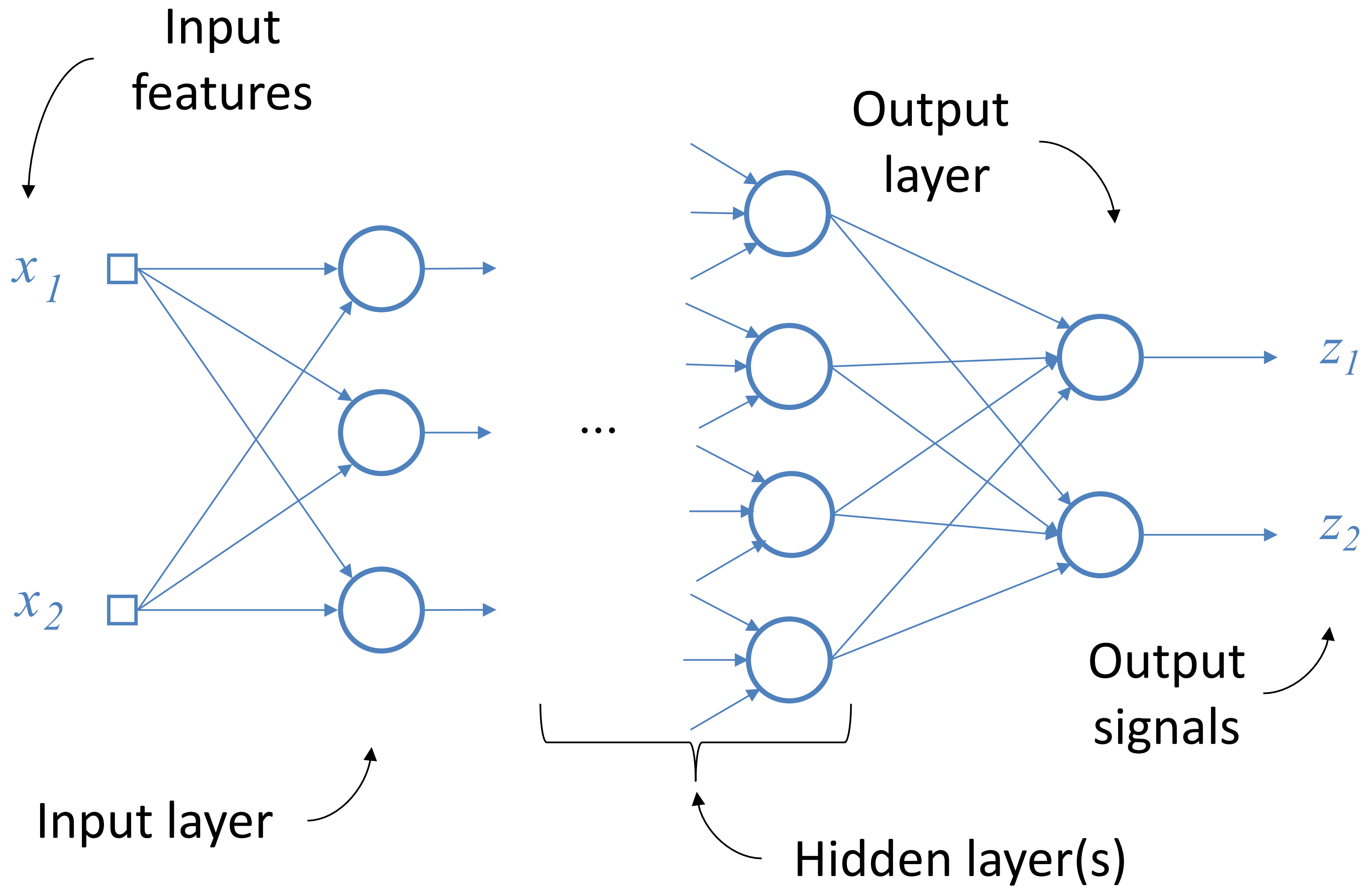
Neural Networks and Learning Systems
TBMI26 / 732A55
2024

Lecture 4

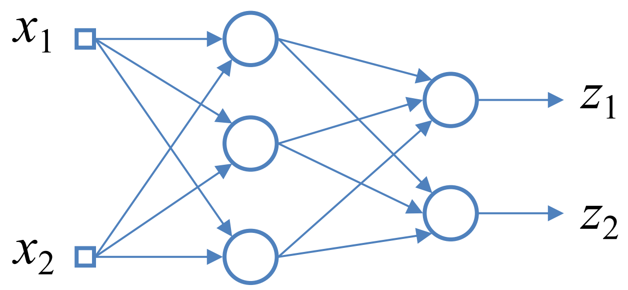
Deep Neural Networks

Magnus Borga
magnus.borga@liu.se

Recap: Neural Networks



Recap: Error Back Propagation



Loss function

# training examples

# output nodes

$$\varepsilon(\mathbf{w}) = \sum_{k=1}^{K} \sum_{m=1}^{M} \left[y_{mk} - z_{mk}(\mathbf{w}) \right]^2$$

all weights

desired output

actual output

Weight update

$$\Delta w_{ij} = -\eta \sum_{k=1}^{K} \frac{\partial \varepsilon(k)}{\partial w_{ij}}$$

Why do we need deep networks?

- 1 hidden layer is enough to produce any classification boundary – in theory...
- In practice, shallow networks are limited in terms of how complex functions they can learn
- Complex boundaries can be more compactly represented with many layers - less nodes in total compared to a 2-layer solution.
- A deep network is able also to learn the basic feature extraction

Challenges in training deep networks

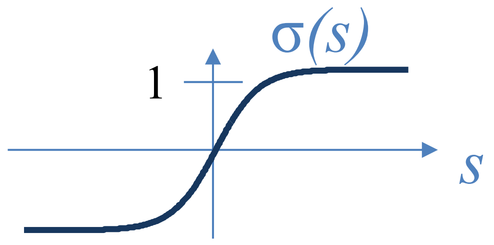
- Huge number of parameters → Extreme degrees of freedom!

  This requires:
  - Huge amounts of training data
  - Powerful computational capacity
  - Ways of reducing the degrees of freedom
  - Regularization
- Basic back-propagation using sigmoid functions does not work in deep networks
  - "Vanishing gradient problem"
  - Instability during training – when parameters change in one layer, the data changes for the following layers

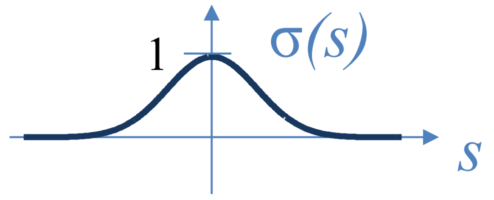
Vanishing gradient problem

Common sigmoid activation functions such as *tanh* squeezes the output to a narrow range. This means that the gradient is very small for most input values.

$$\sigma(s) = \tanh(s)$$



$$\sigma' = 1 - \tanh^2(s) = 1 - \sigma^2$$



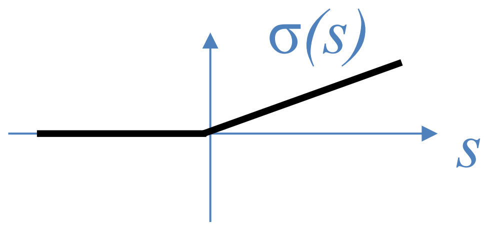
What happens if we apply the chain rule and multiply small error gradients many times when propagating back thru a deep network?

