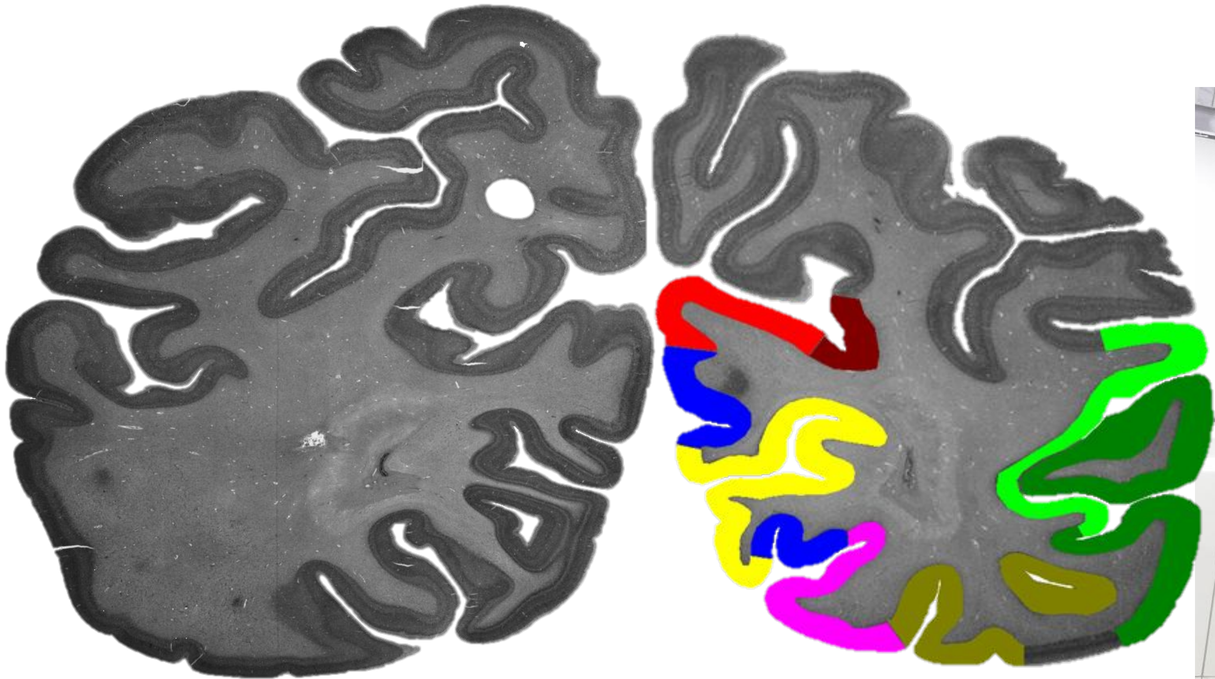


Terabyte-scale image analysis with HPC-enabled Deep Learning for building a map of the human brain

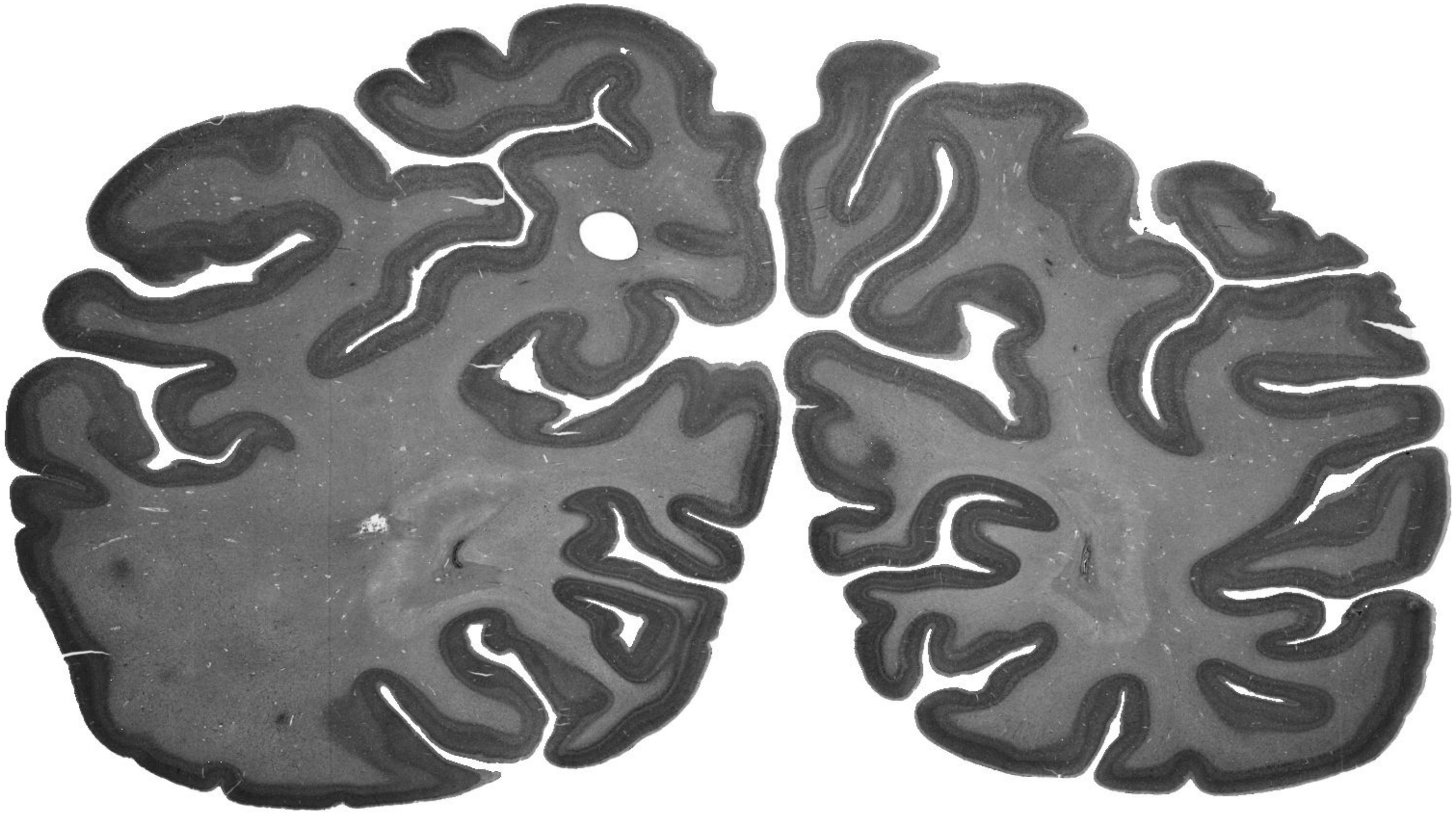
JSC MSA: GPU SEMINAR

16.06.2020 I CHRISTIAN SCHIFFER (TEAM BIG DATA ANALYTICS, INM-1, FORSCHUNGSZENTRUM JÜLICH)

JÜLICH
Forschungszentrum

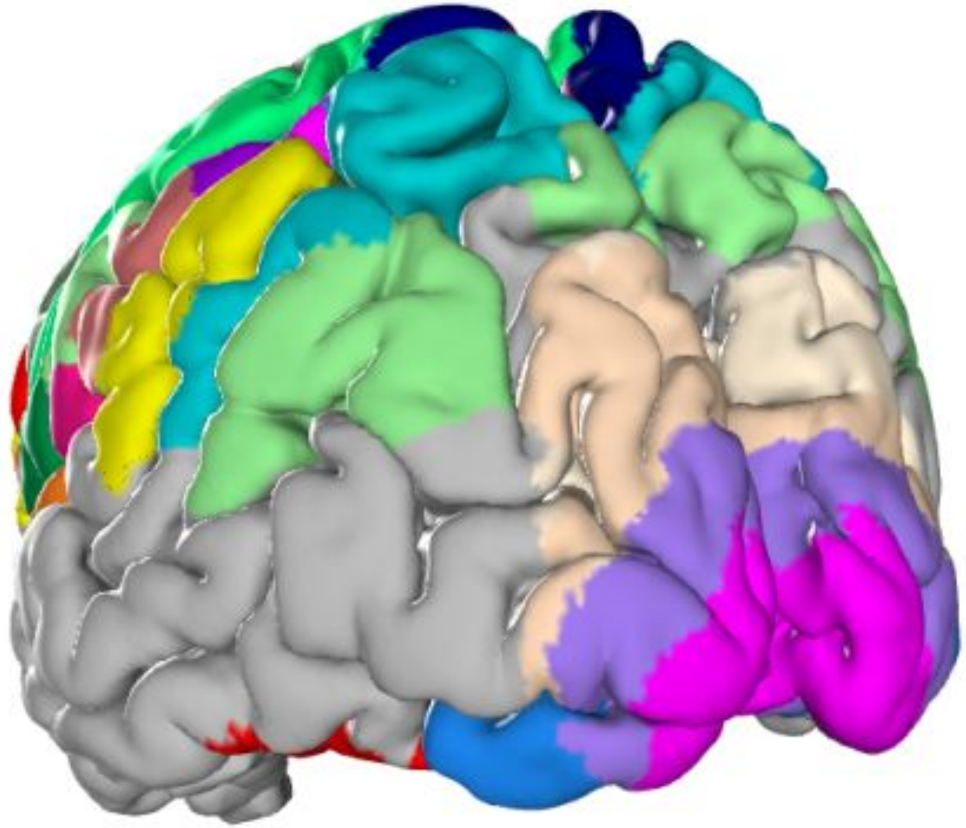
Outline

- **Human Brain Mapping**
- **Deep Learning on HPC**
  - **Frameworks**
  - **Distributed Deep Learning**
- **Deep Learning on "Big Data"**



