Computational MRI

Machine learning in MRI and neural network architecture design

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

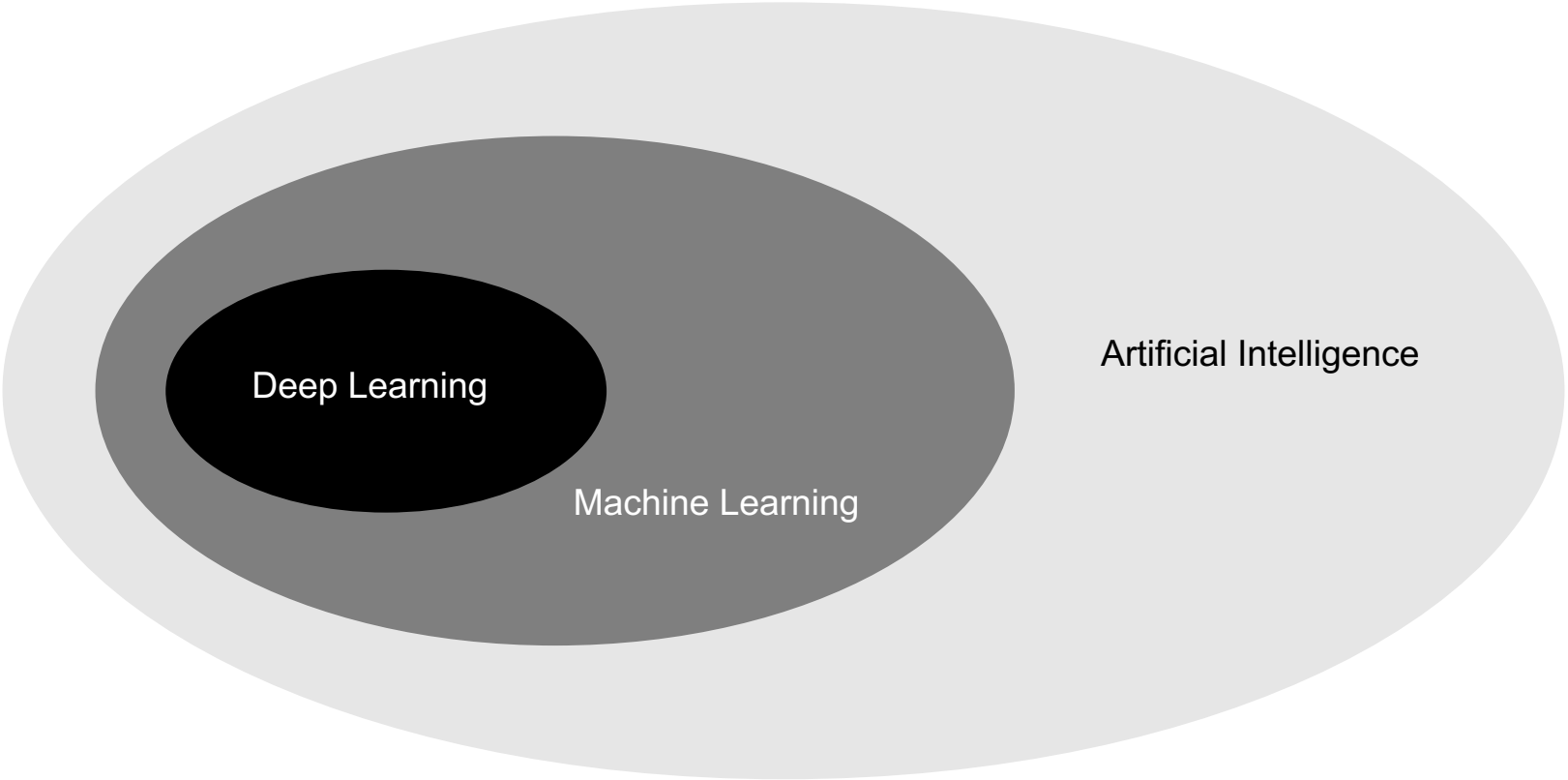
NYU

Outline

- Overview of ML, DL and AI
- Didactic implementation examples (lab exercise)
  - Linear regression
  - Classification of DTI data (MLP)
  - Classification of reconstructed image quality (CNN)
- Generalization: Model complexity, overfitting vs. underfitting
- Architectures: Convolutional vs fully connected

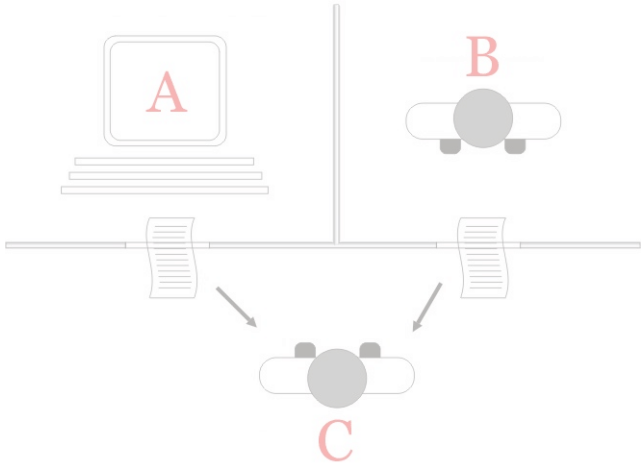
https://github.com/FlorianKnoll/ISMRM2018_Educational_DeepLearning

What is AI?



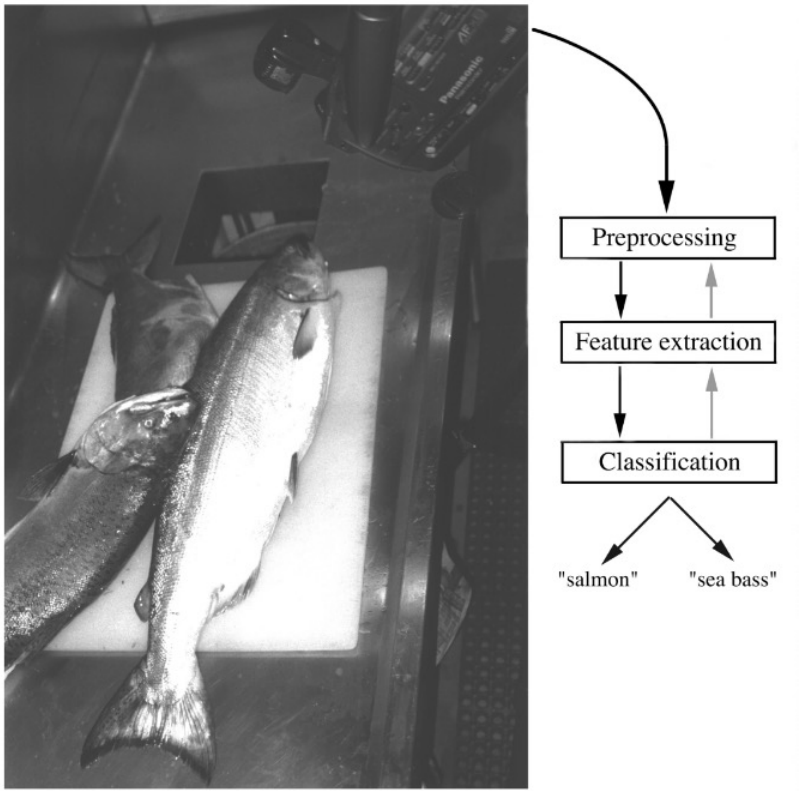
General AI

- Turing Test (Turing)
- Coffee Test (Wozniak)
- Robot College Student Test (Goertzel)
- The Employment Test (Nilsson)
- The flat pack furniture test (Severyns)



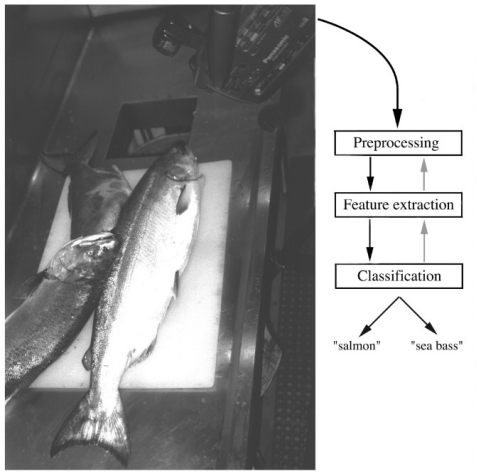
Muehlhauser, Luke. "What is AGI?". Machine Intelligence Research Institute (2014).

ML: A classic computer vision example

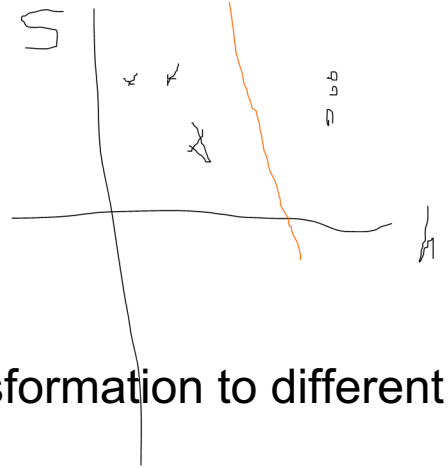


Duda, Hart, Stork: Pattern Classification

Solution 1: Classic computer vision

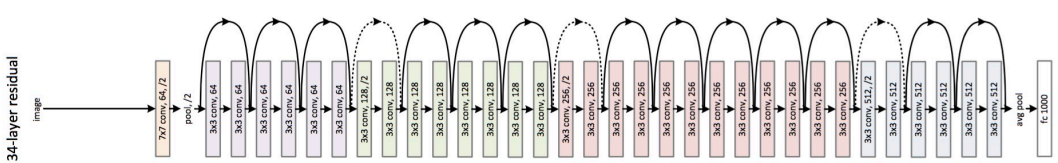


1. Preprocessing: Find individual fish, rotation, lighting
2. Feature extraction: Fit ellipse
3. Classification:
   - linear classifier: kx+d
   - nonlinear classifier or transformation to different feature space