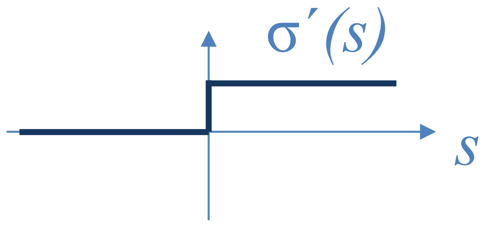
Vanishing gradient problem

- Error gradients become very small for early layers in the network
- → weights in early layers are not updated!

- Solution: Use an activation function with a derivative that survives the back propagation

ReLU – Rectifying Linear Unit





$$\sigma(s) = \max(0, s)$$

$$\sigma' = \begin{cases} 0, & s < 0 \\ 1, & s \geq 0 \end{cases}$$

- The ReLU activation function is often used in the hidden layers of deep neural networks to avoid the vanishing gradient problem
- The output layer often uses a Softmax function to be able to represent the probabilities for the different classes

Instability during training

- Learning works best when the input is centred around zero and has a uniform range
- When parameters in one layer change, the input to the following layers may move away from the optimal range
- The optimization goal becomes a moving target.

Batch normalization

- For each training batch, normalize the output from the hidden layers

$$\hat{x} = \frac{x - \bar{x}}{\sigma_x}$$

- The output is then scaled by

$$y = \gamma\hat{x} + \beta$$

- γ and β are parameters learned by the network

- (It is still debated why batch normalization actually works)

Drop out

- The large number of parameters makes deep networks sensitive to overfitting
- One way of avoiding overfitting is to randomly turn off nodes in hidden layers
- This is called drop out
- Drop out prevents the network to rely on each parameter
- This is a kind of regularization of the solution
- Usually implemented as a dropout layer that randomly switch off output from nodes in the previous layer
- Note: Drop out is only used in training – not when using the network!

L1/L2 regularization

- Add a penalty term to the loss function that depends on the weights
- L1: $$\varepsilon(\mathbf{w}) = \sum_{k=1}^{K} (y_k - z_k(\mathbf{w}))^2 + \lambda \sum_j |w_j|$$
  - Small weights drives to zero
  - Sparse solution → Feature selection
- L2: $$\varepsilon(\mathbf{w}) = \sum_{k=1}^{K} (y_k - z_k(\mathbf{w}))^2 + \lambda \sum_j w_j^2$$
  - Small weights are not penalized
  - Sensitive to outliers

CONVOLUTIONAL NEURAL NETWORKS

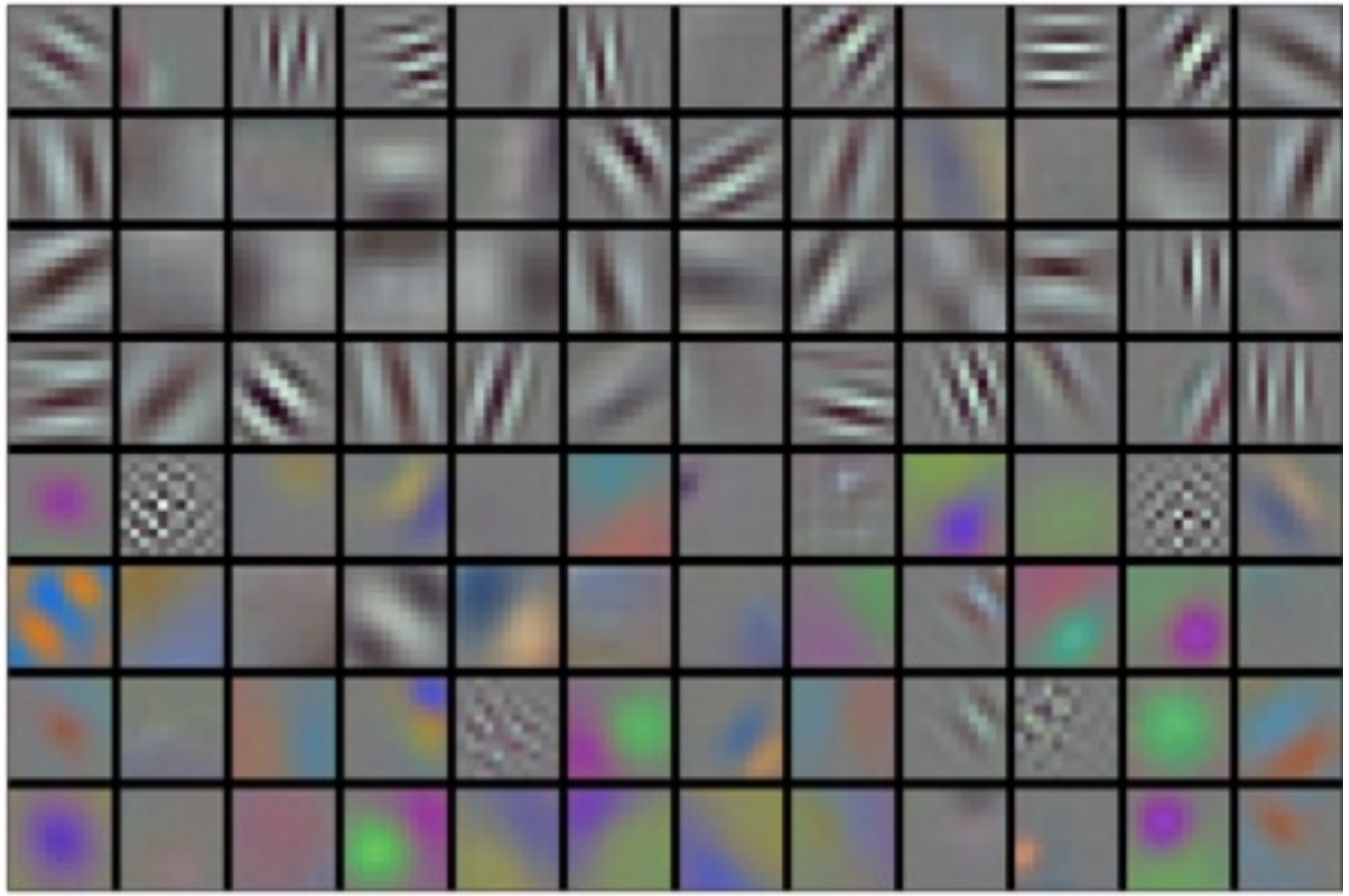
Convolutional Neural Networks

- Consider an image of 512×512 pixels
- The naive approach is to consider each pixel as an independent input feature
- This gives an input feature vector with 262,144 dimensions
- But image features have some important properties
  - Locality
  - Shift invariance
- These properties are not utilized in this naive approach

Locality

- Locally, images consist of simple features such as lines, edges and corners
- Objects are built of spatial relations between such local features
- To detect such local features, we only need to look in a very small neighbourhood of an image
- On a more global scale, we use more abstract features

