How to Jump-Start Your Deep Learning Research

Florian Knoll

Department of Radiology, CBI
Center for Advanced Imaging Innovation and Research (CAI^{2}R)
NYU School of Medicine, New York, USA

CAI^{2}R Center for Advanced Imaging Innovation and Research

NYU School of Medicine
NYU LANGONE MEDICAL CENTER

JOINT ANNUAL MEETING
ISMRM–ESMRMB
16–21 June 2018
SMRT 27th Annual Meeting 15–18 June 2018
www.smrt.org
Paris Expo Porte de Versailles
Paris, France

Declaration of Financial Interests or Relationships

Speaker Name: Florian Knoll

I have no financial interests or relationships to disclose with regard to the subject matter of this presentation.

Surfing the big AI wave

Getting started…

Outline

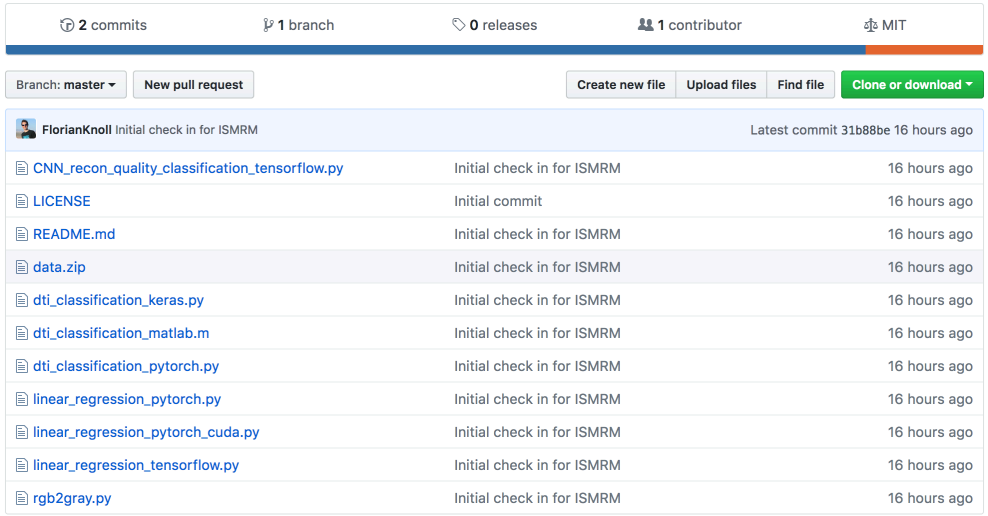
- Didactic implementation examples
  - Linear regression
  - Classification of DTI data (MLP)
  - Classification of reconstructed image quality (CNN)

```
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
```

Outline

- Didactic implementation examples
  - Linear regression
  - Classification of DTI data (MLP)
  - Classification of reconstructed image quality (CNN)

```
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
```



https://github.com/FlorianKnoll/ISMRM2018_Educational_DeepLearning

Outline

- Didactic implementation examples
  - Linear regression
  - Classification of DTI data (MLP)
  - Classification of reconstructed image quality (CNN)

```
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
```

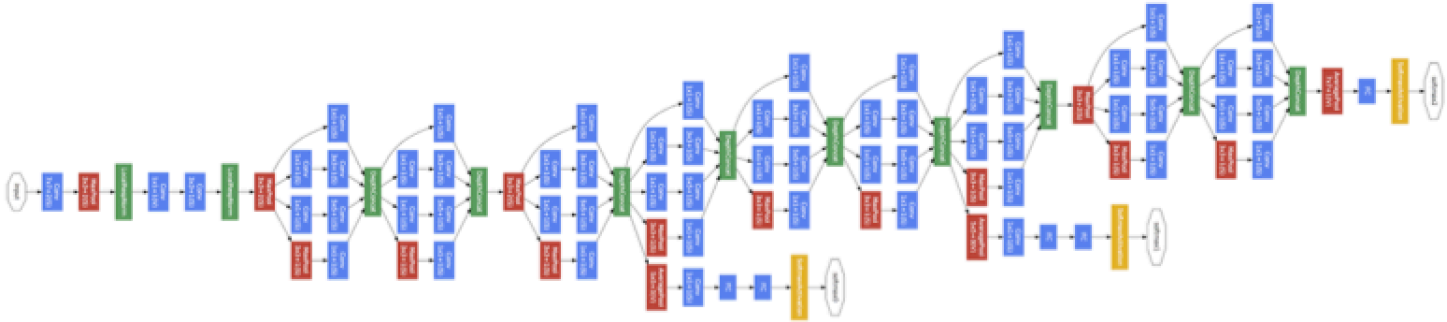
- Software frameworks
- Generalization: Model complexity, overfitting vs. underfitting
- Architectures: Convolutional vs fully connected

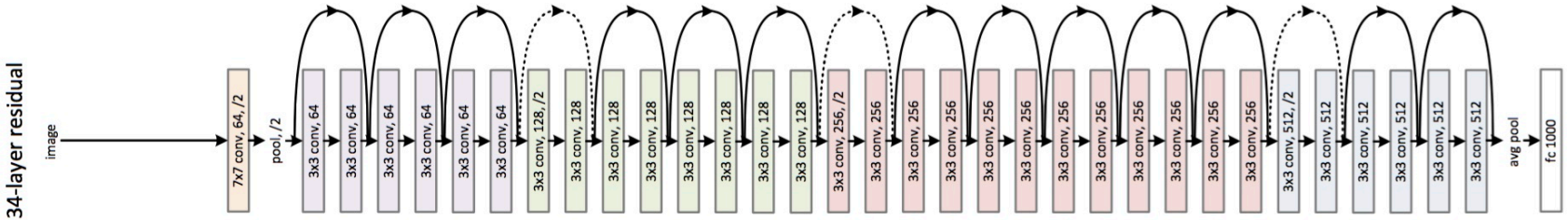
https://github.com/FlorianKnoll/ISMRM2018_Educational_DeepLearning

Software frameworks:

Why would you want to use one?