若是非线性DB，
先把feature map 到新的feature space

Solution 2: Deep Learning



Salmon

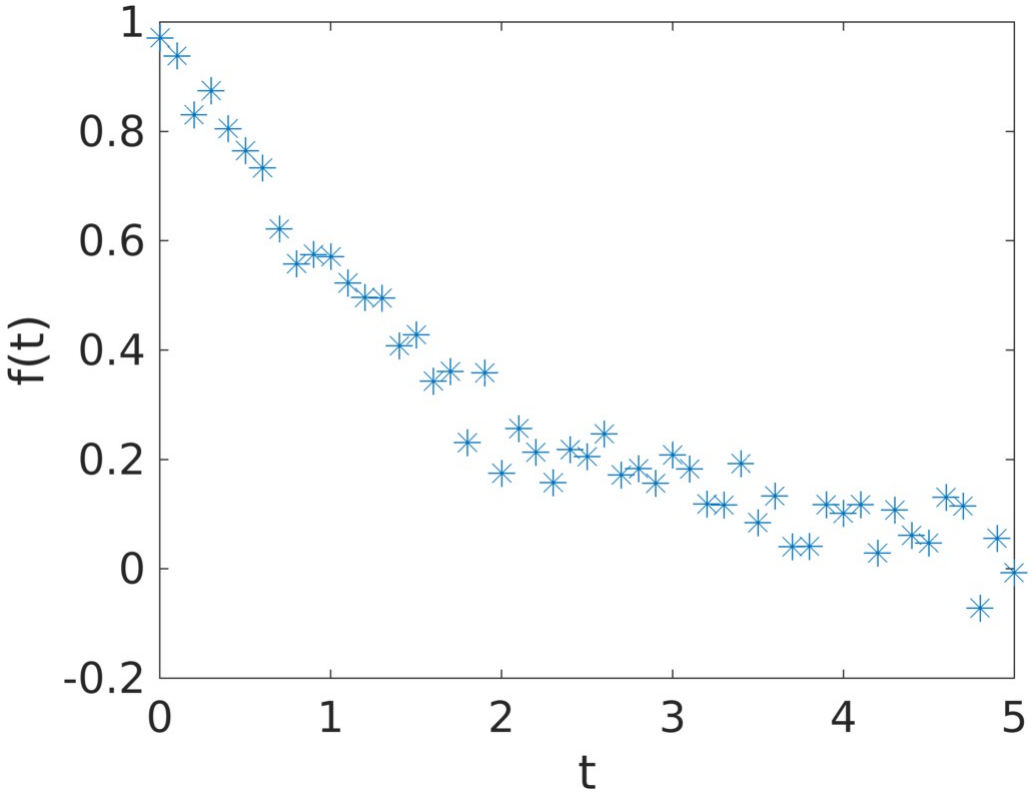
Sea bass

He et al., CVPR 2016

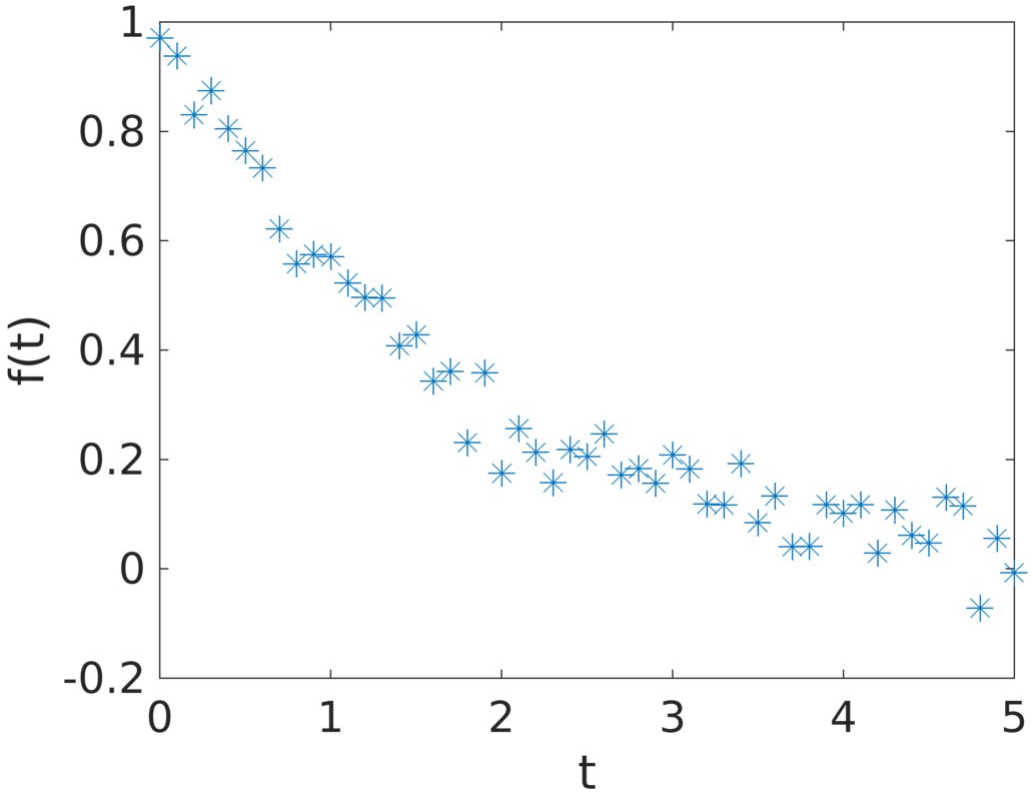
Machine Learning 1: Data

http://www.image-net.org/

Fitting a model to data 1/3

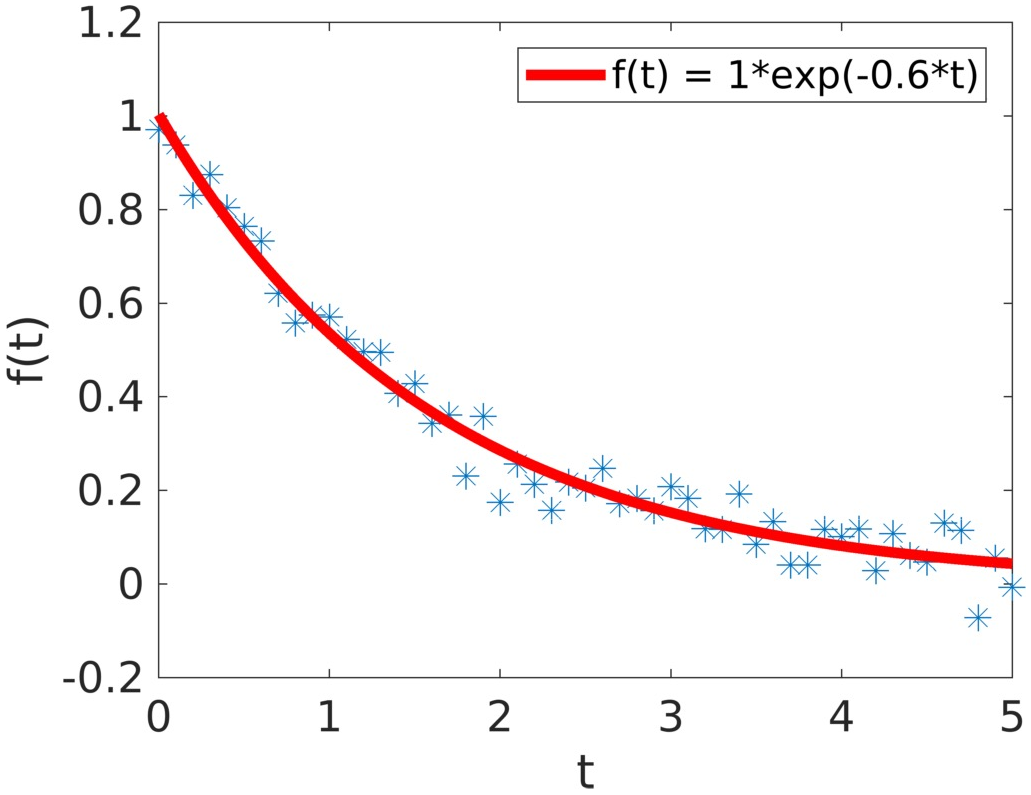


Fitting a model to data 2/3



$$f(t) = ae^{bt}$$

Fitting a model to data 3/3

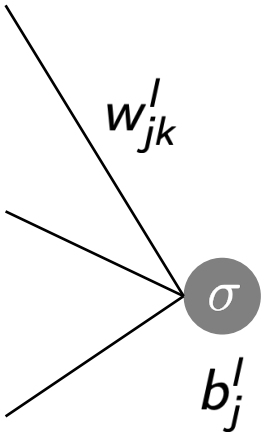


$$f(t) = ae^{bt}$$

$$a = 1$$

$$b = -0.6$$

Neural networks



$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

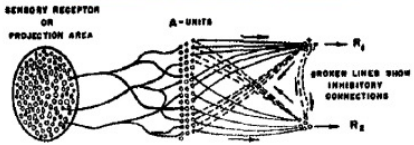


FIG. 2A. Schematic representation of connections in a simple perceptron.

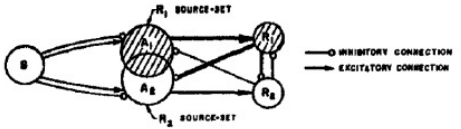
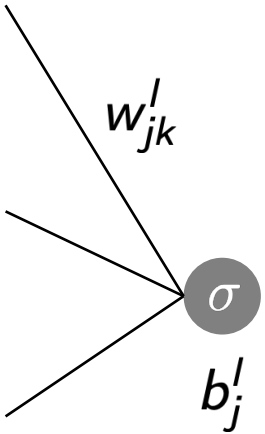


FIG. 2B. Venn diagram of the same perceptron (shading shows active sets for R_1 response).

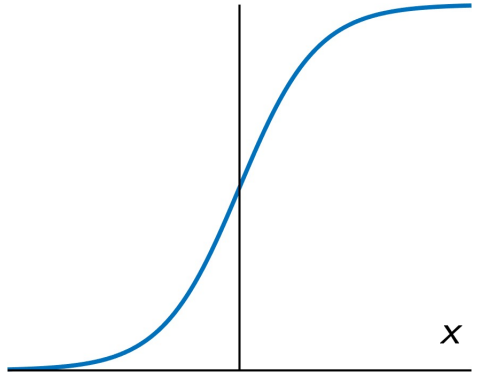
Rosenblatt Psychological review 1957

Neural networks



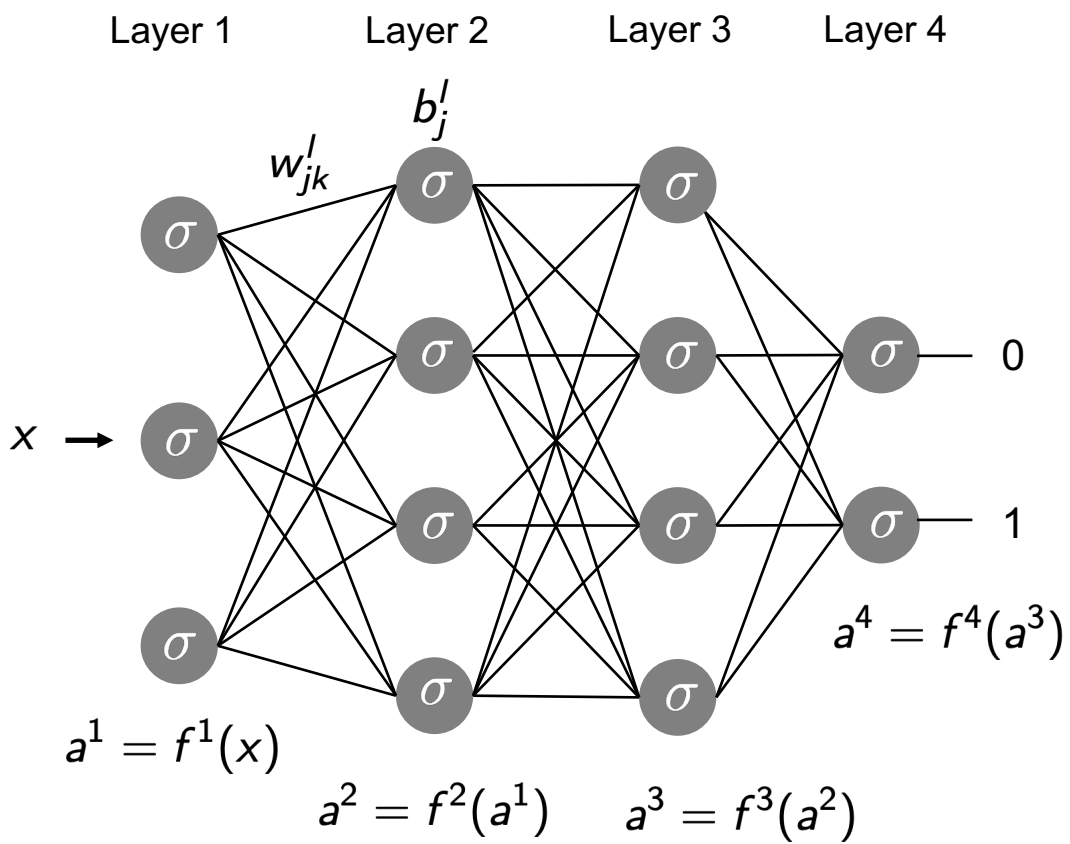
$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