Primitive image features



Shift invariance

- The description of an image feature is shift invariant
  - A corner, or a face, looks the same no matter where in the image it is located
  - While the naive approach can learn to detect an object in a certain position, it cannot recognize that object if it is moved in the image, because that would completely change the input vector
- Hence, we want the training of feature detectors to be independent on position, i.e. shift invariant

Convolution

- In convolution a kernel is shifted over the image
- For each position, the scalar product between the kernel and the corresponding image patch is computed
- $(f * g)(x, y) = \sum_{i,j} f(x - i, y - j)g(i, j)$

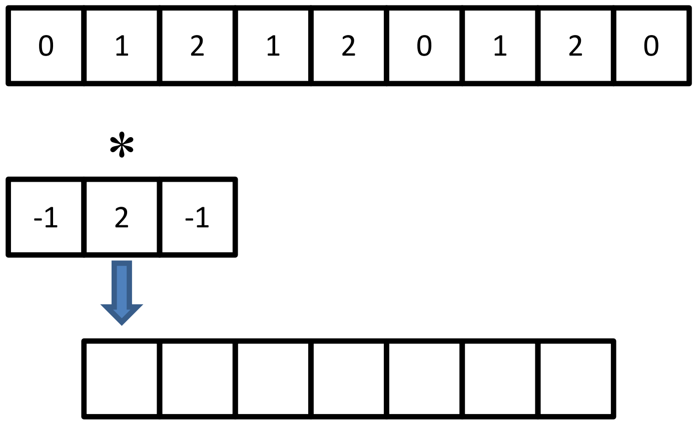
- Remarks:
  – In neural networks, the kernel is not reversed, so it is formally a correlation, not a convolution. But this does not matter in practice.
  – Note that the result is only well defined inside the image, where the complete kernel covers the image, i.e. the result is smaller than the image. This can be handled by padding the image before convolution

Convolution – 1D Example

Input data

Convolution kernel

Resulting data



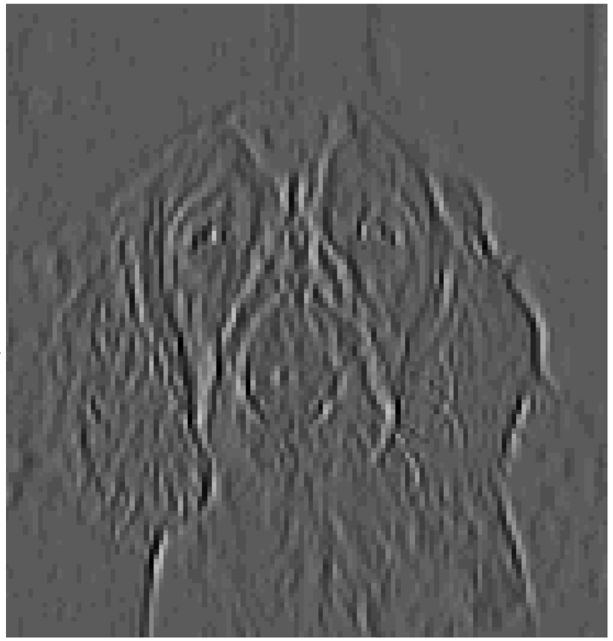
Convolution – Images

Convolution kernel

-1	2	-1
-1	2	-1
-1	2	-1

Image

Result (after down sampling)