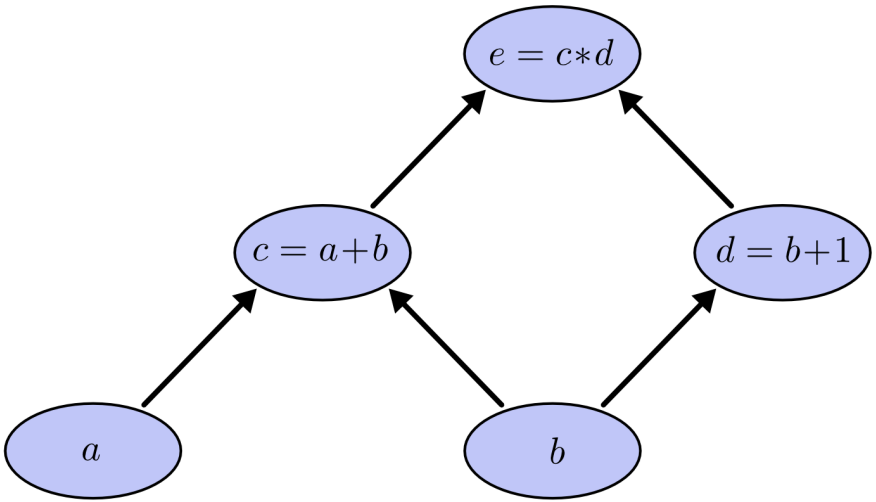
Deep learning models





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

Computational graphs



$$e = (a + b)(b + 1)$$

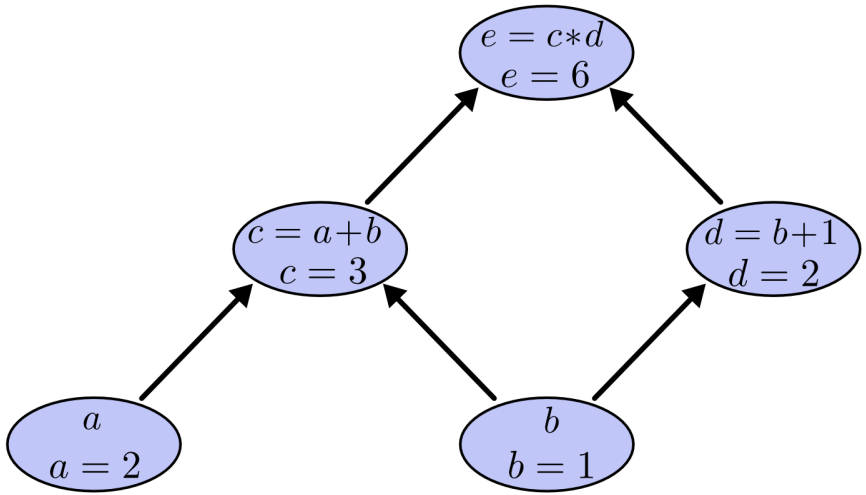
$$c = a + b$$

$$d = b + 1$$

$$e = c \cdot d$$

http://colah.github.io/posts/2015-08-Backprop/

Computational graphs

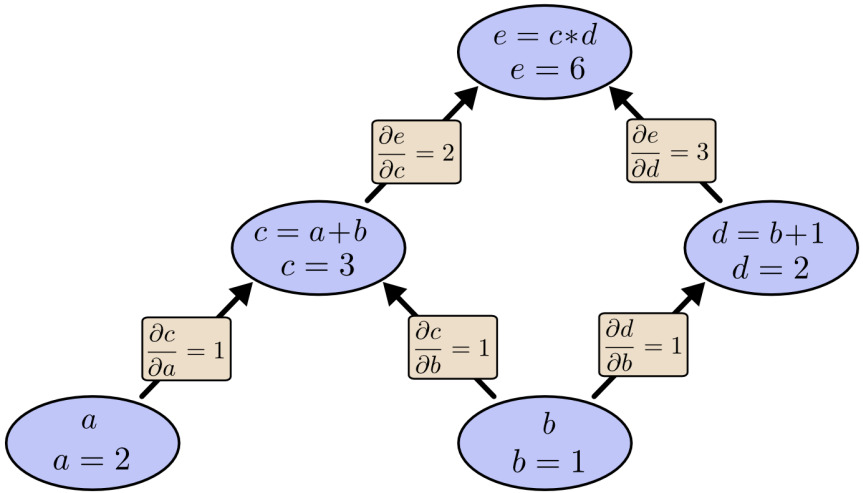


$$a = 2$$

$$b = 1$$

http://colah.github.io/posts/2015-08-Backprop/

Computational graphs

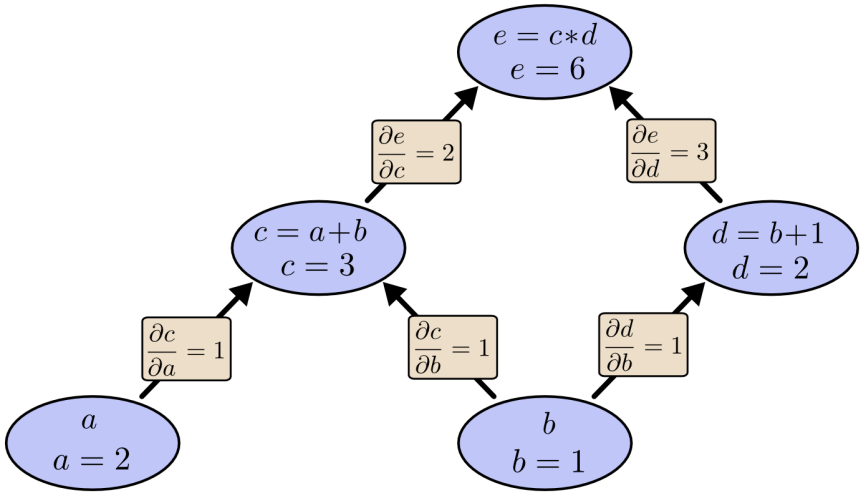


$$c = a + b \qquad \frac{\partial c}{\partial a} = b$$

$$e = c \cdot d \qquad \frac{\partial e}{\partial c} = d$$

http://colah.github.io/posts/2015-08-Backprop/

Computational graphs



$$c = a + b \qquad \frac{\partial c}{\partial a} = b$$

$$e = c \cdot d \qquad \frac{\partial e}{\partial c} = d$$

http://colah.github.io/posts/2015-08-Backprop/

Data and computation

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

SKYNET.NYUMC.ORG

```
[knollf01@skynet ~]$ bhosts
HOST_NAME          STATUS       JL/U    MAX  NJOBS    RUN  SSUSP  USUSP    RSV
skygpu01           closed          -     80      0      0      0      0      0
skygpu02           ok              -     80      0      0      0      0      0
skygpu03           closed          -     80      0      0      0      0      0
skygpu04           closed          -     80      0      0      0      0      0
skygpu05           closed          -     80      0      0      0      0      0
skygpu06           ok              -     80     20     20      0      0      0
skygpu07           ok              -     80      0      0      0      0      0
skygpu08           ok              -     80      0      0      0      0      0
skygpu09           ok              -     80      0      0      0      0      0
skygpu10           ok              -     80      0      0      0      0      0
skygpu11           closed          -    160      0      0      0      0      0
skygpu12           closed          -    160      0      0      0      0      0
skygpu13           ok              -     80      0      0      0      0      0
skygpu14           closed          -     80      0      0      0      0      0
skygpu15           closed          -     80      0      0      0      0      0
skygpu16           ok              -     40     16     16      0      0      0
skygpu17           ok              -     80      0      0      0      0      0
skynet             closed          -     40      0      0      0      0      0
skynet2            closed          -     80      0      0      0      0      0
```

Functionalities of software frameworks

1. Data structures for (large) computational graphs
2. Automatic calculation of gradients
3. GPU support
4. High level support for architecture design
5. Data handling functions (minibatches etc.)
6. Numerical optimizers

Popularity of deep learning frameworks

Library	Rank	Overall	Github	Stack Overflow	Google Results
tensorflow	1	10.87	4.25	4.37	2.24
keras	2	1.93	0.61	0.83	0.48
caffe	3	1.86	1.00	0.30	0.55
theano	4	0.76	-0.16	0.36	0.55
pytorch	5	0.48	-0.20	-0.30	0.98
sonnet	6	0.43	-0.33	-0.36	1.12
mxnet	7	0.10	0.12	-0.31	0.28
torch	8	0.01	-0.15	-0.01	0.17
cntk	9	-0.02	0.10	-0.28	0.17
dlib	10	-0.60	-0.40	-0.22	0.02

September 2017: https://github.com/thedataincubator/data-science-blogs

First generation deep learning frameworks

theano

Université de Montréal

Caffe

Berkeley
UNIVERSITY OF CALIFORNIA

Second generation deep learning frameworks

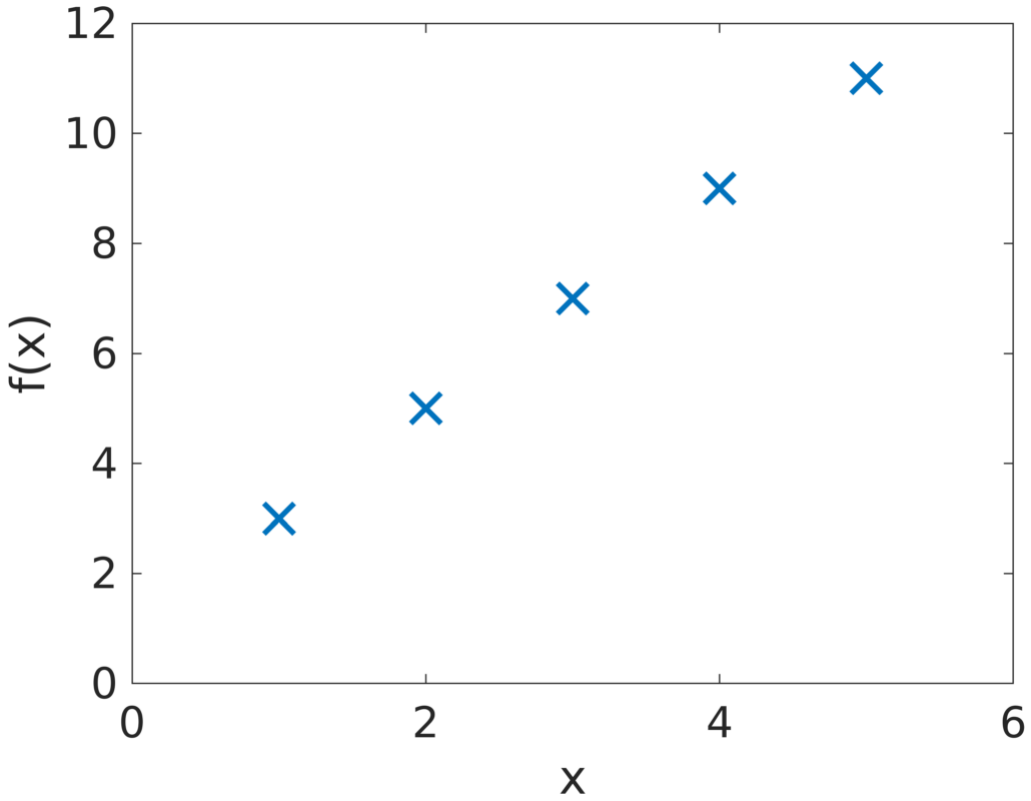
Second generation deep learning frameworks

PYTORCH

Example 1: Linear Regression

A low level view of PyTorch and TensorFlow

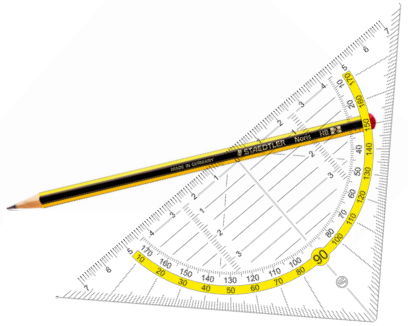
Data



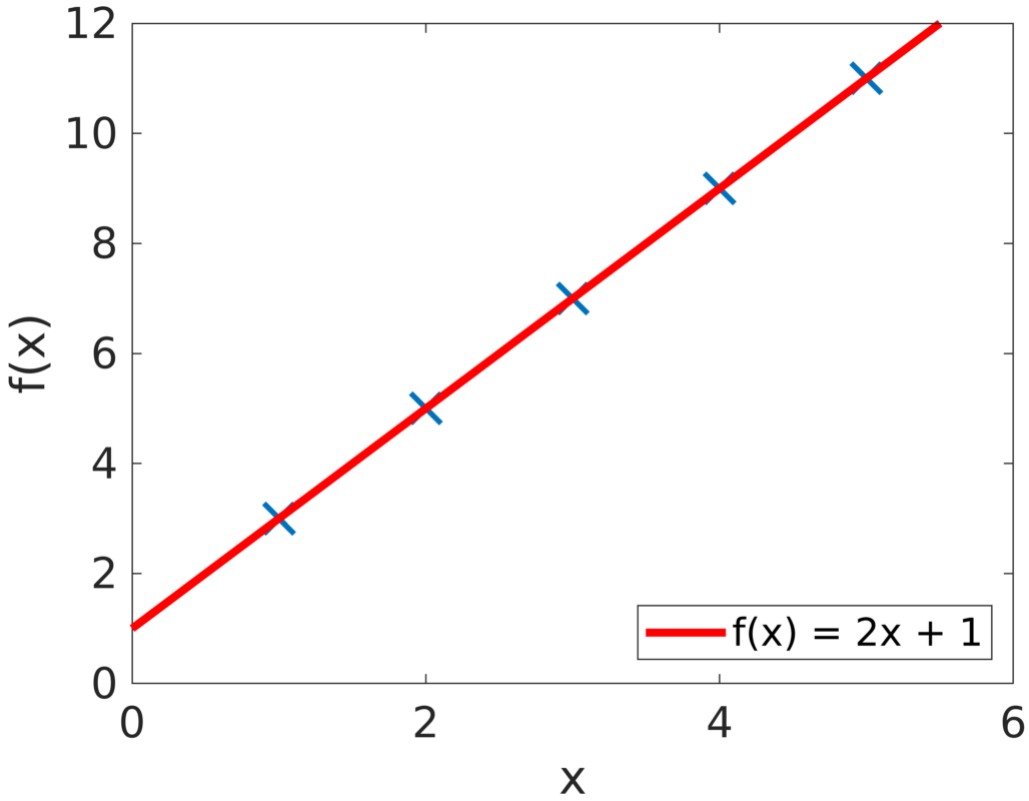
$$x = [1, 2, 3, 4, 5]$$

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

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



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

https://github.com/FlorianKnoll/ISMRM2018_Educational_DeepLearning

Training data and model parameters

```
#%% 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
```

