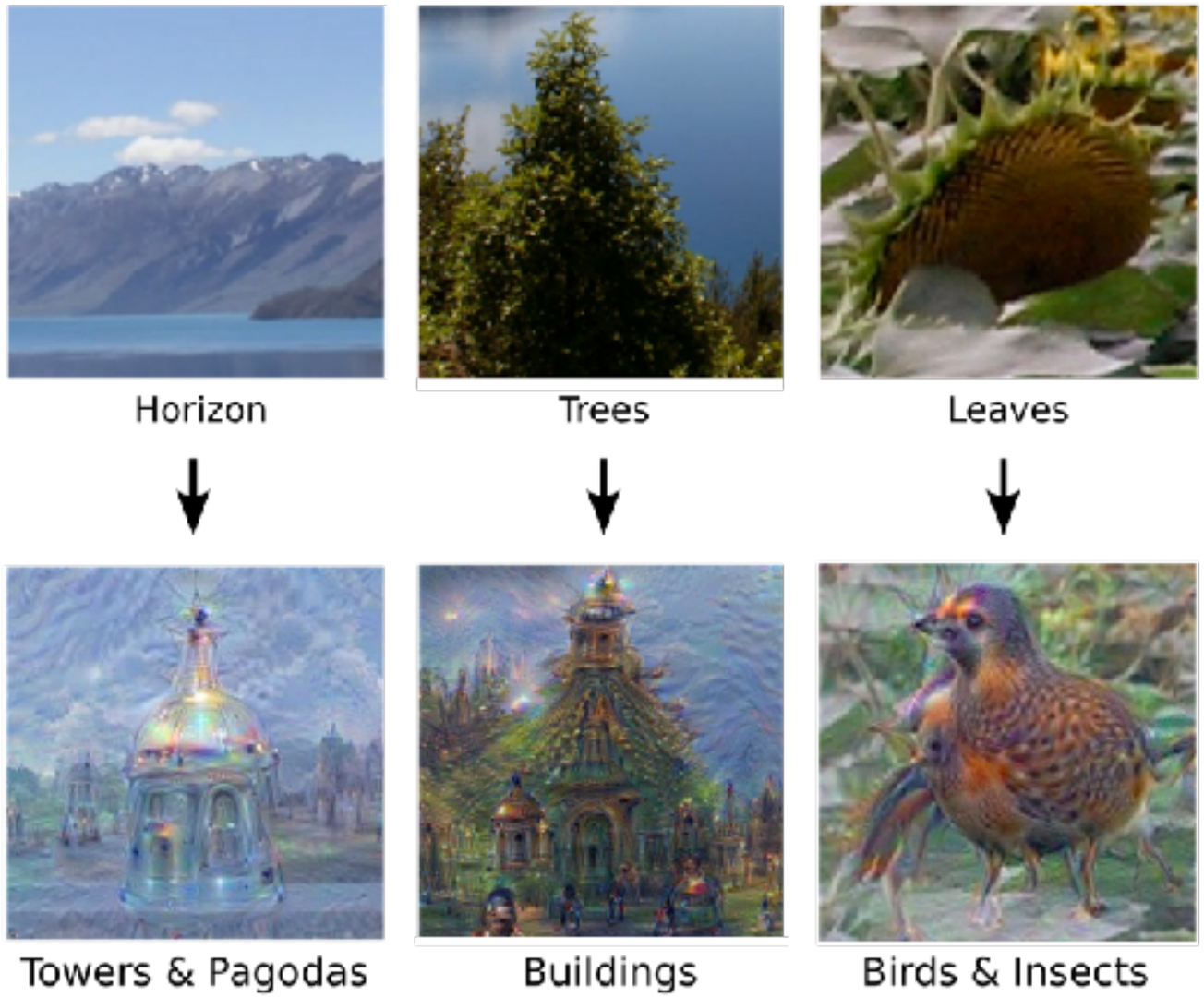
What do CNN neurons see?

- Fix the weights, and apply gradient descent to input x, i.e., optimise

$$\frac{\delta J(W,x)}{\delta x}$$



‘Deep dreams’



https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html

Style transfer

Optimise $\frac{\delta(J_{content}(W,x) + J_{style}(W,x))}{\delta x}$

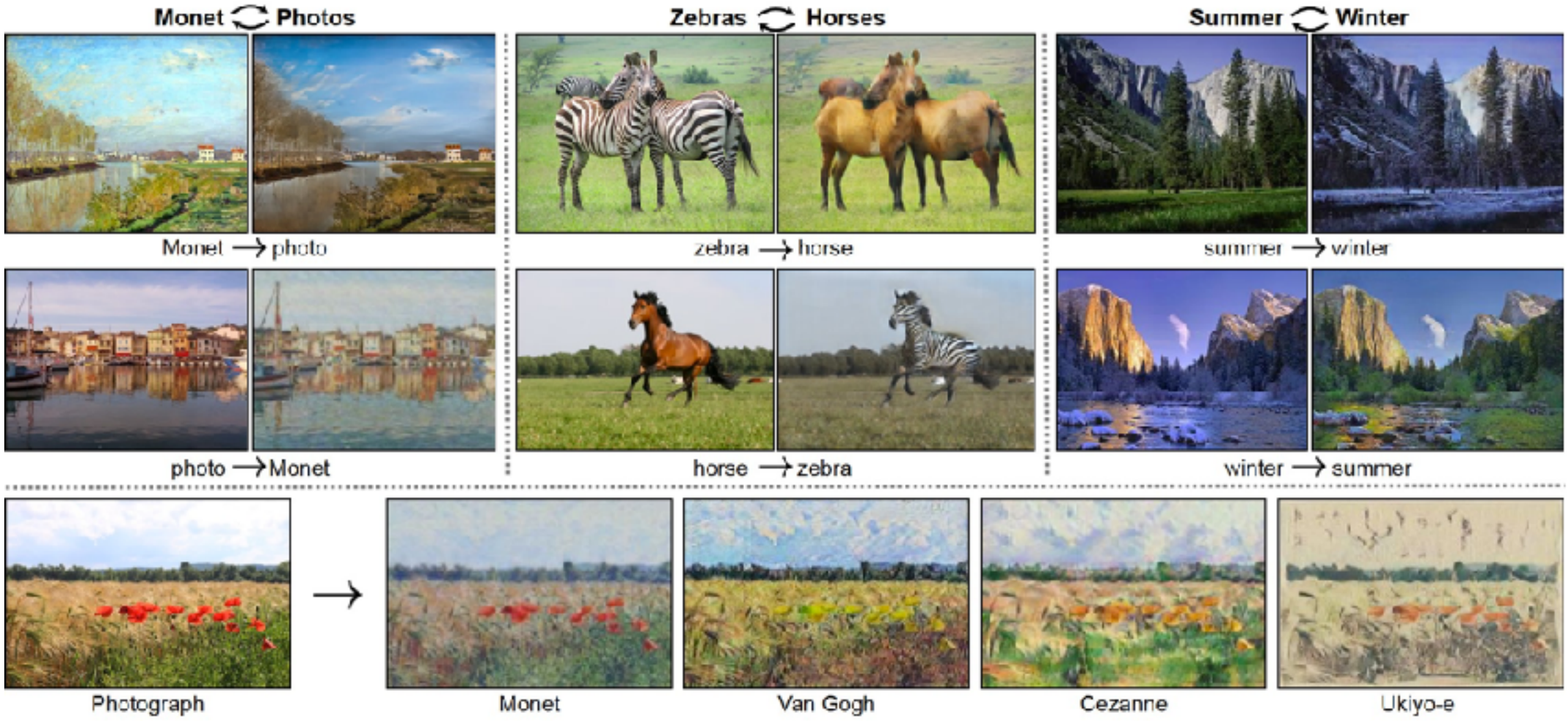
Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. "A neural algorithm of artistic style." arXiv preprint arXiv:1508.06576 (2015).

Style transfer

Optimise $\frac{\delta(J_{content}(W,x) + J_{style}(W,x))}{\delta x}$

Ruder, Manuel, Alexey Dosovitskiy, and Thomas Brox. "Artistic style transfer for videos." German Conference on Pattern Recognition. Springer International Publishing, 2016.