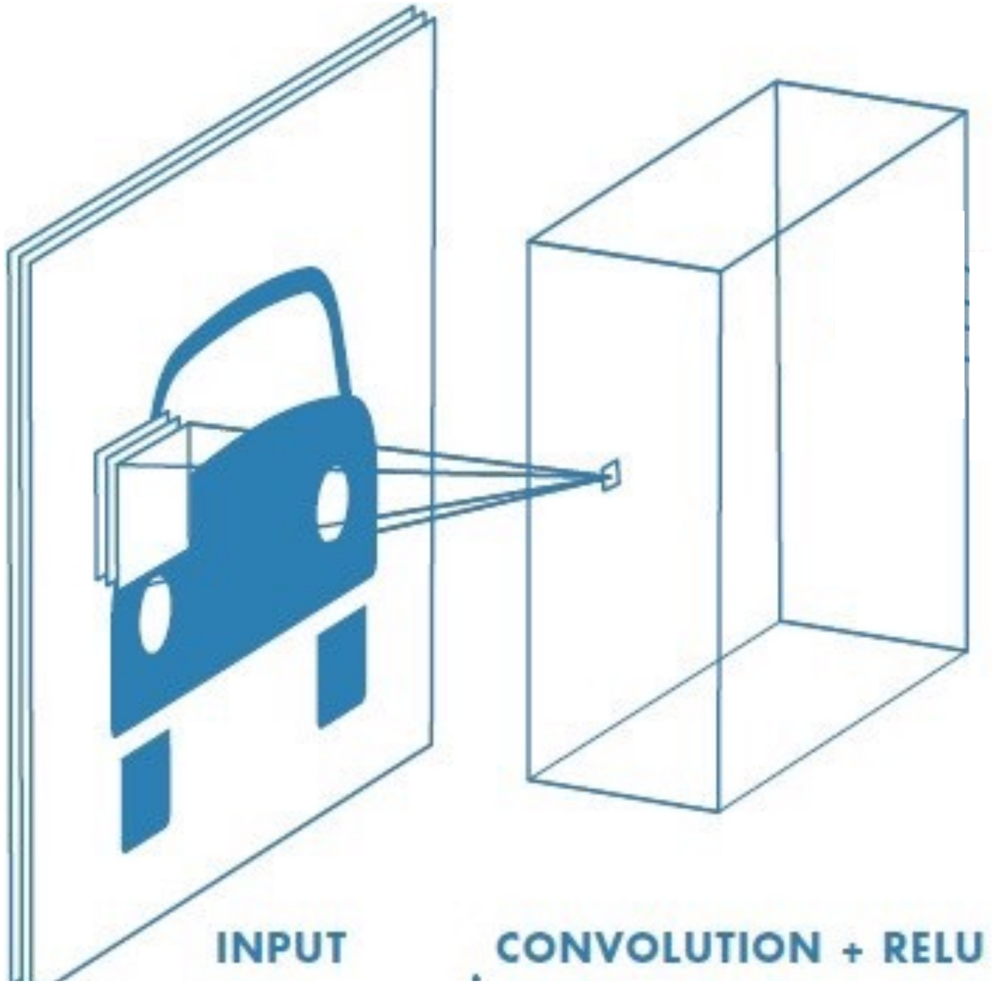


Convolutional networks

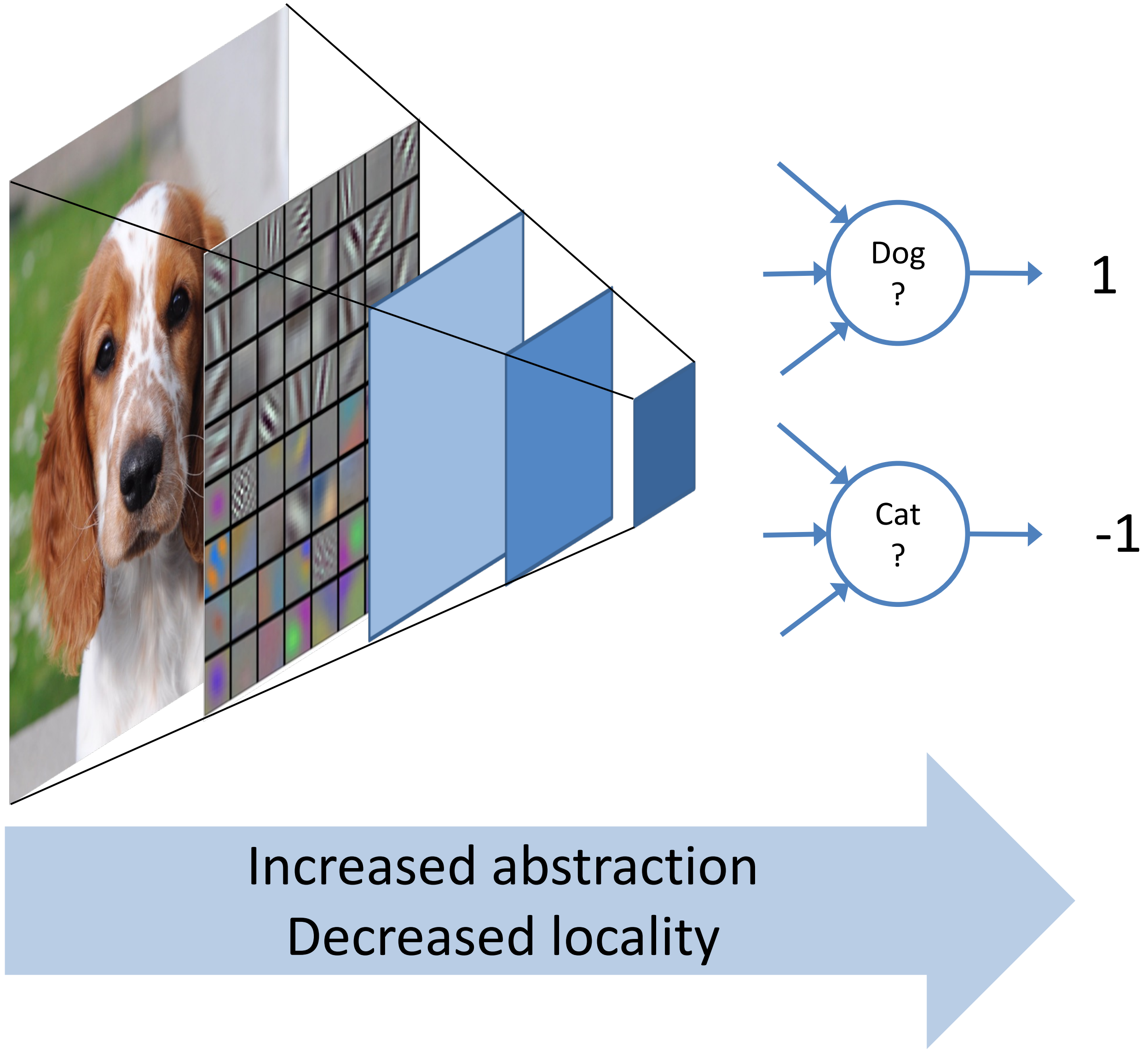
- In a convolutional layer, only the coefficients in the kernels need to be learned by the network
- The kernels are much smaller than the image
  - This reduces the number of parameters to learn and thereby the risk of overfitting
- The kernel is
  - spatially local
  - shift invariant

Convolutional networks

- Each layer consists of multiple kernels each generating a separate feature image
- Each kernel has an additional dimension to cover all channels, e.g. colour channels
- The output of each layer is a stack of such feature images



Abstraction hierarchy

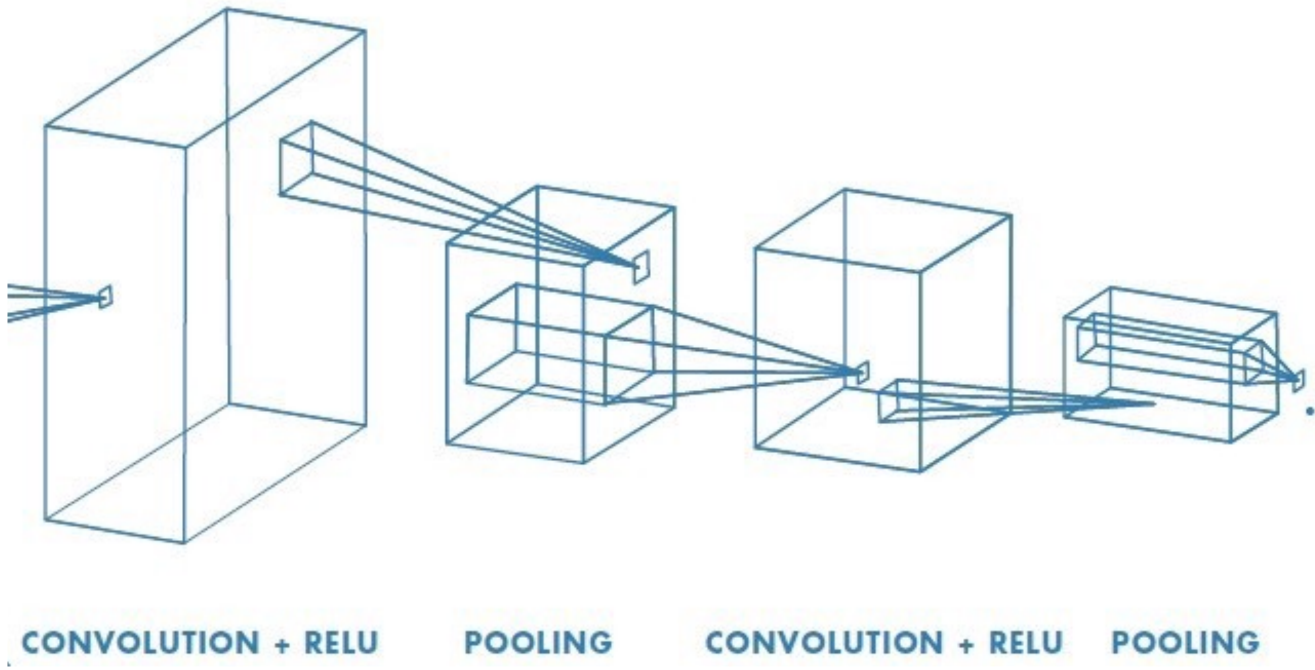


- Local features are pooled and combined to produce more abstract features
- From primitive to complex
- From local to global