PYTORCH

Automatic differentiation

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

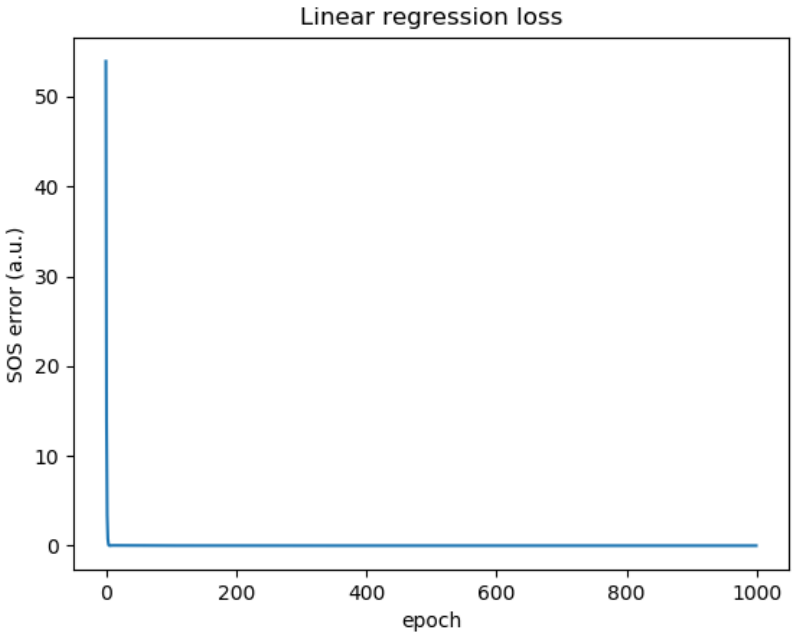
Loop over samples, update parameters

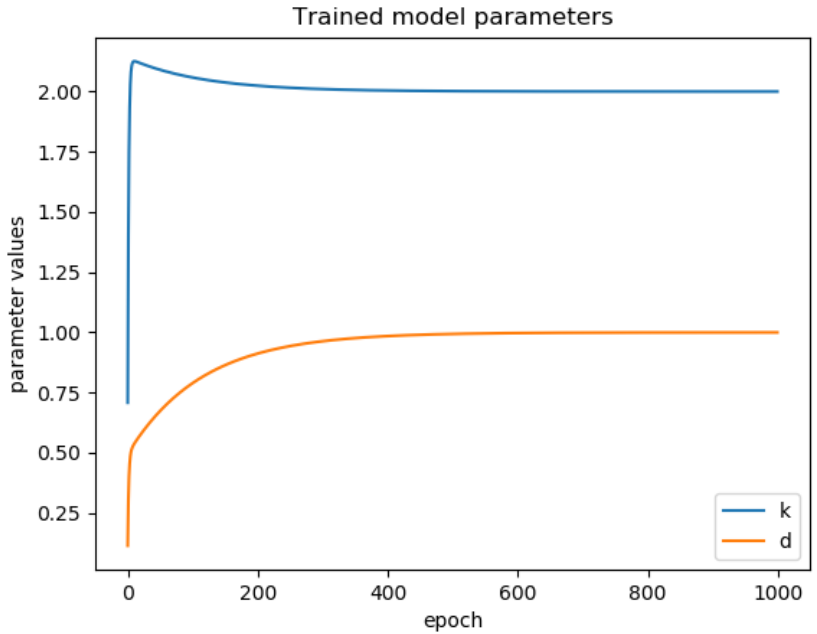
```
for ii in range(training_epochs):
    for x_train_batch, y_train_batch in zip(x_train,y_train):
        loss = loss_fcn(x_train_batch, y_train_batch)
        loss_ii[ii] = loss.item()
        k_ii[ii] = k.data
        d_ii[ii] = d.data
        loss.backward()
        k.data = k.data - lr * k.grad.data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

PYTORCH

The computational graph is dynamically created during the iterations

Linear regression with gradient descent: PyTorch

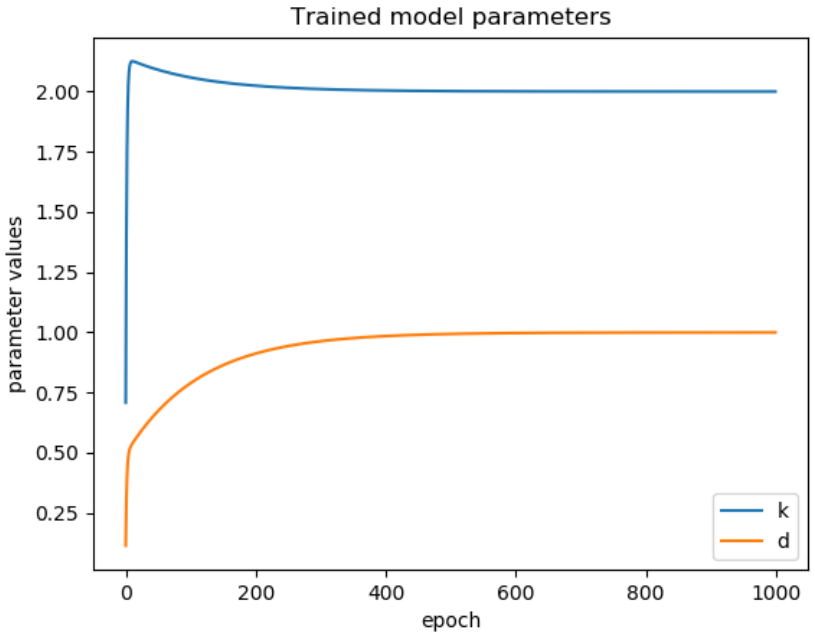




Linear regression in PyTorch

```
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
```

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



Linear regression in PyTorch on the CPU

```
#%% 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
```

Linear regression in PyTorch on the GPU

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

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

d = torch.Tensor([-0.1]).float().to(torch.device("cuda"))
d.requires_grad=True
```

Linear regression in TensorFlow

```
#%% training data
x_train = [1,2,3,4,5]
y_train = [3,5,7,9,11]

#%% Model parameters
k = tf.Variable([0.1], dtype=tf.float32)
d = tf.Variable([-0.1], dtype=tf.float32)

# Model input and output
x = tf.placeholder(tf.float32)
linear_model = (k * x + d)
y = tf.placeholder(tf.float32)

# sum of squares loss
loss = tf.reduce_sum(tf.square(linear_model - y))

#%% optimizer
lr = 0.005
optimizer = tf.train.GradientDescentOptimizer(lr).minimize(loss)

#%% train
training_epochs = 1000
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
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):
  [k_ii_temp, d_ii_temp, loss_ii[ii]] = sess.run([k, d, loss], {x:x_train, y:y_train})
  k_ii[ii] = k_ii_temp.item()
  d_ii[ii] = d_ii_temp.item()
  print('epoch: {}, k={:.3}, b={:.3}, loss={:.3}'.format(ii+1,k_ii[ii], d_ii[ii], loss_ii[ii]))
  sess.run(optimizer, {x:x_train, y:y_train})

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

Static computational graph

Linear regression in TensorFlow

```
#%% training data
x_train = [1,2,3,4,5]
y_train = [3,5,7,9,11]
```

```
#%% Model parameters
k = tf.Variable([0.1], dtype=tf.float32)
d = tf.Variable([-0.1], dtype=tf.float32)

# Model input and output
x = tf.placeholder(tf.float32)
linear_model = (k * x + d)
y = tf.placeholder(tf.float32)

# sum of squares loss
loss = tf.reduce_sum(tf.square(linear_model - y))
```

Linear regression in TensorFlow

```
#%% training data
x_train = [1,2,3,4,5]
y_train = [3,5,7,9,11]
```

```
#%% Model parameters
k = tf.Variable([0.1], dtype=tf.float32)
d = tf.Variable([-0.1], dtype=tf.float32)

# Model input and output
x = tf.placeholder(tf.float32)
linear_model = (k * x + d)
y = tf.placeholder(tf.float32)

# sum of squares loss
loss = tf.reduce_sum(tf.square(linear_model - y))
```

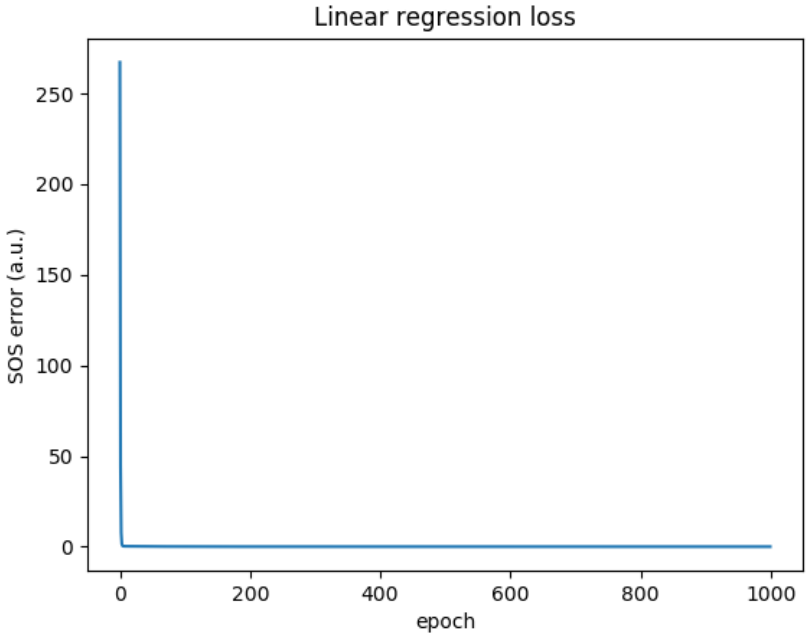
```
#%% optimizer
lr = 0.005
optimizer = tf.train.GradientDescentOptimizer(lr).minimize(loss)
```

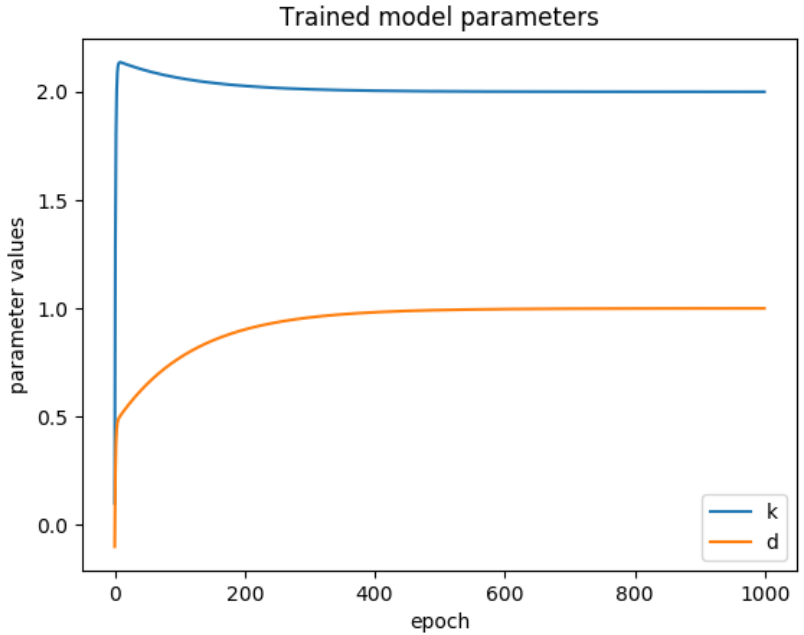
Linear regression in TensorFlow

```
#%% training data
x_train = [1,2,3,4,5]
y_train = [3,5,7,9,11]

#%% Model parameters
k = tf.Variable([0.1], dtype=tf.float32)
d = tf.Variable([-0.1], dtype=tf.float32)

# Model input and output
x = tf.placeholder(tf.float32)
linear_model = (k * x + d)
y = tf.placeholder(tf.float32)

# sum of squares loss
loss = tf.reduce_sum(tf.square(linear_model - y))

#%% optimizer
lr = 0.005
optimizer = tf.train.GradientDescentOptimizer(lr).minimize(loss)

