Leibniz Supercomputing Centre

of the Bavarian Academy of Sciences and Humanities

AI Training Series
Introduction to the LRZ AI Systems

06.05.2024 | Maja Piskac, Ajay Navilarekal, Darshan Thummar, Navdar Karabulut

Upcoming Sessions

10.07.2024 High Performance **Data Analytics Using R** at LRZ

TBA **Fundamentals of Deep Learning -** powered by NVIDIA

… more to follow in fall 2024!

Further details and registration: https://app1.edoobox.com/en/LRZ/

Agenda

1. Introduction to the LRZ AI Systems

- Overview of the LRZ AI Systems
- Access to the LRZ AI Systems
- NVIDIA NGC Cloud
- Introduction to Enroot Containers
- Interactive and Batch Jobs
- Open on Demand
- Exercise: Run a job with an Enroot container

2. Fundamentals of Deep Learning

- Introduction to Convolutional Neural Networks
- Exercises: Train CNNs on a GPU
- Introduction to Transformers
- Exercise: Train a Transformer on a GPU
- Introduction to Reinforcement Learning
- Exercise: Reinforcement Learning

3. Distributed Training of Neural Networks

- Data Parallel Training
- Exercise: DP Training of CNN on 2 GPUs
- Model Parallel Training: Pipeline Parallel and Tensor Parallel
- Exercise: PP Training of Transformer on 2 GPUs

Planned breaks

- 11:30 – 11:45 Coffee Break I
- 12:30 – 13:30 Lunch break
- 15:00 – 15:15 Coffee Break II

Overview of the LRZ Systems

Data Science Storage
(DSS)

Multi-purpose cluster systems might be used for AI workloads as well, but have different focus

LRZ Linux Cluster

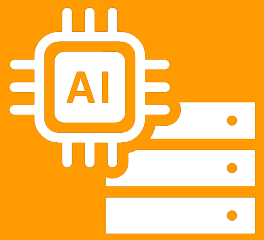
CoolMUC-2 Teramem-2 CoolMUC-3

lxlogin[1-3].lrz.de
lxlogin8.lrz.de

Designed and Configured for AI

LRZ AI Systems

- "Big Data" CPU nodes
- HPE P100 node
- V100 nodes
- DGX-1 P100, DGX-1 V100
- Multiple DGX A100



login.ai.lrz.de
https://login.ai.lrz.de

Flexible system that copes with almost any workload

LRZ Compute Cloud

LRZ Compute Cloud
(w/ some GPUs)

https://cc.lrz.de

Overview of the LRZ AI Systems

	DGX A100 Architecture	DGX A100 Architecture MIG	DGX-1 V100 Architecture	DGX-1 P100 Architecture	HPE Intel Skylake + Nvidia Node	V100 GPU Nodes	CPU Nodes
Number of Nodes	4	1	1	1	1	4	12
CPU cores per node	252	252	76	76	28	19	18 / 28 / 38 / 94
Memory per node	2 TB	1 TB	512 GB	512 GB	256 GB	368 GB	min. 360 GB
GPUs per node	8 NVIDIA A100	8 NVIDIA A100 (16 MIG partitions)	8 Nvidia Tesla V100	8 Nvidia Tesla P100	4 Nvidia Tesla P100	2 Nvidia Tesla V100	--
Memory per GPU	80 GB	40 GB (20GB per MIG partition)	16 GB	16 GB	16GB	16 GB	--
SLURM Partition	lrz-dgx-a100-80x8	lrz-dgx-a100-40x8-mig	lrz-dgx-1-v100x8	lrz-dgx-1-p100x8	lrz-hpe-p100x4	lrz-v100x2	lrz-cpu
Nodes	lrz-dgx-a100-[001-002,004-005]	lrz-dgx-a100-003	dgx-002	dgx-001	p100-001	gpu-[001-003,005]	cpu-[001-012]

DGX A100

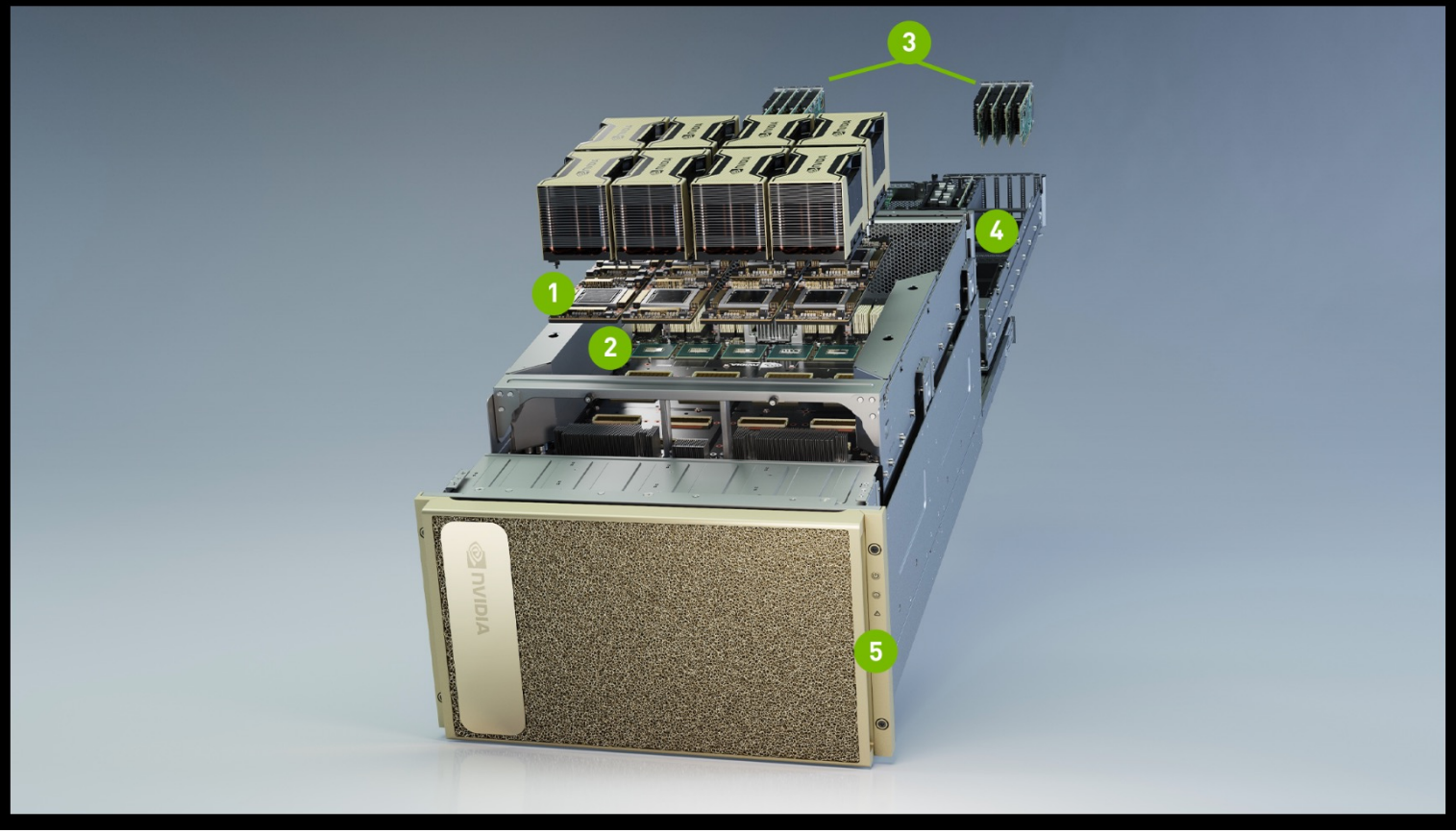
1. **8x NVIDIA A100 GPUs with up to 640GB total GPU memory**
   12 NVIDIA NVLinks® per GPU, 600GB/s of GPU-to-GPU bidirectional bandwidth

2. **6x NVIDIA NVSwitches™**
   4.8TB/s of bidirectional bandwidth, 2X more than previous-generation NVSwitch

3. **10x NVIDIA ConnectX-7 200Gb/s network interface**
   500GB/s of peak bidirectional bandwidth

4. **Dual 64-CORE AMD CPUs and 2TB system memory**
   3.2X more cores to power the most intensive AI jobs

5. **30TB Gen4 NVMe SSD**
   50GB/s of peak bandwidth, 2X faster than Gen3 NVMe SSDs



Storage on the LRZ AI Systems

Storage Pool	Designated Use	Top-level Directory	Size Limit	Automated Backup	Expiration	Additional Information
Home directory	unified home directory with the LRZ Linux Cluster, created when LRZ Linux Cluster access is granted Not suitable for heavy and/or high-frequency I/O operations - use the AI Systems DSS instead.	/dss/dsshome1 /.../<user>	100 GB	yes, backup to tape and file system snapshots	lifetime of LRZ project	File Systems and IO on Linux-Cluster
AI Systems DSS	high-bandwidth, low latency I/O, access is granted upon request through the LRZ Servicedesk	/dss/dssfs04	up to 4 TB	no	until further notice	
Linux Cluster DSS	general purpose, long-term data storage	/dss/dssfs02 /dss/dssfs03	up to 10 TB (or 20TB+ with associated costs)	yes for paid DSS / no for the free tier	lifetime of data project	File Systems and IO on Linux-Cluster (or DSS on demand offer with associated costs)
Exclusive/private DSS systems	specified by the system owner, can be purchased, implemented and housed exclusively for a private group of dedicated users	/dsslegfs01 /dsslegfs02 /dssmcmlfs01	specified by the system owner	specified by the system owner	specified by the system owner	Data Science Storage ("joint project offer")

Access to the LRZ AI Systems – How to access?