Pooling

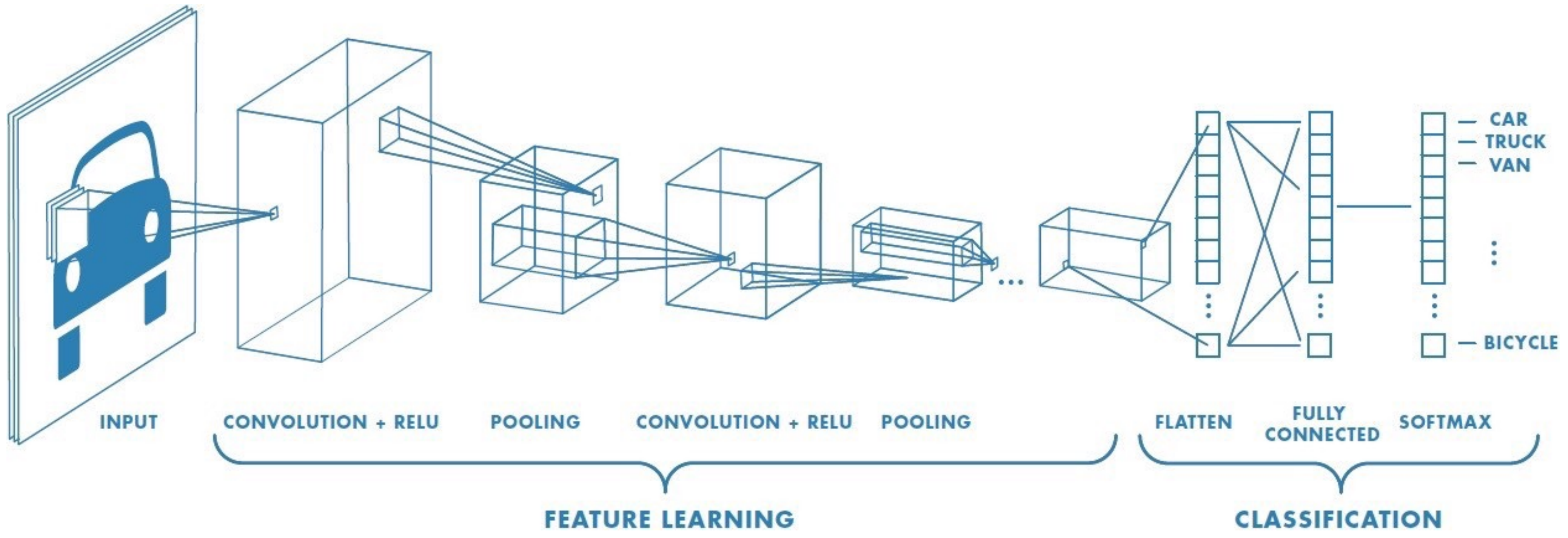
- To decrease locality and increase shift invariance as information propagates forward thru the network, the output from one layer can be pooled together in a pooling layer
- Pooling combines output from nearby units
  - Mean pooling – average of the units
  - Max pooling – max value of the units

- Note – Pooling is done separately for each feature image



Convolutional networks

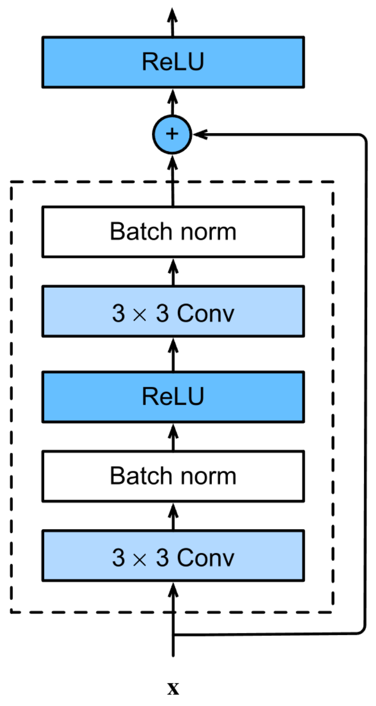
- Early layers learn useful features
- Later layers learn how to combine features for classification



NETWORK ARCHITECTURES

Blocks

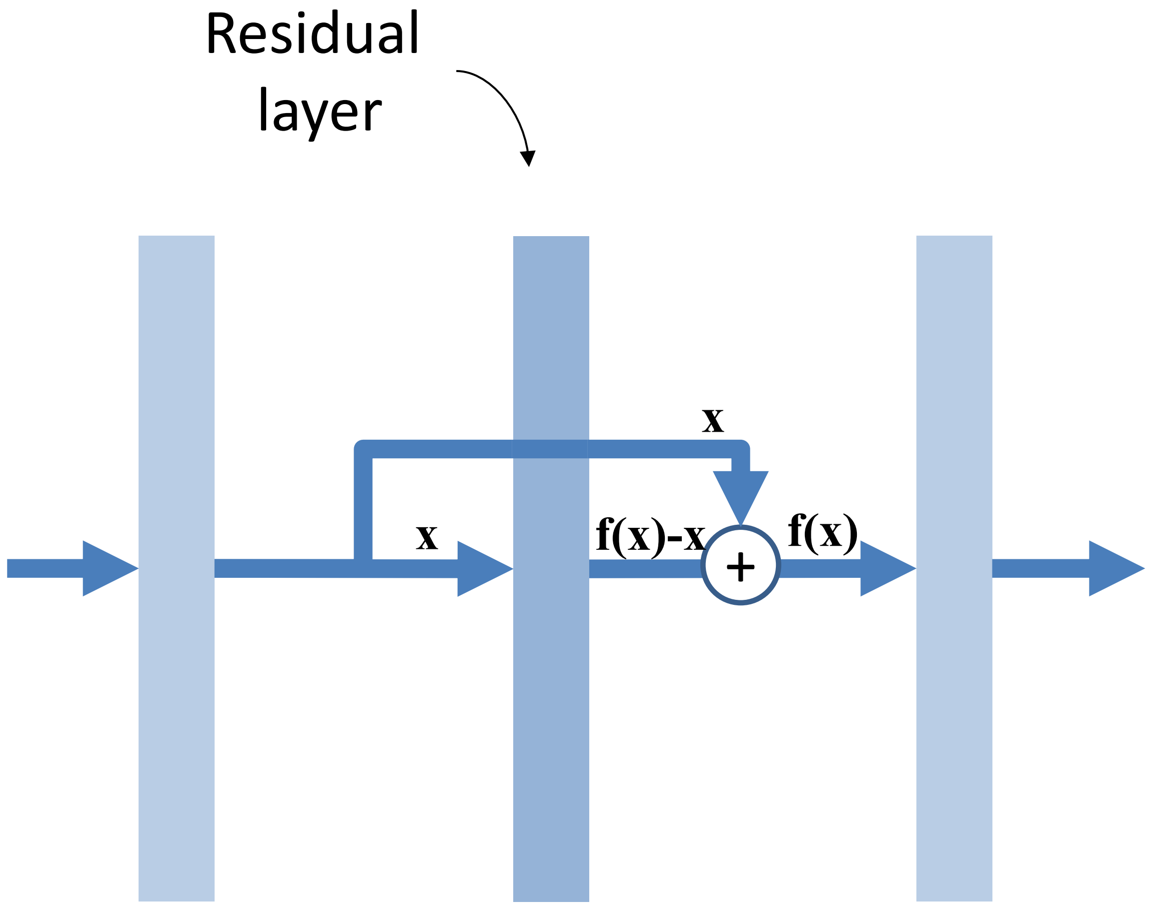
- To simplify the design of complex deep neural networks, they are based on several building <u>blocks</u>
- Each block can consist of one or several layers
- This enables design of networks with 100’s of layers