#%% train
training_epochs = 1000
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
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):
  [k_ii_temp, d_ii_temp, loss_ii[ii]] = sess.run([k, d, loss], {x:x_train, y:y_train})
  k_ii[ii] = k_ii_temp.item()
  d_ii[ii] = d_ii_temp.item()
  print('epoch: {}, k={:.3}, b={:.3}, loss={:.3}'.format(ii+1,k_ii[ii], d_ii[ii], loss_ii[ii]))
  sess.run(optimizer, {x:x_train, y:y_train})

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

- Computational graph is fixed at this stage
- A session is started
- Data is now fed to the graph

Linear regression in TensorFlow



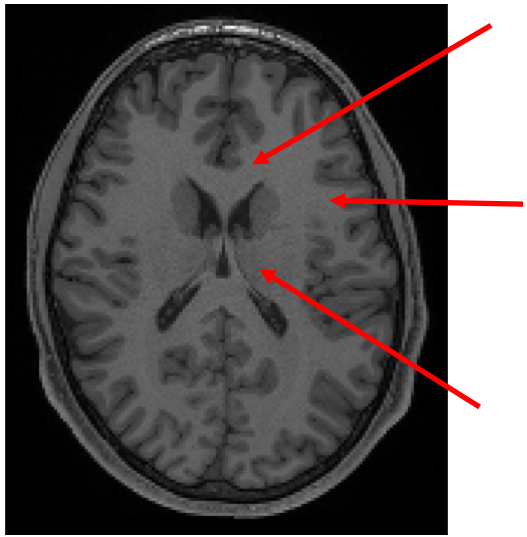


Example 2: Classification of brain tissue from DTI data

High level neural network modules

Classification of brain tissue from HCP DTI data

T1w

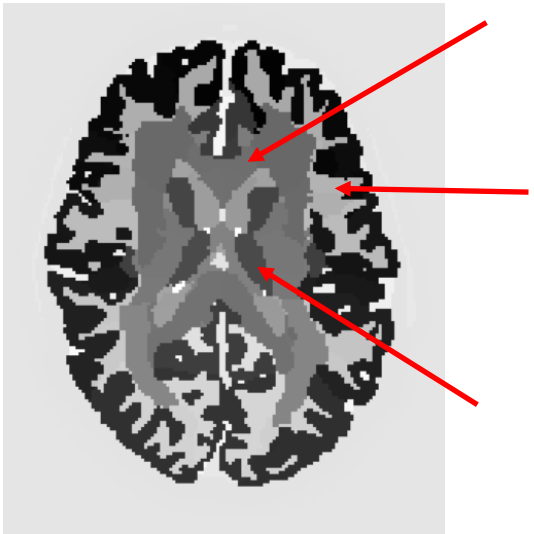


Genu of corpus callusum
(highly aligned WM)

Subcortical WM

Thalamus (GM)

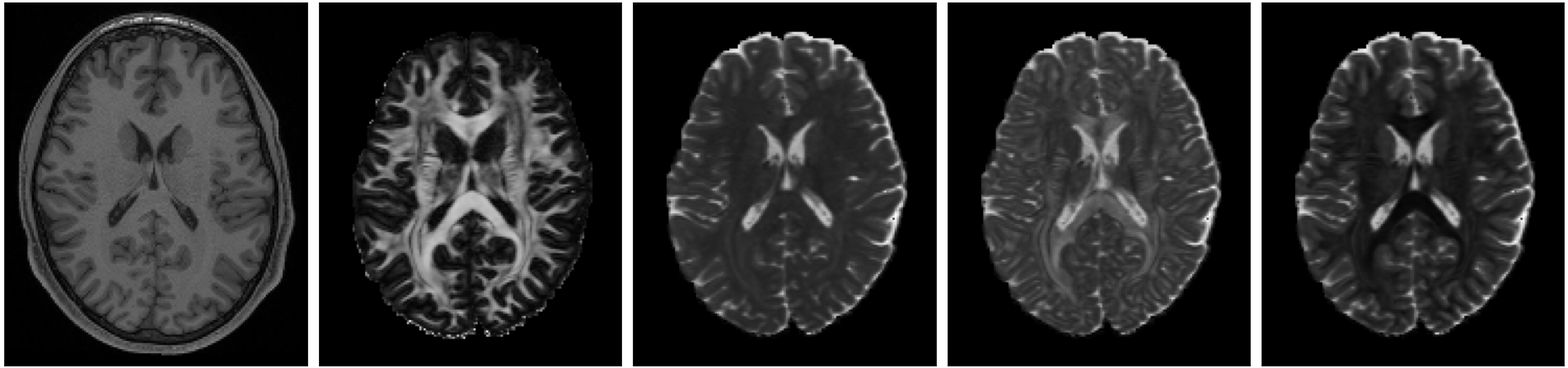
Segmentation



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 $(\lambda_2 + \lambda_3)/2$



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!

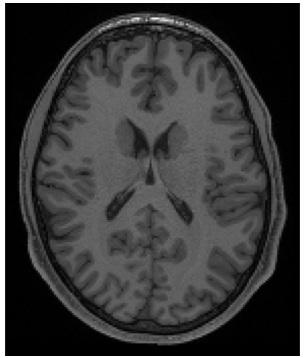
Training, validation and test set

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

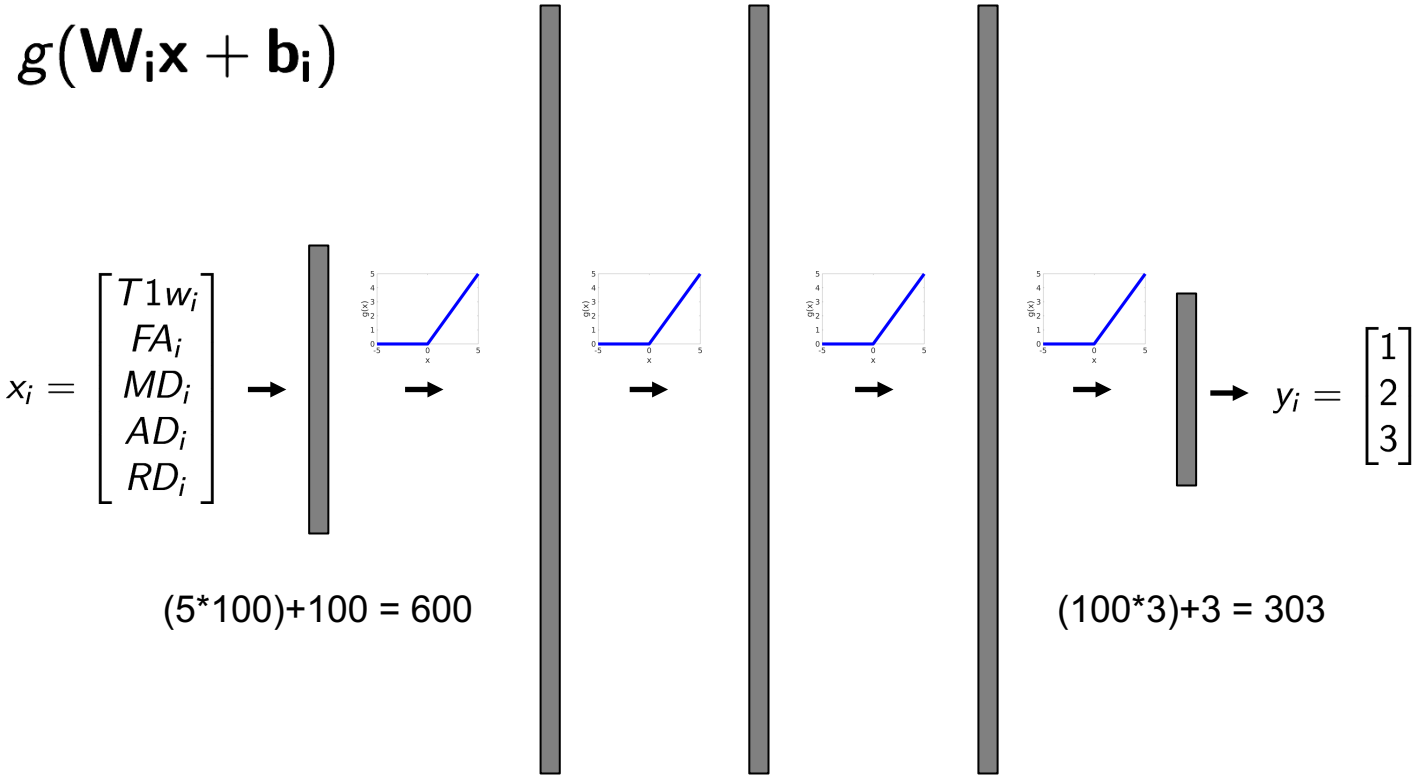
Multi layer perceptron

- 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

Let's build our first neural network (for real!)

→ Neural networks toolbox

PYTORCH → Neural networks package

→ Keras

Let's build our first neural network in Matlab

```
%% NN with 3 HIDDEN LAYER NEURONS
nElements = 100;
nLayers = 3;
inputLayer=imageInputLayer([nFeatures,1,1]);
f1=fullyConnectedLayer(nElements);
f2=fullyConnectedLayer(nElements);
f3=fullyConnectedLayer(nElements);
f4=fullyConnectedLayer(nClasses);
s1=softmaxLayer();
outputLayer=classificationLayer();

architecture = [inputLayer; f1; f2; f3; f4; s1; outputLayer];
disp(architecture);
```

```
epochs = 250;
miniBatchSize = 1024;
InitialLearnRate = 0.001;

% Training options: Note that we set the validation patience stopping
% criterion to the number of epochs. This is a stupid thing to do, but we
% want force the training to go to the defined number of epochs so that it
% is consistent with Tensorflow and Pytorch
options = trainingOptions('adam','MaxEpochs',epochs,'InitialLearnRate',InitialLearnRate,...
    'MiniBatchSize',miniBatchSize,'ExecutionEnvironment','cpu','Plots','training-progress',...
    'ValidationData',{x_val,y_val},'ValidationPatience',epochs);
```

```
%% Train
tic
[net,op] = trainNetwork(x_train,y_train,architecture,options);
toc
```

Kingma and Ba, ICLR 2015

Let’s build our first neural network in PyTorch

```
#%% 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

Model architecture

```
#%% 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)
```

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

Let’s train our network

```
#%% 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)
```

PYTORCH

Let’s build our first neural network in TensorFlow

```
#%% Define model
nElements = 100
nLayers = 3
model = Sequential()
model.add(Dense(nElements, input_dim=nFeatures, activation='relu',name="input_layer"))
model.add(Dense(nElements, activation='relu',name="hidden_layer_01"))
model.add(Dense(nElements, activation='relu',name="hidden_layer_02"))
model.add(Dense(nElements, activation='relu',name="hidden_layer_03"))
model.add(Dense(nClasses, activation='softmax', name="output_softmax"))
model_name = 'dti_FC_{}layers_{}elements'.format(nLayers,nElements)