- User requirements to get the access:
  1. Own / get a Linux Cluster account: https://doku.lrz.de/display/PUBLIC/Access+and+Login+to+the+Linux-Cluster
  2. Submit a service request to LRZ Servicedesk – select "AI topics" and "LRZ AI Systems - Request for Access" from the drop-down lists. Request has to include Linux Cluster account username and a description of the intended usage.

- Login node login.ai.lrz.de accessible via SSH:

```
$ ssh --login_name=xxyyyzz login.ai.lrz.de
```

- Make sure you are connected to the Munich Scientific Network (MWN).
- Provide your LRZ Linux Cluster credentials to log in.

Access to the LRZ AI Systems – Slurm: sinfo, salloc, srun & scancel

$ ssh --login_name=xxyyyzz login.ai.lrz.de

Executes in the login node

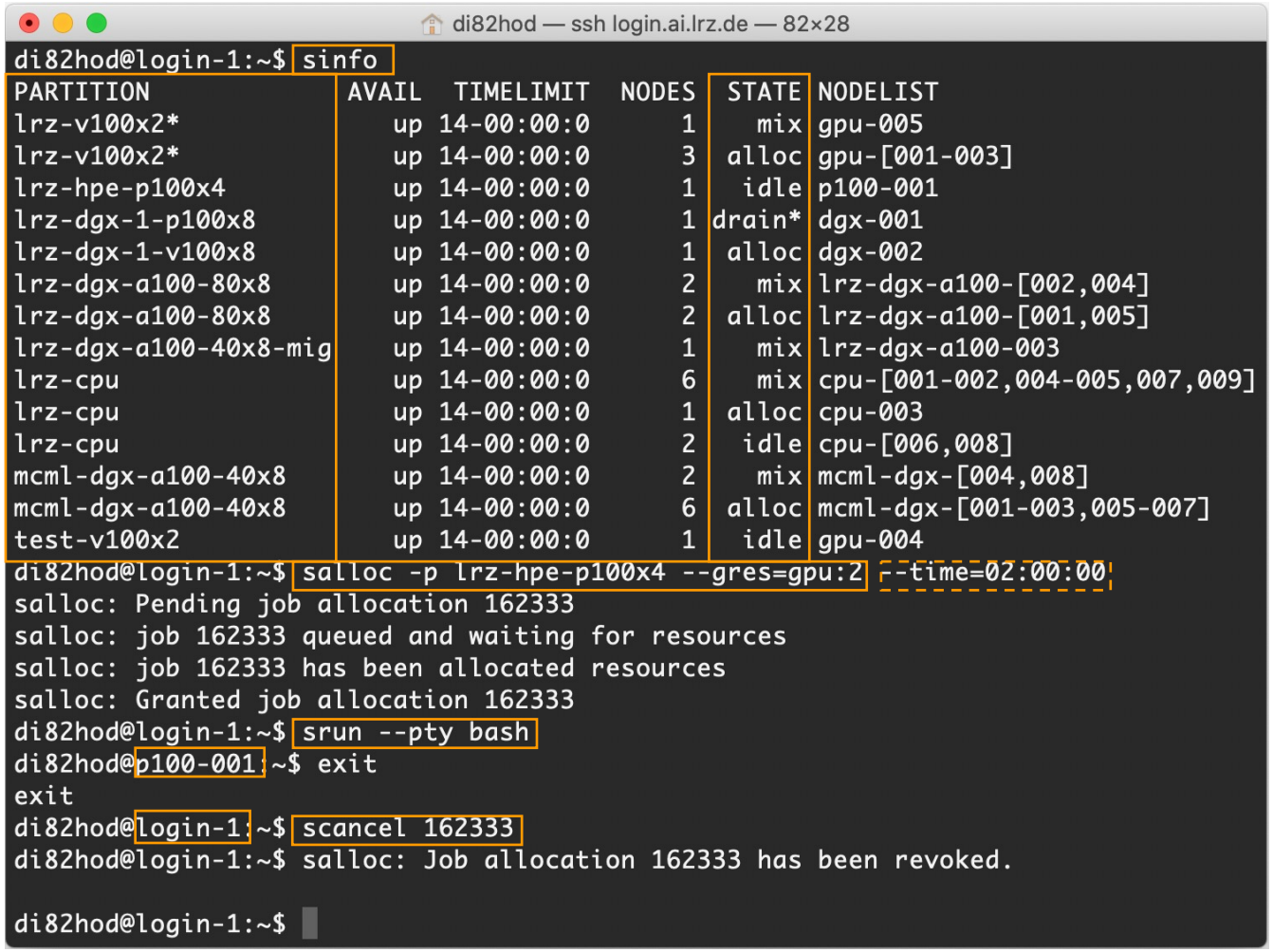
$ sinfo

$ squeue --user=xxyyzzz

$ salloc --partition=lrz-v100x2 --gres=gpu:1