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



Rosenblatt Psychological review 1957

Neural networks



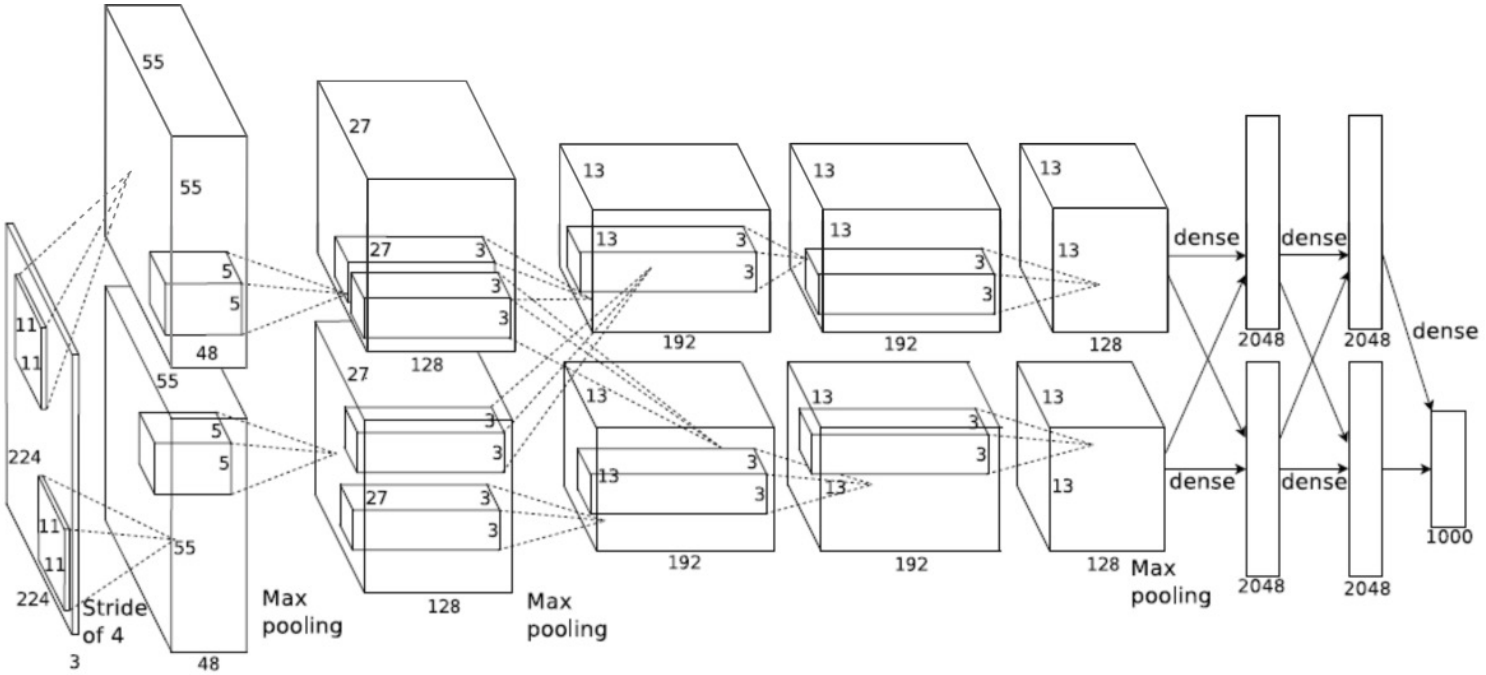
Large number of model parameters

High descriptive capacity

$$a^4 = F(x) = f^4(f^3(f^2(f^1(x))))$$

Cybenko: Math Control Sig Sys 1989

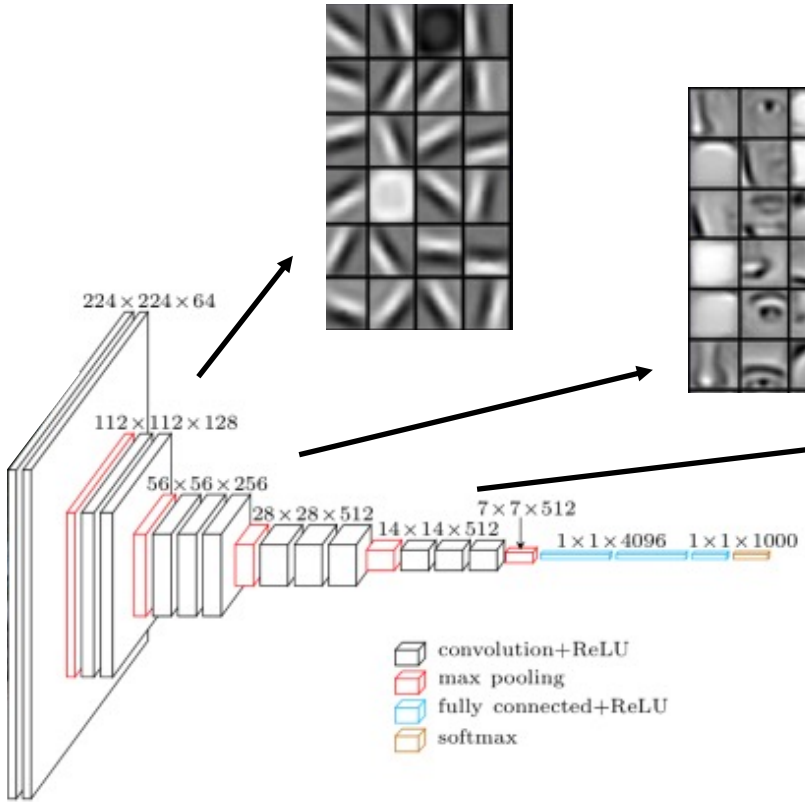
Alexnet



Krizhevsky et al. NIPS 2012

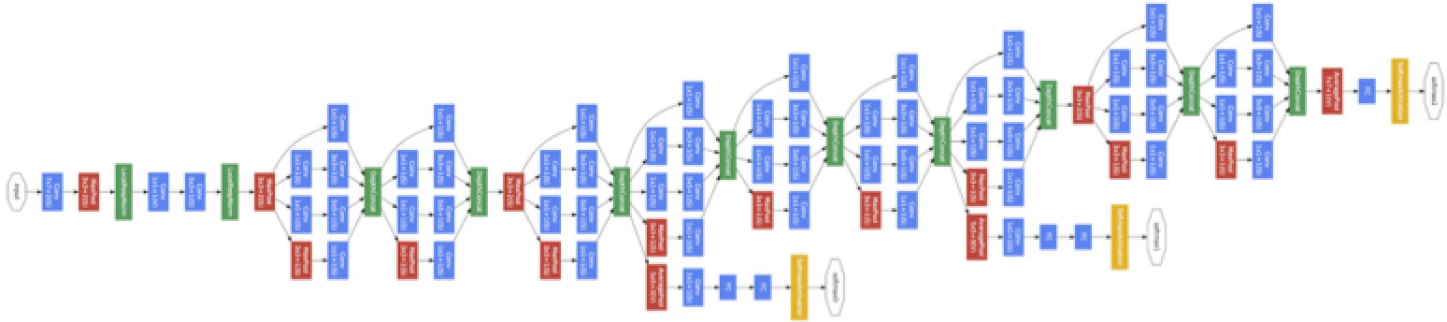
CNN for image categorization

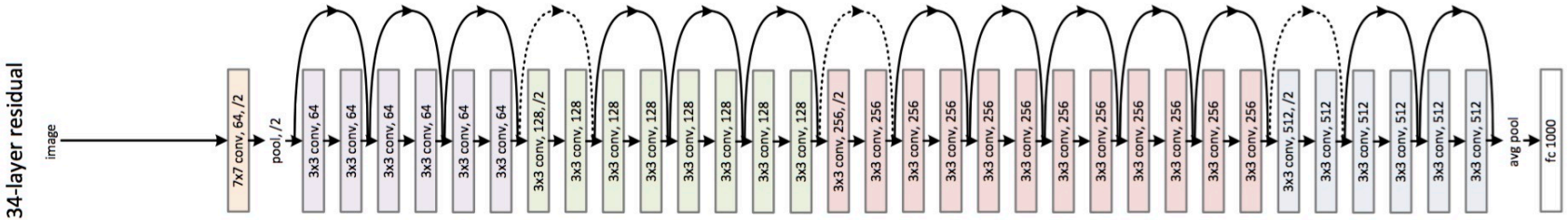
学习到的feature越来越高级



VGG16: K. Simonyan, A. Zisserman ICLR 2015

Googlenet (Inception), Resnet



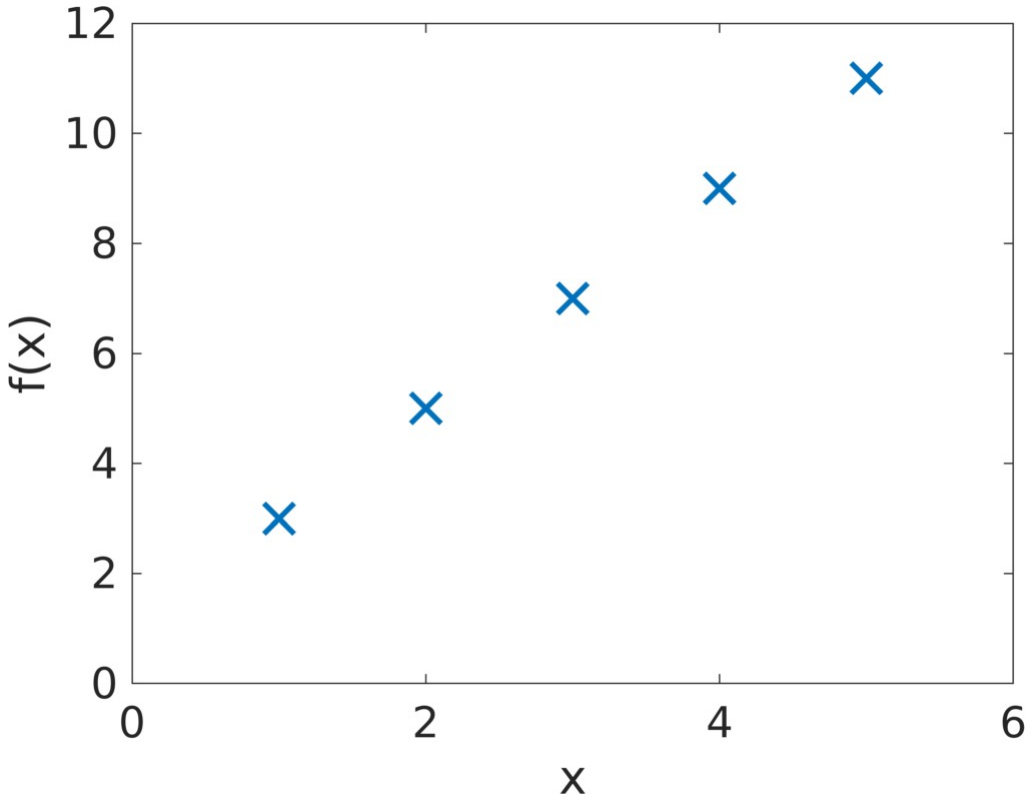


Szegedy et al., CVPR 2015
He et al., CVPR 2016

Example 1: Linear Regression

A low level view of PyTorch

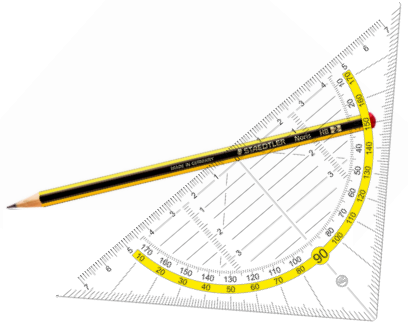
Data



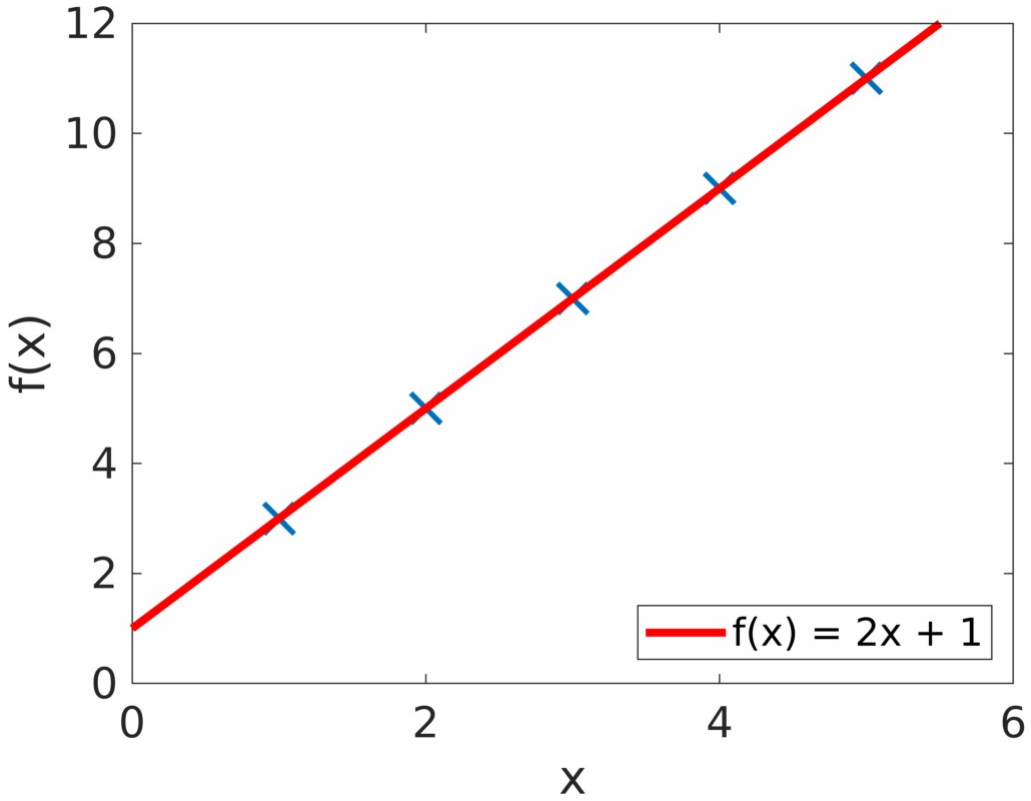
$$x = [1, 2, 3, 4, 5]$$

$$y = [3, 5, 7, 9, 11]$$

$$f(x) = kx + d$$



Data



$$x = [1, 2, 3, 4, 5]$$

$$y = [3, 5, 7, 9, 11]$$

$$f(x) = kx + d$$

$$f(x) = 2x + 1$$

Linear regression with gradient descent: PyTorch

```
#%% training data
x_train = torch.Tensor([1,2,3,4,5]).float()
y_train = torch.Tensor([3,5,7,9,11]).float()

#%% model parameters
k = torch.Tensor([0.1]).float()
k.requires_grad=True

d = torch.Tensor([-0.1]).float()
d.requires_grad=True

#%% model and optimizer
def forward(x):
    return x * k + d

def loss_fcn(x,y):
    y_pred = forward(x)
    return (y_pred-y) * (y_pred-y)

#%% train
training_epochs = 1000
lr = 0.005
loss_ii = np.zeros(training_epochs)
k_ii = np.zeros(training_epochs)
d_ii = np.zeros(training_epochs)

t = time.time()
for ii in range(training_epochs):
    for x_train_batch, y_train_batch in zip(x_train,y_train):
        loss = loss_fcn(x_train_batch, y_train_batch)
        loss_ii[ii] = loss.item()
        k_ii[ii] = k.data
        d_ii[ii] = d.data
        loss.backward()
        k.data = k.data - lr * k.grad.data
        d.data = d.data - lr * d.grad.data

        # Manually set gradient to zero after update step to prevent gradient accumulation
        k.grad.data.zero_()
        d.grad.data.zero_()
    print('epoch: {}, k={:.3}, d={:.3}, loss={:.3}'.format(ii+1,k_ii[ii], d_ii[ii], loss_ii[ii]))

elapsed = time.time() - t
print('Training time: {:.2} s'.format(elapsed))
```

PYTORCH

Model and loss function

```
#%% training data
x_train = torch.Tensor([1,2,3,4,5]).float()
y_train = torch.Tensor([3,5,7,9,11]).float()

#%% model parameters
k = torch.Tensor([0.1]).float()
k.requires_grad=True

d = torch.Tensor([-0.1]).float()
d.requires_grad=True

#%% model and optimizer
def forward(x):
    return x * k + d

def loss_fcn(x,y):
    y_pred = forward(x)
    return (y_pred-y) * (y_pred-y)

#%% train
training_epochs = 1000
lr = 0.005
loss_ii = np.zeros(training_epochs)
k_ii = np.zeros(training_epochs)
d_ii = np.zeros(training_epochs)

t = time.time()
for ii in range(training_epochs):
    for x_train_batch, y_train_batch in zip(x_train,y_train):
        loss = loss_fcn(x_train_batch, y_train_batch)
        loss_ii[ii] = loss.item()
        k_ii[ii] = k.data
        d_ii[ii] = d.data
        loss.backward()
        k.data = k.data - lr * k.grad.data
        d.data = d.data - lr * d.grad.data

        # Manually set gradient to zero after update step to prevent gradient accumulation
        k.grad.data.zero_()
        d.grad.data.zero_()
    print('epoch: {}, k={:.3}, d={:.3}, loss={:.3}'.format(ii+1,k_ii[ii], d_ii[ii], loss_ii[ii]))

elapsed = time.time() - t
print('Training time: {:.2} s'.format(elapsed))
```

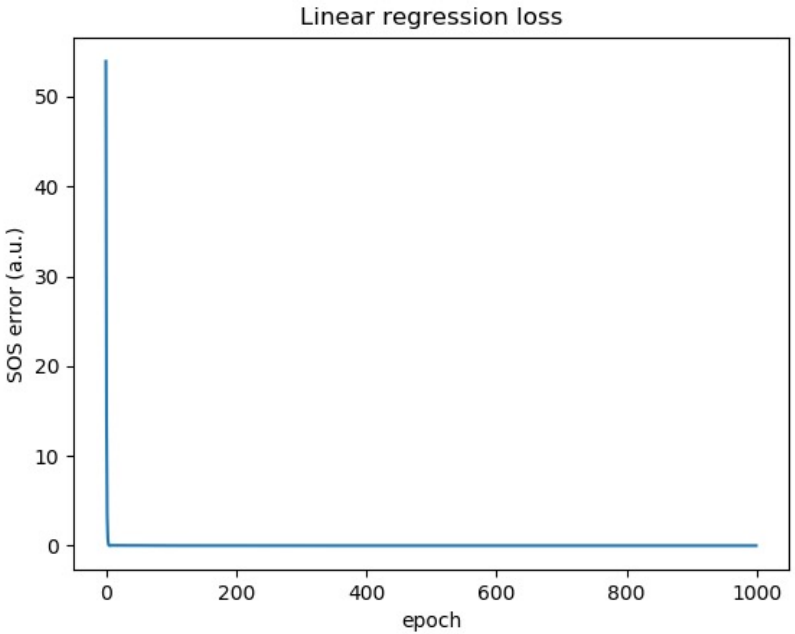
PYTORCH

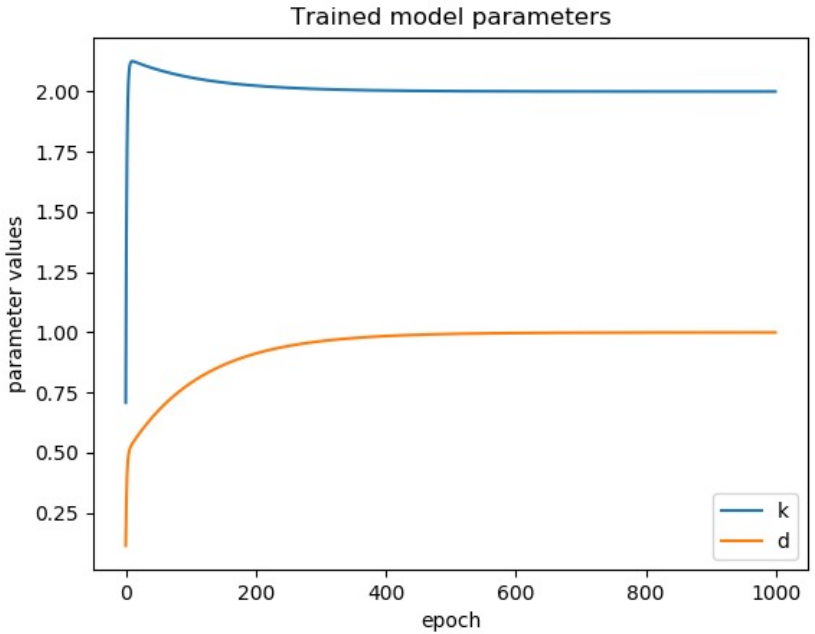
Training the model

```
#%% training data
x_train = torch.Tensor([1,2,3,4,5]).float()
y_train = torch.Tensor([3,5,7,9,11]).float()

#%% model parameters
k = torch.Tensor([0.1]).float()
k.requires_grad=True

d = torch.Tensor([-0.1]).float()
d.requires_grad=True

#%% model and optimizer
def forward(x):
    return x * k + d

def loss_fcn(x,y):
    y_pred = forward(x)
    return (y_pred-y) * (y_pred-y)

#%% train
training_epochs = 1000
lr = 0.005
loss_ii = np.zeros(training_epochs)
k_ii = np.zeros(training_epochs)
d_ii = np.zeros(training_epochs)

t = time.time()
for ii in range(training_epochs):
    for x_train_batch, y_train_batch in zip(x_train,y_train):
        loss = loss_fcn(x_train_batch, y_train_batch)
        loss_ii[ii] = loss.item()
        k_ii[ii] = k.data
        d_ii[ii] = d.data
        loss.backward()
        k.data = k.data - lr * k.grad.data
        d.data = d.data - lr * d.grad.data

        # Manually set gradient to zero after update step to prevent gradient accumulation
        k.grad.data.zero_()
        d.grad.data.zero_()
    print('epoch: {}, k={:.3}, d={:.3}, loss={:.3}'.format(ii+1,k_ii[ii], d_ii[ii], loss_ii[ii]))

elapsed = time.time() - t
print('Training time: {:.2} s'.format(elapsed))
```