# Compile model
# default adam learning rate: 0.001
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
#%% Plot model information
model.summary()
with open('./models/{}.txt'.format(model_name),'w') as fh:
    model.summary(print_fn=lambda x: fh.write(x + '\n'))

plot_model(model, to_file='./models/{}.png'.format(model_name))
```

```
#%% Set up tensorboard
tensorboard_graphdir = './graph_dti/{0}_{1}'.format(time.strftime('%Y-%m-%d_%H-%M-%S'),model_name)
os.makedirs(tensorboard_graphdir)
tbCallBack = keras.callbacks.TensorBoard(log_dir=tensorboard_graphdir, histogram_freq=0, write_graph=True, write_images=True)
```

```
#%% Train model
training_epochs = 250
batch_size=1024

print("Train",model_name,"dti classification")
print("Training data points: {}".format(np.size(y_train,0)))
print("Validation data points: {}".format(np.size(y_val,0)))
print("Test data points: {}".format(np.size(y_test,0)))

training_history=model.fit(x_train, y_train, batch_size=batch_size,  validation_data=(x_val,y_val), epochs=training_epochs, verbose=1,callbacks=[tbCallBack])
```

Model architecture

```
#%% Define model
nElements = 100
nLayers = 3
model = Sequential()
model.add(Dense(nElements, input_dim=nFeatures, activation='relu',name="input_layer"))
model.add(Dense(nElements, activation='relu',name="hidden_layer_01"))
model.add(Dense(nElements, activation='relu',name="hidden_layer_02"))
model.add(Dense(nElements, activation='relu',name="hidden_layer_03"))
model.add(Dense(nClasses, activation='softmax', name="output_softmax"))
model_name = 'dti_FC_{}layers_{}elements'.format(nLayers,nElements)
```

```
# Compile model
# default adam learning rate: 0.001
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Kingma and Ba, ICLR 2015

Let’s train our network

```
#%% Define model
nElements = 100
nLayers = 3
model = Sequential()
model.add(Dense(nElements, input_dim=nFeatures, activation='relu',name="input_layer"))
model.add(Dense(nElements, activation='relu',name="hidden_layer_01"))
model.add(Dense(nElements, activation='relu',name="hidden_layer_02"))
model.add(Dense(nElements, activation='relu',name="hidden_layer_03"))
model.add(Dense(nClasses, activation='softmax', name="output_softmax"))
model_name = 'dti_FC_{}layers_{}elements'.format(nLayers,nElements)

# Compile model
# default adam learning rate: 0.001
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
#%% Plot model information
model.summary()
with open('./models/{}.txt'.format(model_name),'w') as fh:
    model.summary(print_fn=lambda x: fh.write(x + '\n'))

plot_model(model, to_file='./models/{}.png'.format(model_name))
```

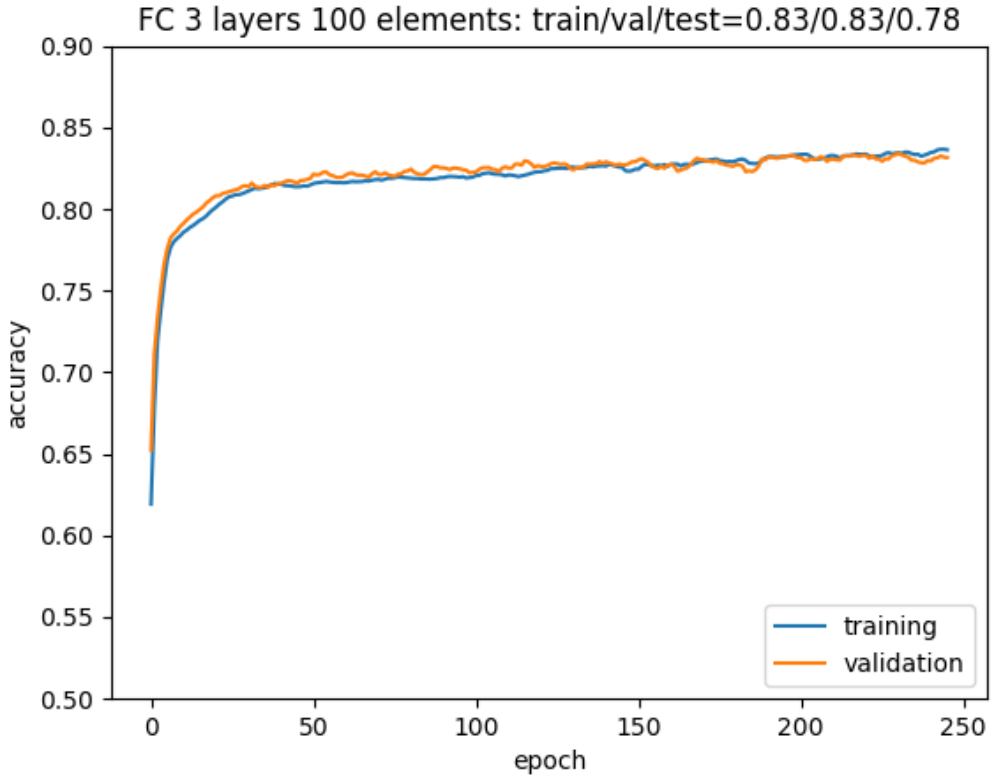
```
#%% Set up tensorboard
tensorboard_graphdir = './graph_dti/{0}_{1}'.format(time.strftime('%Y-%m-%d_%H-%M-%S'),model_name)
os.makedirs(tensorboard_graphdir)
tbCallBack = keras.callbacks.TensorBoard(log_dir=tensorboard_graphdir, histogram_freq=0, write_graph=True, write_images=True)
```

```
#%% Train model
training_epochs = 250
batch_size=1024

print("Train",model_name,"dti classification")
print("Training data points: {}".format(np.size(y_train,0)))
print("Validation data points: {}".format(np.size(y_val,0)))
print("Test data points: {}".format(np.size(y_test,0)))

training_history=model.fit(x_train, y_train, batch_size=batch_size,  validation_data=(x_val,y_val), epochs=training_epochs, verbose=1,callbacks=[tbCallBack])
```

TensorFlow

Keras

Results



- 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 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

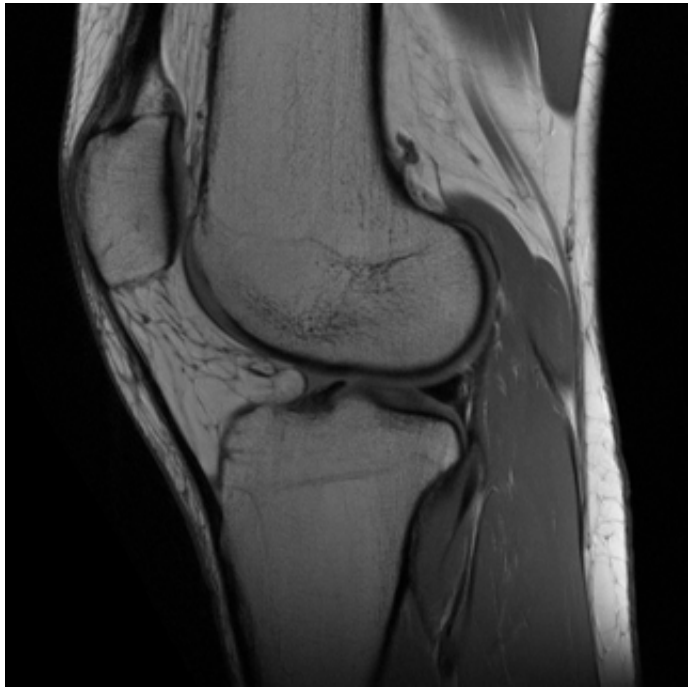
Fully sampled reference

PI-CS (TGV) R=4

Knoll, MRM (2011)

Fully sampled vs 4 times PI-CS accelerated

Fully sampled reference



PI-CS (TGV) R=4

Knoll, MRM (2011)

#1091: TH 11:39, Plenary hall

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 the audience :-)

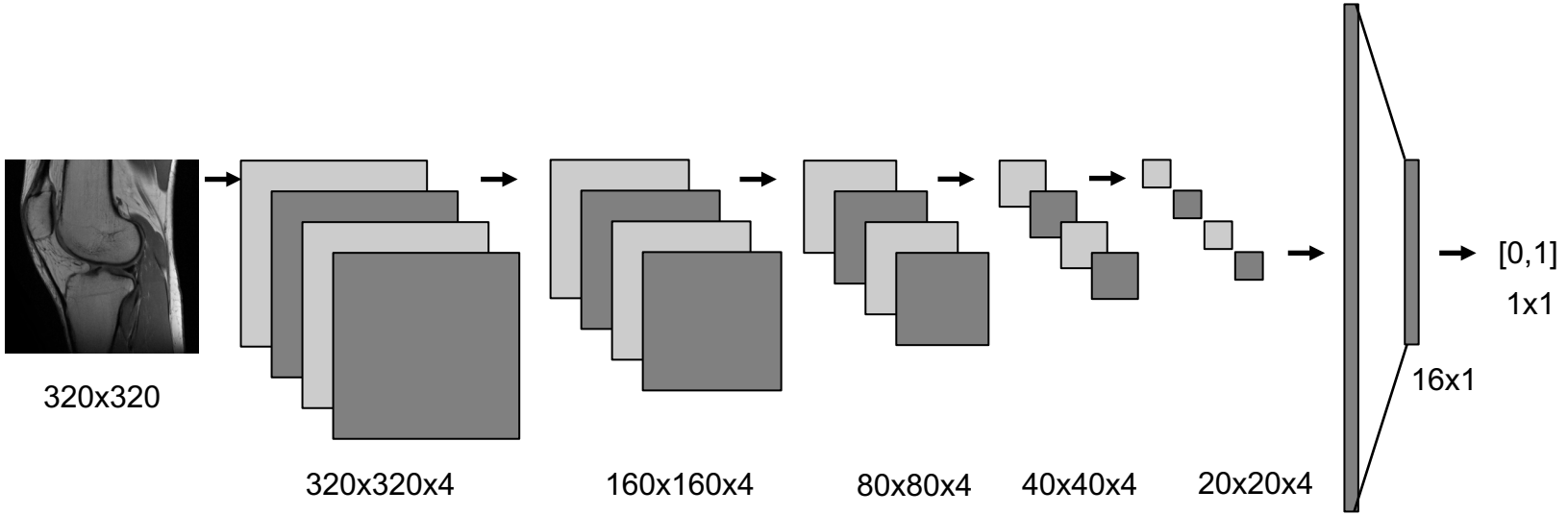
$$\{(x_1, y_1), \ldots (x_N, y_N)\}$$

$$N = 520$$

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

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

Our first convolutional neural network (CNN)