$ srun --pty bash

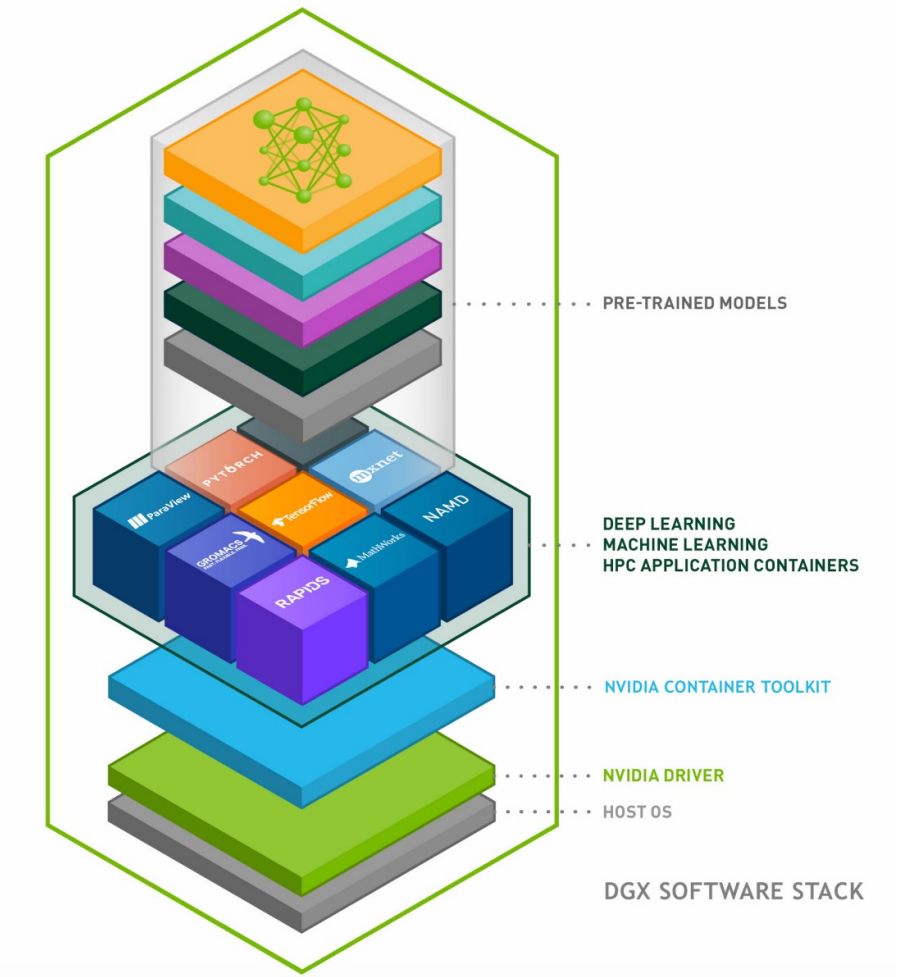
$ scancel job_id



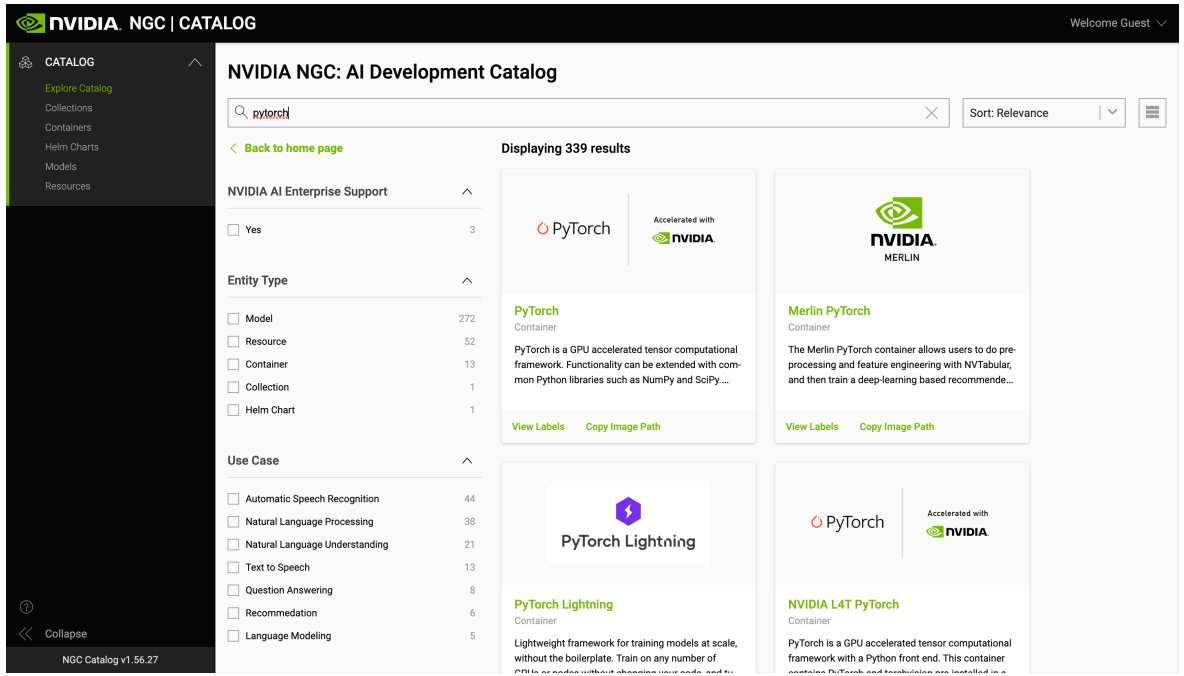
https://slurm.schedmd.com/overview.html

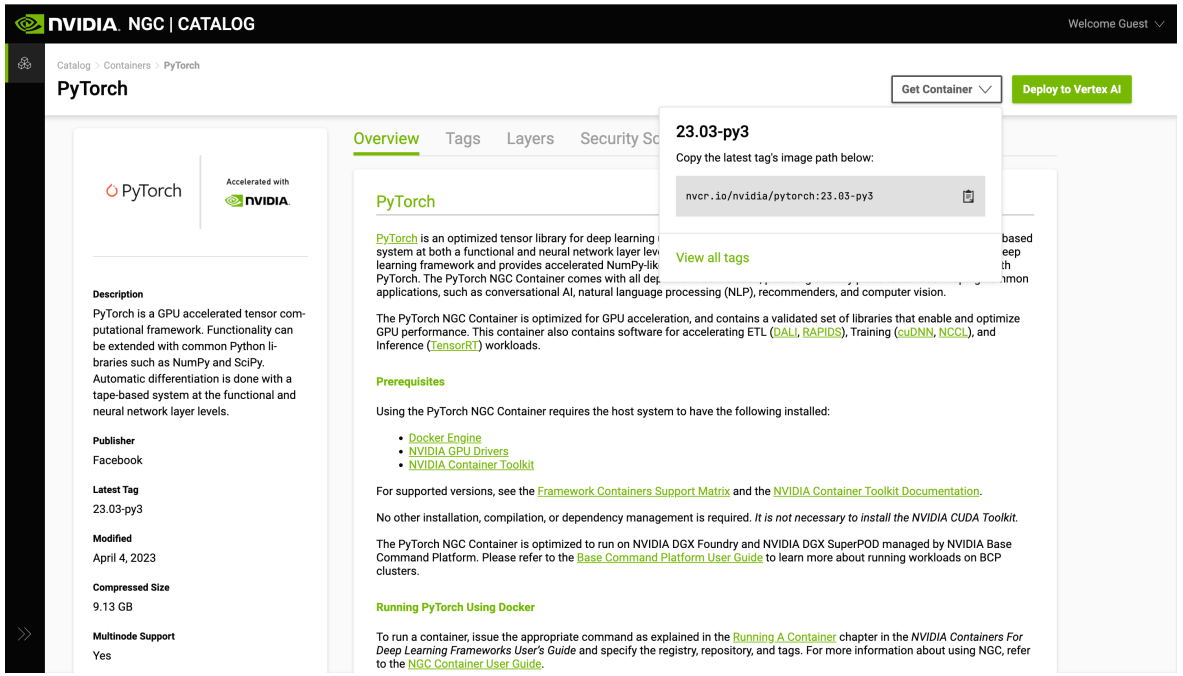
Nvidia NGC Containers

- The NGC catalogue provides access to **GPU accelerated** software that speeds up end-to-end workflows with performance optimized **containers, pretrained AI models, and SDKs** that can be deployed on any NVIDIA's GPU powered systems.
- The **NVIDIA Container Toolkit** includes a container runtime library and utilities to automatically configure containers to leverage NVIDIA GPUs.
- The **NVIDIA CUDA Toolkit**, incorporated within each GPU-accelerated container in NGC, is the development environment for creating high performance NVIDIA GPU-accelerated applications.
- https://catalog.ngc.nvidia.com