Building a Human Brain Atlas for Cytoarchitecture

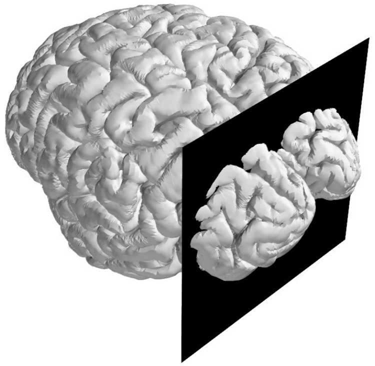
- Three-dimensional model of the human brain
- Data from multiple modalities in common space
- One aspect: **Cytoarchitectonic areas**
- **Cytoarchitectonic mapping** to delineate cortical areas in high-resolution histological sections

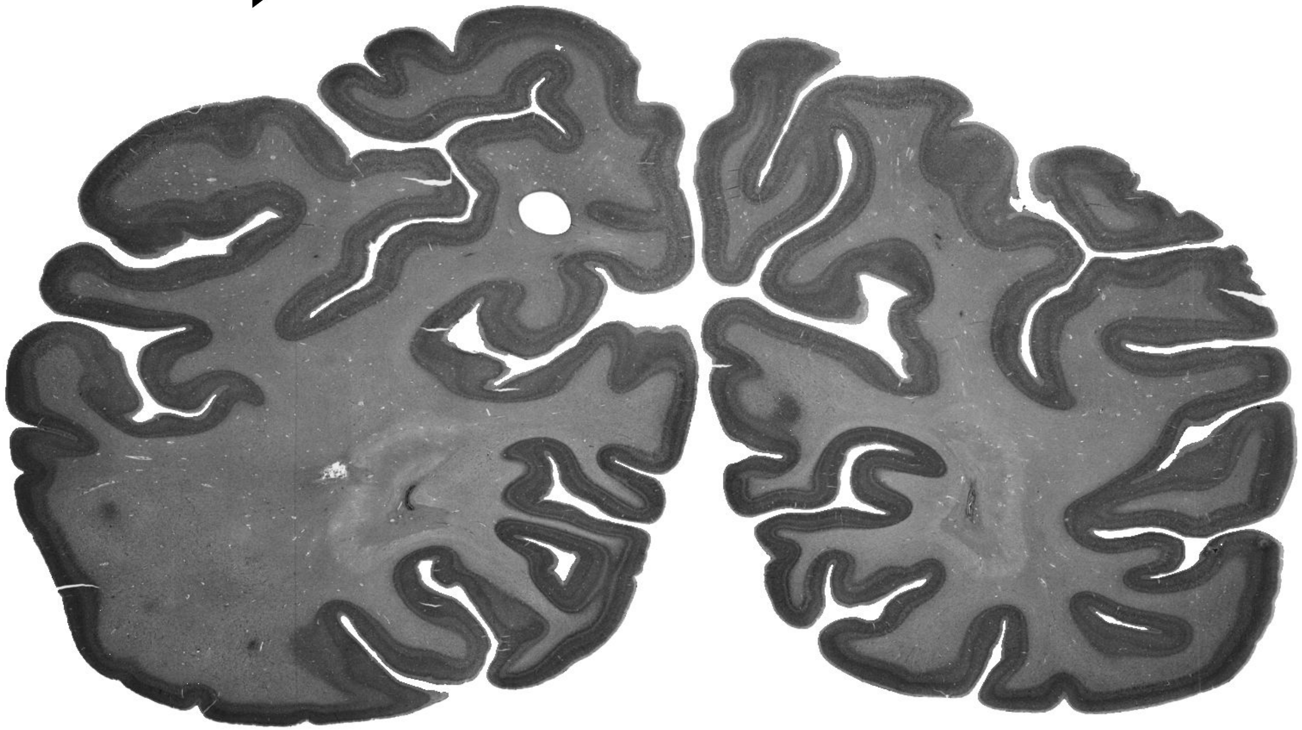


JuBrain Probabilistic Atlas
http://www.jubrain.fz-juelich.de

Histological Human Brain Sections

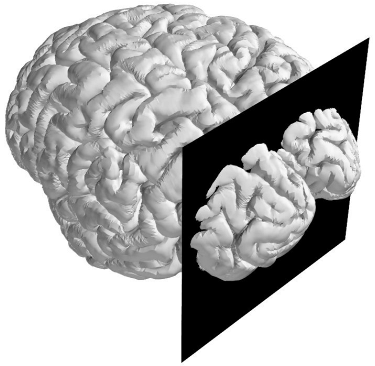
- Cut brain in ~7400 20µm thick sections
- Stain cell bodies and scan in light microscope at 1µm resolution

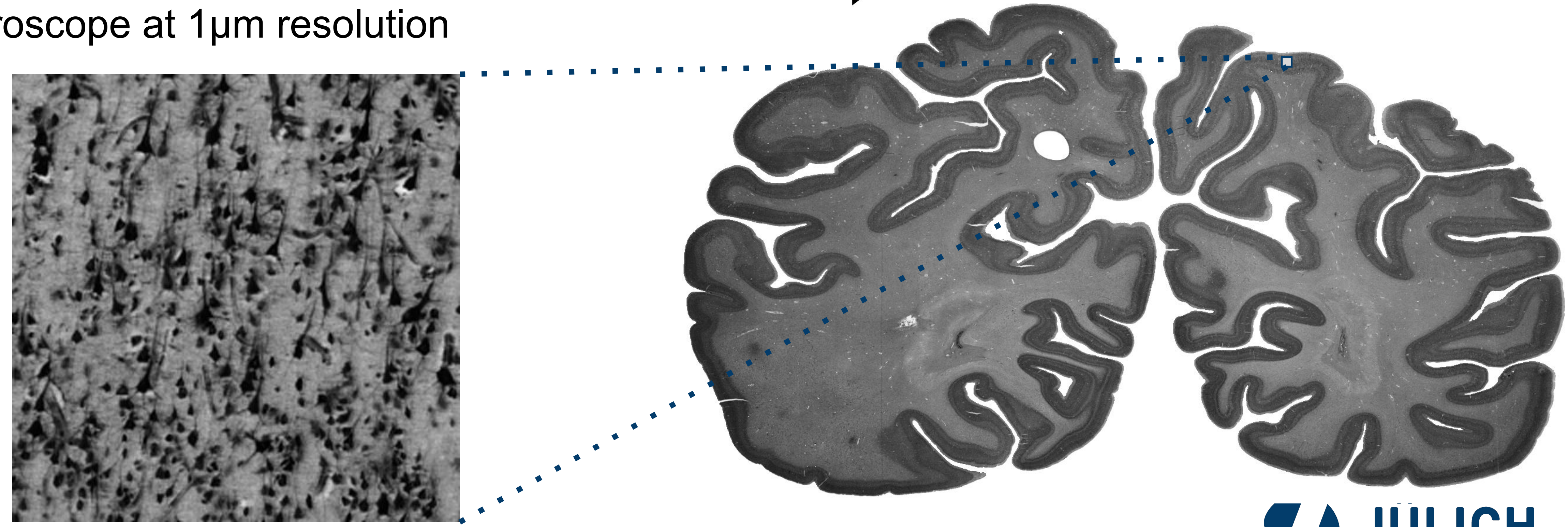




Histological Human Brain Sections

- Cut brain in ~7400 20µm thick sections
- Stain cell bodies and scan in light microscope at 1µm resolution