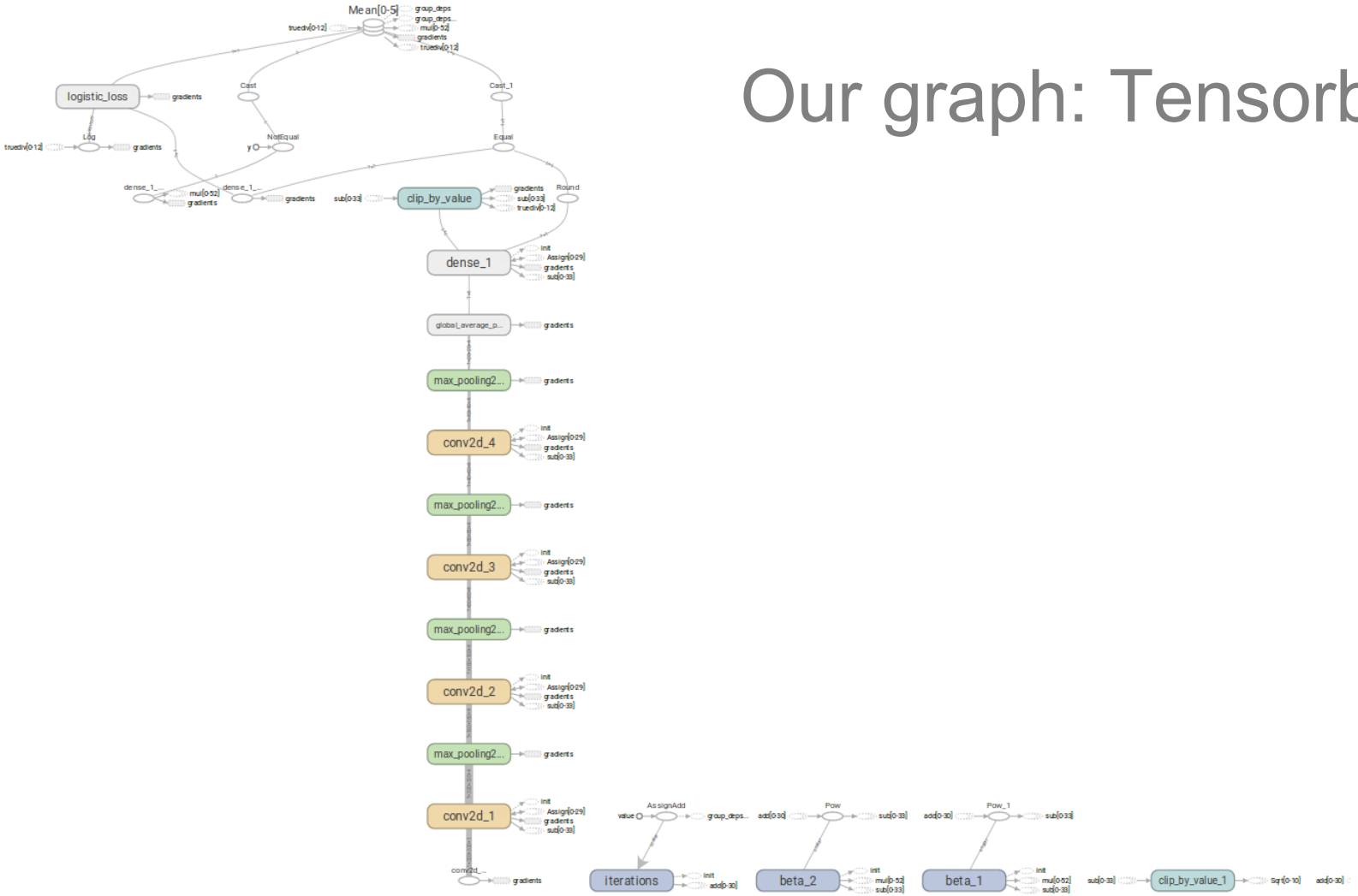
4 3x3x4 convolutional layers, Relu activation
2D MaxPooling
1 fully connected layer with 16 elements
Sigmoid Output

Total number of parameters: 26117

Our first convolutional neural network (CNN)

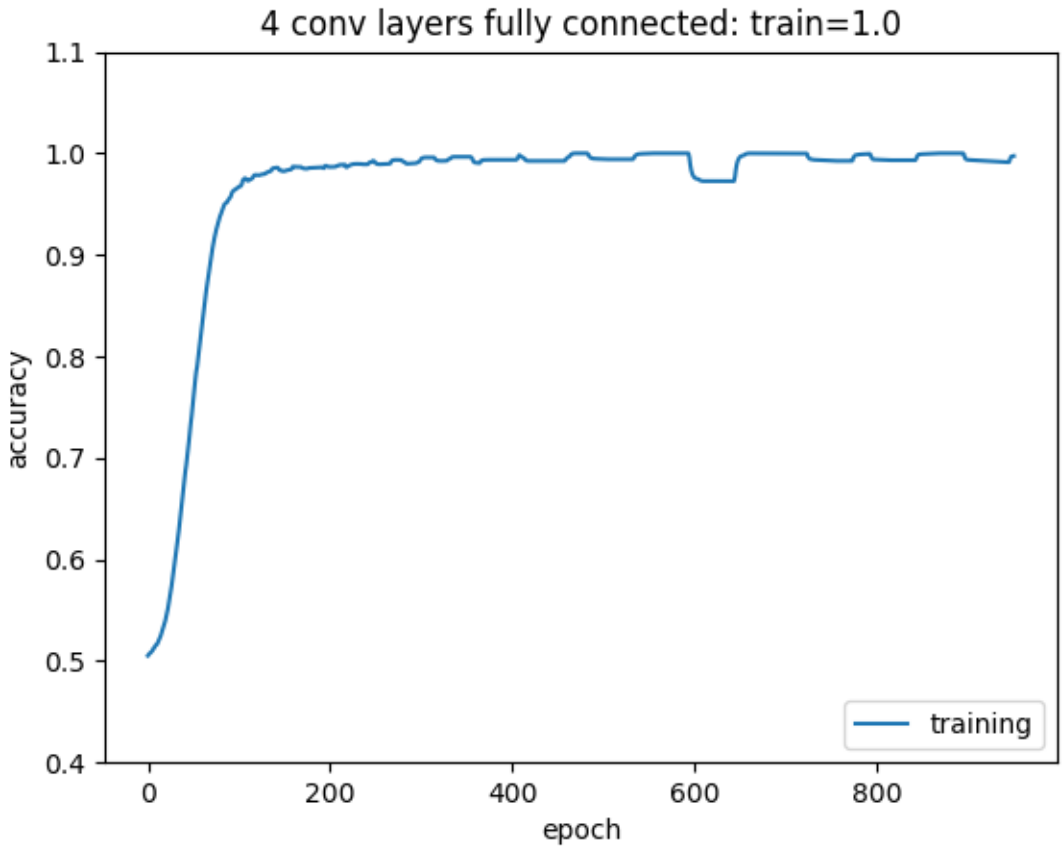
```
#%% Define model architecture
model = Sequential()
model.add(Convolution2D(4, (3, 3), activation='relu', padding='same', input_shape=(nR,nC,1)))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Convolution2D(4, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Convolution2D(4, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Convolution2D(4, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))

# Global Average Pooling
#model_name = 'CNN1layers_global_avg';
#model_name = 'CNN4layers_global_avg';
#model.add(GlobalAveragePooling2D())

# Fully connected and dense layer
model_name = 'CNN4layers_FC';
model.add(Flatten())
model.add(Dense(16, activation='relu'))

# Output layer
model.add(Dense(1, activation='sigmoid'))
```

Our graph: Tensorboard

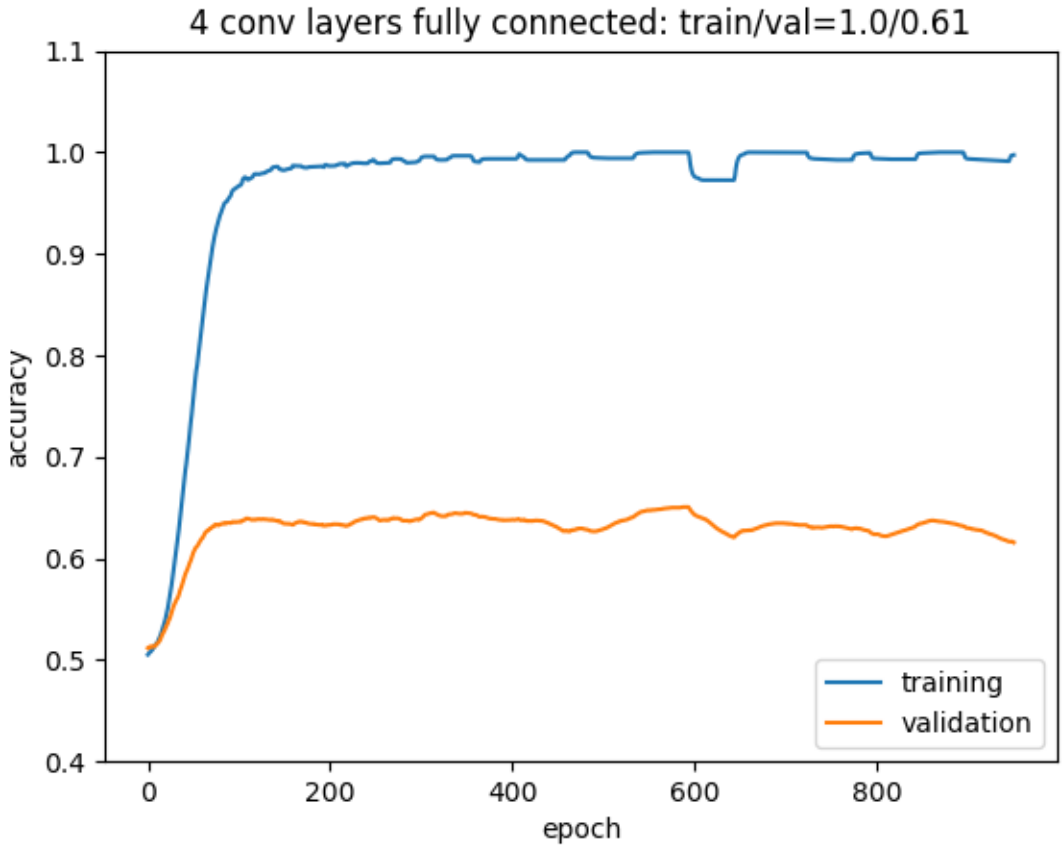


TensorFlow

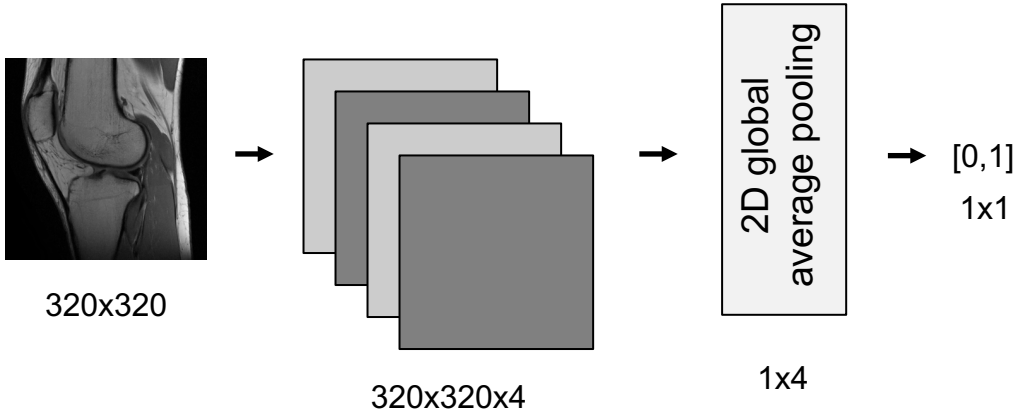
Results


4 conv layers fully connected: train=1.0
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