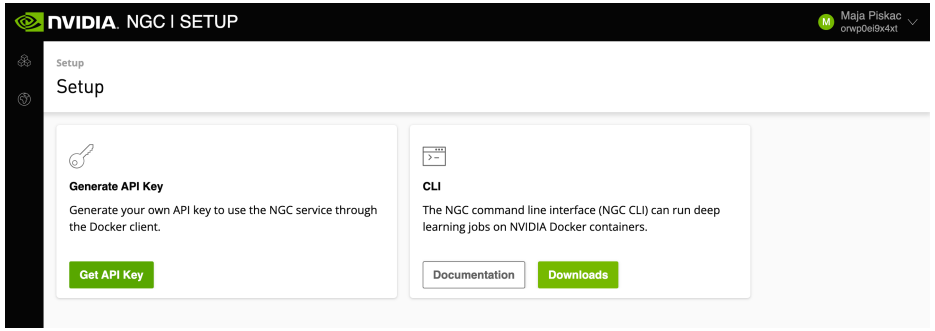


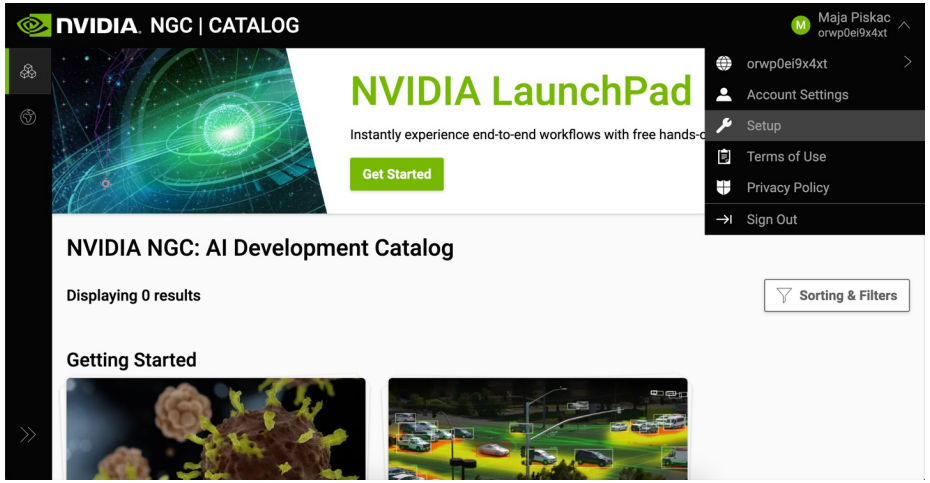
Nvidia NGC Containers

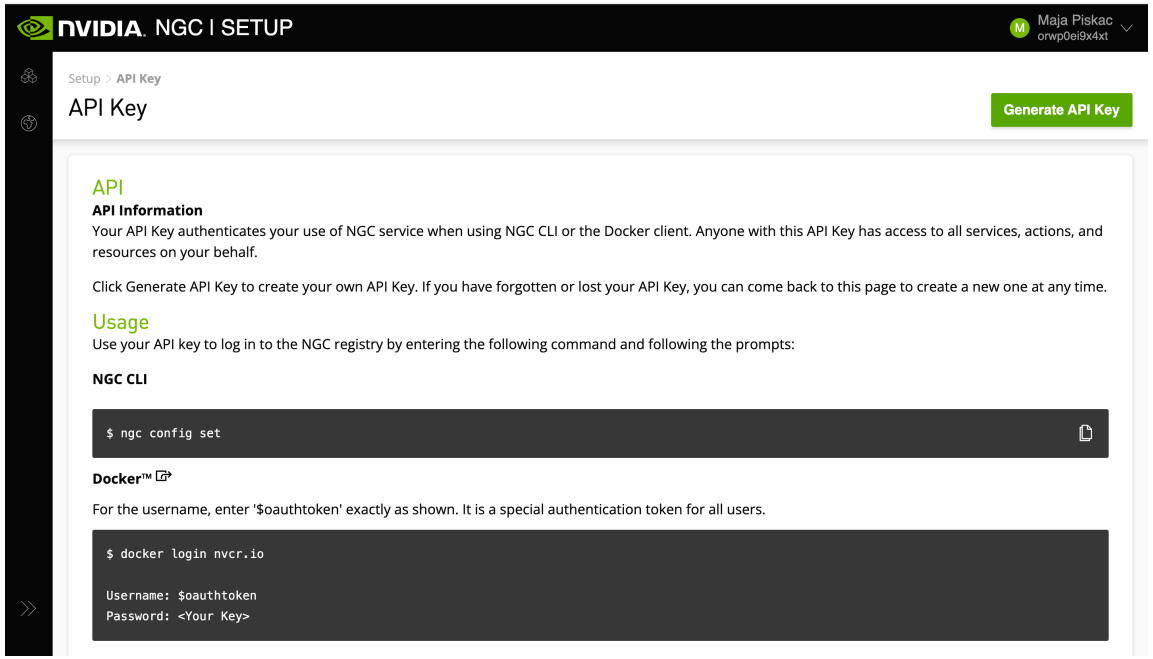




Nvidia NGC Containers – Setting up credentials







Nvidia NGC Containers – Setting up credentials

- Create the file *enroot/.credentials* within your $HOME and insert the following lines in it:

```
machine nvcr.io login $oauthtoken password < KEY>
machine authn.nvidia.com login $oauthtoken password <KEY>
```

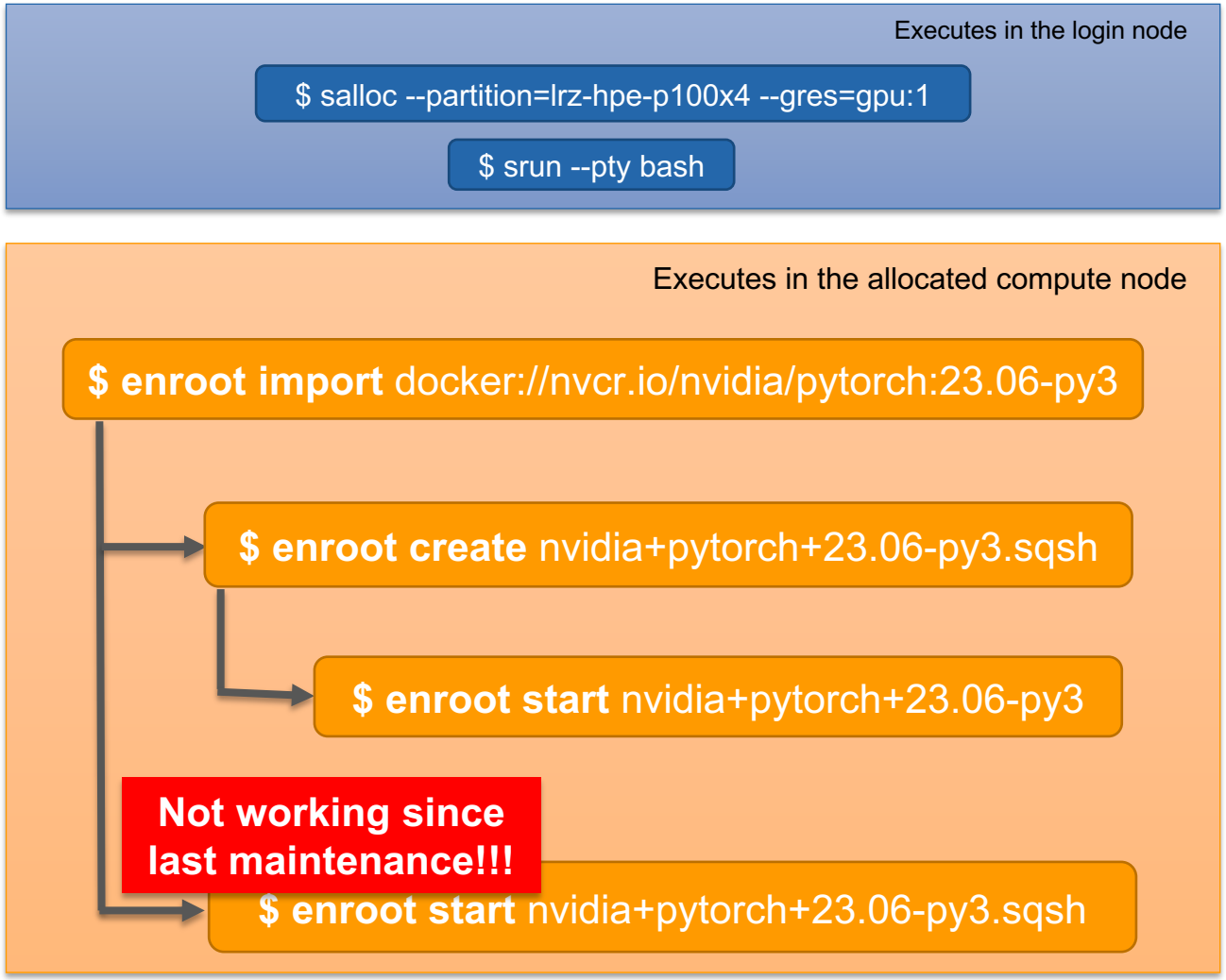
- Where <KEY> is the API key generated and copied in the previous step.
- Introduce a new line after < KEY>.

- Now you can import containers from Nvidia NGC on compute nodes of LRZ AI Systems, e.g. a Pytorch container, with:

```
$ enroot import docker://nvcr.io/nvidia/pytorch:23.06-py3
```