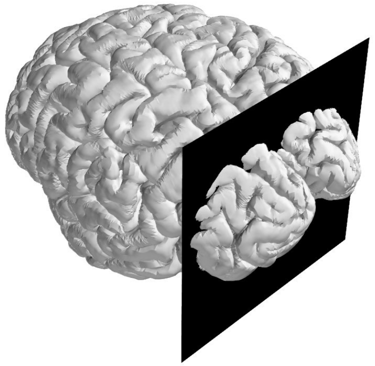


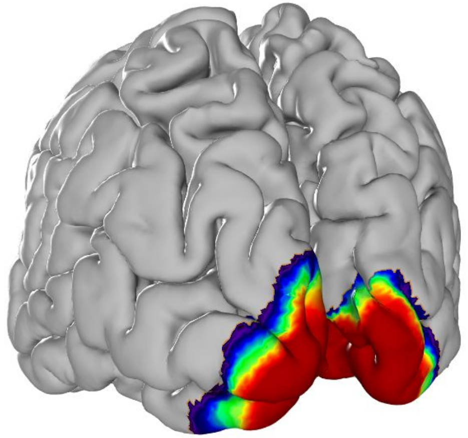


JÜLICH Forschungszentrum

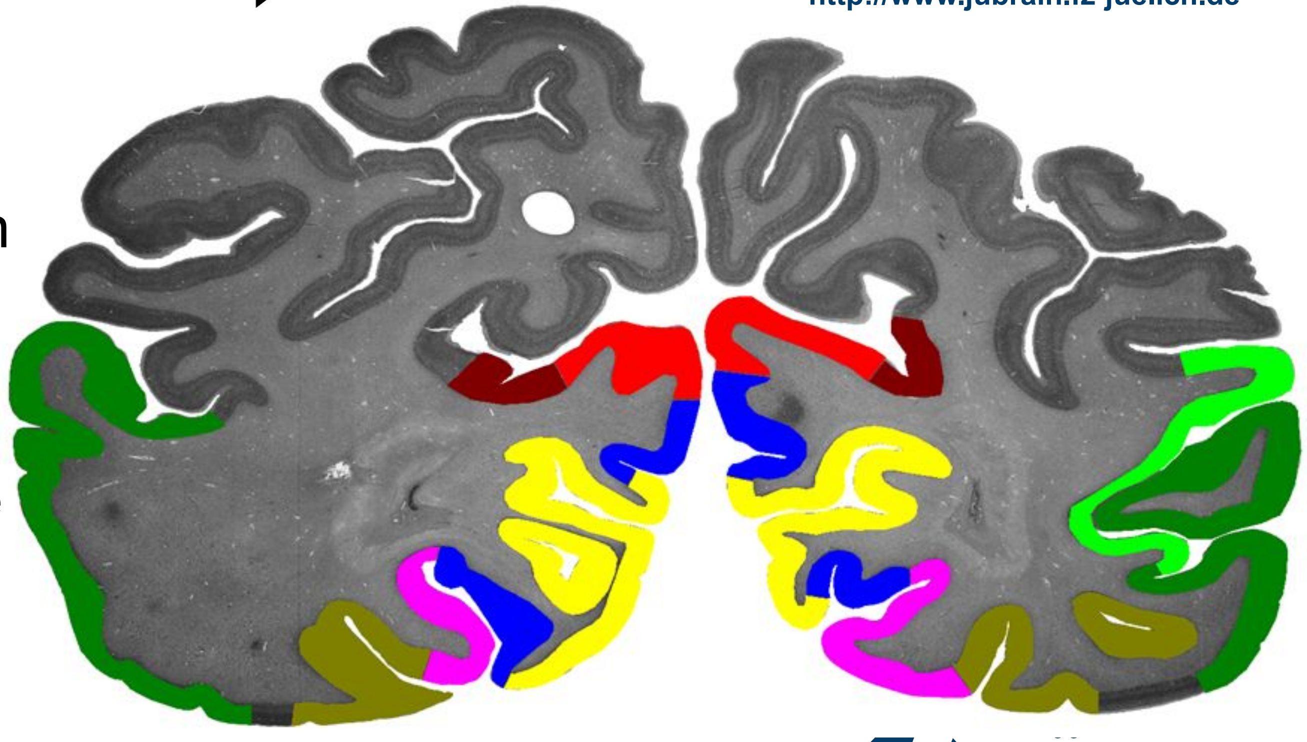
Histological Human Brain Sections

- Cut brain in ~7400 20µm thick sections
- Stain cell bodies and scan in light microscope at 1µm resolution
- Delineate brain areas in every 60th section of 10 different brains
- Average annotations in common reference space to obtain **probabilistic maps**