CycleGAN



Zhu, Jun-Yan, et al. "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks." arXiv preprint arXiv:1703.10593 (2017).

Evolution of architectures for image classification

IMAGENET **Large Scale Visual Recognition Challenge**

Year 2010

NEC-UIUC

Dense grid descriptor: HOG, LBP

Coding: local coordinate, super-vector

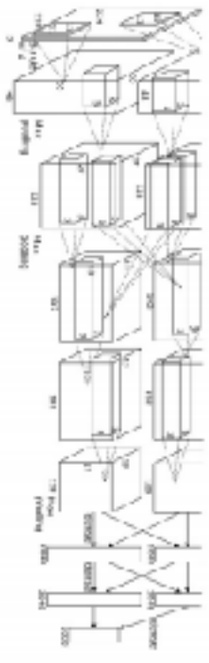
Pooling, SPM

Linear SVM

[Lin CVPR 2011]

Year 2012

SuperVision



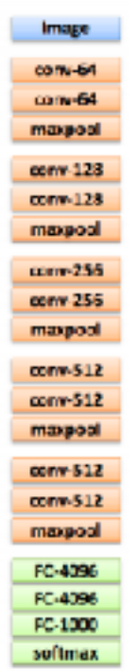
[Krizhevsky NIPS 2012]

Year 2014

GoogLeNet

[Szegedy arxiv 2014]

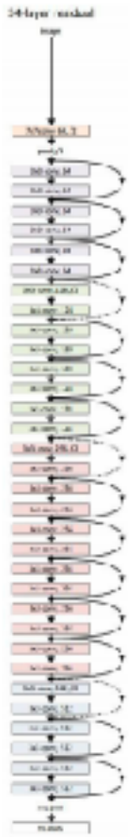
VGG



[Simonyan arxiv 2014]

Year 2015

MSRA

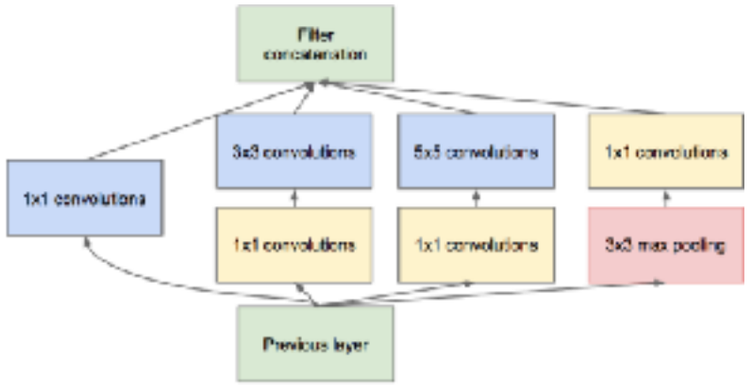


Trend: Narrower and deeper.

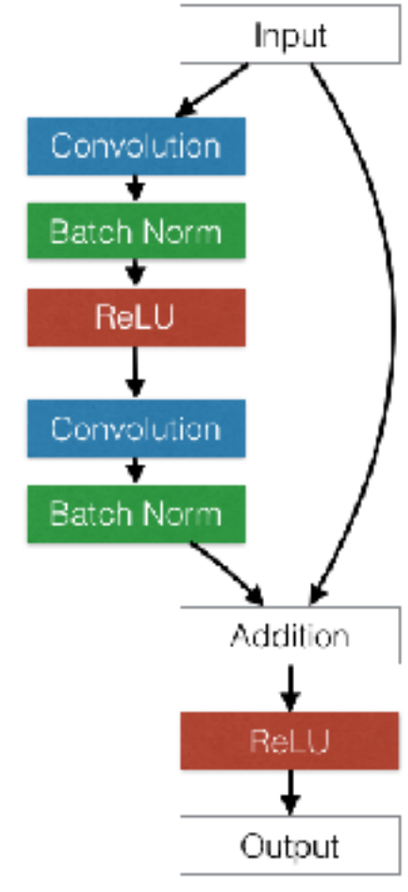
Some architecture details

- AlextNet (2012)
    - 8 layers, 60M parameters. 15.3% error.
- VGGNet (2014):
    - 16 layers, 140M parameters. 7.4%
- GoogleNet (2014):
    - 22 layers, 4M parameters
    - Inception module.

      6.66% error.
- ResNet (2015):
    - 152 layers. 3.57% error.
    - 2M parameters



Inception module



ResNet module

Notes

- Batch normalisation and pooling layers may not be necessary (all convolutional networks)

- ResNet have theoretical support for handling very high number of layers

- Trend: Narrower, deeper, modularisation, fewer parameters

Recurrent neural networks

Motivations
Handle sequential data

Predict value of an output variable given sequential past observations.

Example: Text translation, time series prediction, genomics data, speech recognition, control, customer behaviour, …

Input: Multivariate sequence $X_i \in \mathbb{R}^{T*s}$
Output: Vector $y_i \in \mathbb{R}^{s'}$

T: Length of sequence
s: Dimension of input space
s': Dimension of output space

Motivations
Handle sequential data

Example: Character prediction in text sequences

Use past 10 characters to predict next one.

a b Z Have you ever **been in Ma**drid?

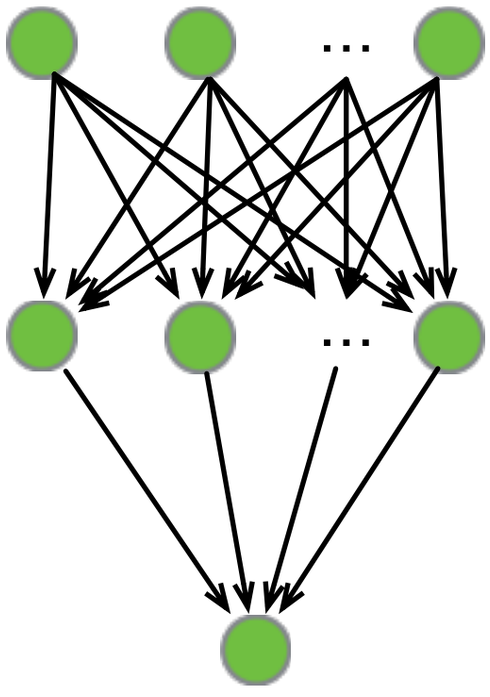
$$X_i = \begin{bmatrix} 0 & 1 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 1 & 0 & \dots & 0 \end{bmatrix} \in \mathbb{R}^{10 \times 52}$$

$$y_i = (0, 0, 0, 1, \dots, 0) \in \mathbb{R}^{52}$$ (letter ‘d’)

‘one-hot’ encoding of characters: 52 dimensions.
Input dimension: 10*52 , output dimension 52