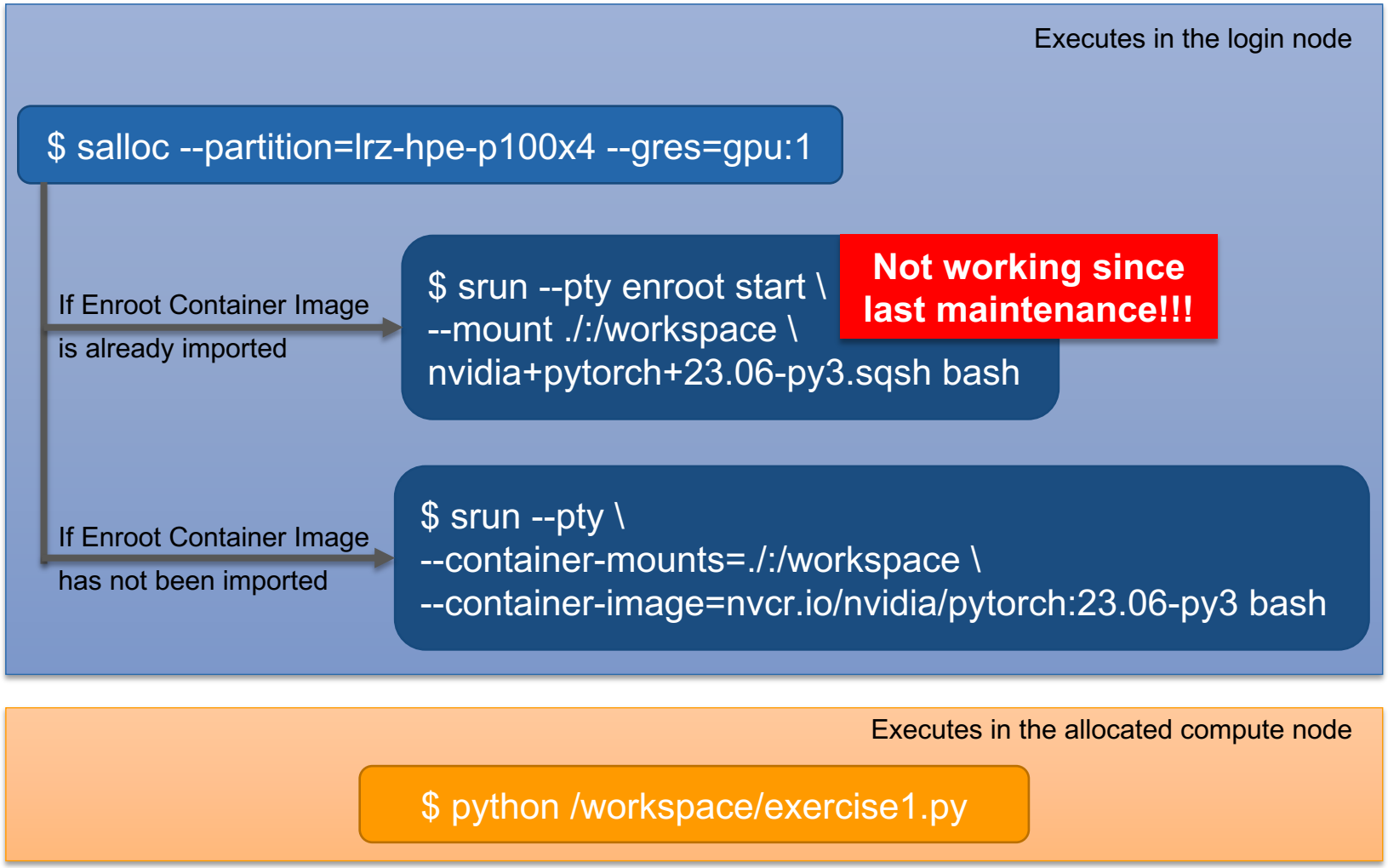
Introduction to Enroot: The Software Stack Provider for the AI Systems

- Enroot container runtime operates completely in user space.
- It allows to run containers defined by container images from the NVIDIA NGC Cloud or from the Docker Hub.
- Not available on the login node, but on the compute nodes!
- The Enroot Workflow:
  I. **Import** an Enroot Container Image – resulting in **sqsh** file
  II. Create an Enroot Container with **create**,
  III. Run software inside an existing Enroot Container with **start**.



Running Applications as Interactive Jobs

- Interactive jobs are submitted to an existing allocation of resources using the **srun** command.
- We can **mount** existing data from outside of the container into container.
- Enroot container creation and job submission in a single step can be done via a plugin called *pyxis.*



Running Applications as Batch Jobs

- Batch jobs are the preferred and quicker way of using the LRZ AI Systems.
- Batch job is queued and executed when the resources are available.
- It does the allocation and running of the job for you (instead of salloc and srun).

- The **sbatch** command submits jobs described in a **sbatch script file**.
- You need to specify the partition and number of GPUs that you want to use.
- Two additional required arguments: *output* and *error messages* file.

```
#!/bin/bash
#SBATCH -p lrz-hpe-p100x4
#SBATCH --gres=gpu:1
#SBATCH -o exercise1.out
#SBATCH -e exercise1.err

srun \
--container-mounts='./:/workspace' \
--container-image='nvcr.io/nvidia/pytorch:23.06-py3' \
python /workspace/exercise1.py
```

Executes in the login node

```
$ sbatch exercise1.sbatch
```

Dealing with base images from catalogues other than NGC

- If your image does not supply the CUDA Toolkit, do not install it within the image, because this fixes paths to the existing NVIDIA driver on the target machine and might crash if the NVIDIA driver is upgraded.
- Instead add the following **environment variables** within the container, and the container runtime will copy within the container the needed libraries. Refer to https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/docker-specialized.html for more information on the accepted values of these variables.

Executes in the allocated resource

```
$ enroot import docker://someone/image-no-cuda
$ enroot create --name base_container someone+image-no-cuda.sqsh
$ enroot start base_container bash
echo "NVIDIA_DRIVER_CAPABILITIES=compute,utility" >> /etc/environment
echo "NVIDIA_REQUIRE_CUDA=cuda>=9.0" >> /etc/environment
echo "NVIDIA_VISIBLE_DEVICES=all" >> /etc/environment
$ exit
$ enroot export --output base_image.sqsh base_container
```

Creating an extended Enroot image

- If your workload depends on a package not provided by the used image.

Executes in the allocated resource

```
$ enroot create --name custom_container nvidia+pytorch+23.06-py3.sqsh
$ enroot start custom_container
$ pip install --no-cache-dir lightning
$ HOROVOD_GPU_OPERATIONS=NCCL pip install --no-cache-dir horovod
$ exit
$ enroot export --output pytorch+lightning+custom.sqsh custom_container
```

- For installing some applications you need to be root within the container (e.g., installing software using the apt package manager in Debian and Ubuntu-based containers.) In this case, add the --root flag.

Executes in the allocated resource

```
$ enroot start --root my_container
$ apt update
$ apt install python3-dev
```

Access to AI Systems through interactive web servers

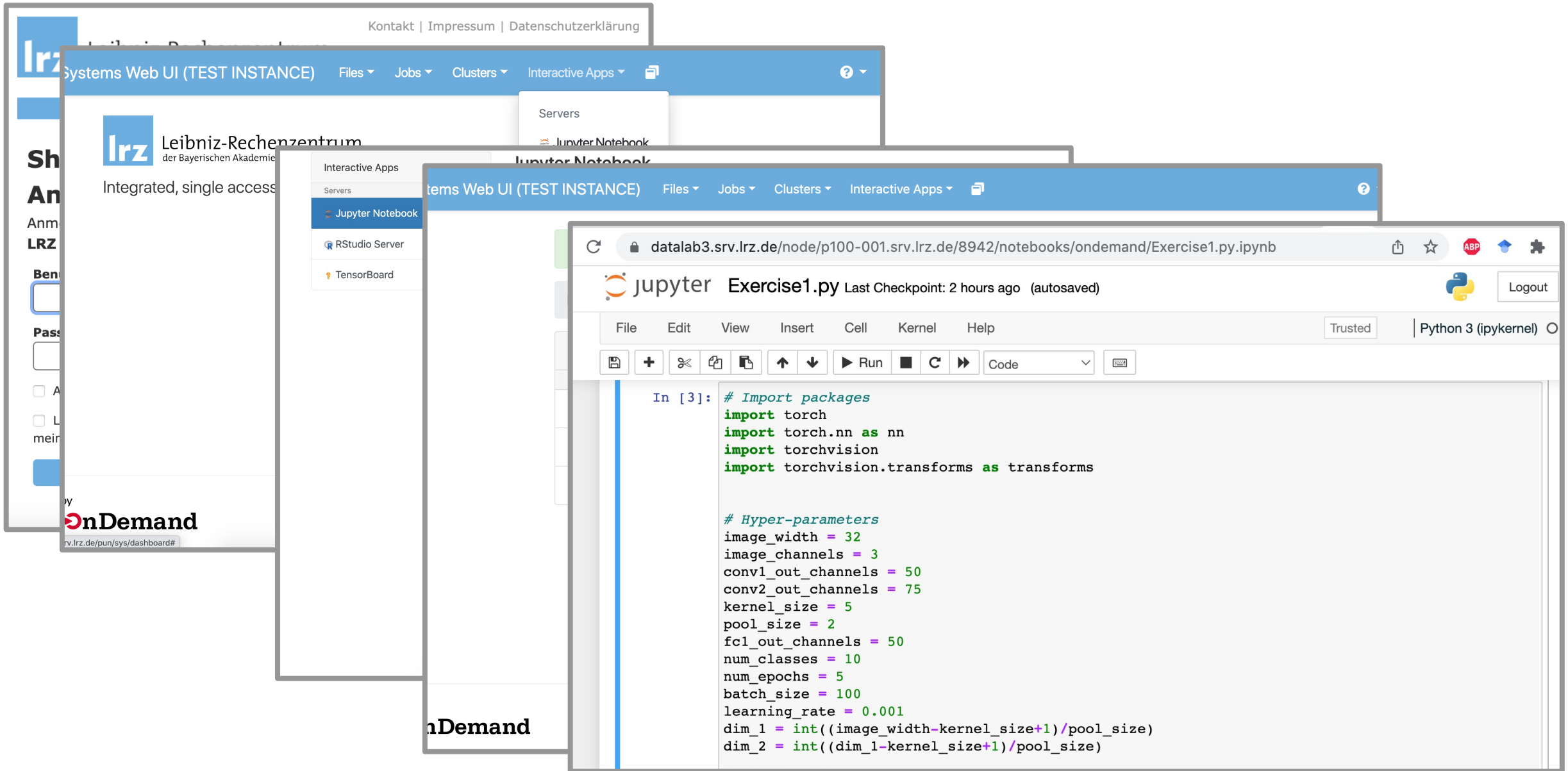
- Jupyter Notebook, JupyterLab, RStudio Server and TensorBoard

- Available at https://login.ai.lrz.de

- To start e.g. a Jupyter Notebook session select from the top panel:
  "Interactive Apps" => “Jupyter Notebook”

- For a typical use-case:
  - select the type of resources (CPU only or CPU + single GPU)
  - specify your workload (a combination of CPU core and RAM requirements)
  - select the container environment you want to work with (e.g. available PyTorch or Tensorflow container, or a custom container)
  - finally, specify the number of hours you plan to work (be aware that your session will be shut down when this time limit is reached, and any unsaved work will then be lost).