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

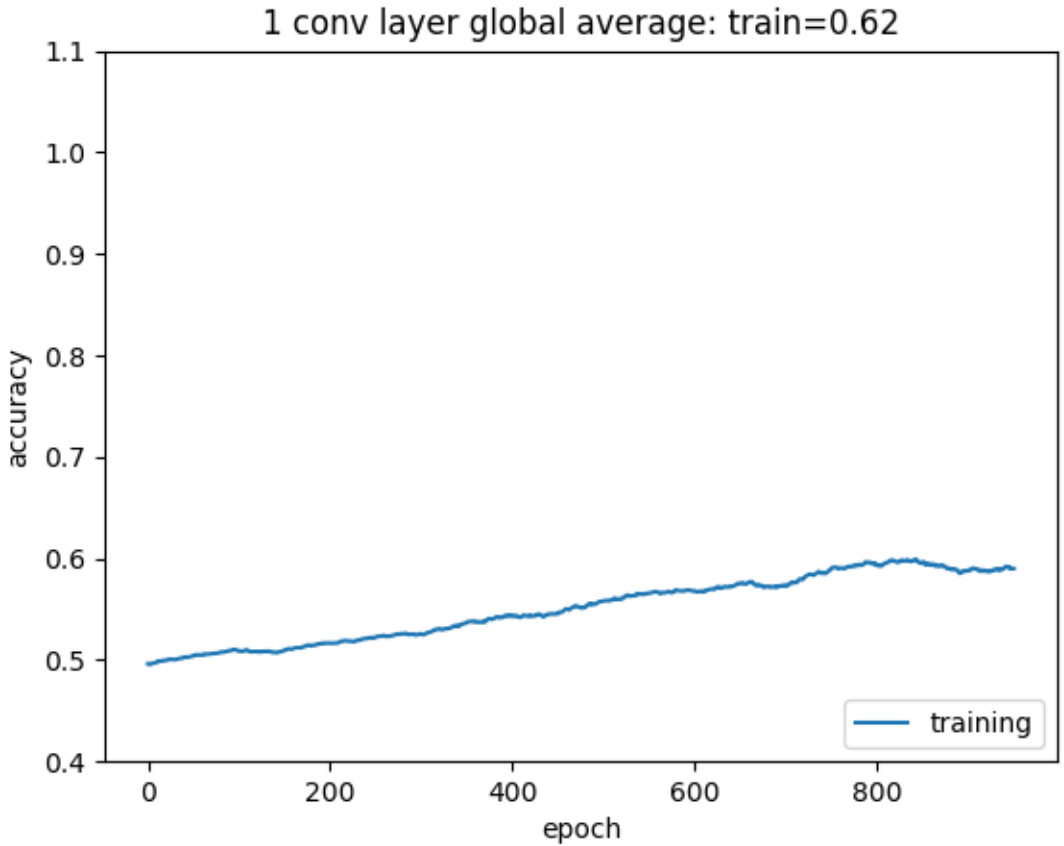
Reduce complexity of our model



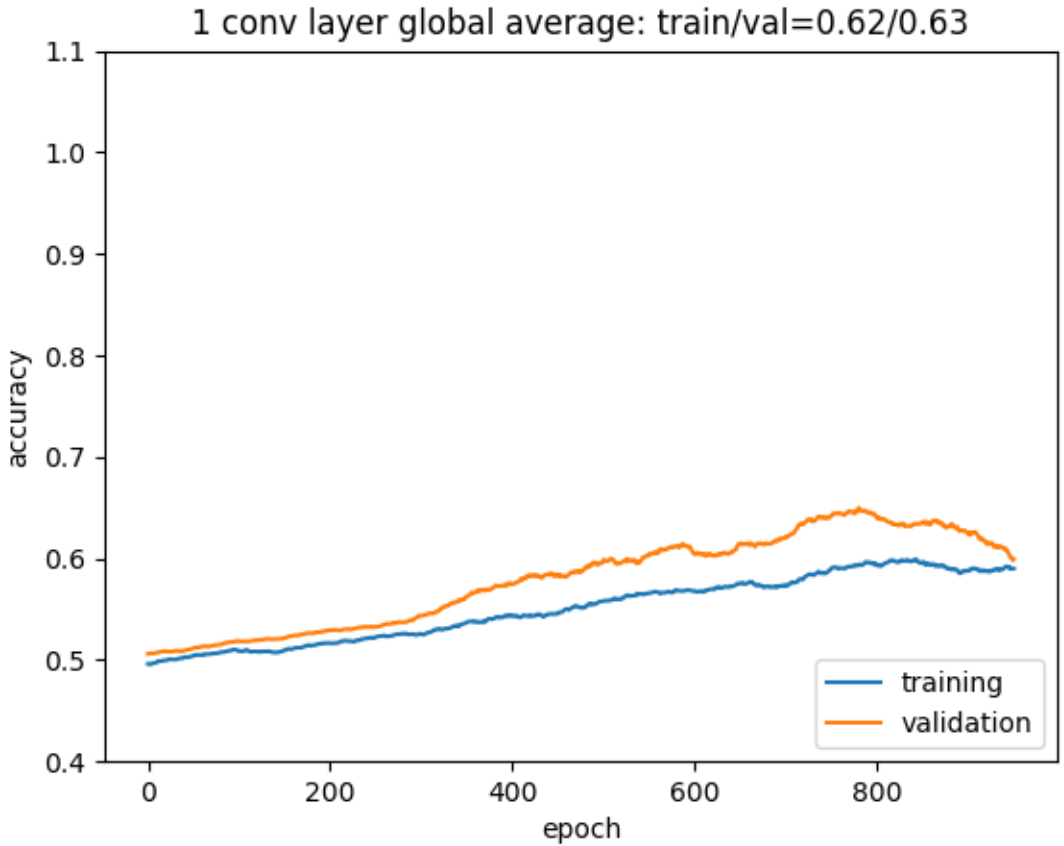
1 3x3x4 convolutional layer, Relu activation
2D Global Average Pooling
Sigmoid Output

Total number of parameters: 45

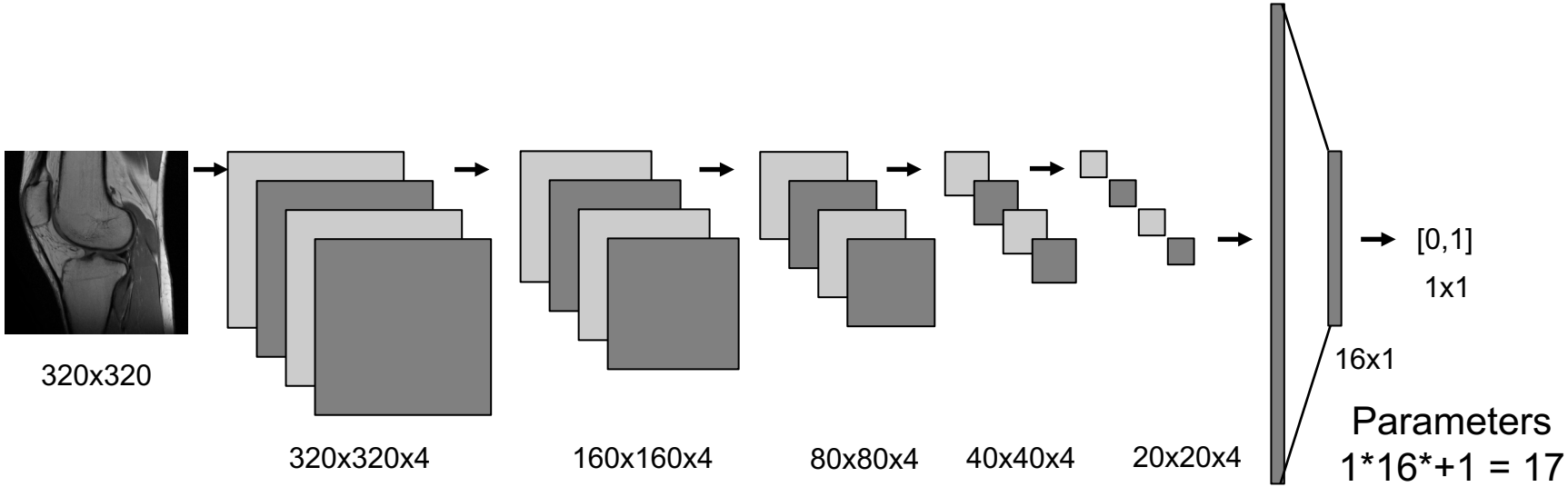
Results


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



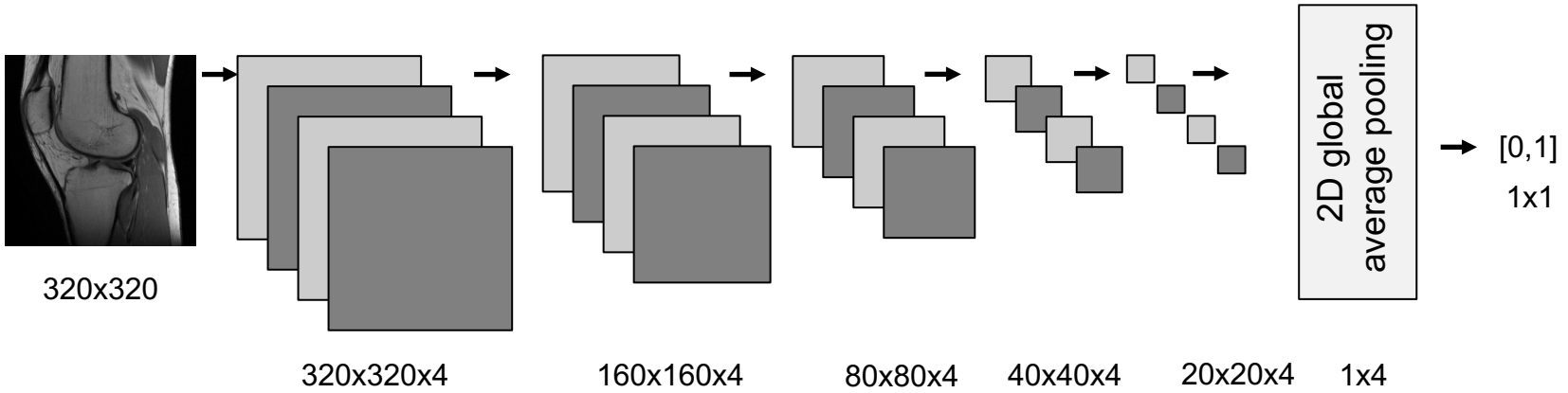
Parameters
(3*3+1)*4 = 40

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

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

98% of parameters in fully connected layer!