A block in ResNet

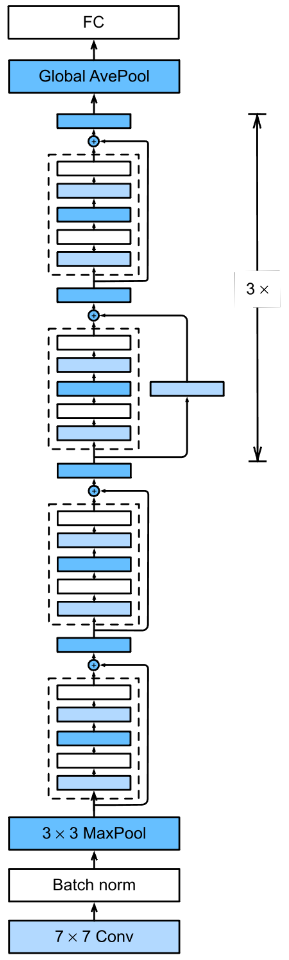
Image from d2l.ai

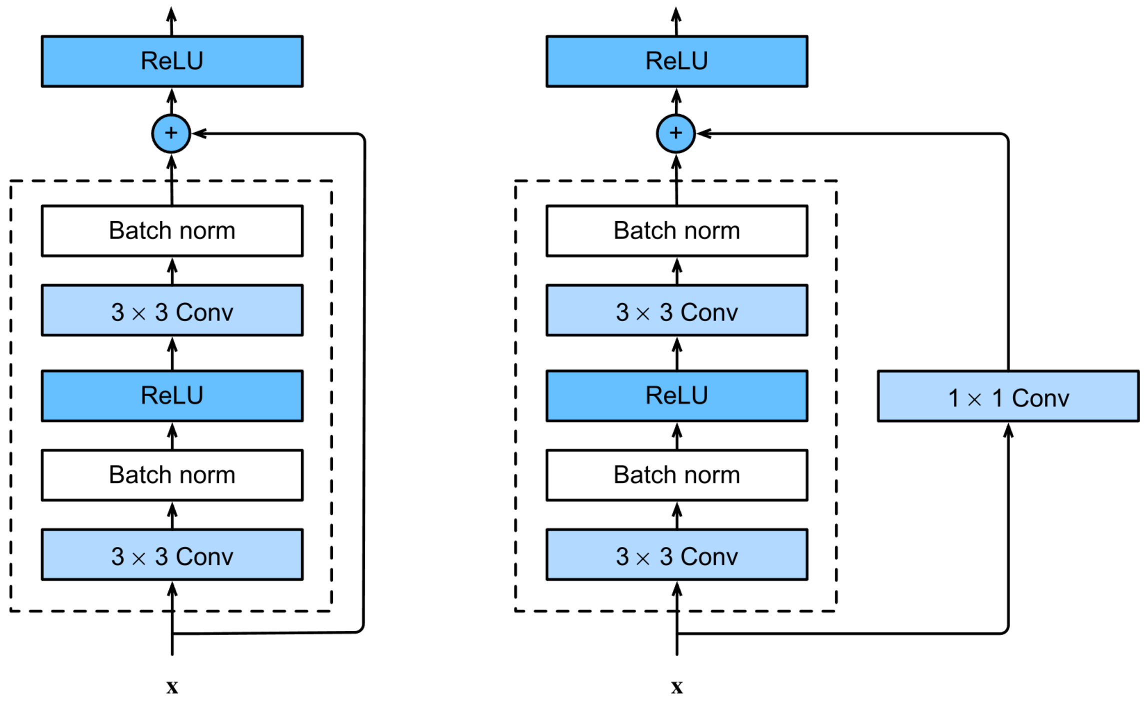
Residual Networks



- Intuition: The identity function should be the simplest to learn
- The residual layer only needs to learn the residual f(x) – x
- Early in the training, the short cuts make the network act as a more shallow network
- Easier to back propagate the error in very deep networks