Access to AI Systems through interactive web servers



Public Datasets and Containers on the LRZ AI Systems

- Dedicated DSS container for storing public datasets and Enroot container images of interest to researchers.
- Procedure to request the addition of public datasets / Enroot images:
  - make sure the dataset / Enroot image is licensed for public usage and requires no individual license nor registration, and in case of an Enroot image make sure the image is not provided by the Nvidia NGC, Dockerhub or another public repository directly
  - open a ticket with the LRZ Servicedesk, providing the location of the dataset / Dockerfile for building the image, and a justification for public interest (including the expected target audience)
  - provide instructions for downloading the dataset (ideally shell script) / building the image (if non-standard).

Dataset	Location	Version	Licence
AlphaFold	/dss/dssfs04/pn69za/pn69za-dss-0004/datasets/alphafold_2024	Last update March 2024 following the instructions here https://github.com/deepmind/alphafold#genetic-databases	https://github.com/deepmind/alphafold#license-and-disclaimer
COCO-Stuff	dss/dssfs04/pn69za/pn69za-dss-0004/datasets/cocostuff/	2020 Update (train2017.zip, val2017.zip, annoations_trainval2017.zip, stuff_annotations_trainval2017.zip)	https://cocodataset.org/#termsofuse
Visual Genome	dss/dssfs04/pn69za/pn69za-dss-0004/datasets/visualgenome/	Version 1.4 of dataset completed as of July 2017	Creative Commons Attribution 4.0 International License

Summary

Command line

```
$ ssh -l xxyyyzz login.ai.lrz.de
```

Interactive job

Executes in the login node login-1

```
$ sinfo
```

```
$ salloc --partition=dgx-1-p100 --gres=gpu:1
```

```
$ srun --pty --container-mounts=./:/workspace \
--container-image=nvcr.io/nvidia/pytorch:23.03-py3 bash
```

Executes in the allocated resource

```
$ python /workspace/exercise1.py
```

Batch job

```
#!/bin/bash
#SBATCH -p lrz-dgx-a100-80x8
#SBATCH --gres=gpu:1
#SBATCH -o exercise1.out
#SBATCH -e exercise1.err

srun --container-mounts='./:/workspace' \
--container-image='nvcr.io/nvidia/pytorch:23.06-py3' \
python /workspace/exercise1.py
```

Executes in the login node datalab2

```
$ sbatch exercise1.sbatch
```

Web browser

https://login.ai.lrz.de

https://doku.lrz.de/display/PUBLIC/LRZ+AI+Systems

Exercise: Start an interactive job and create an Enroot container

1. Ssh into the login node login.ai.lrz.de and clone the exercises from the GitHub repo https://github.com/LRZ-BADW/ai-systems.git
   Alternatively, you could download from https://doku.lrz.de/display/PUBLIC/AI+Infrastructure+Material.

2. Allocate and run a job with 1 GPU.

3. Import a Pytorch container image from NGC (or Pytorch Docker container), create, extend and start the container.

Agenda

1. Introduction to the LRZ AI Systems

- ✓ Overview of the LRZ AI Systems
- ✓ Access to the LRZ AI Systems
- ✓ NVIDIA NGC Cloud
- ✓ Introduction to Enroot Containers
- ✓ Interactive and Batch Jobs
- ✓ Open on Demand
- ✓ Exercise: Run a job with an Enroot container

2. Fundamentals of Deep Learning

- ❑ Introduction to Convolutional Neural Networks
- ❑ Exercises: Train CNNs on a GPU
- ❑ Introduction to Transformers
- ❑ Exercise: Train a Transformer on a GPU
- ❑ Introduction to Reinforcement Learning
- ❑ Exercise: Reinforcement Learning

3. Distributed Training of Neural Networks

- ❑ Data Parallel Training
- ❑ Exercise: DP Training of CNN on 2 GPUs
- ❑ Model Parallel Training: Pipeline Parallel and Tensor Parallel
- ❑ Exercise: PP Training of Transformer on 2 GPUs

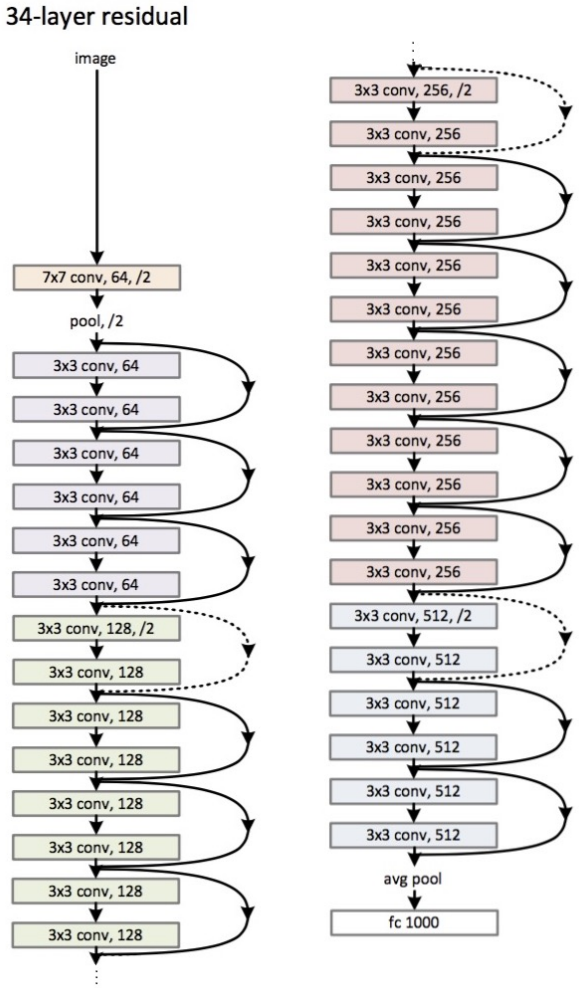
Convolutional Neural Network Architectures – VGG (Visual Geometry Group)

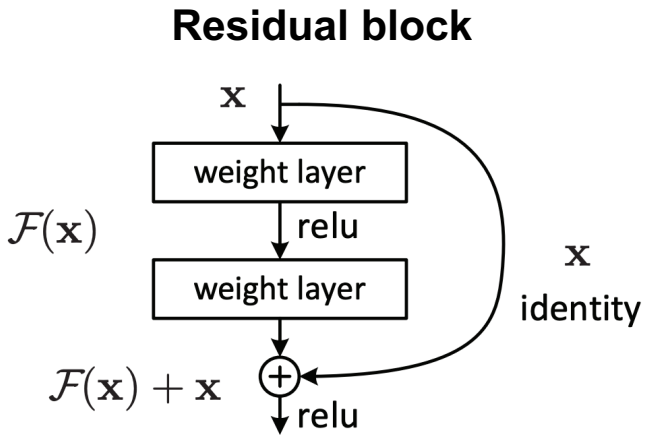
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