PYTORCH

Linear regression with gradient descent: PyTorch

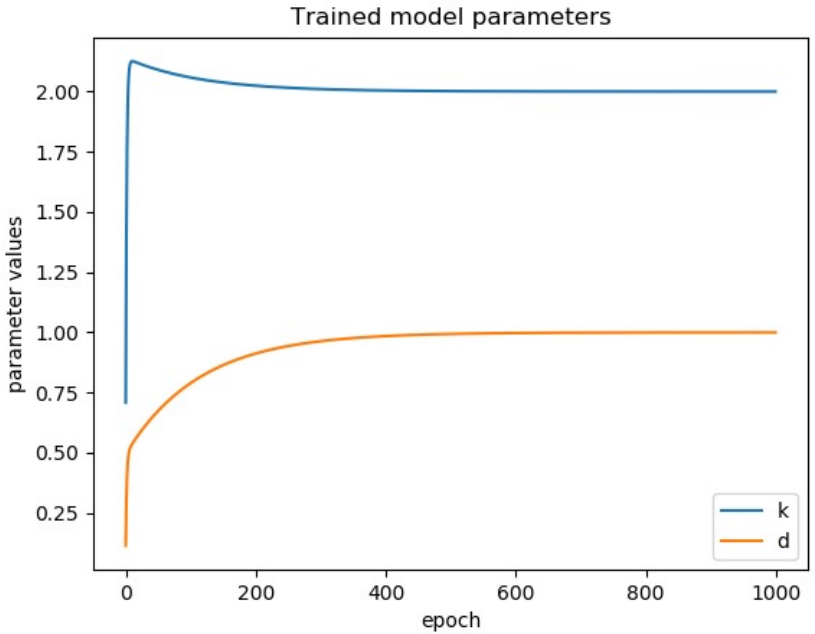




Linear regression in PyTorch

```
epoch: 993, k=2.0, d=1.0, loss=1.11e-09
epoch: 994, k=2.0, d=1.0, loss=1.11e-09
epoch: 995, k=2.0, d=1.0, loss=9.9e-10
epoch: 996, k=2.0, d=1.0, loss=9.9e-10
epoch: 997, k=2.0, d=1.0, loss=9.9e-10
epoch: 998, k=2.0, d=1.0, loss=9.9e-10
epoch: 999, k=2.0, d=1.0, loss=9.31e-10
epoch: 1000, k=2.0, d=1.0, loss=9.9e-10
Training time: 1.4 s
```

$$f(x) = 2x + 1$$

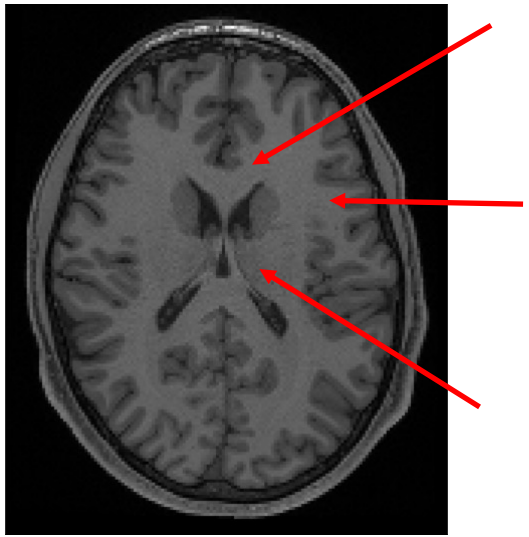


Example 2: Classification of brain tissue from DTI data

High level neural network modules

Classification of brain tissue from HCP DTI data

T1w

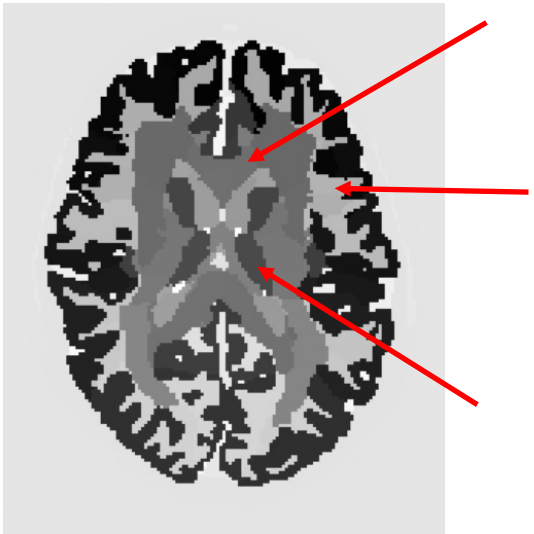


Genu of corpus callusum
(highly aligned WM)

Subcortical WM

Thalamus (GM)

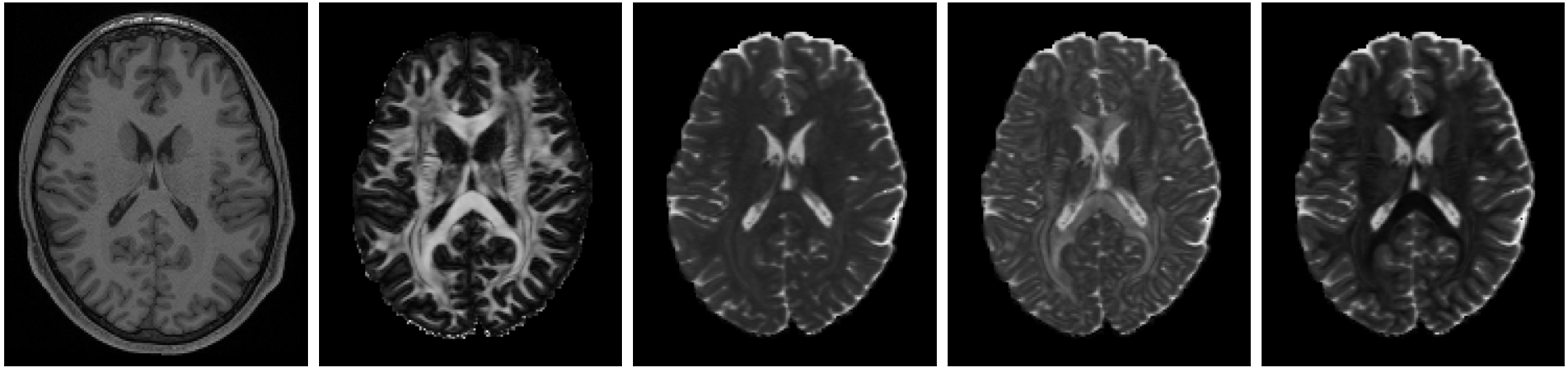
Segmentation



https://www.humanconnectome.org/

Classification of brain tissue from HCP DTI data

T1w | FA | MD $(\lambda_1 + \lambda_2 + \lambda_3)/3$ | AD (λ_1) | RD $(\lambda_2 + \lambda_3)/2$



fMRI

https://www.humanconnectome.org/

Structure and inspection of the data

$$\{(x_1, y_1), \ldots (x_N, y_N)\}$$

$$x_i = [T1w_i, FA_i, MD_i, AD_i, RD_i]$$

T1w (a.u.)	FA (-)	MD $\left(\frac{\mu m^2}{ms}\right)$	AD $\left(\frac{\mu m^2}{ms}\right)$	RD $\left(\frac{\mu m^2}{ms}\right)$	Class	Class label
898	0.22	1.066592	1.33	0.94	Thalamus	1
1007	0.68	0.39	0.72	0.22	CC	2
867	0.38	0.58	0.82	0.45	Cortical WM	3
…	…	...	...	...	…	…

Normalization!

Training, validation and test set

$$\{(x_1, y_1), \ldots (x_N, y_N)\}$$

1. Training set: Train the model
2. Validation set: Use as performance criterion to tune training, for hyperparameter selection,…
3. Test set: Don’t use during training at all!

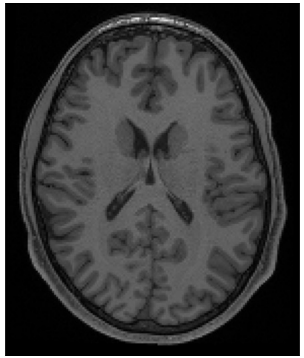
Training, validation and test set

1. Training set: 80% randomized samples from 3 subjects
2. Validation set: 20% randomized samples from 3 subjects
3. Test set: Randomized samples from 1 different subject

$$\{(x_1, y_1), \ldots (x_N, y_N)\}$$

$$N = 23700 \qquad N_{train} = 14044$$
$$N_{val} = 3511 \qquad N_{test} = 6145$$

Check balancing of classes



$$N_{thal} = 11662$$

$$N_{cc} = 4582$$

$$N_{subcort-wm} = 7455$$

- Create data duplicates
- Weight samples during training

Buda et al., https://arxiv.org/abs/1710.05381 (2017)

Let’s build our first neural network

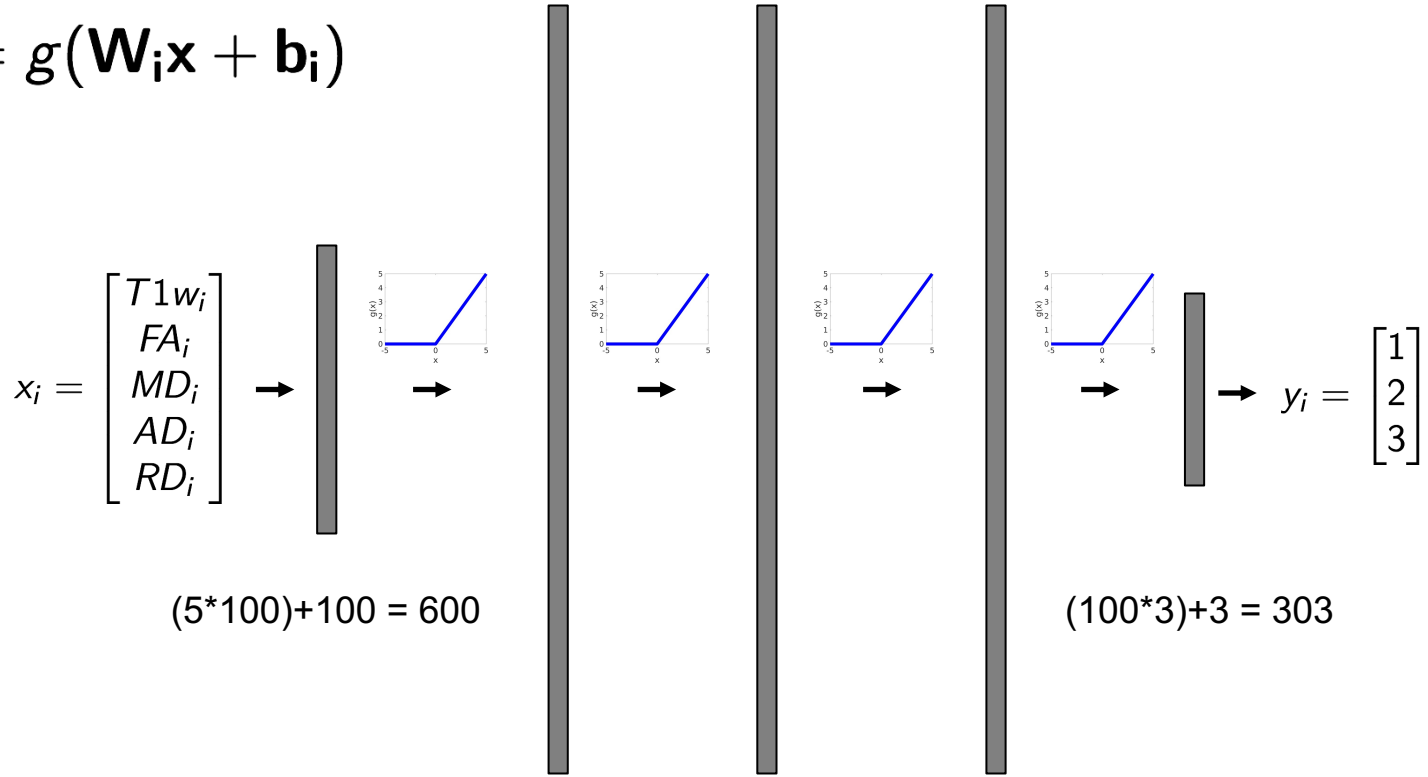
Multi layer perceptron

- 1 input layer
- 3 hidden layers with 100 elements
- 1 output layer
- Non-linearity: ReLu
- Softmax output

Bishop, C. M. *Neural Networks for Pattern Recognition*. Oxford University Press. (1995)

Model Architecture

$$y_i(\mathbf{x}) = g(\mathbf{W_i x} + \mathbf{b_i})$$



Total number of parameters: 31203

PyTorch implementation

```
#%% Define model
nElements = 100
nLayers = 3
model_name = 'dti_FC'
model = torch.nn.Sequential(
    torch.nn.Linear(nFeatures, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nClasses, bias=True),
)
print(model)
```

```
#%%choose optimizer and loss function
training_epochs = 250
lr = 0.001
batch_size = 1024
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

```
#%%Create minibatch data loading for training and validation
dataloader_train = data_utils.TensorDataset(x_train, y_train)
dataloader_train = data_utils.DataLoader(dataloader_train, batch_size=batch_size, shuffle=False,num_workers=4)
```

```
#%% Train model
loss_train = np.zeros(training_epochs)
acc_train = np.zeros(training_epochs)
loss_val = np.zeros(training_epochs)
acc_val = np.zeros(training_epochs)

for epoch in range(training_epochs):
    for local_batch, local_labels in dataloader_train:
        # feedforward - backpropagation
        optimizer.zero_grad()
        out = model(local_batch)
        loss = criterion(out, local_labels)
        loss.backward()
        optimizer.step()
        loss_train[epoch] = loss.item()
        # Training data accuracy
        [dummy, predicted] = torch.max(out.data, 1)
        acc_train[epoch] = (torch.sum(local_labels==predicted).numpy() / np.size(local_labels.numpy(),0))

        # Validation
        out_val = model(x_val)
        loss = criterion(out_val, y_val)
        loss_val[epoch] = loss.item()
        [dummy, predicted_val] = torch.max(out_val.data, 1)
        acc_val[epoch] = ( torch.sum(y_val==predicted_val).numpy() / setsize_val)

    print ('Epoch {}/{} train loss: {:.3}, train acc: {:.3}, val loss: {:.3}, val acc: {:.3}'.format(epoch+1, training_epochs, loss_train[epoch], acc_train[epoch], loss_val[epoch], acc_val[epoch]))
```

PYTORCH

Set up optimizer and data support functions

```
#%% Define model
nElements = 100
nLayers = 3
model_name = 'dti_FC'
model = torch.nn.Sequential(
    torch.nn.Linear(nFeatures, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nClasses, bias=True),
)
print(model)

#%%choose optimizer and loss function
training_epochs = 250
lr = 0.001
batch_size = 1024
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

#%%Create minibatch data loading for training and validation
dataloader_train = data_utils.TensorDataset(x_train, y_train)
dataloader_train = data_utils.DataLoader(dataloader_train, batch_size=batch_size, shuffle=False,num_workers=4)

#%% Train model
loss_train = np.zeros(training_epochs)
acc_train = np.zeros(training_epochs)
loss_val = np.zeros(training_epochs)
acc_val = np.zeros(training_epochs)

for epoch in range(training_epochs):
    for local_batch, local_labels in dataloader_train:
        # feedforward - backpropagation
        optimizer.zero_grad()
        out = model(local_batch)
        loss = criterion(out, local_labels)
        loss.backward()
        optimizer.step()
        loss_train[epoch] = loss.item()
        # Training data accuracy
        [dummy, predicted] = torch.max(out.data, 1)
        acc_train[epoch] = (torch.sum(local_labels==predicted).numpy() / np.size(local_labels.numpy(),0))