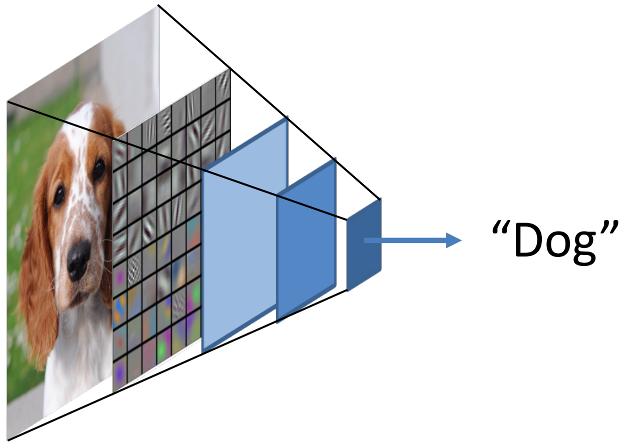
ResNet





Images from d2l.ai

Image segmentation



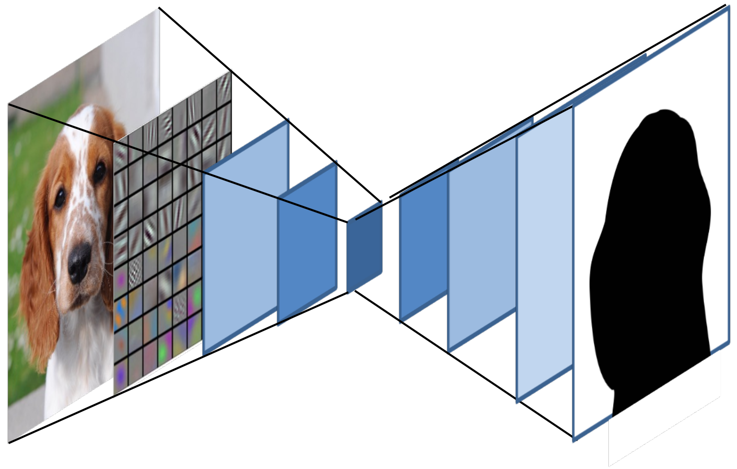


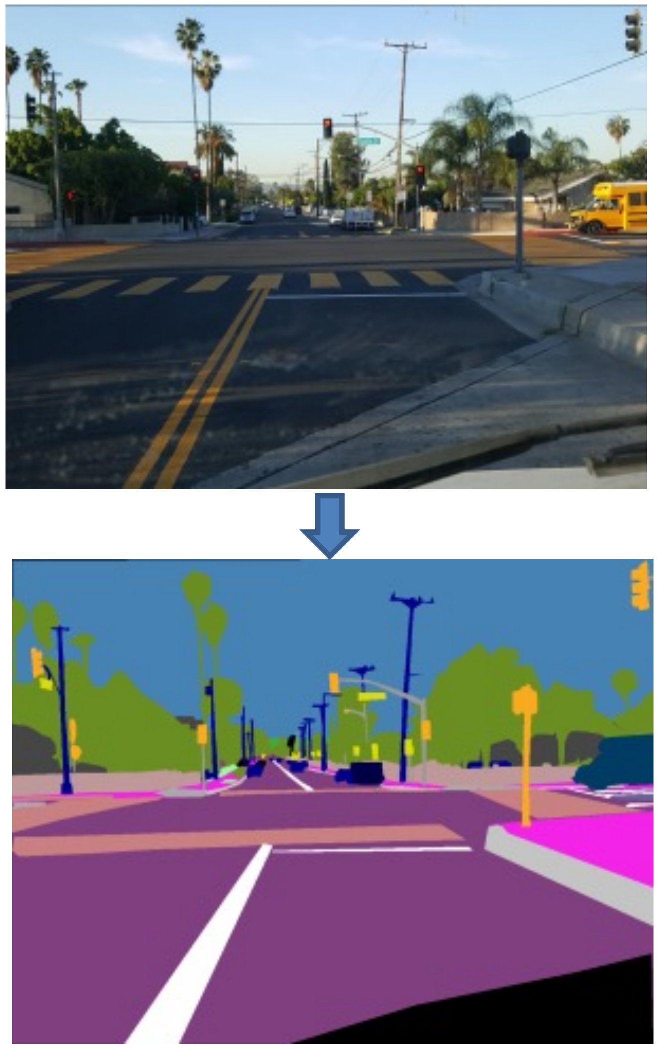
Image classification

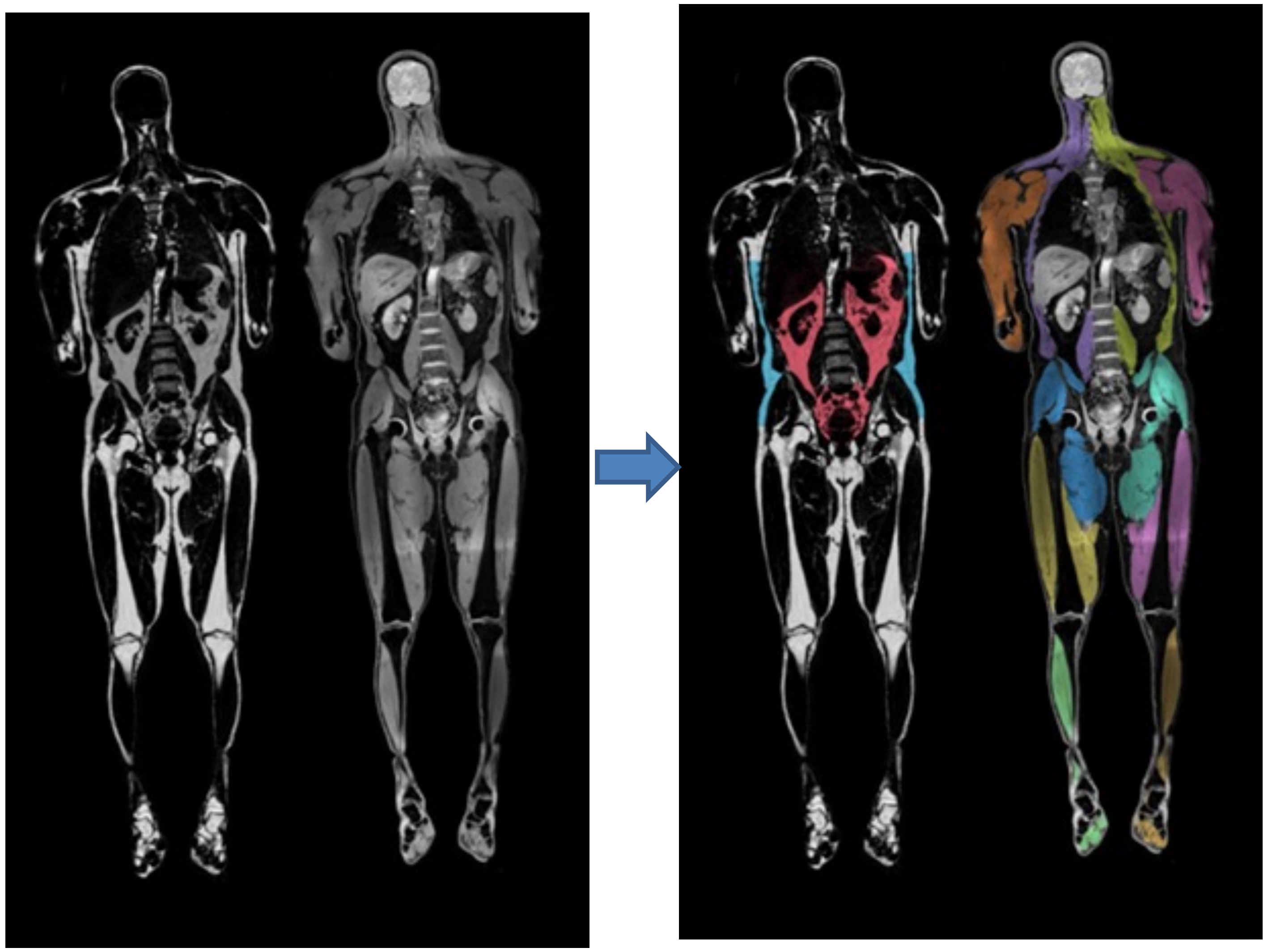
- "What does this image depict?"
- High-dimensional input
- Low-dimensional output

Image segmentation

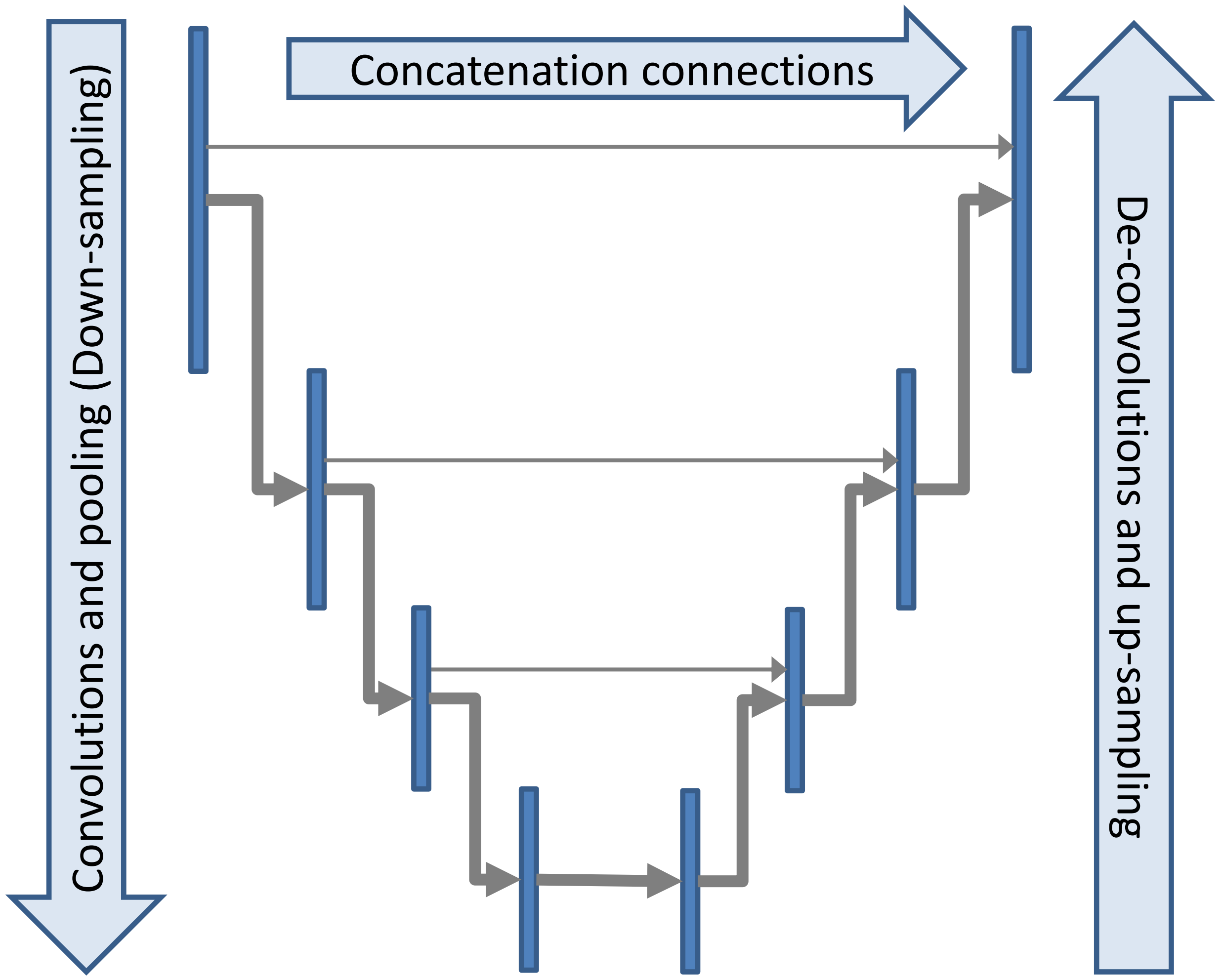
- Where is the dog?
- Classification of each pixel
- Foreground / Background
- Output dimensionality same as input