Motivations
Handle sequential data

Classical feedforward model could be used by ‘flattening’ temporal dimension



$$x_i \in \mathbb{R}^{s1}$$

$$X_i \in \mathbb{R}^{T*s}$$

$$s_1 = T * s$$

$$W^{(1)} \in \mathbb{R}^{s_2 \times s_1}$$

$$a_i = f(W^{(1)} x_i) \in \mathbb{R}^{s2}$$

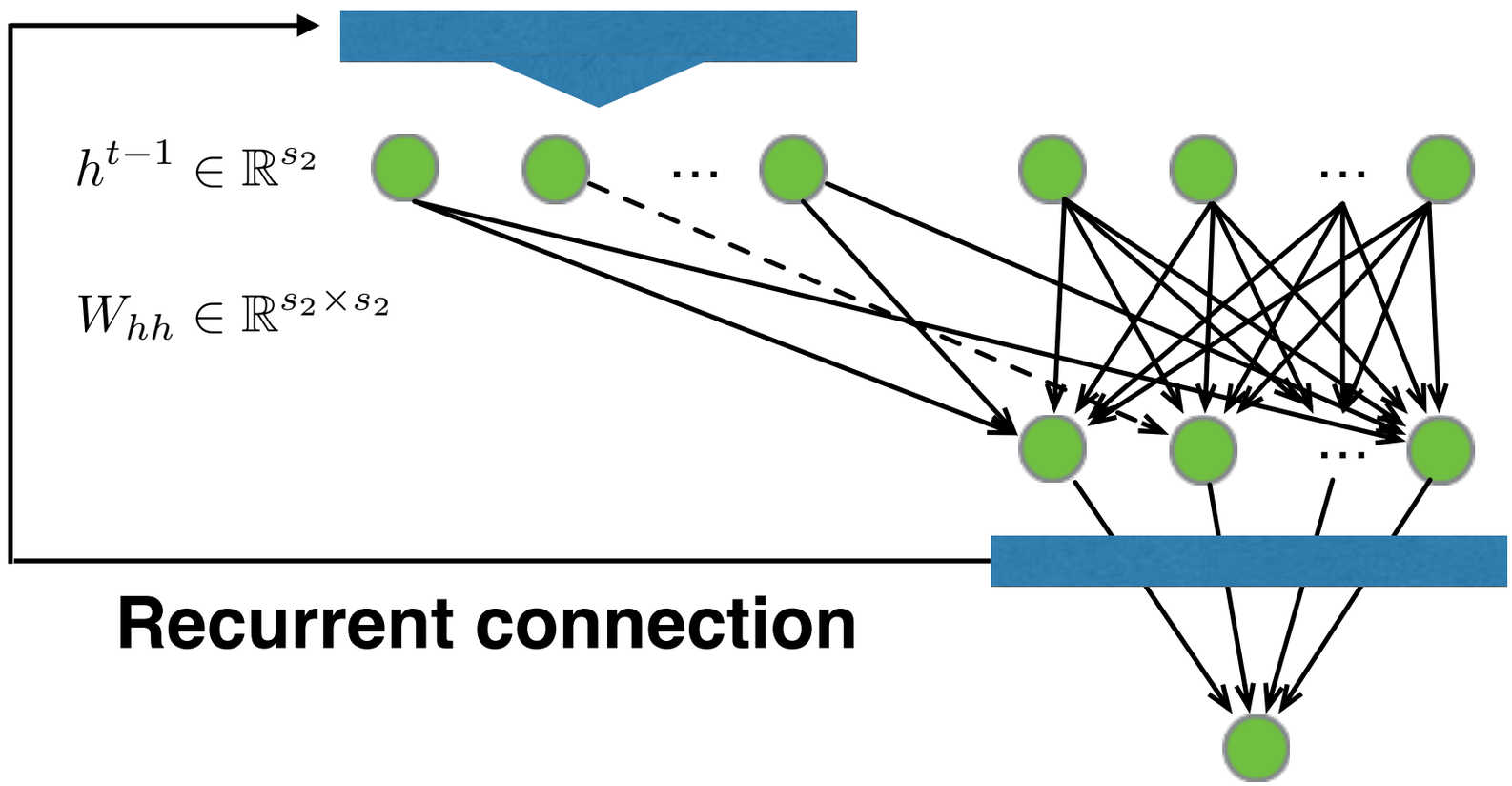
$$W^{(2)} \in \mathbb{R}^{1 \times s_2}$$

$$\hat{y}_i = f(W^{(2)} a_i) \in \mathbb{R}$$

But could not handle large T (which could even be unbounded). Explosion of parameter space.

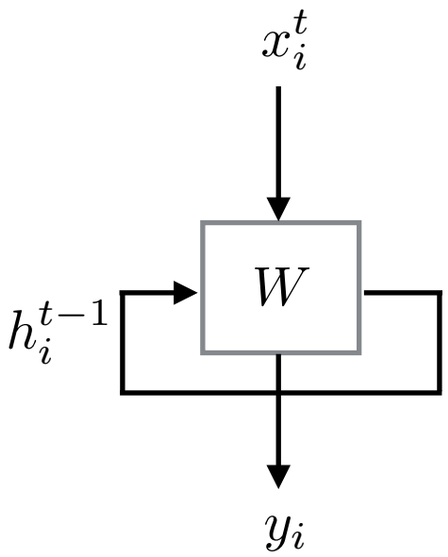
Motivations
Handle sequential data

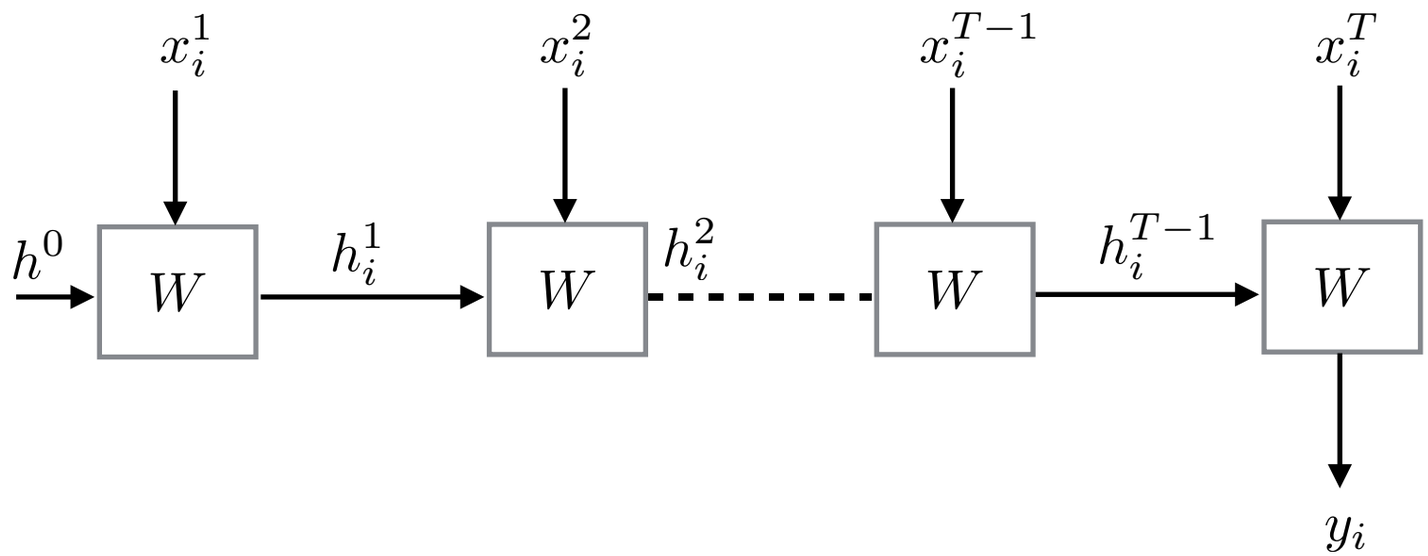
Instead, use a ‘recurrent’ connection



Hidden layer output is fed back into input layer.
Acts as a ‘memory’ layer (or memory ‘cell’).
Number of parameters independent of sequence length T

Recurrent unit
Compact representation





‘Rolled’ ‘Unrolled’

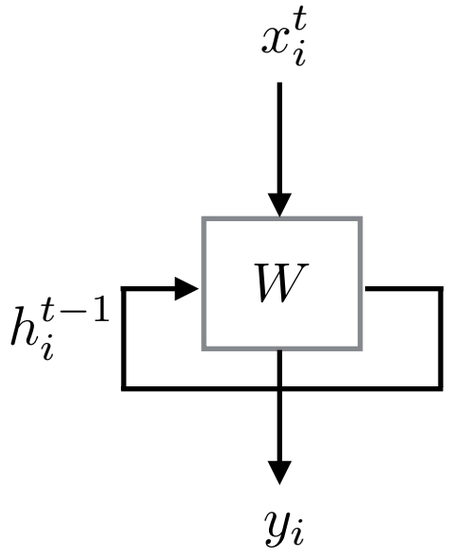
Weights: $W = \{W_{xh}, W_{hh}, W_{hy}\}$

Optimisation with backpropagation $\Delta W = \nabla_W J(W, x_i, y_i)$
(through time)

Notes

- h_0 initialization: Either random (when starting learning), or from a known past state. It is otherwise part of the back propagation optimisation to provide sensible values for new test data.