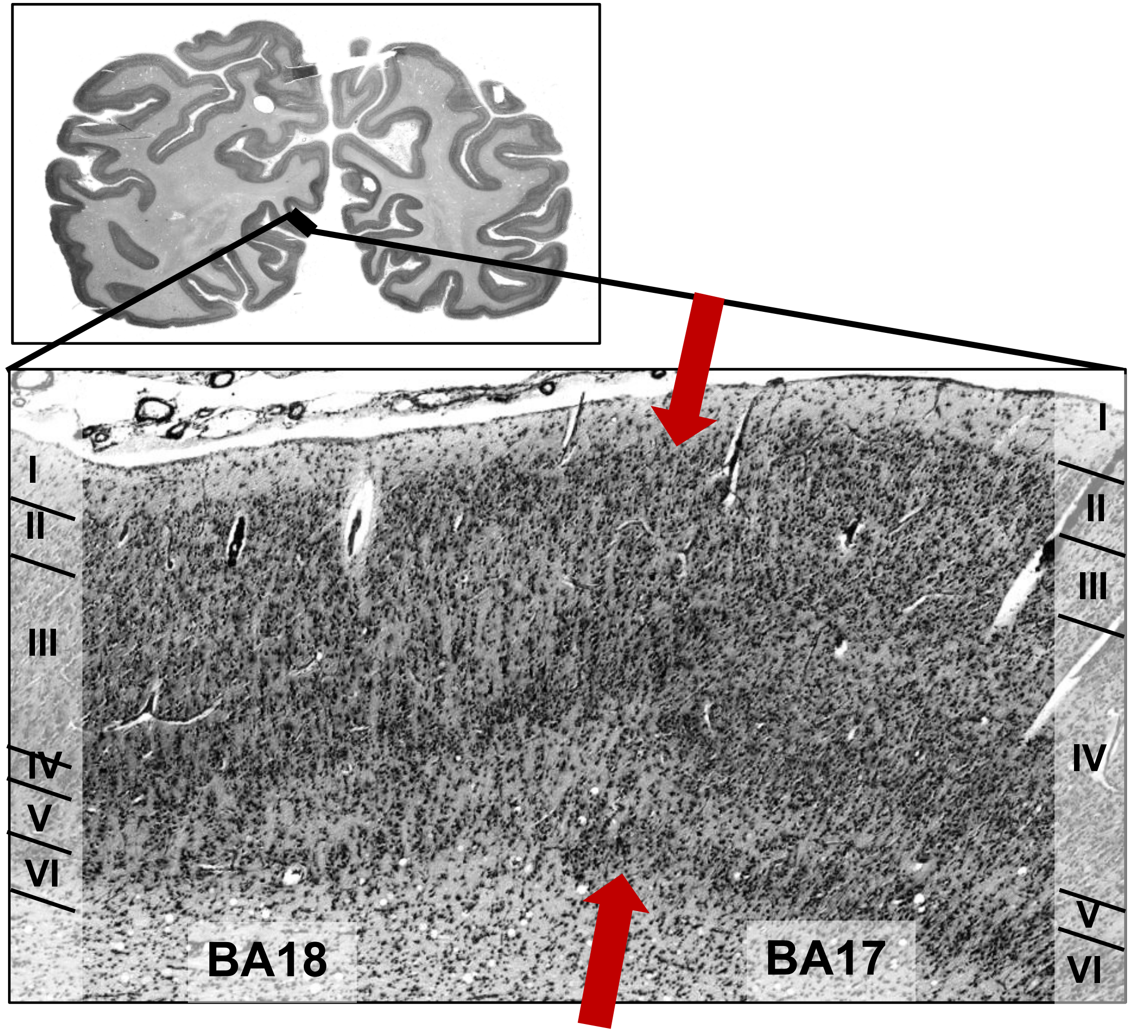


JuBrain Probabilistic Atlas
http://www.jubrain.fz-juelich.de



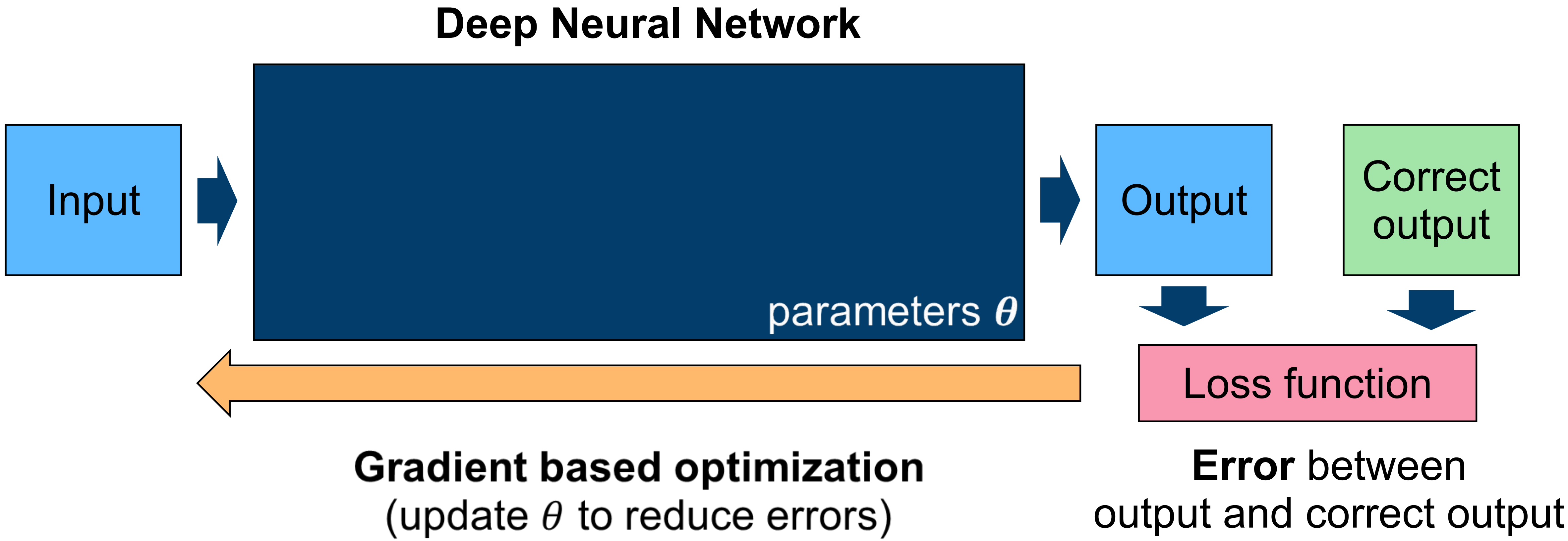
Observer Independent Method

- Distinguished by variations of cell distribution in cortical laminae and with respect to columnar organization
- Schleicher et al., 1999: Observer independent method for parcellation
- **Time and labor intensive, does not scale with high throughput imaging**
- **Idea:** Use **Deep Learning** to speed up and support mapping process



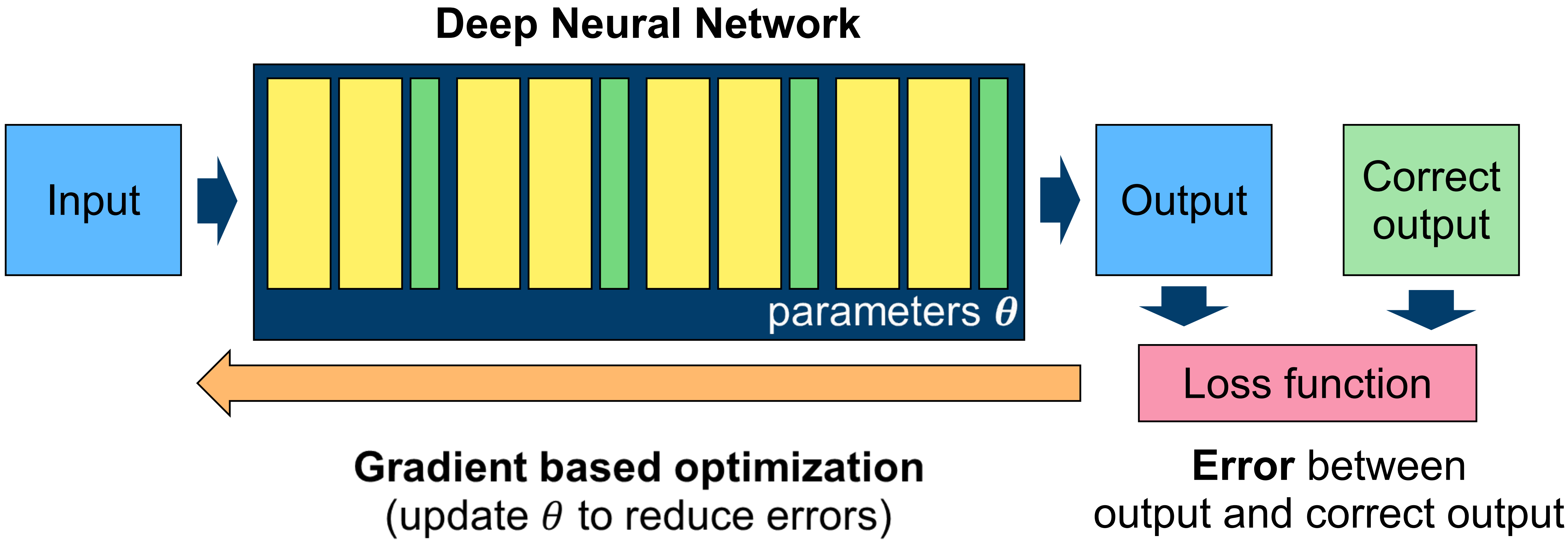
A. Schleicher et al., "Observer-independent method for microstructural parcellation of cerebral cortex: a quantitative approach to cytoarchitectonics", Neuroimage, vol 9, no 1, pp. 165-177, 1999

Basic principle of Deep Learning



JÜLICH Forschungszentrum

Basic principle of Deep Learning



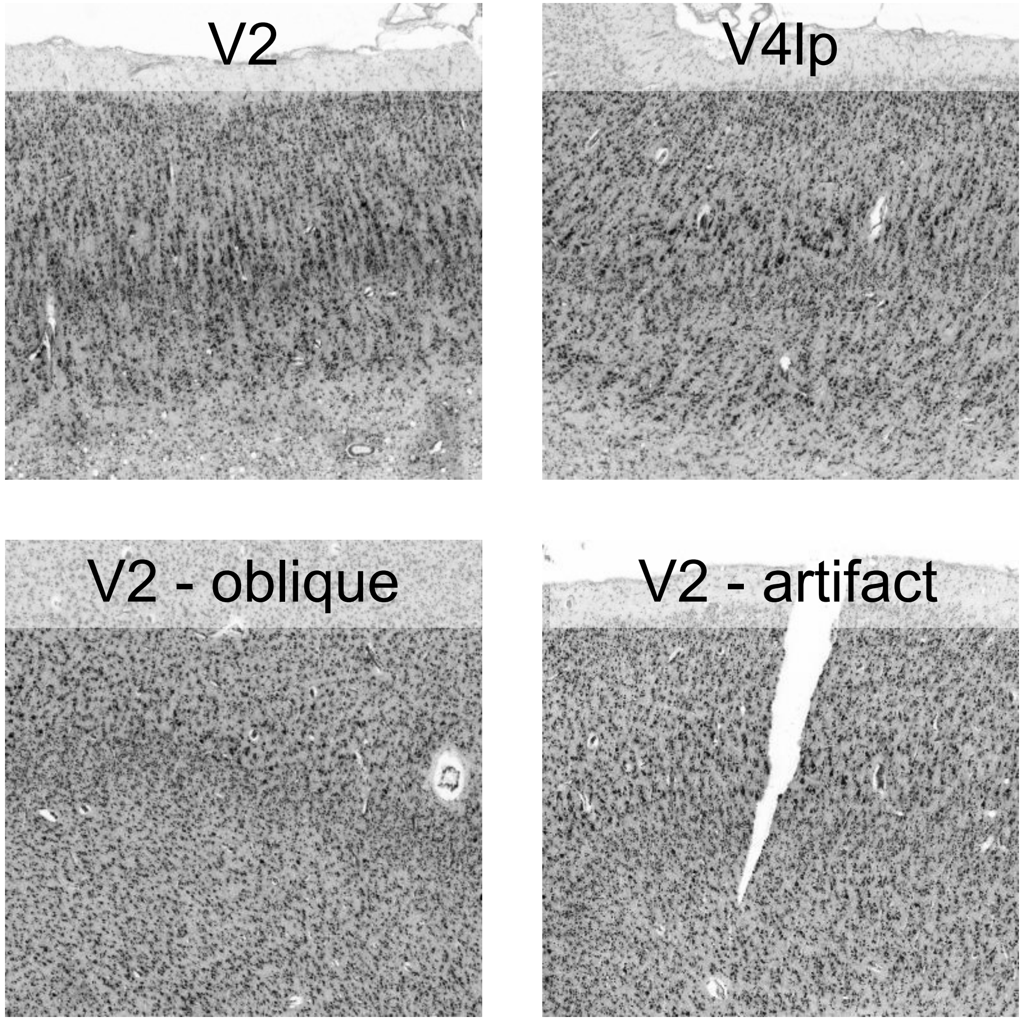
Dataset for automatic cytoarchitecture classification