        # Validation
        out_val = model(x_val)
        loss = criterion(out_val, y_val)
        loss_val[epoch] = loss.item()
        [dummy, predicted_val] = torch.max(out_val.data, 1)
        acc_val[epoch] = ( torch.sum(y_val==predicted_val).numpy() / setsize_val)

    print ('Epoch {}/{} train loss: {:.3}, train acc: {:.3}, val loss: {:.3}, val acc: {:.3}'.format(epoch+1, training_epochs, loss_train[epoch], acc_train[epoch], loss_val[epoch], acc_val[epoch]))
```

PYTORCH

Kingma and Ba, ICLR 2015

Let’s train our network

```
%% Define model
nElements = 100
nLayers = 3
model_name = 'dti_FC'
model = torch.nn.Sequential(
    torch.nn.Linear(nFeatures, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nElements, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(nElements, nClasses, bias=True),
)
print(model)
```

```
%%choose optimizer and loss function
training_epochs = 250
lr = 0.001
batch_size = 1024
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

```
%%Create minibatch data loading for training and validation
dataloader_train = data_utils.TensorDataset(x_train, y_train)
dataloader_train = data_utils.DataLoader(dataloader_train, batch_size=batch_size, shuffle=False,num_workers=4)
```

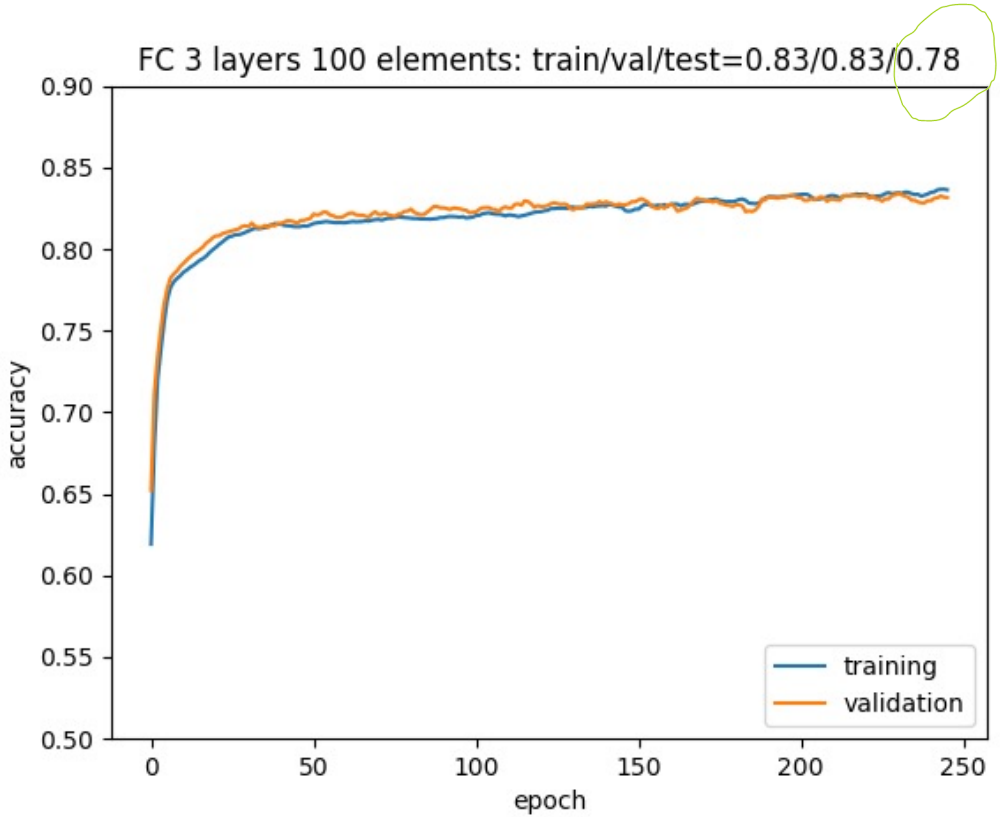
```
%% Train model
loss_train = np.zeros(training_epochs)
acc_train = np.zeros(training_epochs)
loss_val = np.zeros(training_epochs)
acc_val = np.zeros(training_epochs)

for epoch in range(training_epochs):
    for local_batch, local_labels in dataloader_train:
        # feedforward - backpropagation
        optimizer.zero_grad()
        out = model(local_batch)
        loss = criterion(out, local_labels)
        loss.backward()
        optimizer.step()
        loss_train[epoch] = loss.item()
        # Training data accuracy
        [dummy, predicted] = torch.max(out.data, 1)
        acc_train[epoch] = (torch.sum(local_labels==predicted).numpy() / np.size(local_labels.numpy(),0))

        # Validation
        out_val = model(x_val)
        loss = criterion(out_val, y_val)
        loss_val[epoch] = loss.item()
        [dummy, predicted_val] = torch.max(out_val.data, 1)
        acc_val[epoch] = ( torch.sum(y_val==predicted_val).numpy() / setsize_val)

    print ('Epoch {}/{} train loss: {:.3}, train acc: {:.3}, val loss: {:.3}, val acc: {:.3}'.format(epoch+1, training_epochs, loss_train[epoch], acc_train[epoch], loss_val[epoch], acc_val[epoch]))
```

PYTORCH

Kingma and Ba, ICLR 2015

Results

test set里面有其他病人，模型的泛化能力不够好

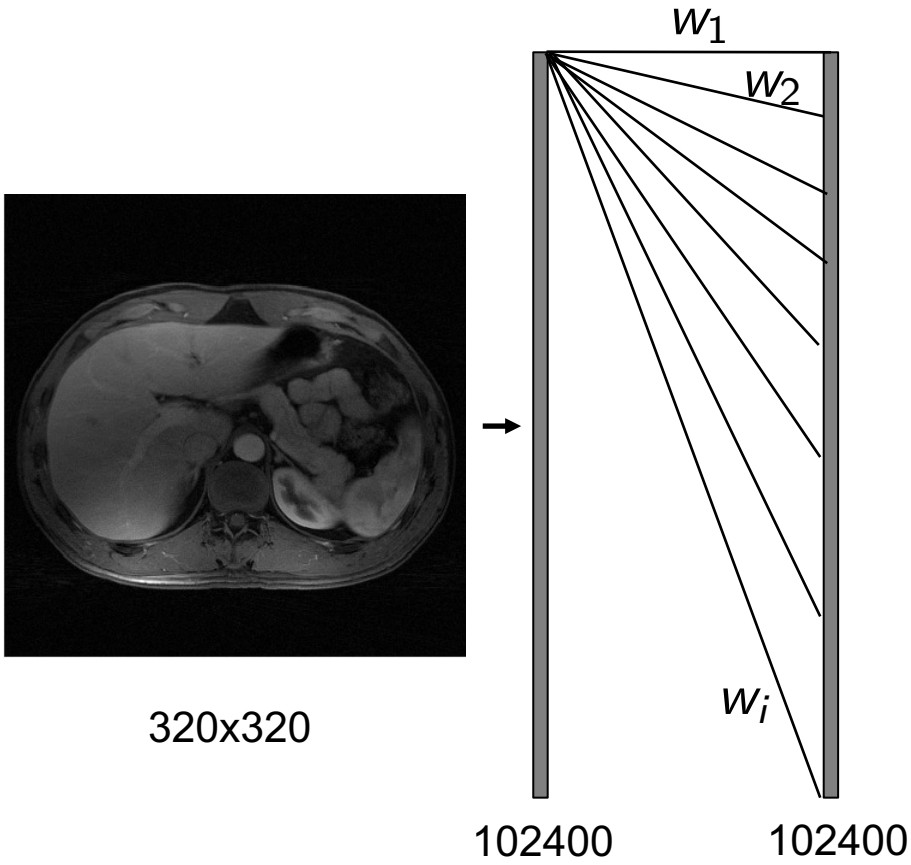


- Higher model complexity?
- Inspect misclassified samples
- Trends for different label categories?
- Experiment with features
- Don’t use test performance to adjust model architecture!

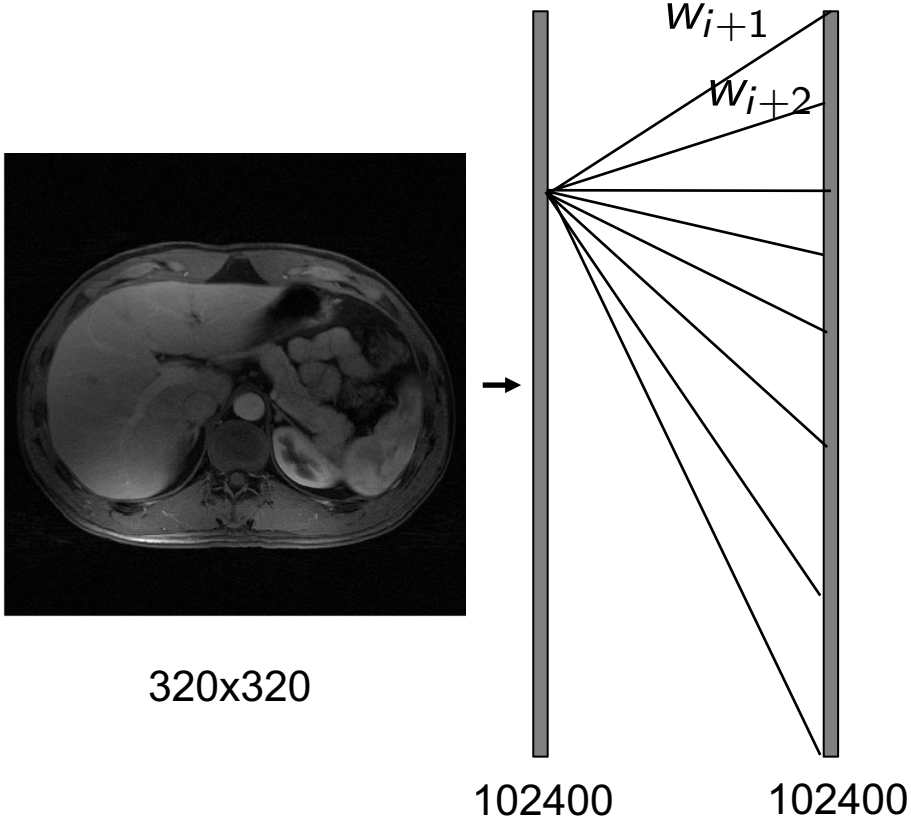
Example 3: Classification of image quality of accelerated reconstructions

Convolutional Neural Networks (CNNs), model complexity

Fully connected Neural Networks



Fully connected Neural Networks

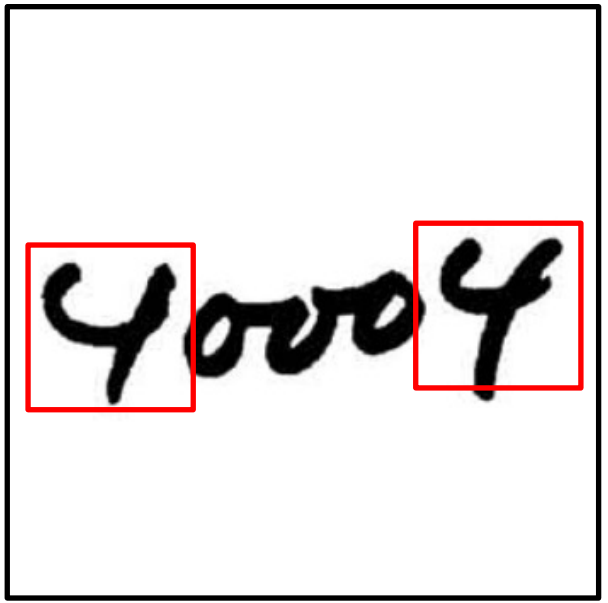


Number of parameters:
$102400 * 102400 \approx 1.05 * 10^{10}$

More efficient use of parameters!

Convolutional neural networks

Figure 1: Examples of original zipcodes from the testing set.



Local connectivity

Share parameters

Lecun: NIPS 1989

Convolutional neural networks

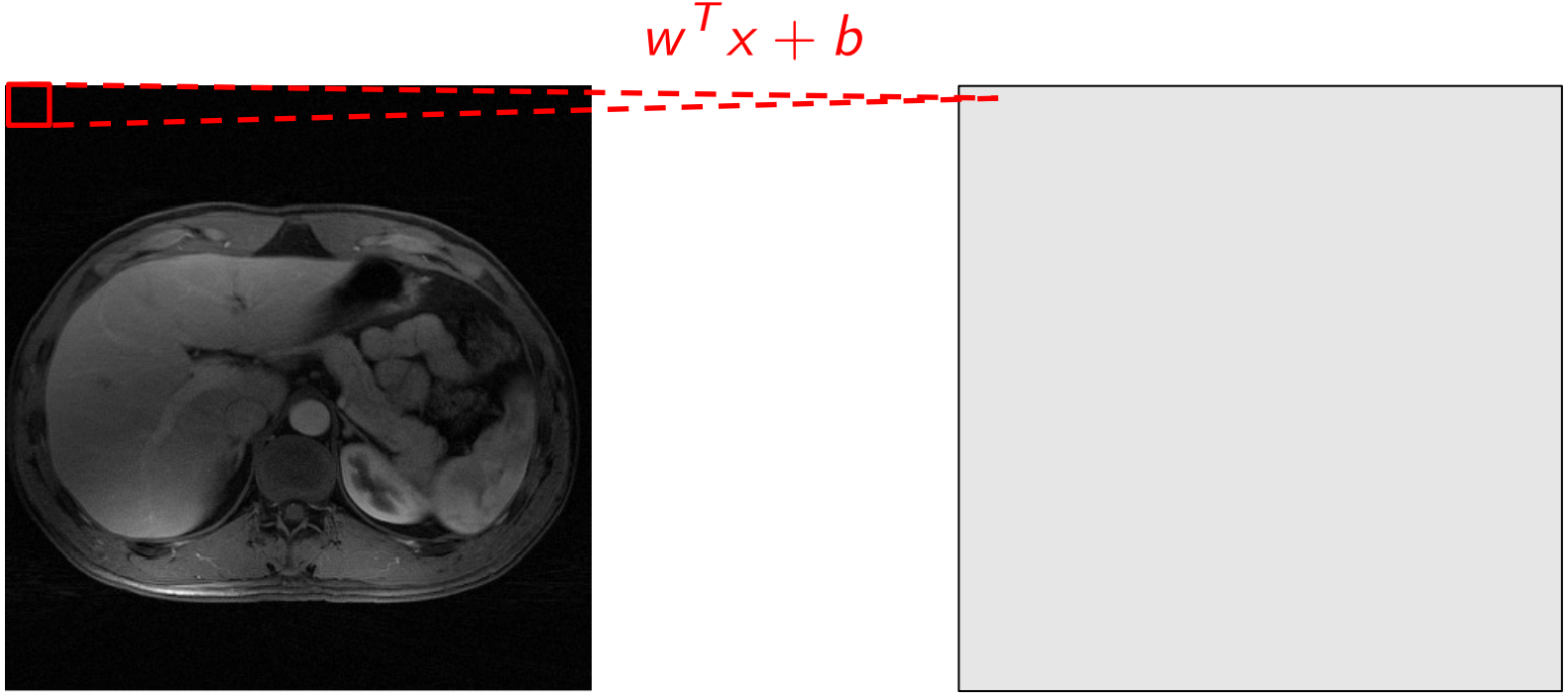
Figure 1: Examples of original zipcodes from the testing set.