- There are vanishing/exploding gradient issues because of past states. Can be mitigated with LSTM (next slide).



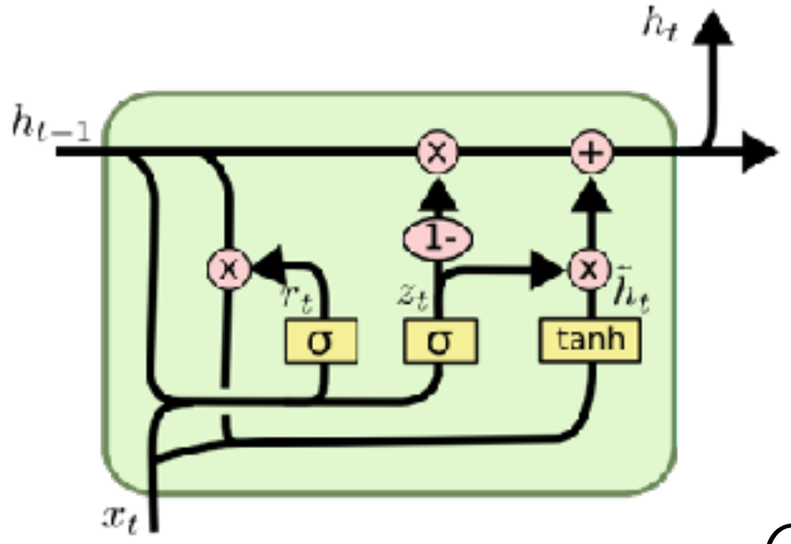
RNN variations
Memory cell

Many possible variations in the processing within the memory cell.

State of the art: LSTM (Long short term memory) *(Hochreiter et al., 1997)*

Other: Gated recurrent units (LSTM simplified) *(Chung et al., 2014)*
No other compelling alternatives yet. *(LSTM, a search space Odyssey, Greff et al., 2015).*

Benefit: Allows long range memorisation



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

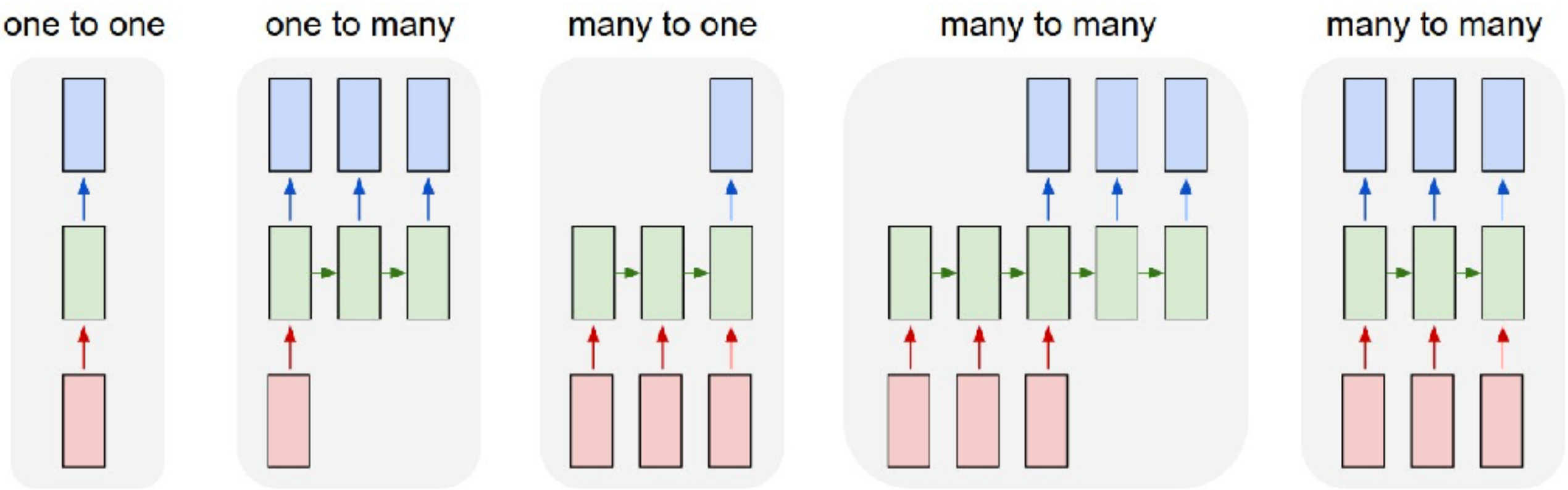
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Credit: Christopher Olah