INM-1 brain collection

- Collection of donor brains
- Data sizes for different scanning protocols
  - every 15th section: **~3.5 TB**
  - every section: **~50 TB**
  - every section w. z-scanning (30 layers): **~1.5 PB**
- **~400 sections** with partial brain area annotations

Challenges

- Complex and ambiguous cell patterns
- Inter-individual differences between brains
- High variability due to staining, sectioning artifacts, changing angle between sectioning plane and brain surface (*oblique cuts*)



Technical Challenges

- Training on whole images (~10 GB) is impossible, we train on large high-resolution **image patches**
  - **ImageNet:** 224x224 pixels
  - **Ours:** ~2000x2000 pixels (4x4 mm²)
- Dataset does not fit into **memory** and has to be **read demand**
- **Pre-processing** of large images (e.g. data augmentation) is computationally expensive
- **GPU memory** is limited, few patches fit on a single GPU

HPC to the rescue!

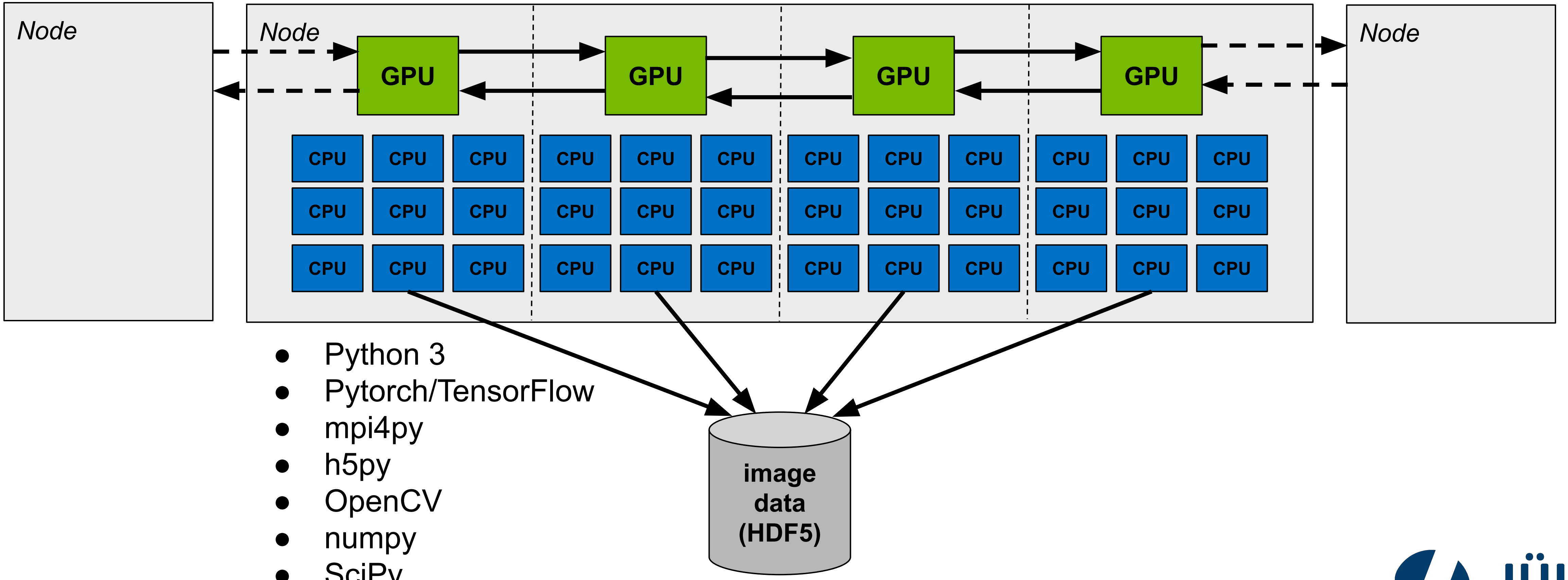
- Data is stored close to the JSC HPC systems
  → **Fast I/O**

- I/O and pre-processing can be parallelized across many CPUs
  → **Fast training sample creation**

- Training can be parallelized across many GPUs
  → **Fast training pipeline**

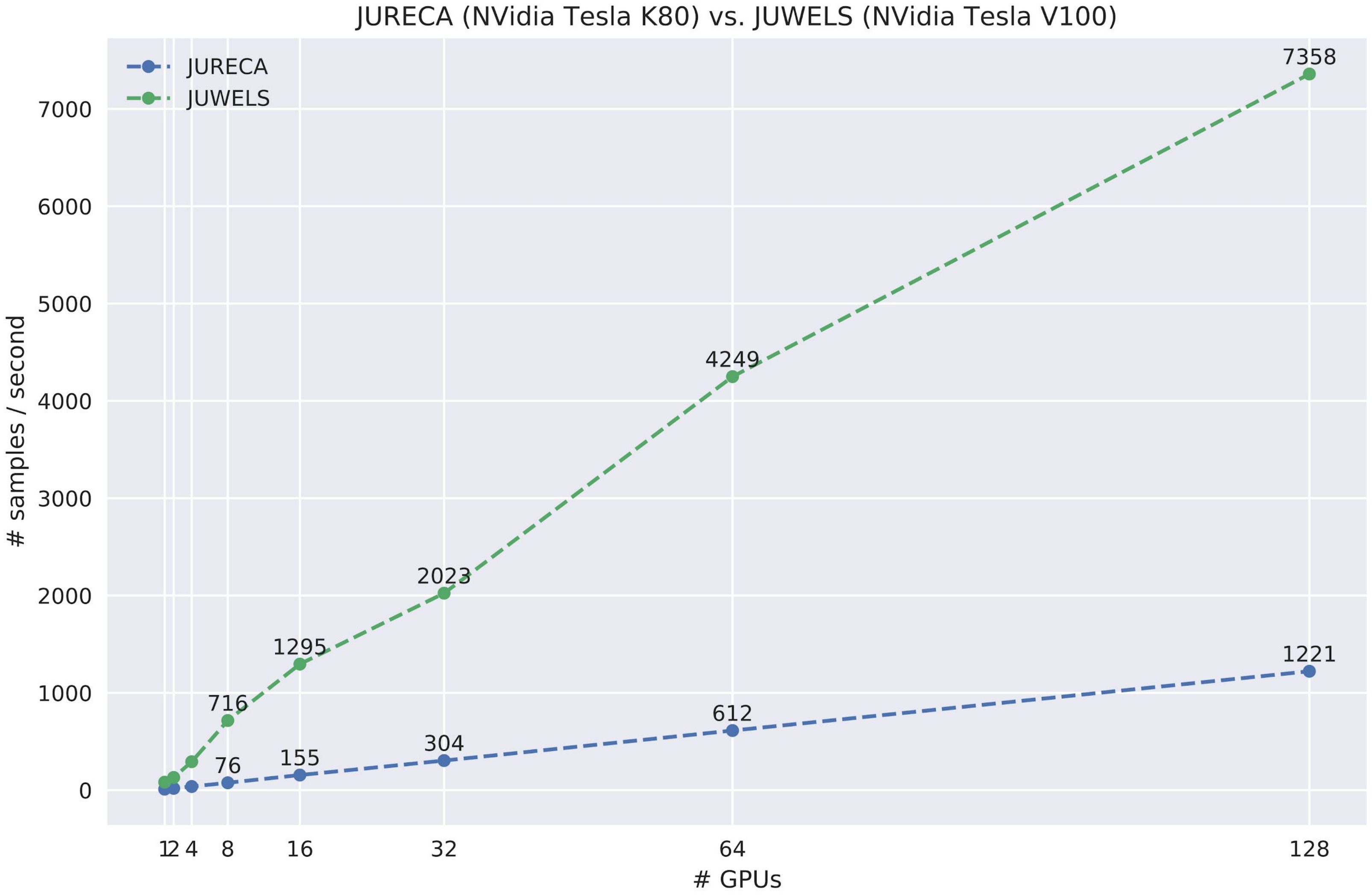
- Large scale experiments of hyperparameter exploration can be parallelized across many nodes
  → **Fast iterative development loop**