- Main idea: increasing depth of CNNs
- 3 × 3 conv. layers with a stride of 1 - small receptive fields (compared to prev. 11 × 11 with a stride of 4 in AlexNet)
- 1 × 1 conv. to make the decision function more non-linear without changing the receptive fields
- ReLU activation function
- ImageNet dataset

https://arxiv.org/pdf/1409.1556.pdf

Convolutional Neural Network Architectures – ResNet

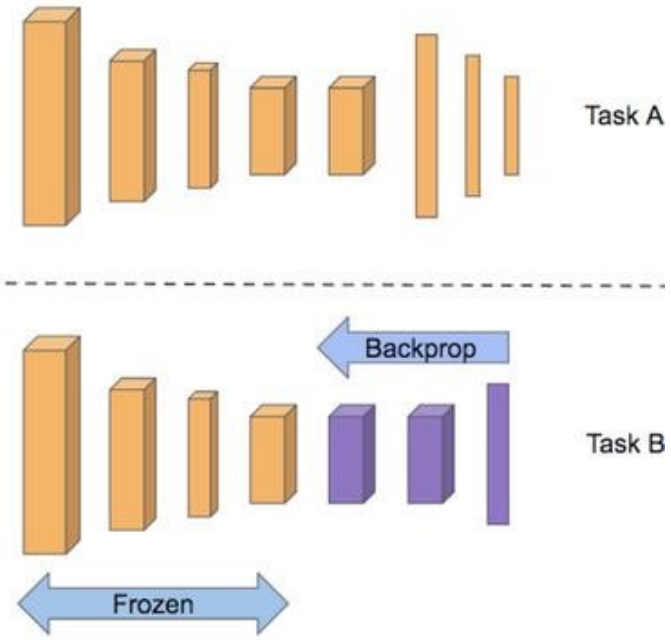




- Main idea: “identity shortcut connection”
- Avoids vanishing gradient problem
- The network gets the input along with the learning on the residual and if the input function was the appropriate function, it can change the weights of the residual function to be zero.
- ImageNet dataset

https://arxiv.org/pdf/1512.03385.pdf

Transfer learning – Pretrained Convolutional Neural Networks



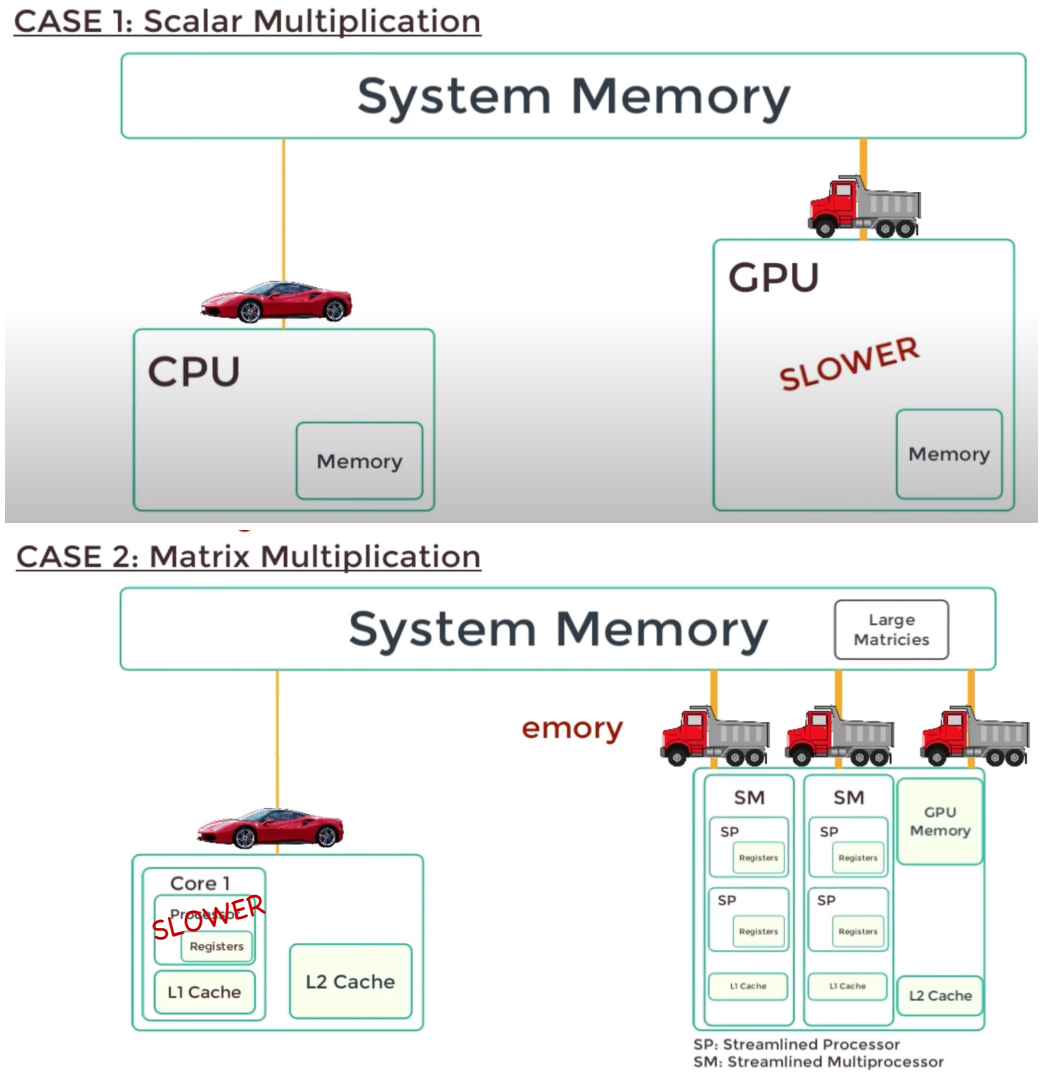
- Pre-trained model is a saved network that was previously trained on a large dataset.
- Feature Extraction: Use the feature maps from the pre-trained model to detect features in the new samples. Add a new classifier, which will be trained from scratch to make predictions.
- Fine-Tuning: Unfreeze a few top layers of a pretrained model and jointly train both the newly-added classifier layers and the last layers of the base model. This allows us "fine-tune" the higher-order feature maps to make them more relevant for the specific task.

```
import torchvision.models as models
model = models.resnet34(weights='IMAGENET1K_V1')
```

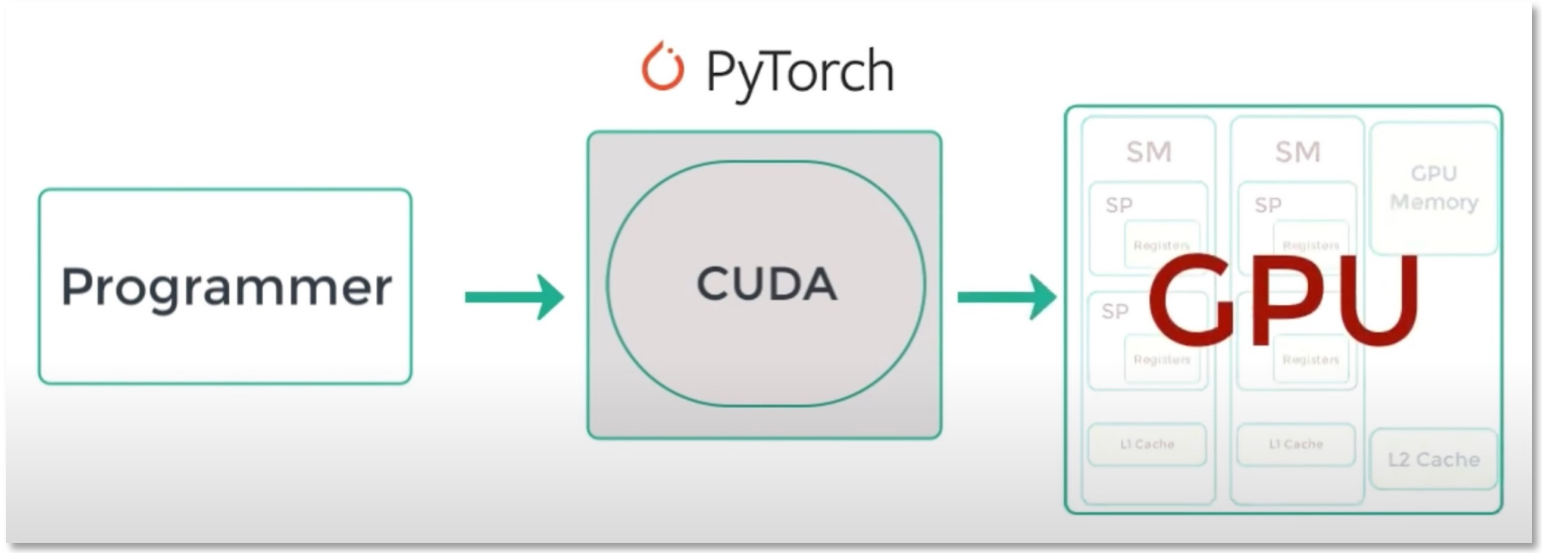
```
import torchvision.models as models
model = models.vgg19(weights='IMAGENET1K_V1')
```

Neural Networks and GPUs - Why?