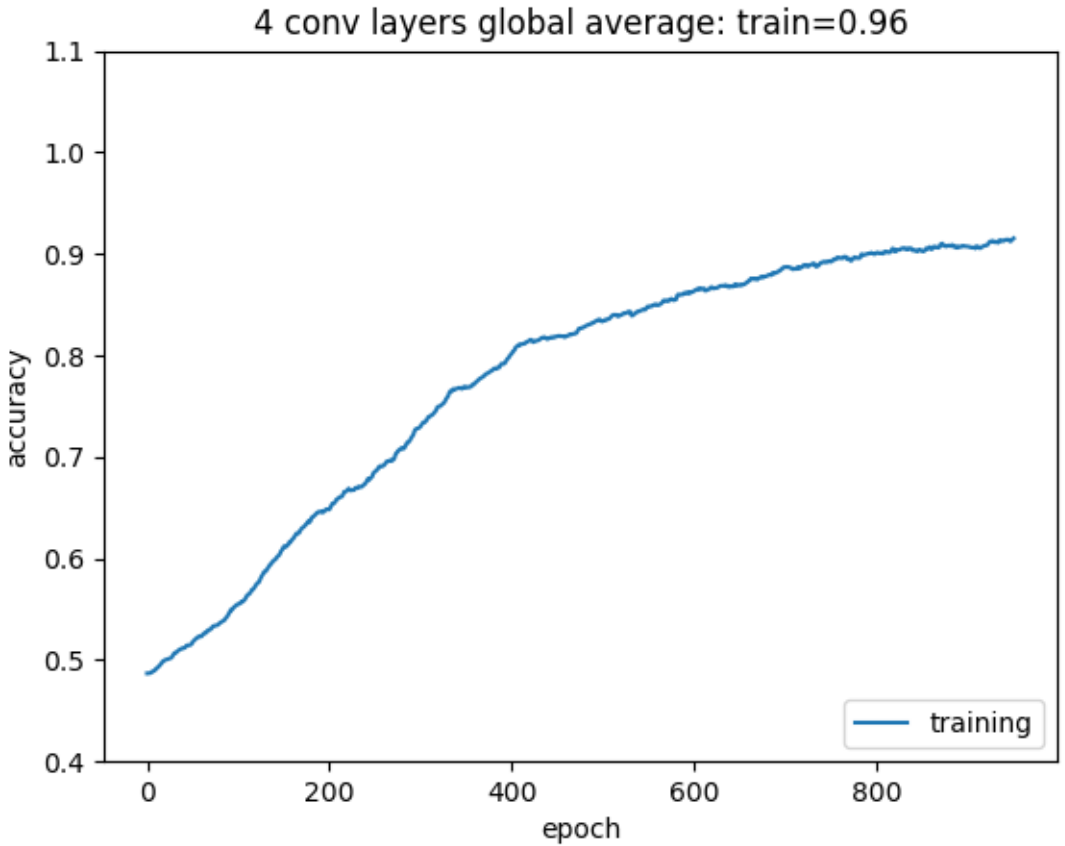
Replace FC layer with global average pooling



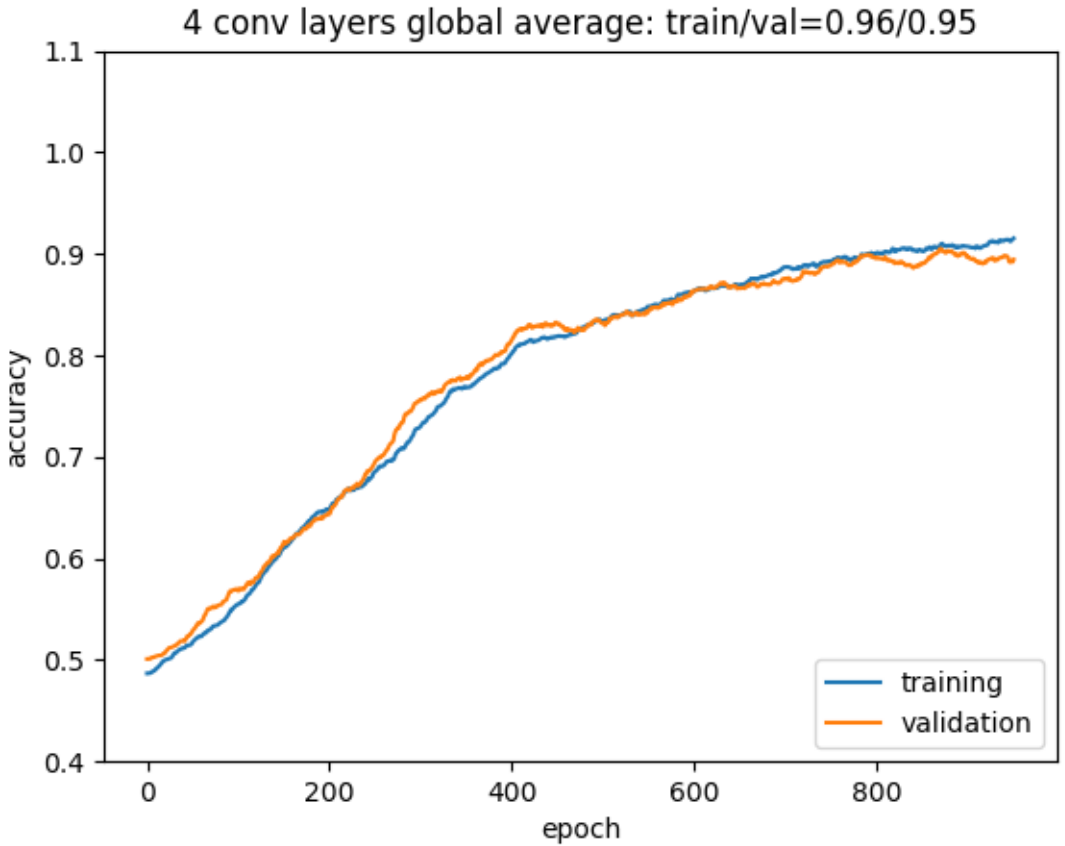
4 3x3x4 convolutional layers, Relu activation
2D MaxPooling
2D Global Average Pooling

Total number of parameters: 489

Results


4 conv layers global average: train=0.96
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



- 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

- Examples for linear regression, MLPs, CNNs
- Software frameworks
- High level view on generalization
- Model complexity, overfitting vs. underfitting
- Model parameters: Convolutional vs fully connected layers

Upcoming workshops

ISMRM Workshop on Machine Learning, Part II
25-28 October 2018
Capital Hilton, Washington, D.C., USA

Machine Learning for Medical Image Reconstruction (MLMIR)

MICCAI 2018 Granada SPAIN

- MICCAI paper results released: 22nd May 2018
- Paper submission deadline: 11th June 2018
- Decisions to authors: 2nd July 2018
- Camera-ready submission: 17th July 2018
- Workshop: 16th September 2018 (am), Granada, Spain

ML solves the problems the world really cares about!

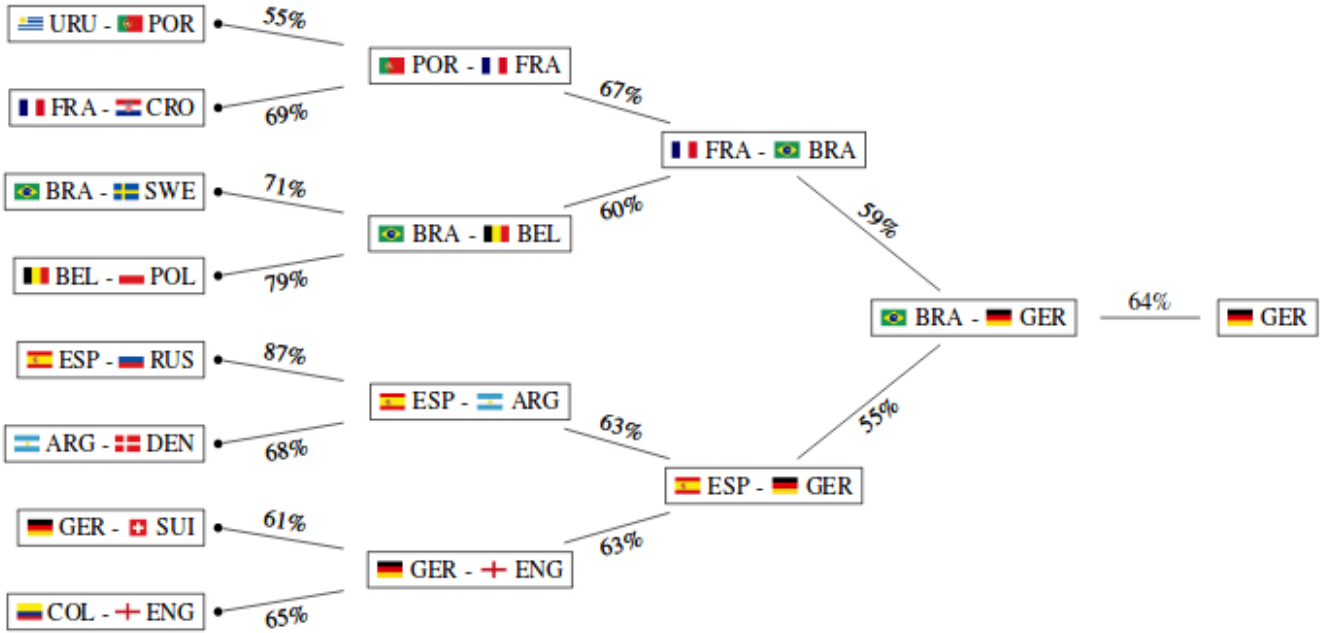


Figure 5: Most probable course of the knockout stage together with corresponding probabilities for the FIFA World Cup 2018 based on 100,000 simulation runs.

Groll et al.: https://arxiv.org/abs/1806.03208

Acknowledgements

- NYU
  - Mary Bruno
  - Joe Katsnelson
  - Matt Muckley
  - Michael Recht
  - Dan Sodickson
- TU Graz
  - Kerstin Hammernik
  - Erich Kobler
  - Thomas Pock
- Siemens healthineers
  - Matthias Fenchel
- Grant Support
  - NIH P41 EB017183

Thank you!

https://github.com/FlorianKnoll/ISMRM2018_Educational_DeepLearning