HPC enabled training workflow



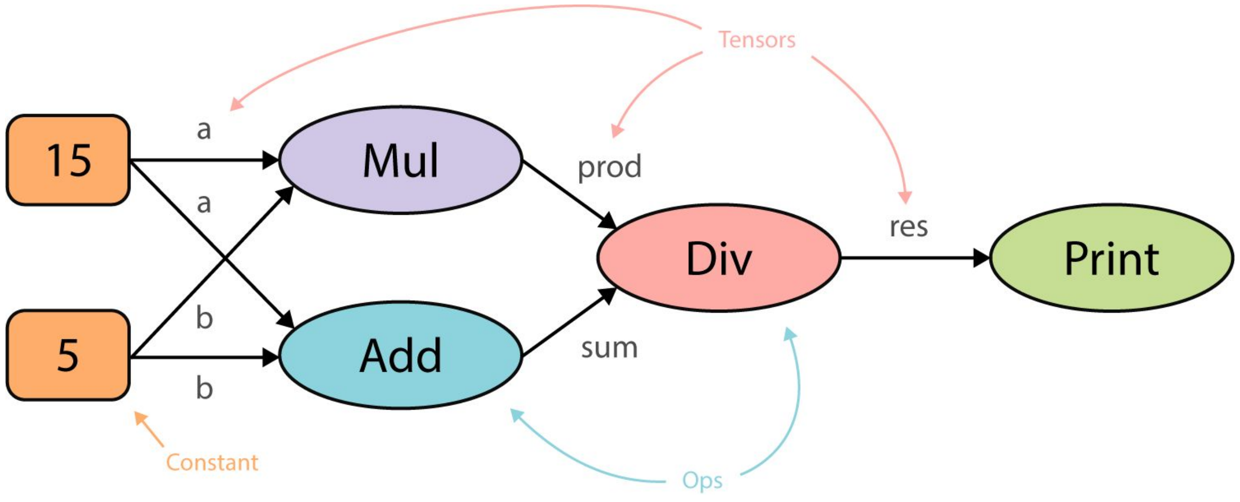
- Python 3
- Pytorch/TensorFlow
- mpi4py
- h5py
- OpenCV
- numpy
- SciPy

Workflow performance on JURECA and JUWELS



Deep Learning Frameworks

- Model neural network as **computation graph**
  - Nodes are **operations** (e.g. matrix multiply)
  - Edges are **tensors**
  - Enables **automatic differentiation**
  - Static or dynamic construction
- Many operations can be efficiently executed on GPUs, for example
  - Matrix multiplication (Fully-connected layer)
  - Convolution
- Focused on Deep Learning, but applicable to **many other applications**



Source: Understand TensorFlow by mimicking its API from scratch

Common libraries used by Deep Learning Frameworks

- **CUDA**
- **cuDNN** (NVIDIA CUDA Deep Neural Network library)
- **NCCL** (NVIDA Collective Communication Library)

Popular Deep Learning Frameworks

Popular

- **TensorFlow 1.x/2.x** (Google)
- **(tf.)keras** (created François Chollet, now at Google)
- **pytorch** (Facebook)
- **MxNet**+GluonCV+GluonNLP+GluonTS (Apache)
- **CNTK** (Microsoft)

Older frameworks (but you still find code for them)

- **theano** (Montreal Institute for Learning Algorithms)
- **Caffe** (relies more heavily on C++)

TensorFlow 1.x

- Computation graph is **statically** defined (define-and-run)
- Graph can be automatically optimized before execution
- **Shortcomings** (addressed in TensorFlow 2.x)
    - Hard to debug
    - Heavy use of global variables and states
    - Overloaded API

```python
import tensorflow as tf

# Input nodes
a = tf.placeholder(tf.int16)
b = tf.placeholder(tf.int16)

# Define computation graph
add = tf.add(a, b)
mul = tf.multiply(a, b)
div = tf.divide(add, mul)

# Execute graph with concrete input
with tf.Session() as sess:
   print(sess.run(div, feed_dict={a: 15,
b: 5}))
```

TensorFlow 2.x

- Computation graph is **dynamically** defined
- Computation can be structured in **functions**
  - Improved code structure
  - Functions can be compiled for improved performance
  - Compilation can be temporarily disabled for debugging
- API cleanup
  - tf.keras main API for neural nets

```python
import tensorflow as tf

# Dynamic computation graph
a = tf.convert_to_tensor( 15)
b = tf.convert_to_tensor( 5)

add = tf.add(a, b)
mul = tf.multiply(a, b)
div = tf.divide(add, mul)

# Compile graph using function
@tf.function()
def compute(a, b):
 add = tf.add(a, b)
 mul = tf.multiply(a, b)
 return tf.divide(add, mul)

compute(15, 5)
```

(tf.)keras

- **API specification** for building and training neural networks
- Standalone implementation supports **multiple backends**
- Part of TensorFlow 2.x as **tf.keras**
- Allows training models with very few lines of code

```
import tensorflow as tf

# Get the data
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Define the model
model = tf.keras.models.Sequential([
 tf.keras.layers.Flatten( input_shape=(28, 28)),
 tf.keras.layers.Dense( 128, activation='relu'),
 tf.keras.layers.Dropout( 0.2),
 tf.keras.layers.Dense( 10)
])

# Define loss function
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()

# Compile model (set optimizer, loss function and metrics)
model.compile(optimizer='adam', loss=loss_fn,
metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=5)

# Apply the model
model.evaluate(x_test, y_test, verbose=2)
```

PyTorch

- Computation graph is **dynamically** defined
- **numpy-oriented** interface (*“numpy with GPUs”*)
- Fine-grained control through low-level API
- Optional third-party libraries to avoid boilerplate code (e.g. **Lightning**)

```
import torch

# Define variables
a = torch.from_numpy(15)
b = torch.from_numpy(5)

# Move to third GPU
a = a.to("cuda:2")
b = b.to("cuda:2")

# Compute (on GPU)
add = torch.add(a, b)
mult = torch.mul(a, b)
div = torch.div(add, mult)
```

TensorFlow or PyTorch?

TensorFlow (+tf.keras)

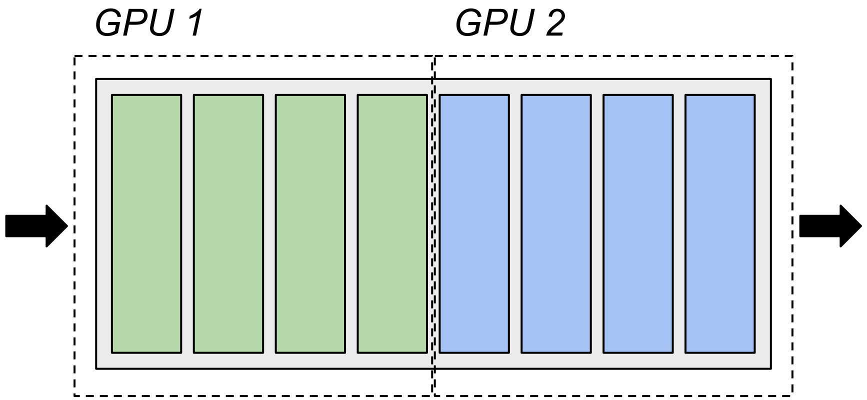
- Allows quick and easy experimentation with standard processing pipelines and models
- Many things can be accomplished in few lines of code
- Offers various ways to deploy trained models to production (e.g. TensorFlow.js for the web or TensorFlow Lite for mobile and IoT)
- More exotic experiments can be hard to implement

PyTorch

- Development feels more “phytonic”
- Lower level API allows more fine grained control
- Non-standard experiments are often easier to implement
- Higher flexibility comes at the cost of more boilerplate code (e.g. training loop)