Image segmentation – Examples


groundai.com


AMRA Medical

U-Net


Convolutions and pooling (Down-sampling)
Concatenation connections
De-convolutions and up-sampling

- Convolutional network for image segmentation
- The first part is a normal contracting network with convolutions and pooling
- The second, expanding part performs up-sampling and de-convolution
- Features from the contracting part are concatenated with up-sampling results in the expanding part

Auto Encoders

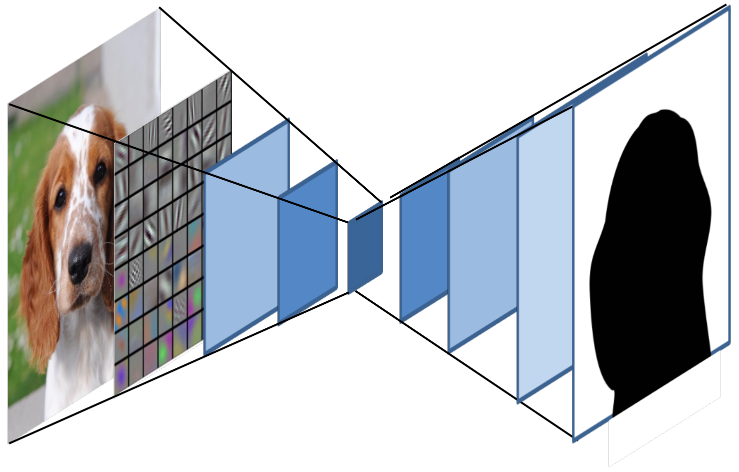
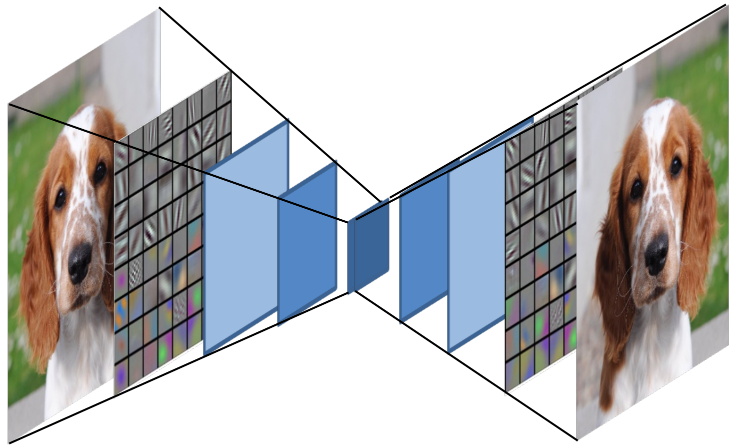


Image segmentation

- Training data {$\mathbf{x}_i$, $\mathbf{y}_i$}
- Binary output (foreground / background)

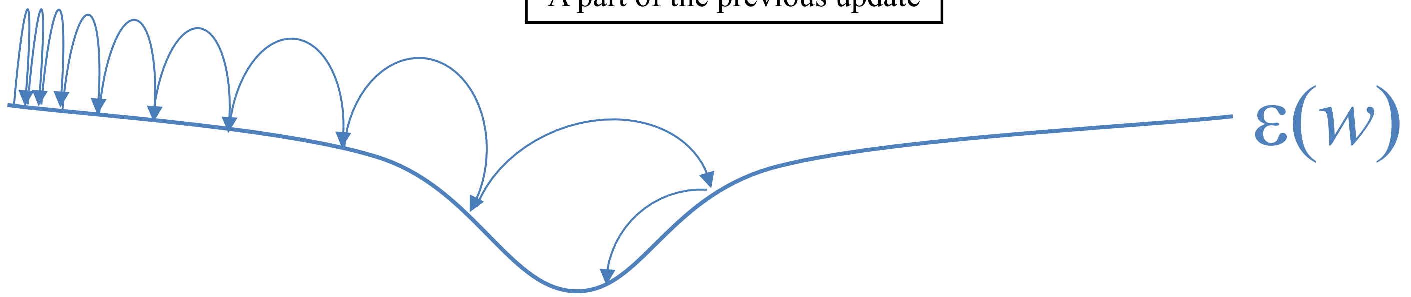


Auto encoder

- Training data {$\mathbf{x}_i$, $\mathbf{x}_i$}
- Does not require labelled training data – Unsupervised learning!
- The smallest layer must be able to represent all training data
- The bottle neck forces an efficient coding of the original data (images)

Faster convergence

- Normalize input features, e.g. batch normaliztion
- Change to another activation function, e.g. ReLu
- Residual layers – skip connections
- Separate and adaptive step length η for each weight:
  - If the derivative has the same sign in several consecutive steps, η should increase. If the derivative change sign, η should decrease.
- Introduce a *momentum term*: $\Delta w_{ji}(t) = \underbrace{\alpha \Delta w_{ji}(t-1)}_{\boxed{\text{A part of the previous update}}} - \eta \frac{\partial \varepsilon(t)}{\partial w_{ji}}$



Deep Neural Networks – Summary

- Based on the multi-layer perceptron and back propagation
- Additional smart tricks
  - ReLU activation function
  - Batch normalization
  - Weight sharing e.g. convolutional networks
  - Skipping layers – Residual Networks
  - Larger building blocks
  - ...
- Requires huge amounts of training data and lots of computational power!