RNN variations
Input/outputs

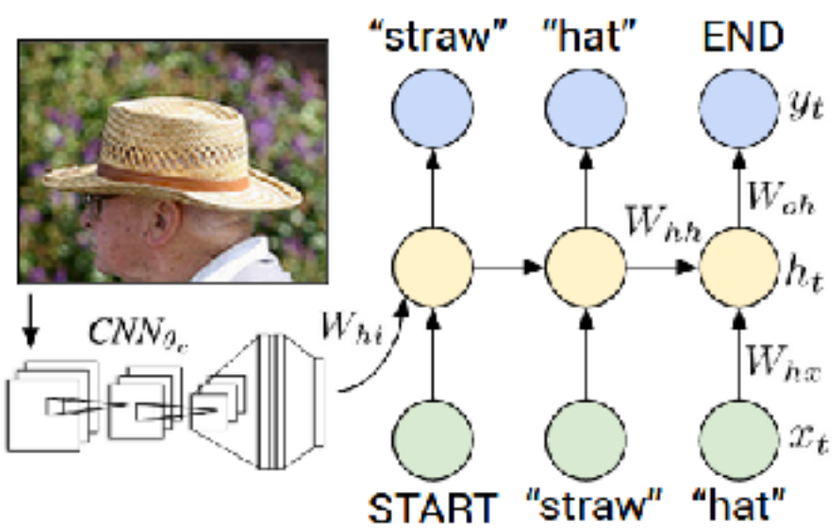


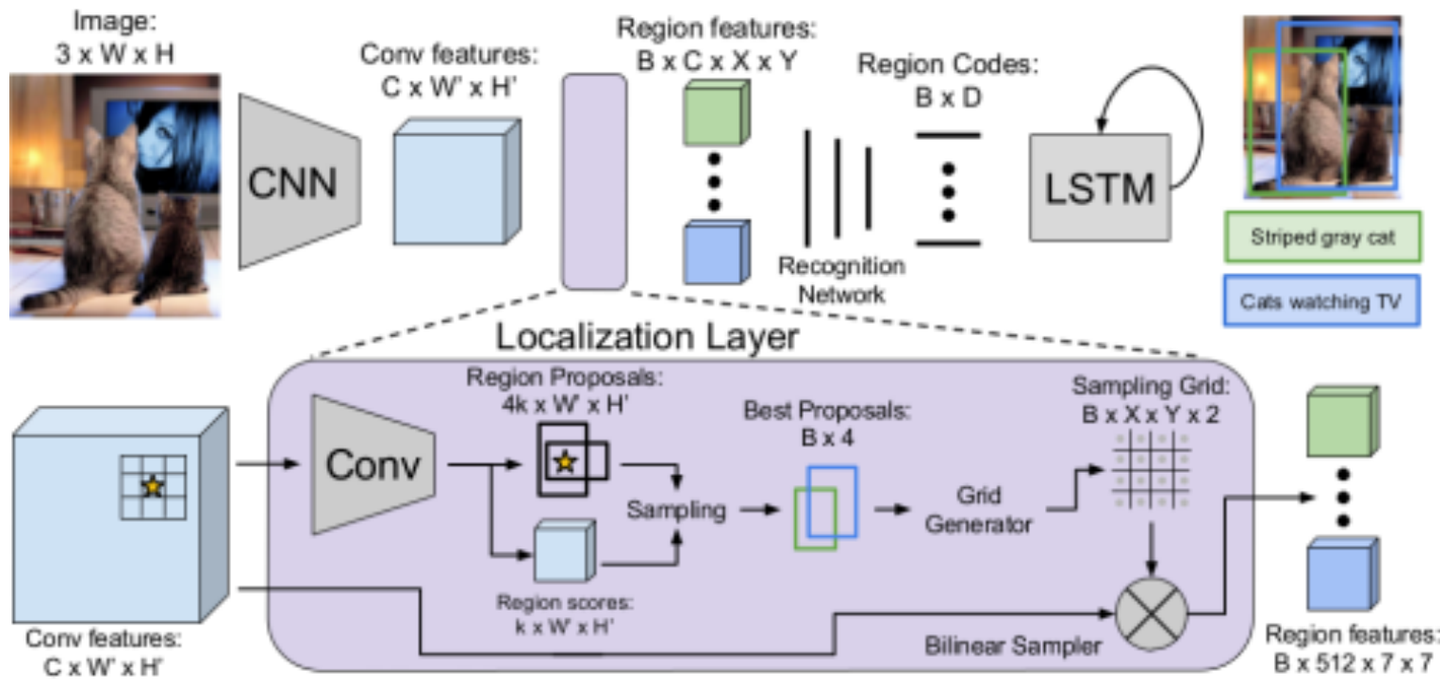
Credit: Andrej Karpathy

Example of applications:
(i) Image recognition (ii) Image captioning (iii) sentiment analysis (iv) Machine translation (v) Video frames classification

Recent DNN applications

Image captioning
Karpathy, 2016





Credit: Andrej Karpathy

Generates captions of images.
Mix of RNN and convolutional modules.

Recent DNN applications

Baidu Deep Speech 2

End-to-end Deep Learning for English and Mandarin Speech Recognition

English and Mandarin speech recognition

Transition from English to Mandarin made simpler by end-to-end DL

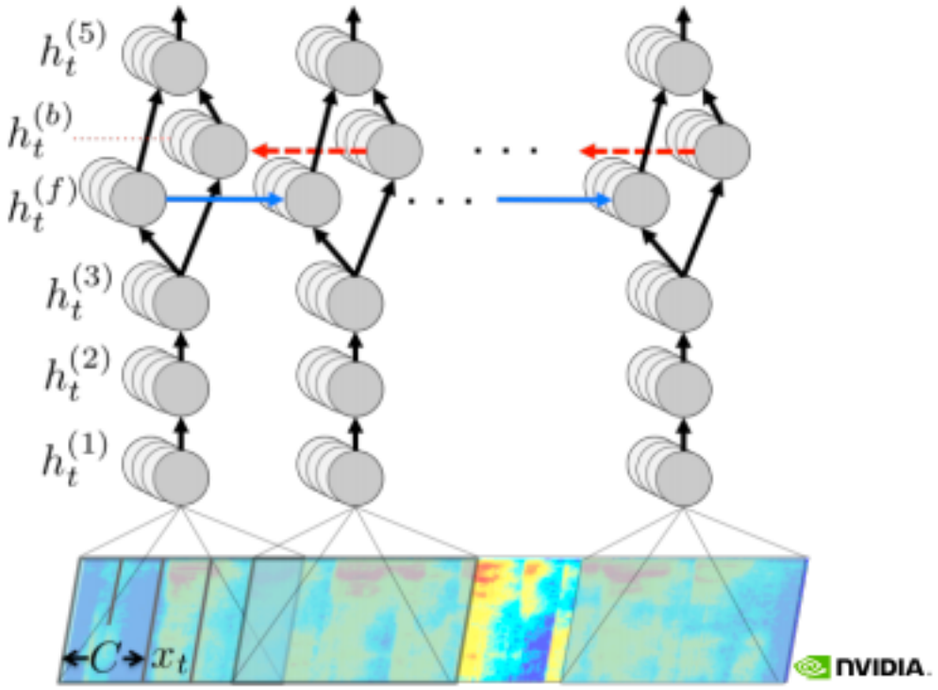
No feature engineering or Mandarin-specifics required

More accurate than humans

Error rate 3.7% vs. 4% for human tests

http://svail.github.io/mandarin/

http://arxiv.org/abs/1512.02595



Credit: Nvidia

Recent DNN applications

AlphaGo

First Computer Program to Beat a Human Go Professional

Training DNNs: 3 weeks, 340 million training steps on 50 GPUs

Play: Asynchronous multi-threaded search

Google DeepMind

Simulations on CPUs, policy and value DNNs in parallel on GPUs

Single machine: 40 search threads, 48 CPUs, and 8 GPUs

Distributed version: 40 search threads, 1202 CPUs and 176 GPUs

Outcome: Beat both European and World Go champions in best of 5 matches

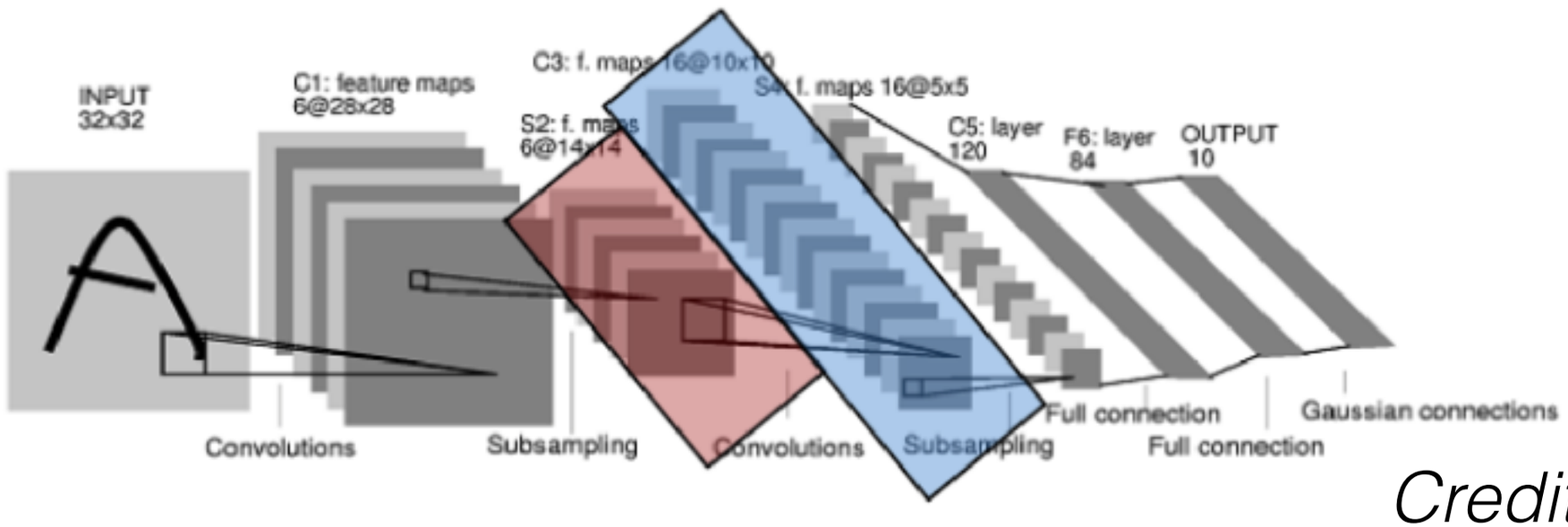
http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html
http://deepmind.com/alpha-go.html