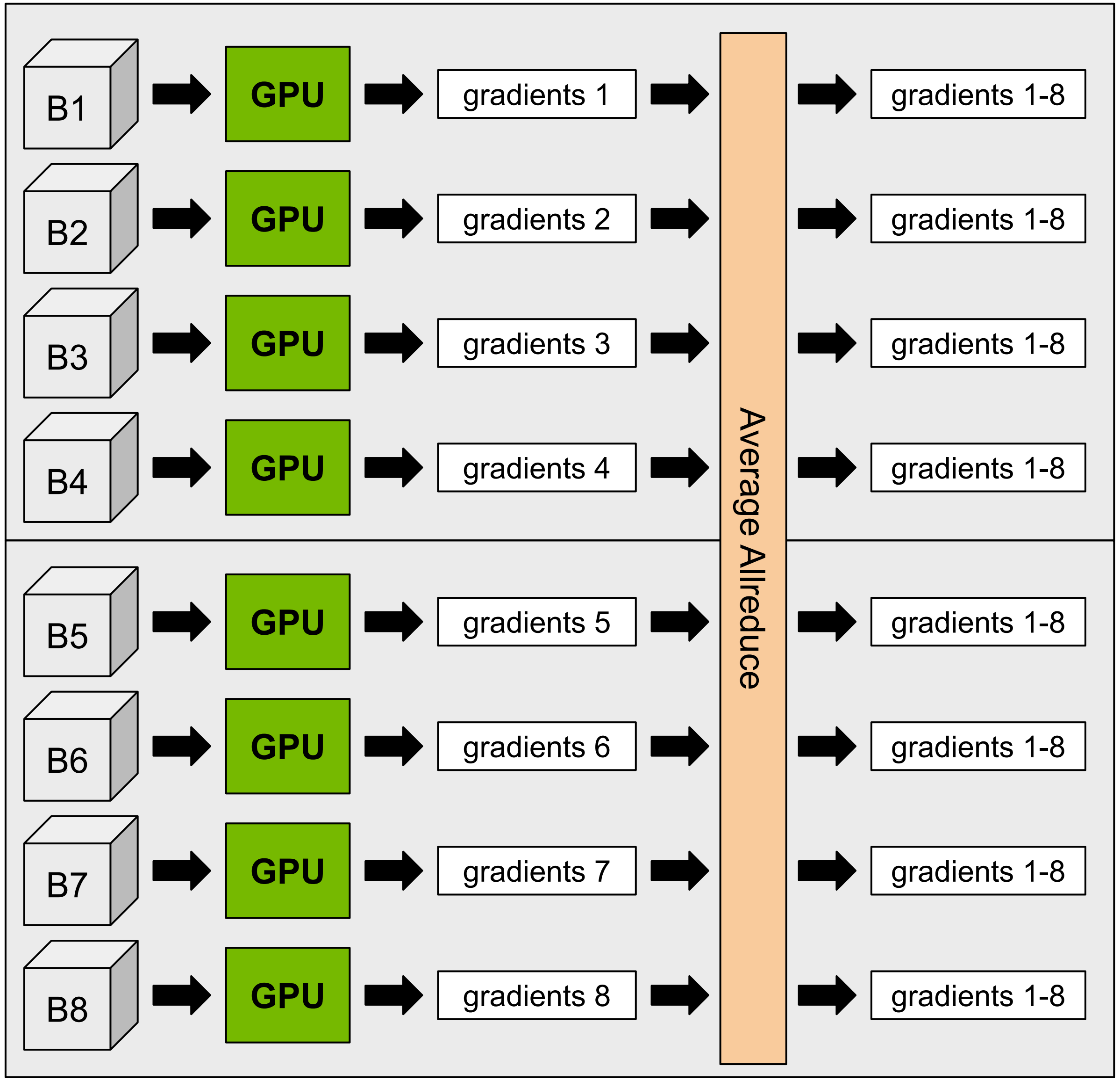
Distributed Deep Learning

- Distributed Deep Learning enables training across multiple GPUs on one node or across multiple nodes
- Reduces training time or allows training of larger models
- **Data parallelism**
  - Each GPU gets a replica of the model
  - Each GPU processes different samples
  - Gradients are averaged before updating the weights
- **Model parallelism** (rarely used in practice)
  - Split one model across multiple GPUs
  - Useful for very large models which do not fit on one GPU



Data parallel training

- Well supported in all frameworks
- Most common variant: **Synchronized Gradient Descent**
- Gradient averaging can be efficiently implemented, eg. with MPI or NCCL



Distributed training in TensorFlow with Horovod

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Define the model
model = tf.keras.Sequential([
   tf.keras.layers.Conv2D( 32, [3, 3], activation='relu'),
   ...
   tf.keras.layers.Dropout( 0.5),
   tf.keras.layers.Dense( 10, activation='softmax')
])

opt = tf.optimizers.Adam( learning_rate=0.1)

model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),
              optimizer=opt,
              metrics=['accuracy'],
              experimental_run_tf_function =False)

model.fit(x_train, y_train, epochs=5)
```

Note: Horovod also supports PyTorch and MXNet

Assign GPU

Modify optimizer

Init weights

```
import tensorflow as tf
import horovod.tensorflow.keras  as hvd

# Initialize Horovod
hvd.init()
# Assign GPU to this process
gpus = tf.config.experimental.list_physical_devices( 'GPU')
tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()],  'GPU')

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Define the model
model = ...

opt = tf.optimizers.Adam( learning_rate=0.1)
# Make optimizer distributed
opt = hvd.DistributedOptimizer(opt)

model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),
              optimizer=opt,
              metrics=['accuracy'],
              experimental_run_tf_function =False)

# Make sure all models start with the same weights
callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback( 0), ]
model.fit(x_train, y_train, epochs=5, callbacks=callbacks)
```

Distributed training in PyTorch

```python
from torch import nn
from torchvision.models import resnet50
import torch.optim as optim
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel

data = ...

# Create a model
model = resnet50()
# Move model to first GPU
mode = model.cuda()

# Create optimizer
opt = optim.SGD(model.parameters(), lr=0.1)
loss_fn = nn.MSELoss().cuda()

# Training loop
for x, y in data:
   opt.zero_grad()
   y_ = model(x)
   loss = loss_fn(y, y_)
   loss.backward()
   opt.step()
```

```python
from torch import nn
from torchvision.models import resnet50
import torch.optim as optim
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel
```

Init

```python
# Initialize distributed environment
rank = ...   # e.g. by mpi4py
size = ...
dist.init_process_group("nccl", "tcp://127.0.0.1:12345", rank, size)

data = ...

# Create a model
model = resnet50()
```

Modify model

```python
# Model wrapper takes care of averaging gradients
model = DistributedDataParallel(model, device_ids=[rank, ])
# Move model to correct GPU
mode = model.to(rank)