Local connectivity

Share parameters

Lecun: NIPS 1989

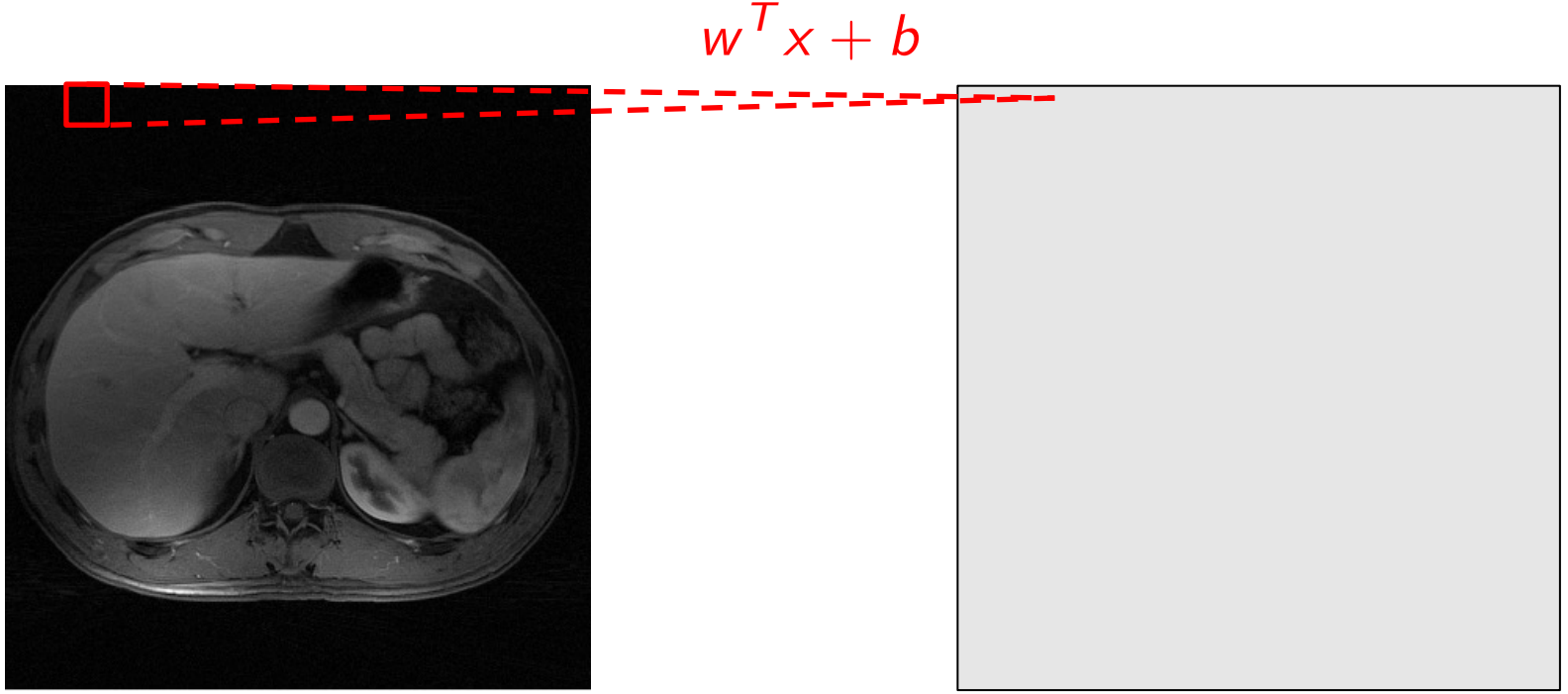
Convolutional layers



320x320 image
3x3 filter *w*

Lecun: NIPS 1989

Convolutional layers

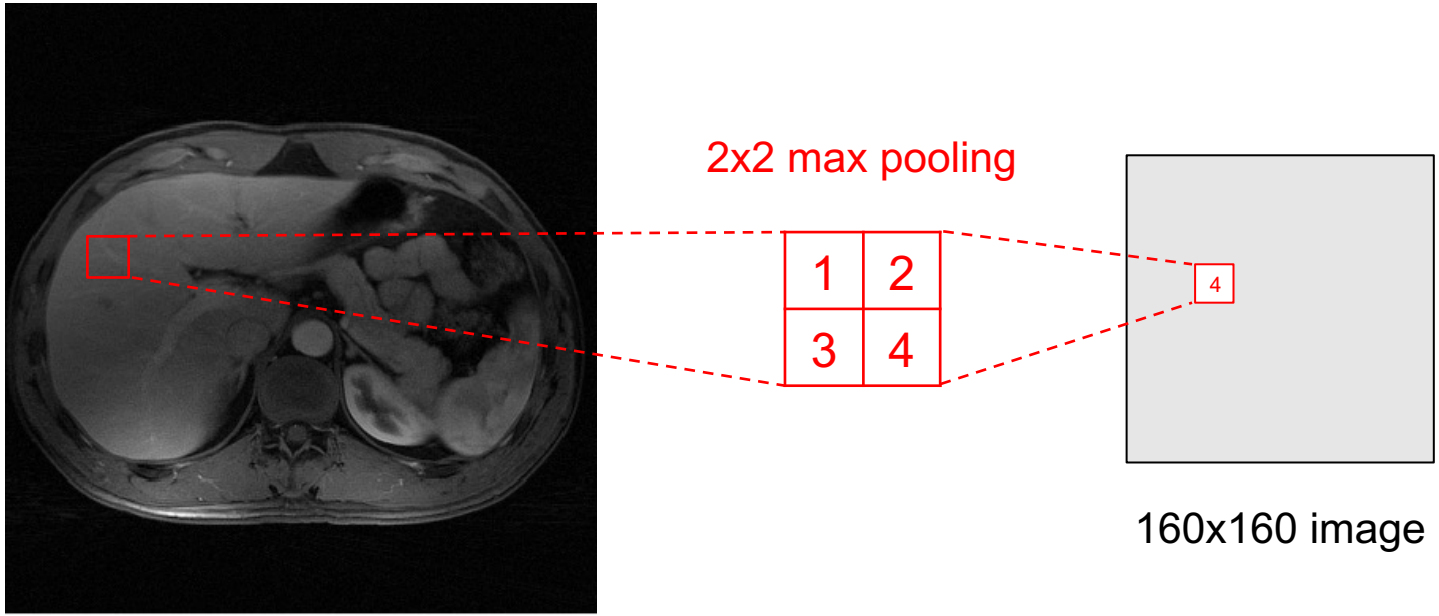


320x320 image
3x3 filter w

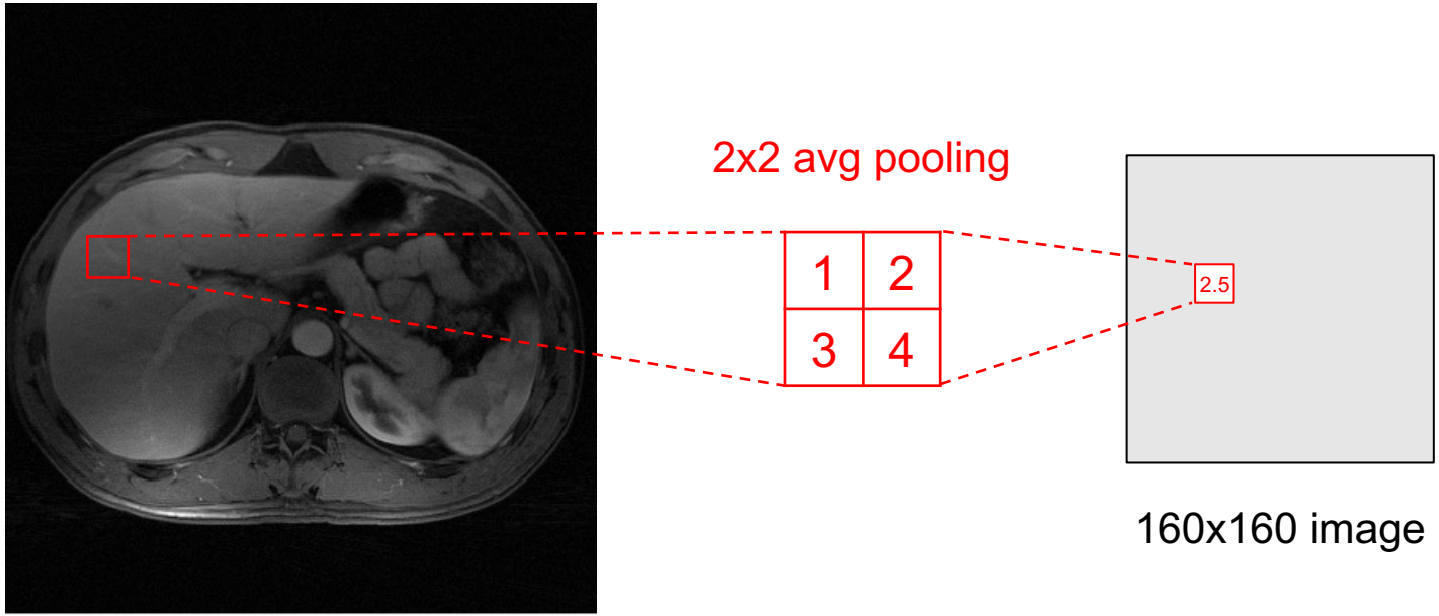
Model parameters: 3*3+1 = 10

Lecun: NIPS 1989

Pooling layers



Pooling layers



No model parameters

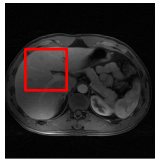
Pooling layers

320x320 image
3x3 filter w

2x2
pooling

160x160 image
3x3 filter w

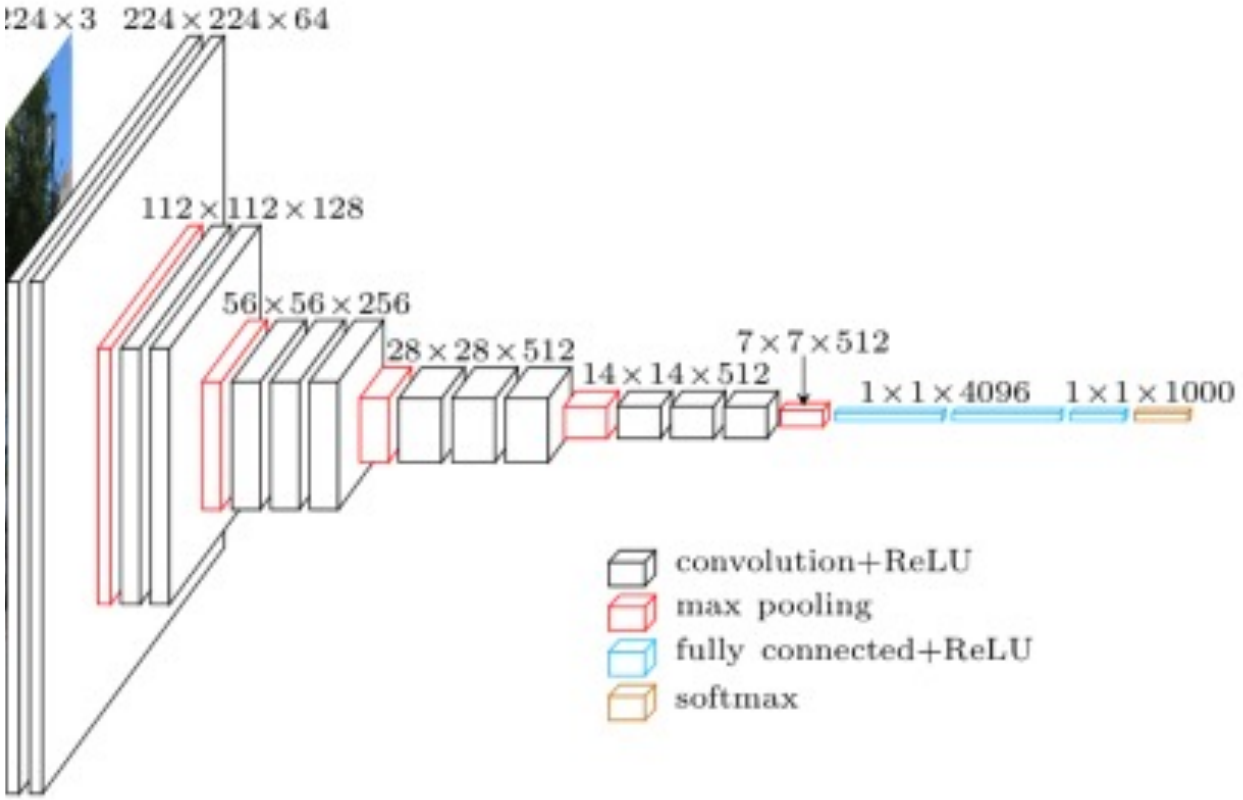
2x2
pooling



80x80 image
3x3 filter w

- Increases receptive field
- Multi scale image analysis
- Compression of information

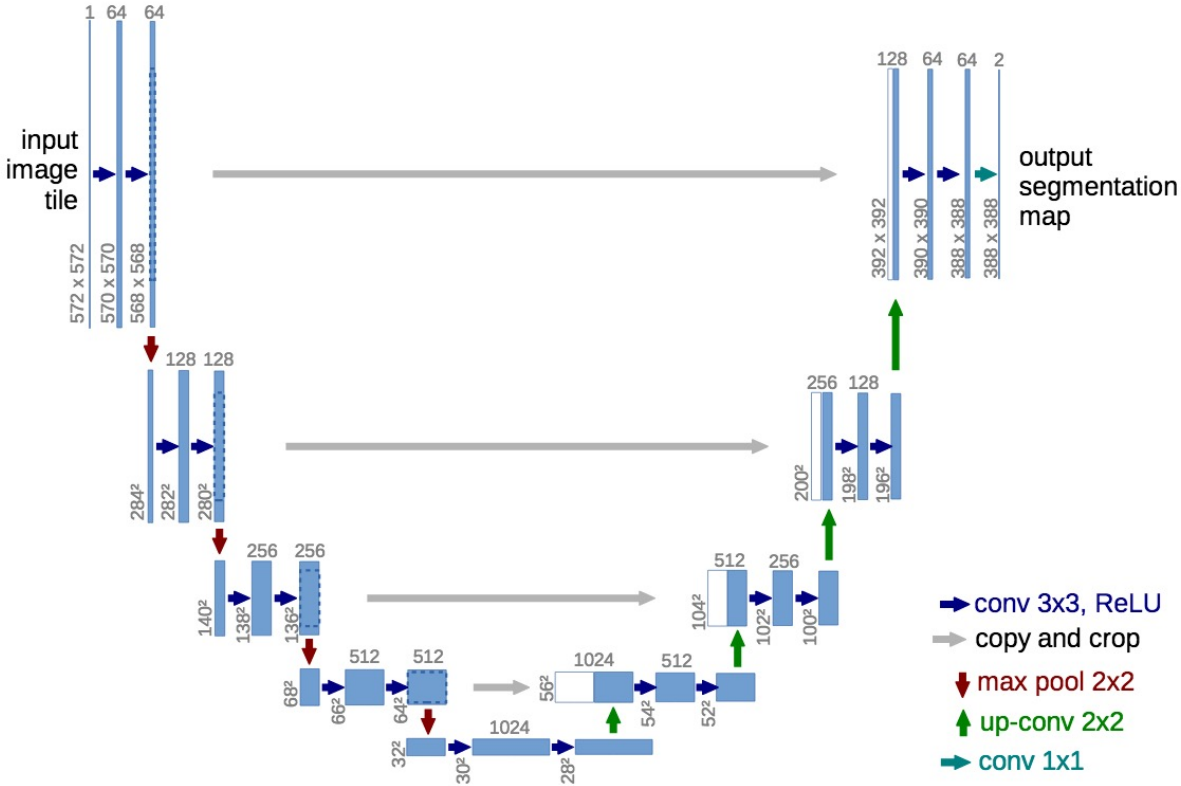
Example architecture: VGG



Popular for image categorization

Simonyan ICLR 2015

Example architecture: U-Net



Popular for image processing

and segmentation

Ronneberger MICCAI 2015

Example 3: Classification of image quality of accelerated reconstructions

Fully sampled vs 4 times PI-CS accelerated

CS:compress sensing

Fully sampled reference

PI-CS (TGV) R=4

Knoll, MRM (2011)