Credit: Nvidia

Scalability

Distributed computations

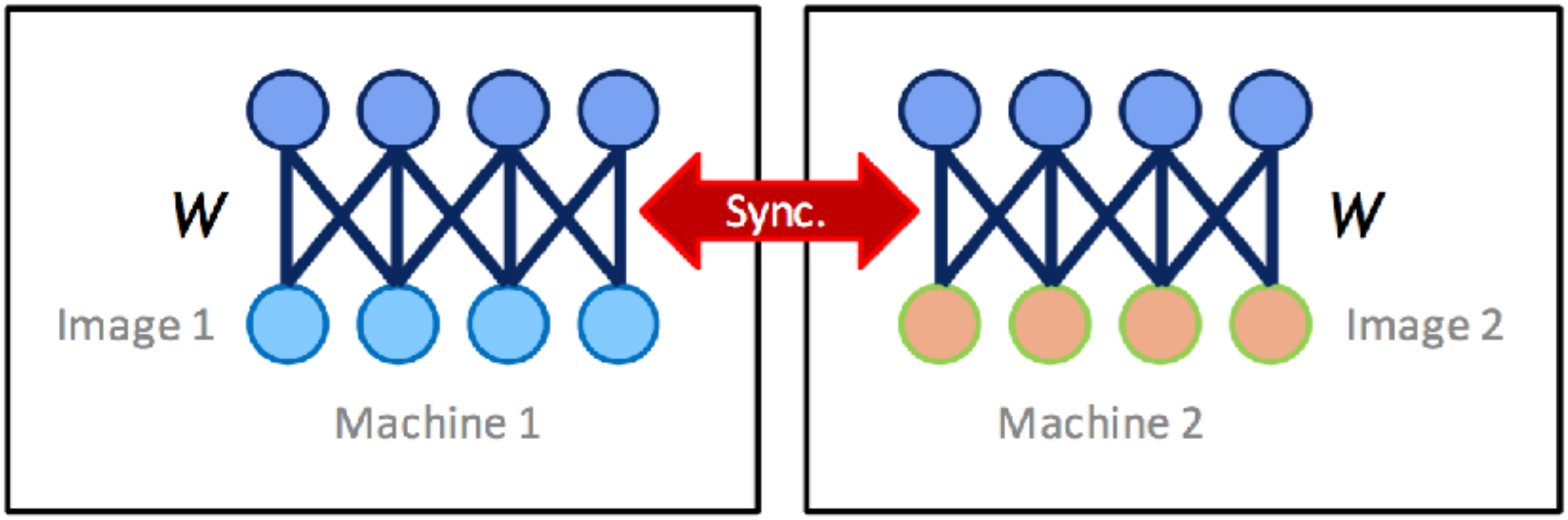
- Lots of parallelism available in a DNN
  - Data parallelism
  - Model parallelism
  - Hyper parameter parallelism
  - Low-level parallelism: GPUs



Credit: NVidia

Data parallelism

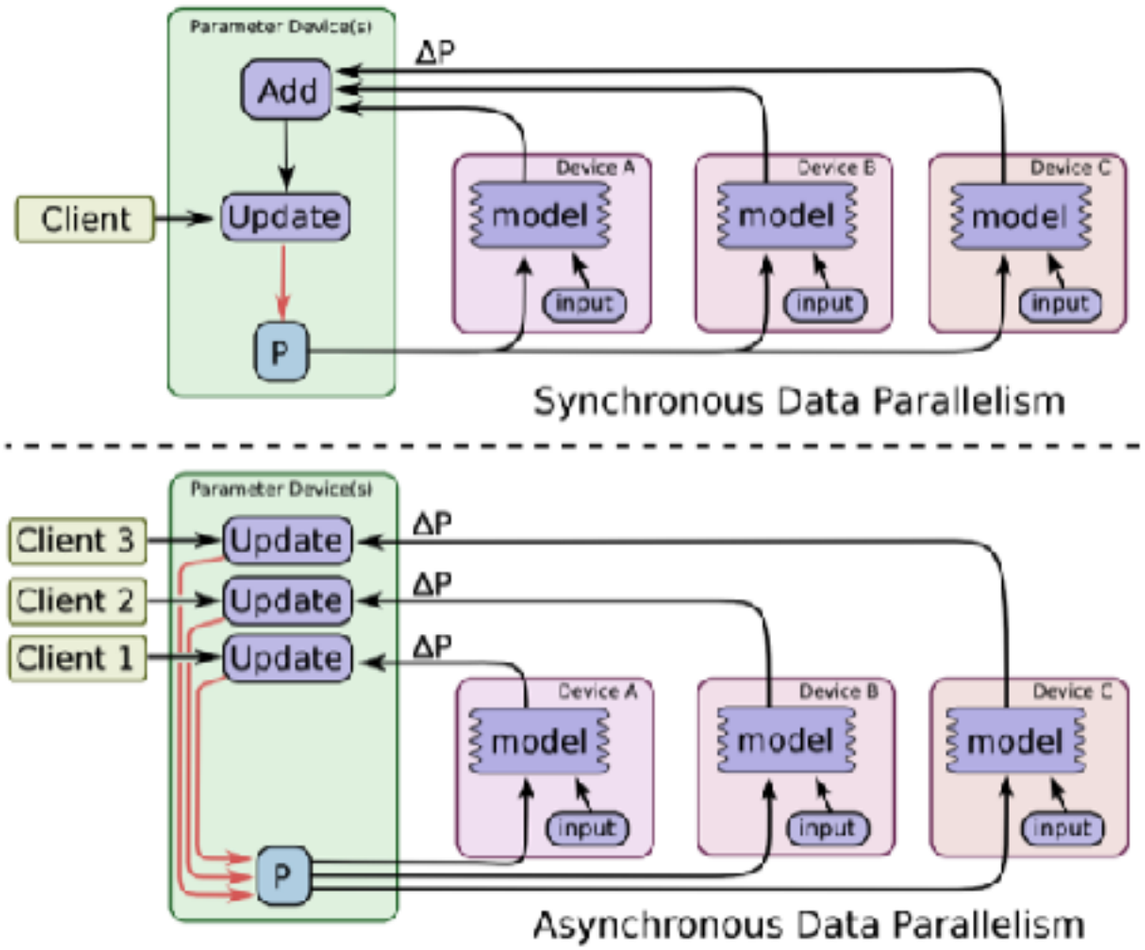
- Different batches of data processed by different machines



- Notes
  - Need to sync models across machines (parameter server)
  - Linear scaling
  - Model may not fit in memory

Data parallelism

- Two types:
  - Synchronous
  - Asynchronous

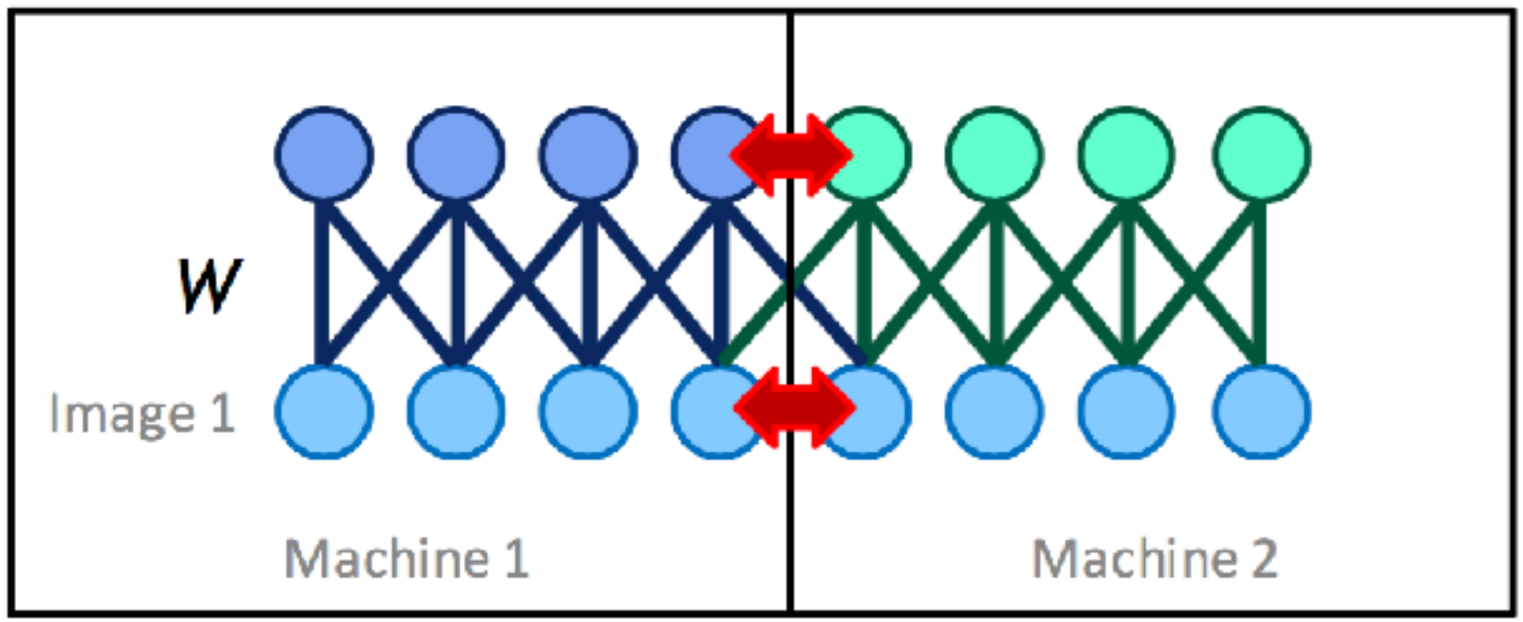


- Notes
  - Synchronous has better convergence
  - Asynchronous is faster, and more resilient to failure

Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems. 2012.