# Create optimizer
opt = optim.SGD(model.parameters(), lr=0.1)
loss_fn = nn.MSELoss().to(rank)

# Training loop
for x, y in data:
   opt.zero_grad()
   y_ = model(x)
   loss = loss_fn(y, y_)
   loss.backward()
   opt.step()
```


JÜLICH Forschungszentrum

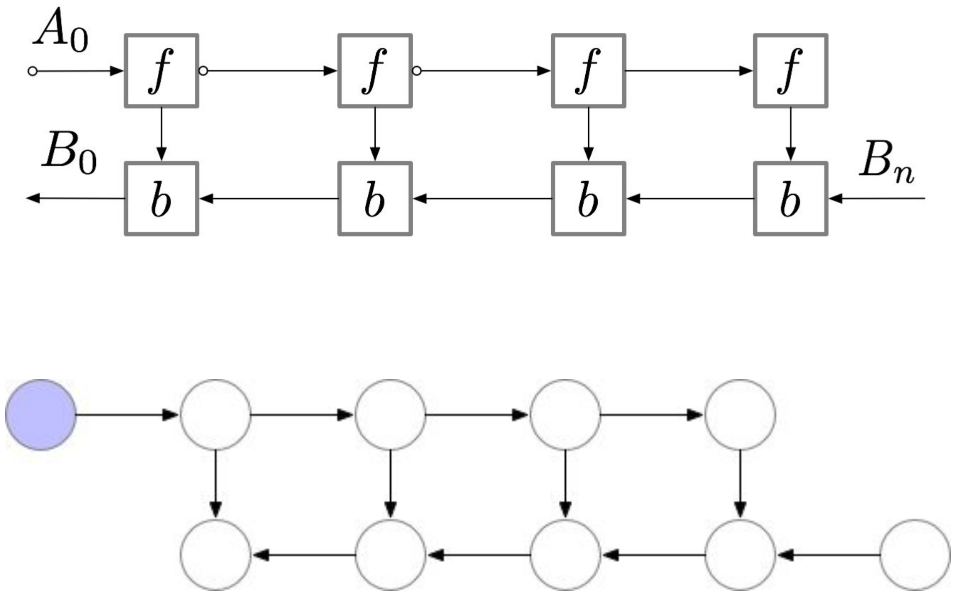
Deep Learning on “Big Data”

- Most Deep Learning applications rely on **large datasets**, but **individual samples** are mostly not very large
  - **Example:** ImageNet contains millions of images, but each image is not extremely large (e.g. 224x224 pixels)

- Some applications have to handle large datasets <u>and</u> large sample size, for example
  - medical imaging (2D and 3D data)
  - remote sensing
  - astronomy
  - …
- **GPU memory** often becomes the limiting factor when training deep models for such applications

Trade speed for memory by gradient checkpointing

- Intermediate layer outputs are usually **kept in memory**, as they are needed for gradient computation (chain rule of calculus)
- Hidden representations of large high-dimensional data (e.g. images or 3D volumes) can take massive amounts of space
- **Idea:** Trade speed for memory by **discarding** intermediate outputs and **recompute** them on demand during gradient computation

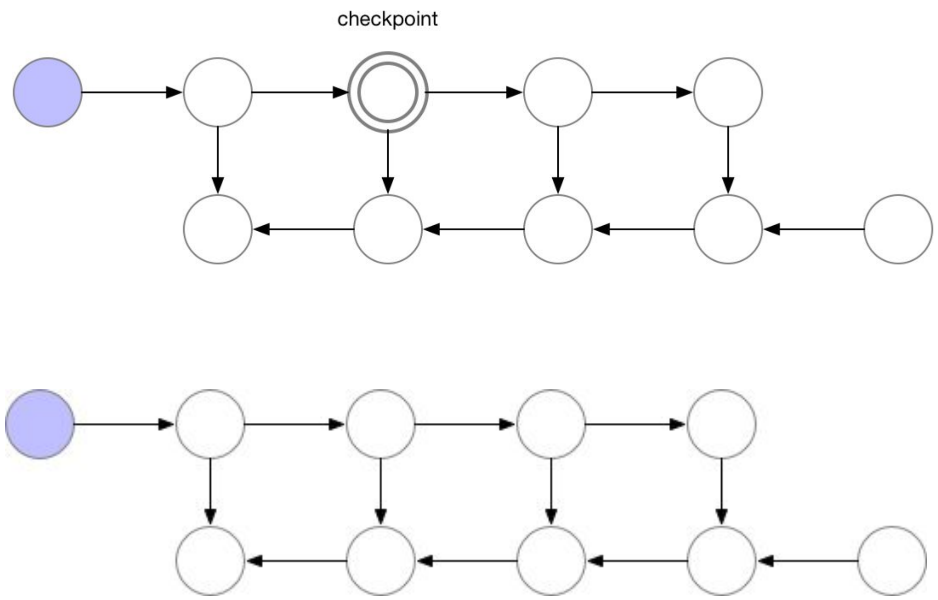


Source: Make huge neural nets fit in memory

JÜLICH Forschungszentrum

Trade speed for memory by gradient checkpointing

- Intermediate layer outputs are usually **kept in memory**, as they are needed for gradient computation (chain rule of calculus)
- Hidden representations of large high-dimensional data (e.g. images or 3D volumes) can take massive amounts of space
- **Idea:** Trade speed for memory by **discarding** some intermediate outputs and **recompute** them on demand during gradient computation



Source: Make huge neural nets fit in memory

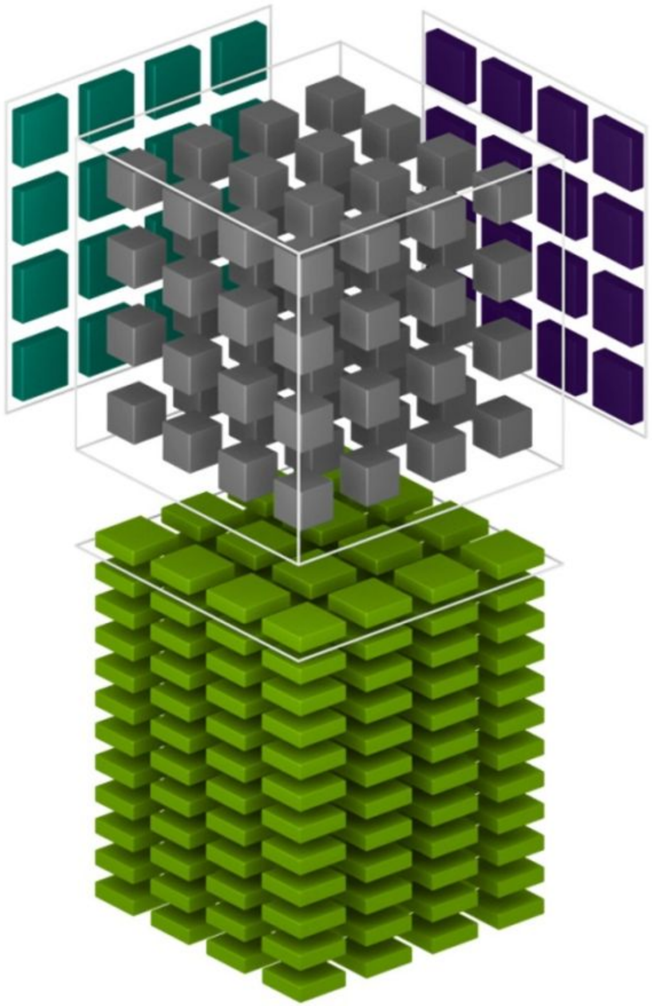
Gradient checkpointing in PyTorch

```python
from torch import nn


class Model(nn.Module):
   def __init__(self):
       super().__init__()
       self.conv1 = nn.Conv2d(1, 16, kernel_size=3)
       self.conv2 = nn.Conv2d(16, 32, kernel_size=3)
       self.conv3 = nn.Conv2d(32, 64, kernel_size=3)
       self.conv4 = nn.Conv2d(64, 128,
kernel_size=3)

   def forward(self, x):
       x = self.conv1(x)
       x = self.conv2(x)
       x = self.conv3(x)
       x = self.conv4(x)

       return x
```

```python
from torch import nn
import torch.utils.checkpoint as cp

class Model(nn.Module):
   def __init__(self):
       super().__init__()
       self.conv1 = nn.Conv2d(1, 16, kernel_size=3)
       self.conv2 = nn.Conv2d(16, 32, kernel_size=3)
       self.conv3 = nn.Conv2d(32, 64, kernel_size=3)
       self.conv4 = nn.Conv2d(64, 128,
kernel_size=3)

   def _make_cp_fun(self):
       def _cp_fun(x):
           x = self.conv1(x)
           x = self.conv2(x)
           return x
       return _cp_fun

   def forward(self, x):
       x = cp.checkpoint(self._make_cp_fun(), x)
       x = self.conv3(x)
       x = self.conv4(x)

       return x
```

Mixed Precision Training

- Deep Learning typically uses float32 (single precision) for parameters and layer outputs
- Using float16 (half precision) **speeds up computation** and **halves memory requirements**
- **Mixed precision training**
  - Use float16 for gradients and layer outputs
  - Keep parameters in float32
  - Internally scale loss and gradients to prevent underflow
- **TensorCores** in modern NVIDIA GPUs (Volta, Turing, Ampere) specifically speed up half precision operations



Mixed Precision Training in PyTorch (1.6-nightly build)

```python
from torch.cuda import amp

# Creates model and optimizer in default precision
model = ...
optimizer = ...
data = ...

# Creates a GradScaler once at the beginning of training.
scaler = amp.GradScaler()

for input, target in data:
    optimizer.zero_grad()

    # Runs the forward pass with autocasting.
    with amp.autocast():
        output = model(input)
        loss = loss_fn(output, target)

    # Scales loss.  Calls backward() on scaled loss to create scaled gradients.
    scaler.scale(loss).backward()

    # scaler.step() first unscales the gradients of the optimizer's assigned params.
    scaler.step(optimizer)

    # Updates the scale for next iteration.
    scaler.update()
```

Christian Schiffer

Email: c.schiffer@fz-juelich.de

Phone: +49 2461 61-3486

Team *Big Data Analytics*

Institute of Neuroscience and Medicine (INM-1)