Fully sampled vs 4 times PI-CS accelerated

Fully sampled reference

PI-CS (TGV) R=4

Knoll, MRM (2011)

Fully sampled vs 4 times PI-CS accelerated

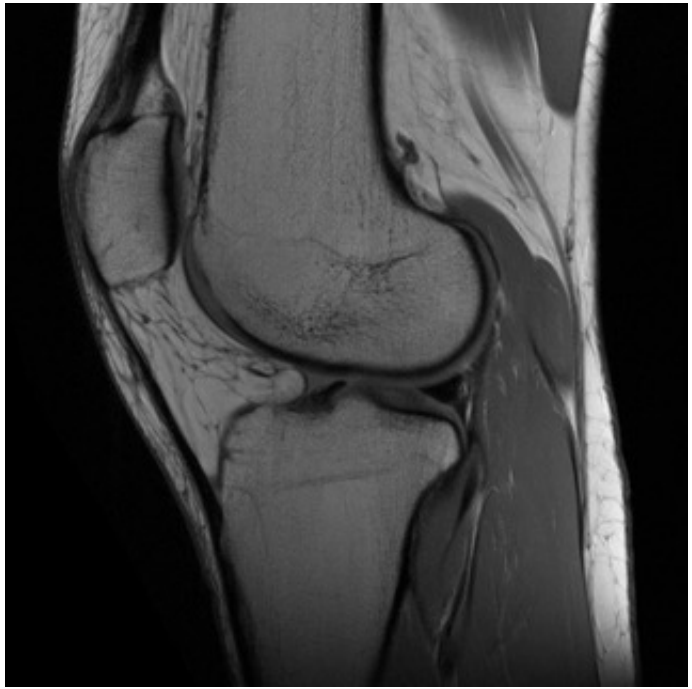
Fully sampled reference

PI-CS (TGV) R=4

Knoll, MRM (2011)

Fully sampled vs 4 times PI-CS accelerated

Fully sampled reference



PI-CS (TGV) R=4

Knoll, MRM (2011)

Training, validation and test set

1. Training set: 2/3 randomized samples
2. Validation set: 1/3 randomized samples
3. No separate test set: Experiment is left to you :-)

$$\{(x_1, y_1), \ldots (x_N, y_N)\}$$

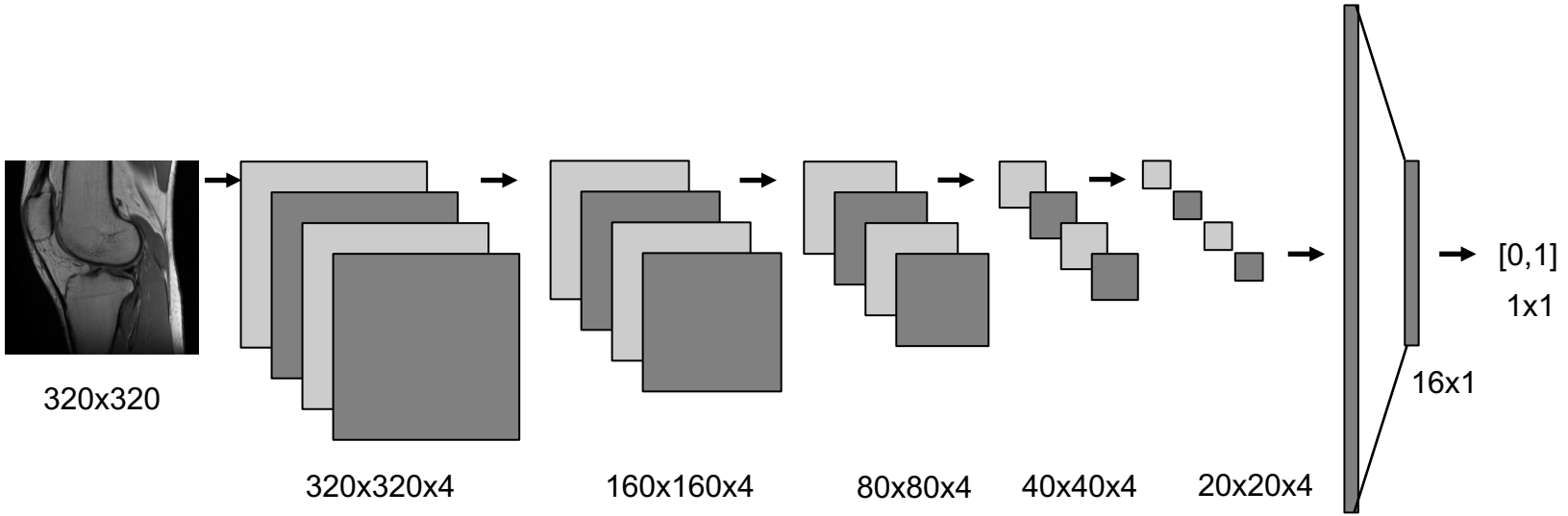
$$N = 520$$

$$N_{train} = 390 \quad N_{ref} = 260$$

$$N_{val} = 130 \quad N_{tgv} = 260$$

total generalize validation?

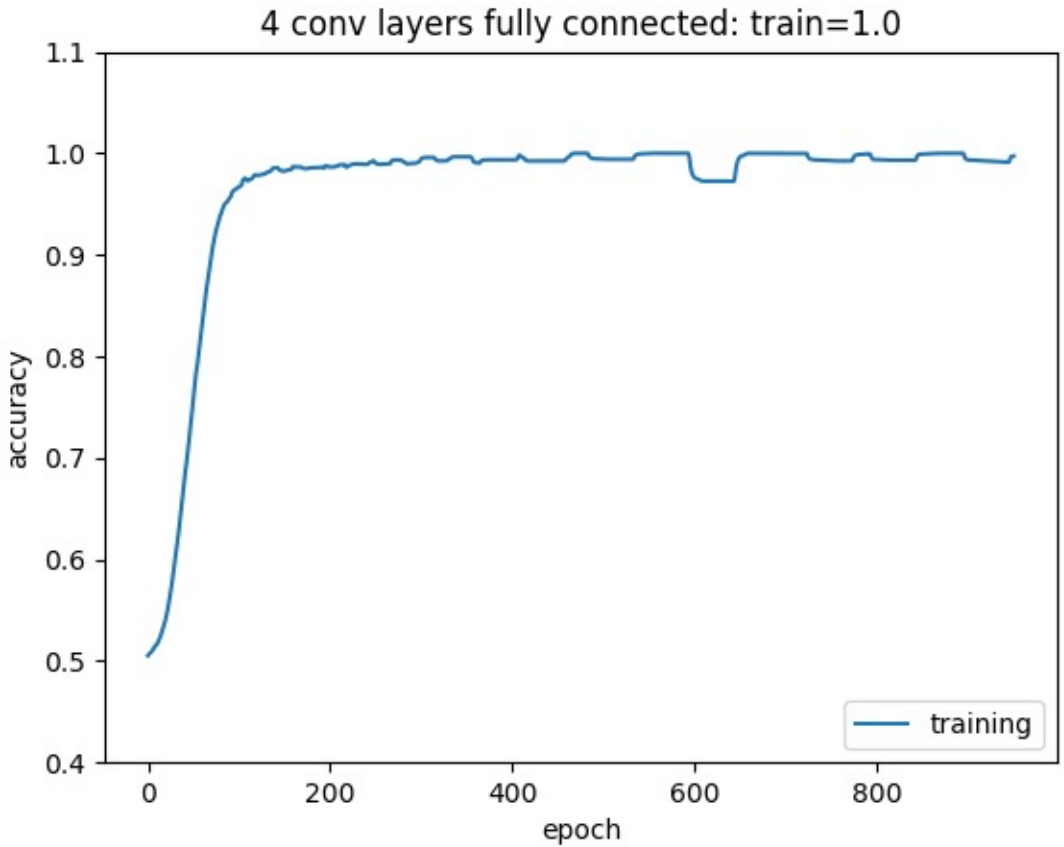
Our first convolutional neural network (CNN)



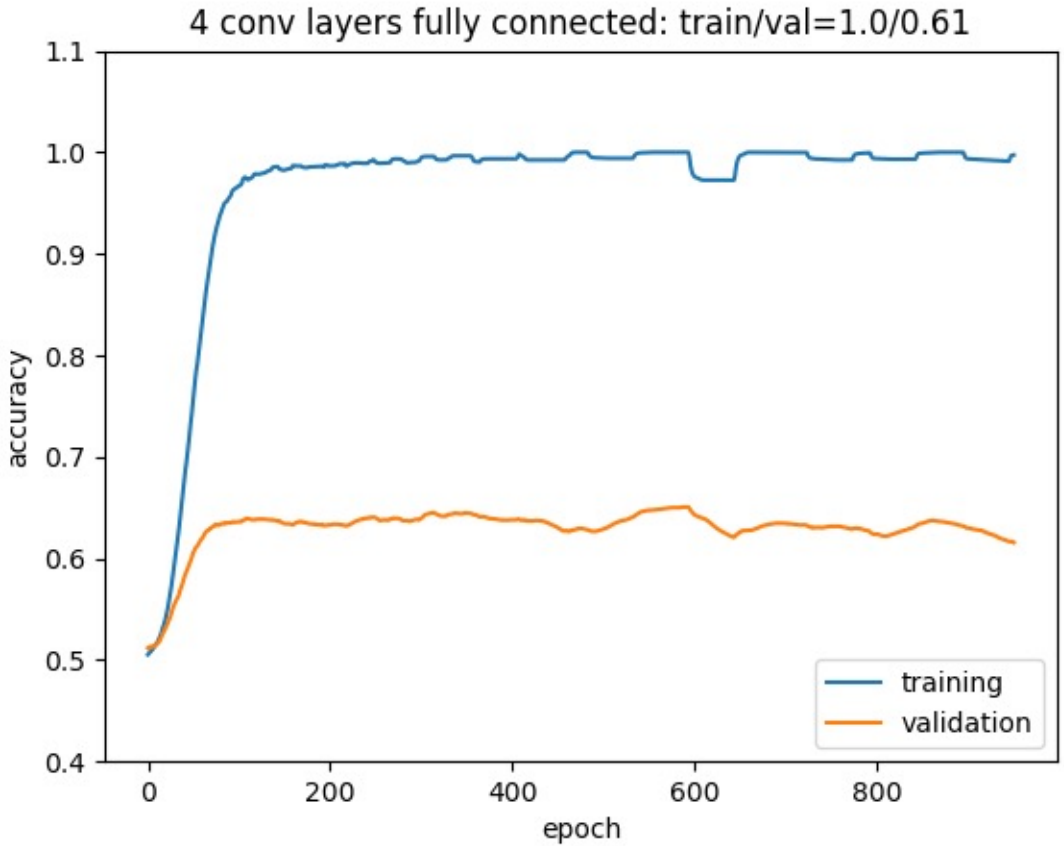
4 3x3x4 convolutional layers, Relu activation
2D MaxPooling
1 fully connected layer with 16 elements
Sigmoid Output

Total number of parameters: 26117

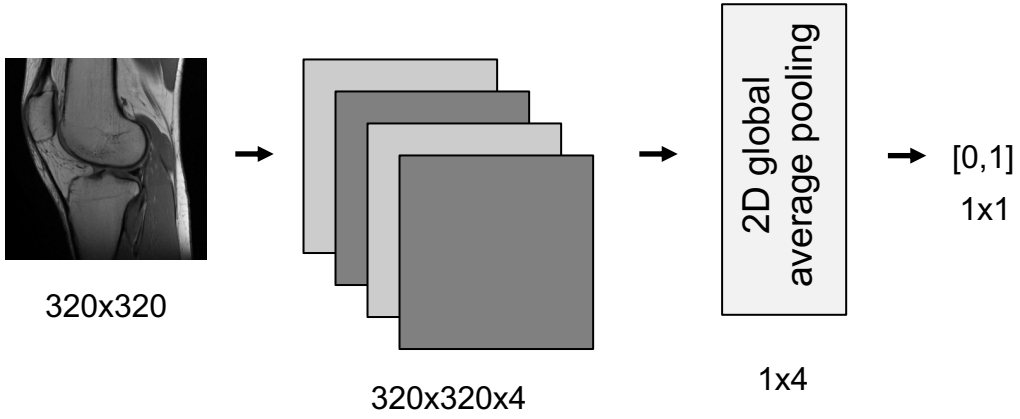
Results


4 conv layers fully connected: train=1.0
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
accuracy
0
200
400
600
800
epoch
training

Results


4 conv layers fully connected: train/val=1.0/0.61
accuracy
epoch
training
validation
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
200
400
600
800