Model parallelism

- Different parts of the DNN computed on different machines



- Notes
  - Allows large models to be distributed
  - Much more frequent communication - need for sync
  - Mostly effective for fully connected layers

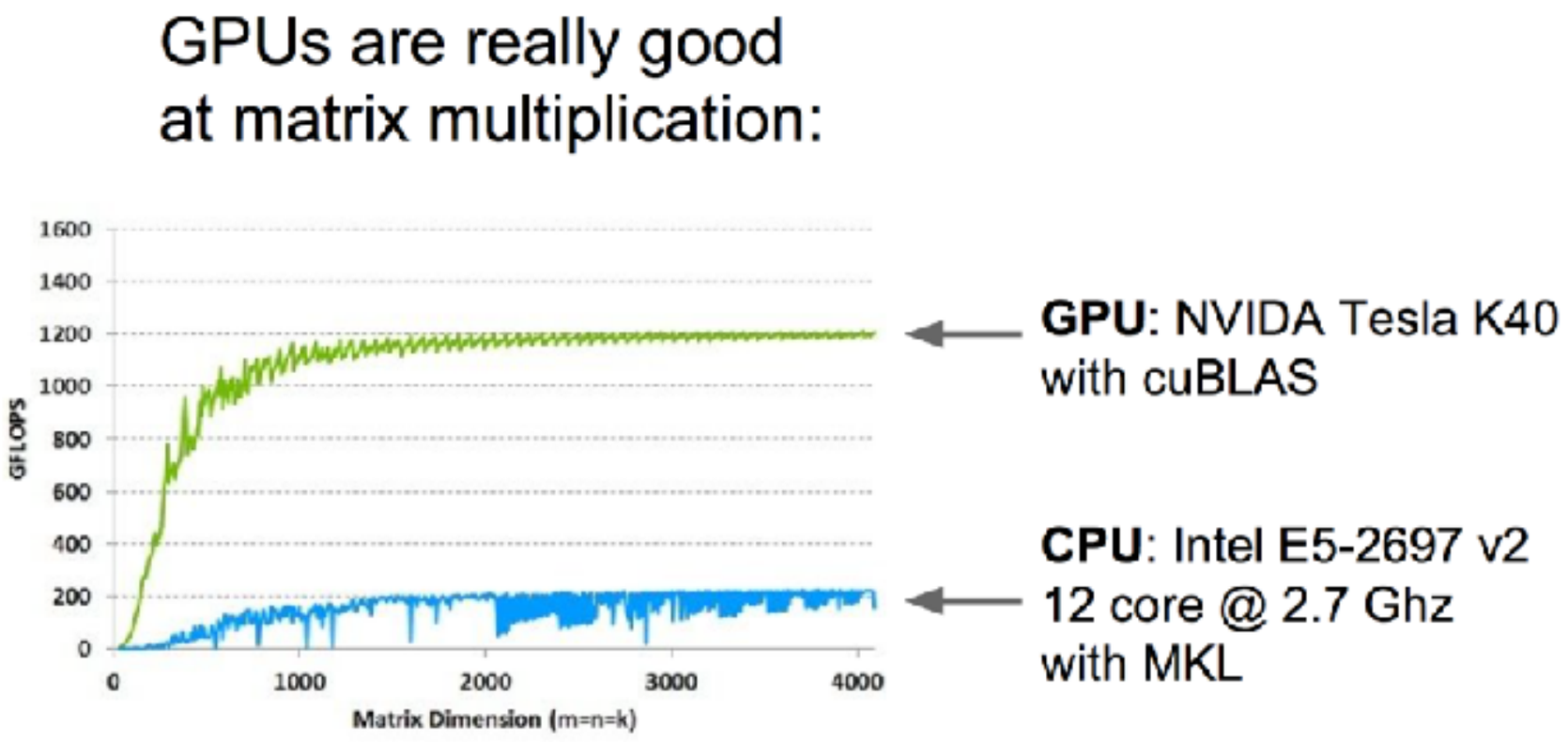
Hyper parameter parallelism
(aka, conveniently parallel)

- Train and test different DNNs in parallel

- Notes
  - May be used on different CPU/GPU/Machines
  - Probably the most obvious and effective way!

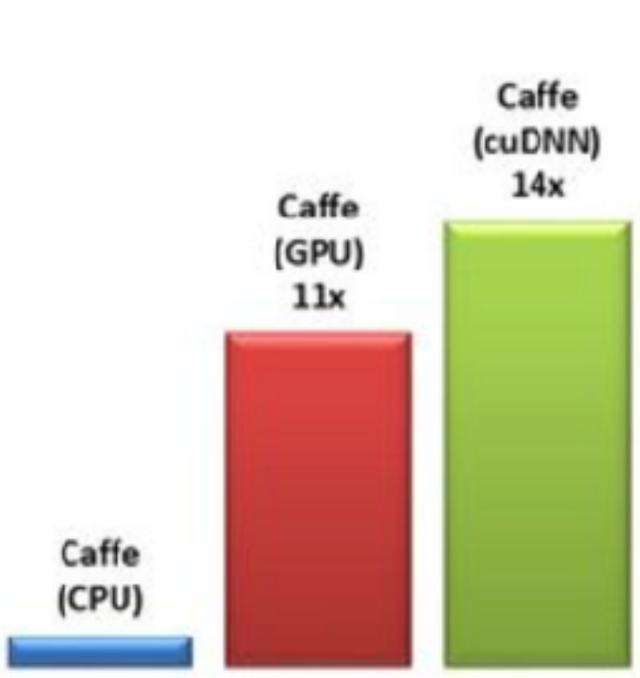
Low level parallelism: GPUs

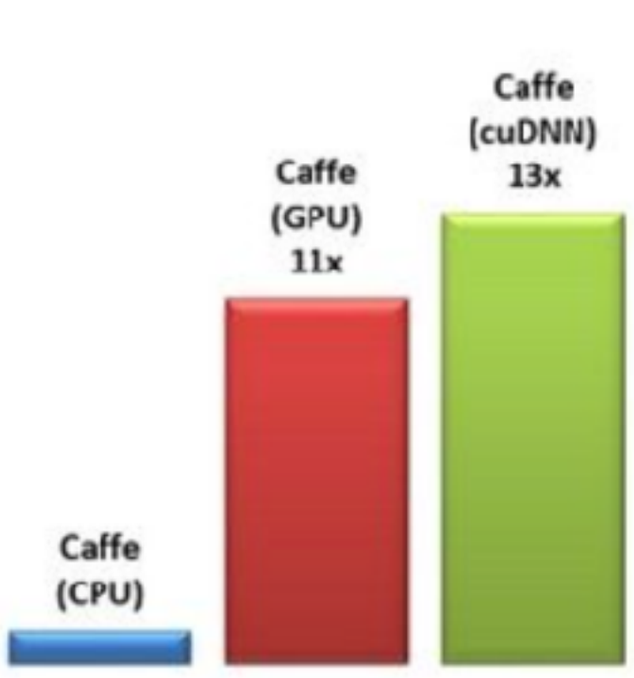
- CPU: Few, fast cores (1-16). Sequential processing.
- GPUs. Nvidia titan X, GTX1080. Many, slow cores (thousands). Originally for graphics. Good at parallel computation
- GPU libraires: CUDA (only Nvidia), OpenCL.