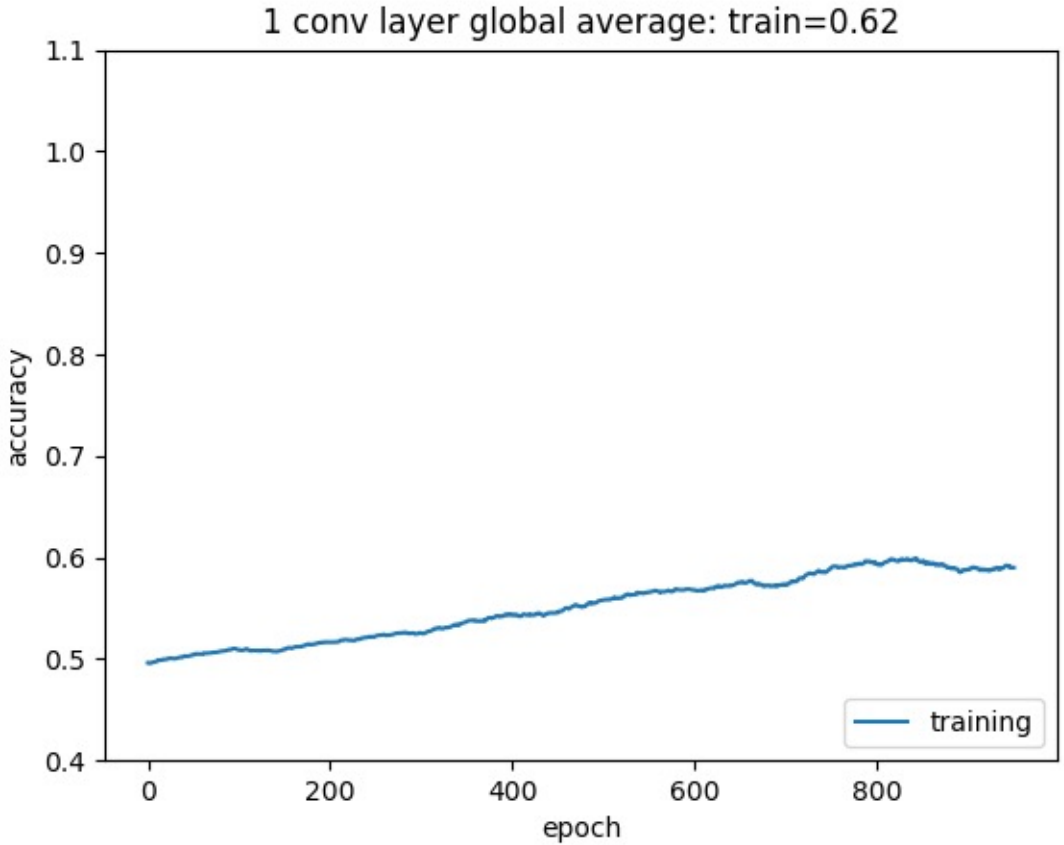
Reduce complexity of our model



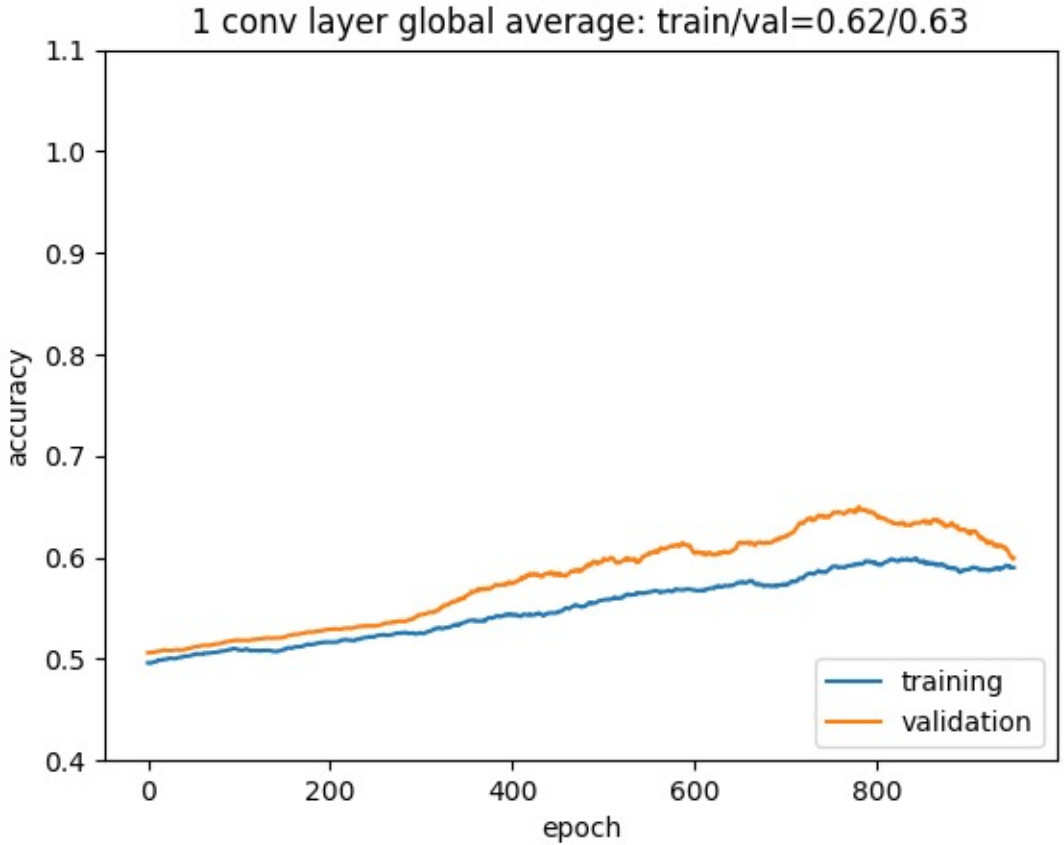
1 3x3x4 convolutional layer, Relu activation
2D Global Average Pooling
Sigmoid Output

Total number of parameters: 45

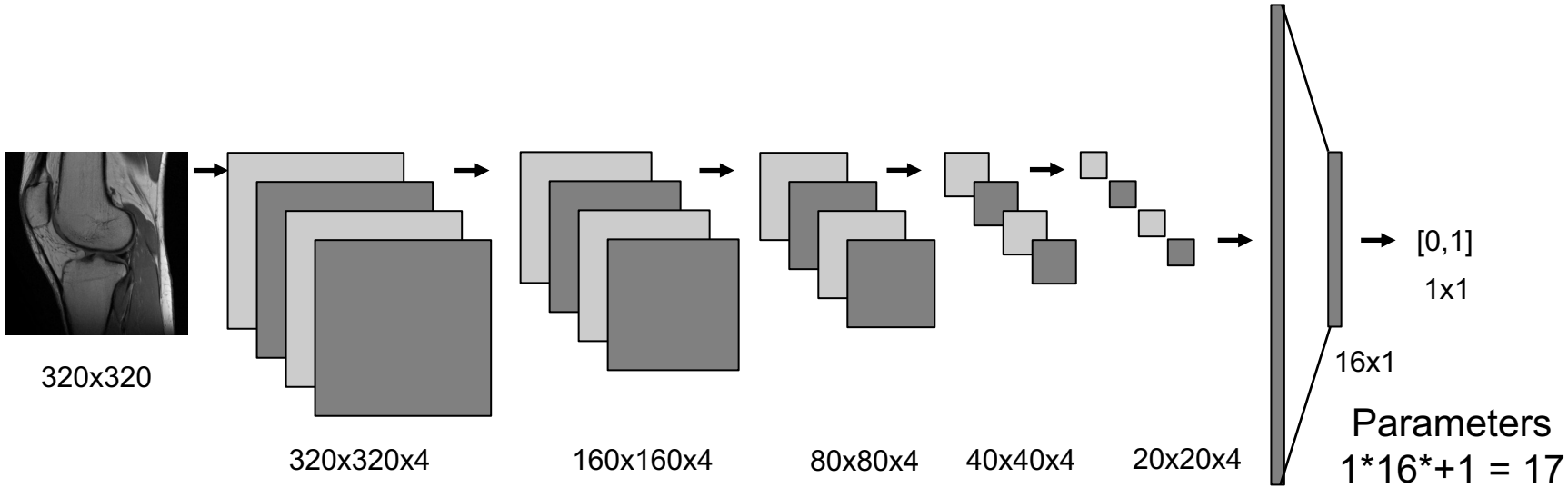
Results


1 conv layer global average: train=0.62
accuracy
epoch
training
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
200
400
600
800

Results


1 conv layer global average: train/val=0.62/0.63
accuracy
epoch
training
validation
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
200
400
600
800

A closer look at the parameter distribution of the first model



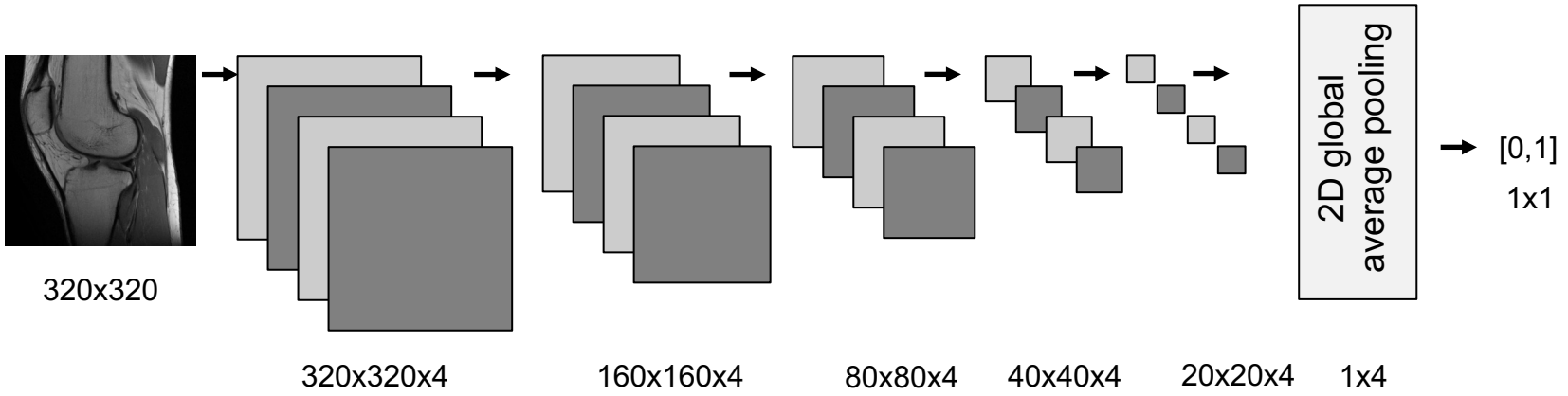
Parameters
(3*3+1)*4 = 40

Parameters
(3*3*4+1)*4 = 148

Parameters
20*20*4*16+16 = 25616

98% of parameters in fully connected layer!

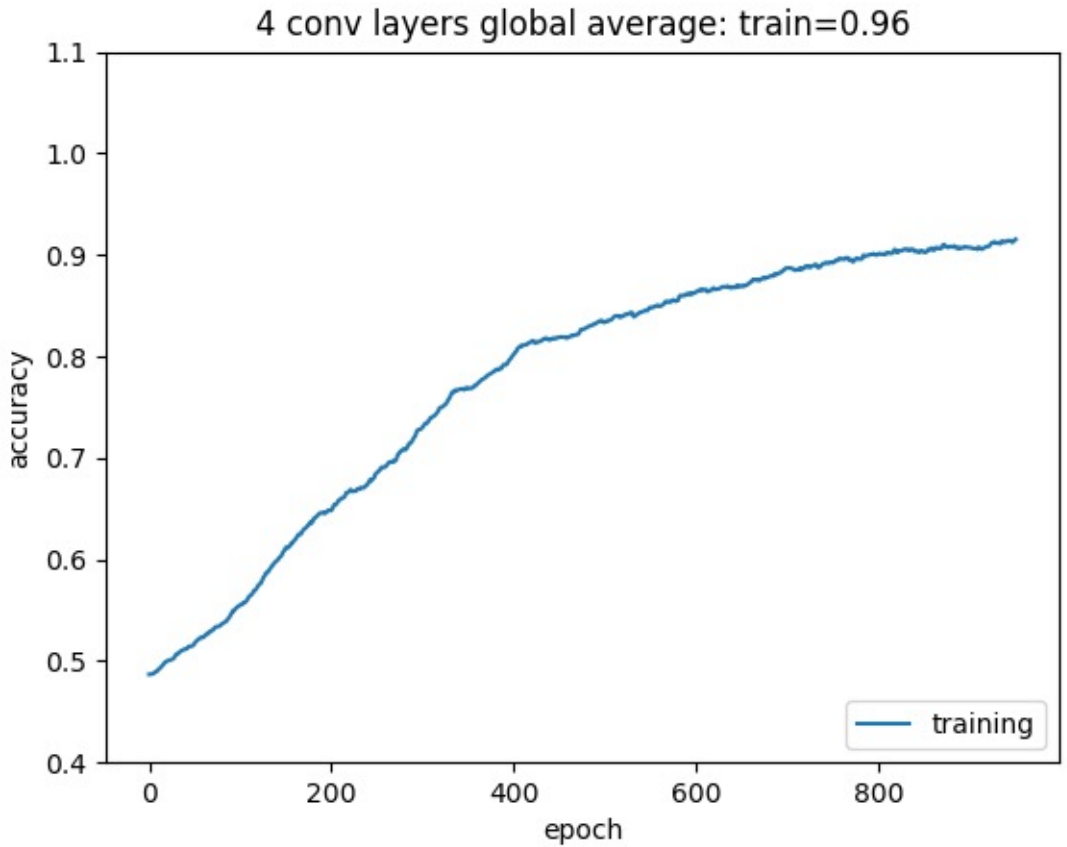
Replace FC layer with global average pooling



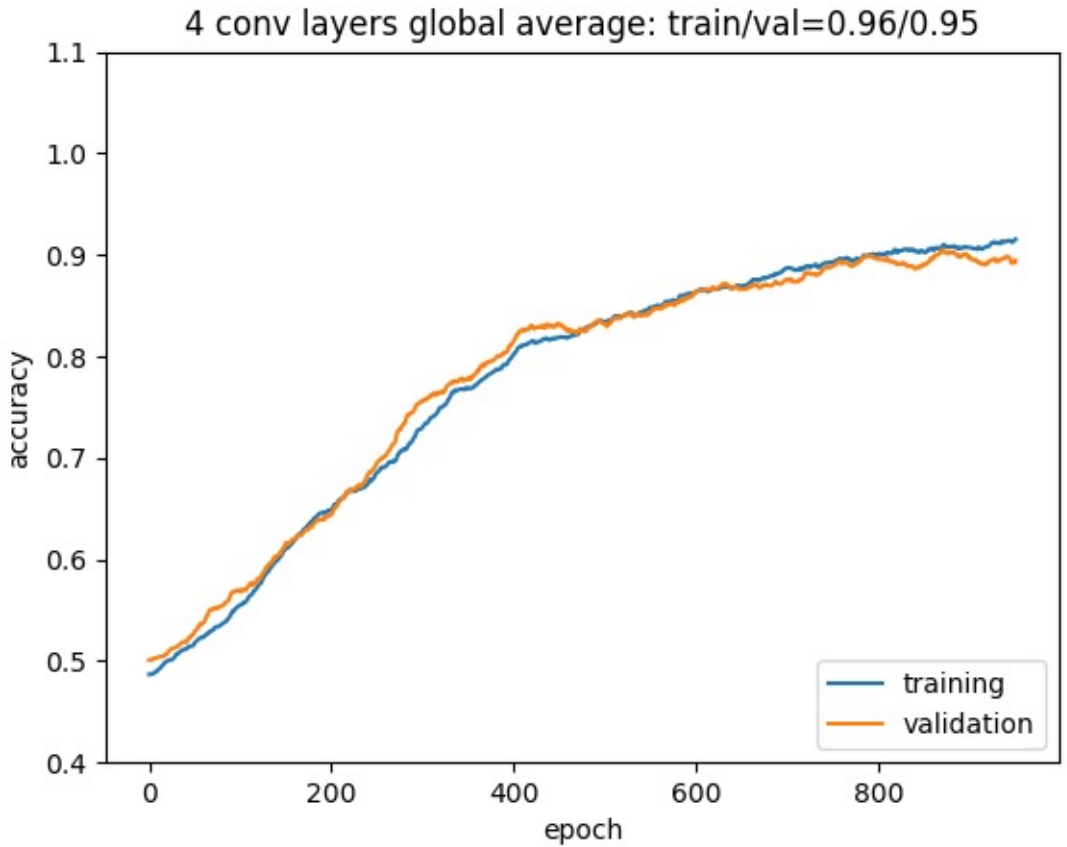
4 3x3x4 convolutional layers, Relu activation
2D MaxPooling
2D Global Average Pooling

Total number of parameters: 489

Results


4 conv layers global average: train=0.96
accuracy
epoch
training
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
200
400
600
800

Results



independent test set

- Use separate test set
- Inspect misclassified samples
- Number of conv layers, filter size,…
- Use higher model complexity combined with regularization (e.g. dropout)

Srivastava et al., JMLR (2014)

Summary

- Introduction to DL/ML/AI
- Examples for linear regression, MLPs, CNNs
- High level view on generalization
- Model complexity, overfitting vs. underfitting
- Model parameters: Convolutional vs fully connected layers