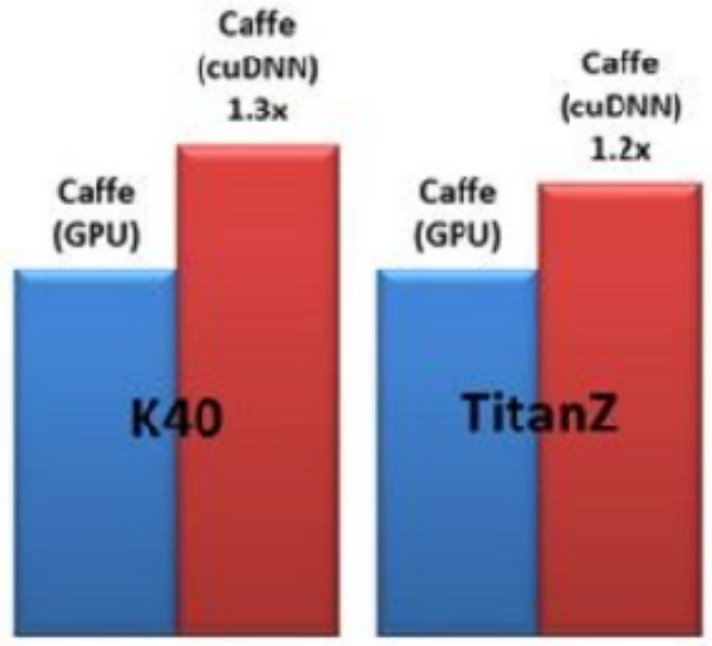


Low level parallelism: GPUs

GPUs are really good at convolution (cuDNN):







All comparisons are against a 12-core Intel E5-2679v2 CPU @ 2.4GHz running Caffe with Intel MKL 11.1.3.

DNN toolboxes

DNN libraries

TensorFlow™

Microsoft CNTK

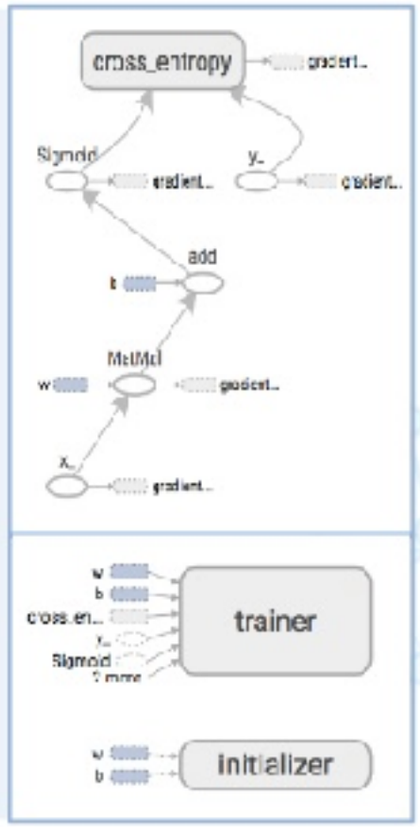
- Written in Python, C; Interface Python
- Large, very active communities
- Based on computation graphs

https://www.tensorflow.org/

https://github.com/Microsoft/CNTK

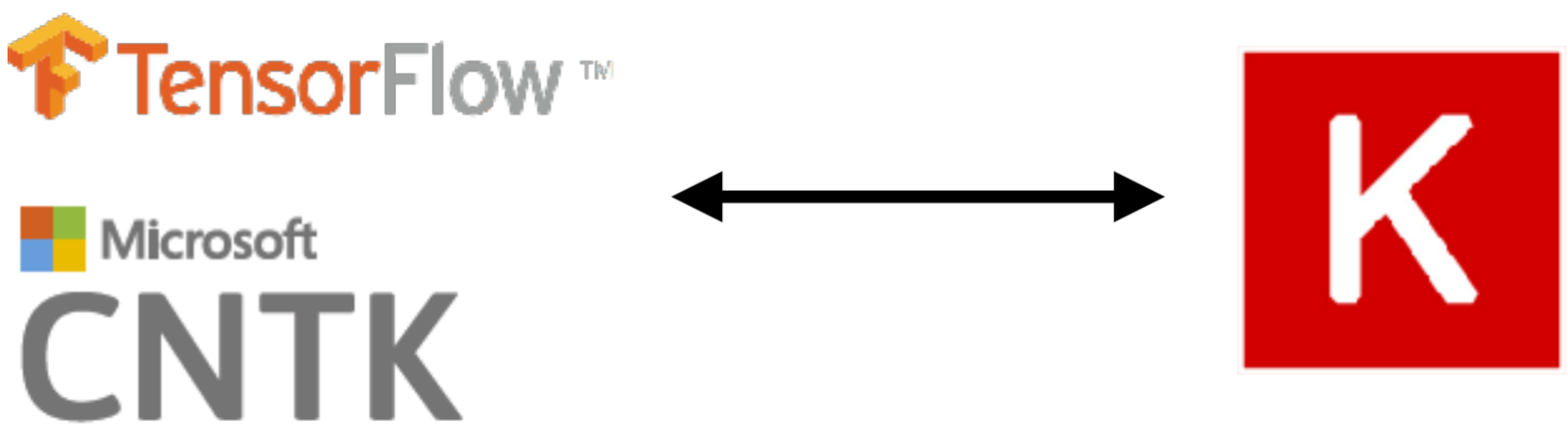
Computation Graph

```
# placeholder
x_ = tf.placeholder(tf.float32,[None,2])
y_ = tf.placeholder(tf.float32,[None,1])
# variable
w = tf.Variable(tf.random_uniform([2,1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1,1]))
# operations
y = tf.nn.sigmoid(tf.matmul(x_,w)+b)
# error function
cross_entropy = -tf.reduce_sum(y_*tf.log(y) + (1-
y_) * tf.log(1 - y) )
# trainer
optimizer = tf.train.GradientDescentOptimizer(0.1)
train = optimizer.minimize(cross_entropy)
# initalizer
init = tf.initialize_all_variables()
```



Example: TensorFlow

DNN libraries



- Written in Python, C; Interface Python
- Large, very active communities
- Based Computational graph

https://www.tensorflow.org/

https://github.com/Microsoft/CNTK

Keras

- DL library on top of Tensorflow and CNTK
- Easier to stack layer
- Easier to train and test

http://keras.io

DNN libraries

Caffe

- Written: C++; Interface: C++, Python
- Many pretrained models
- Mostly for CNNs, backed by Berkeley

dmlc mxnet

- W/I in C++, Python Julia, JS, GO, R, Scala
- Active development, backed by Amazon

- Written: C, Lua. Fastest implementation.

See Smola lecture on DL

Others: See https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

Summary
Deep learning: Pros

- Robust
    - No need to engineer feature before learning. Features are automatically learned during training
- Generalisable
    - Transfer learning: Same network may be reused for other tasks
- Scalable
    - Performance improves with more data. Massively parallelisable

Summary
Deep learning: Cons

- Network structure is an open research issue
    - No guidelines to decide the structure of the network. Trial and error
- Interpretation
    - Hard to interpret how data is processing and what neurons do. Adversarial examples.
- Computational resources and training times
    - Training of large networks require substantial computational resources

Resources

- Why you should understand backpropagation? Karpathy, 2016

  https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b#.r60yxzz3j

- CS231n - Convolutional Neural Networks for Visual Recognition, Stanford 2016, Andrej Karpathy and Fei-Fei li.
  http://cs231n.stanford.edu/

- Blog of Christopher Olah (Google Brain) : http://colah.github.io/ , now http://distill.pub/

Resources

- Books (online)
  - Neural Networks and Deep Learning. Michael Nielsen. 2017. http://neuralnetworksanddeeplearning.com/
  - Deep Learning. Ian Goodfellow Yoshua Bengio and Aaron Courville. 2016. http://www.deeplearningbook.org/