- GPUs in comparison to CPUs:
  - GPU allows parallel running of repetitive calculations within an application
  - CPU can be thought of as the taskmaster of the entire system, coordinating a wide range of general-purpose computing tasks
  - GPU performs a narrower range of more specialized tasks (e.g., matrix multiplications)
- CPUs are faster than GPUs in scalar multiplications.
- GPUs are faster than CPUs in matrix multiplications.



Neural Networks and GPUs - How?



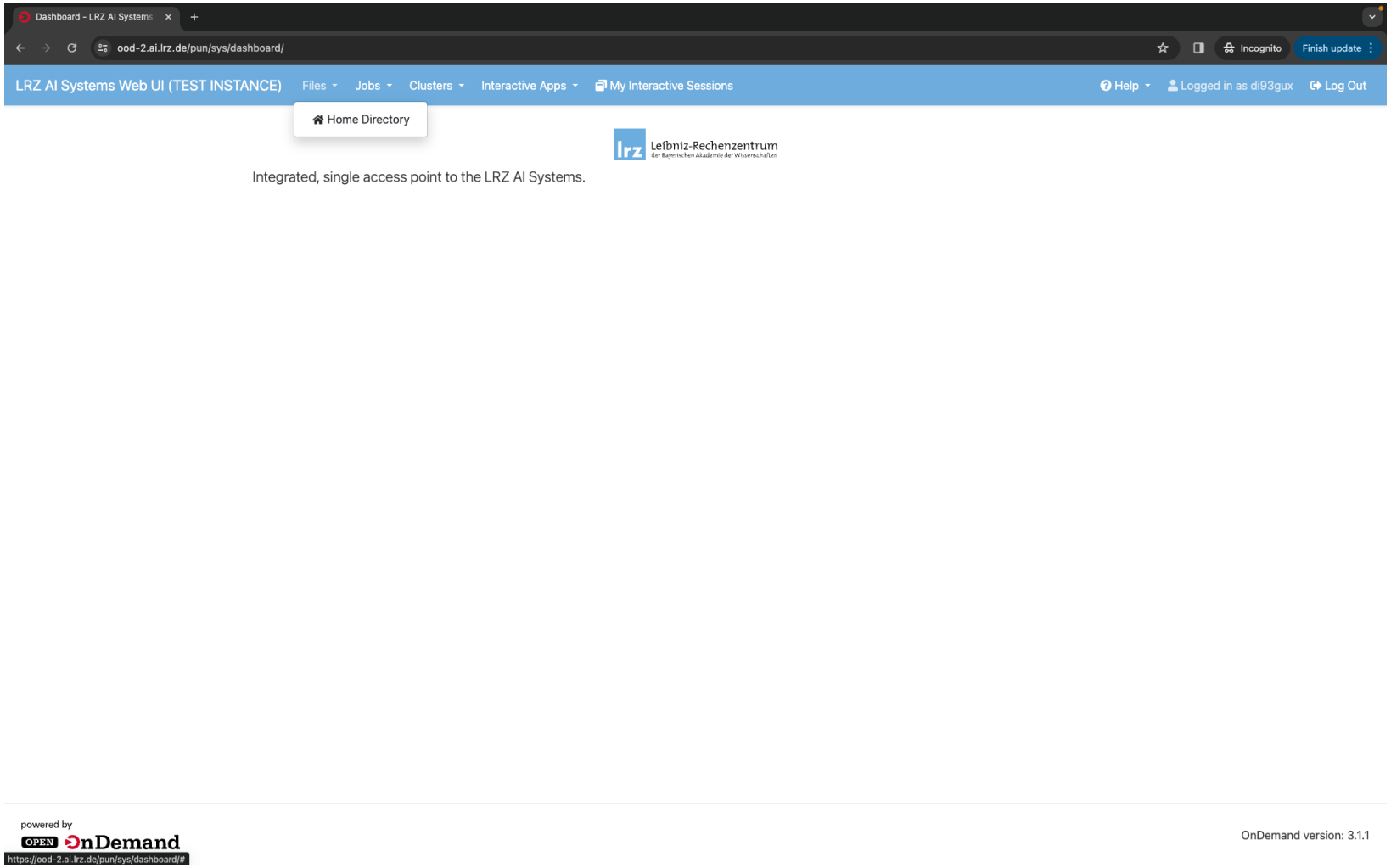

```
# Device configuration - cpu
device = torch.device('cpu')

# Device configuration - gpu
device = torch.device('cuda')

# Model to device
model = model.to(device)

# Data to device
images = images.to(device=device)
labels = labels.to(device=device)
```

CNN Exercises in Pytorch

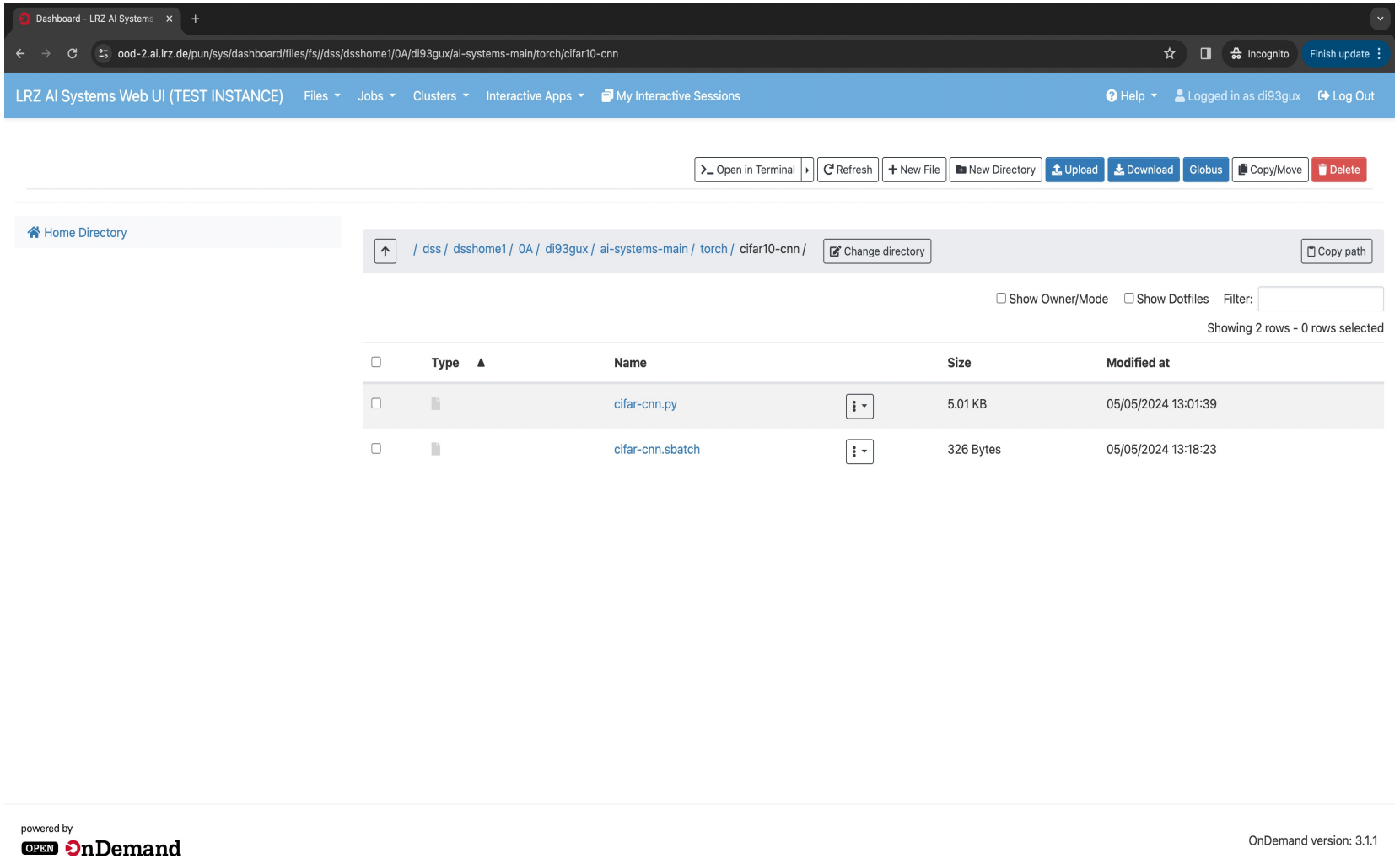
Exercise: Run CNN exercise in Pytorch

- Run exercises cifar-cnn.sbatch as batch jobs.
  - https://login.ai.lrz.de

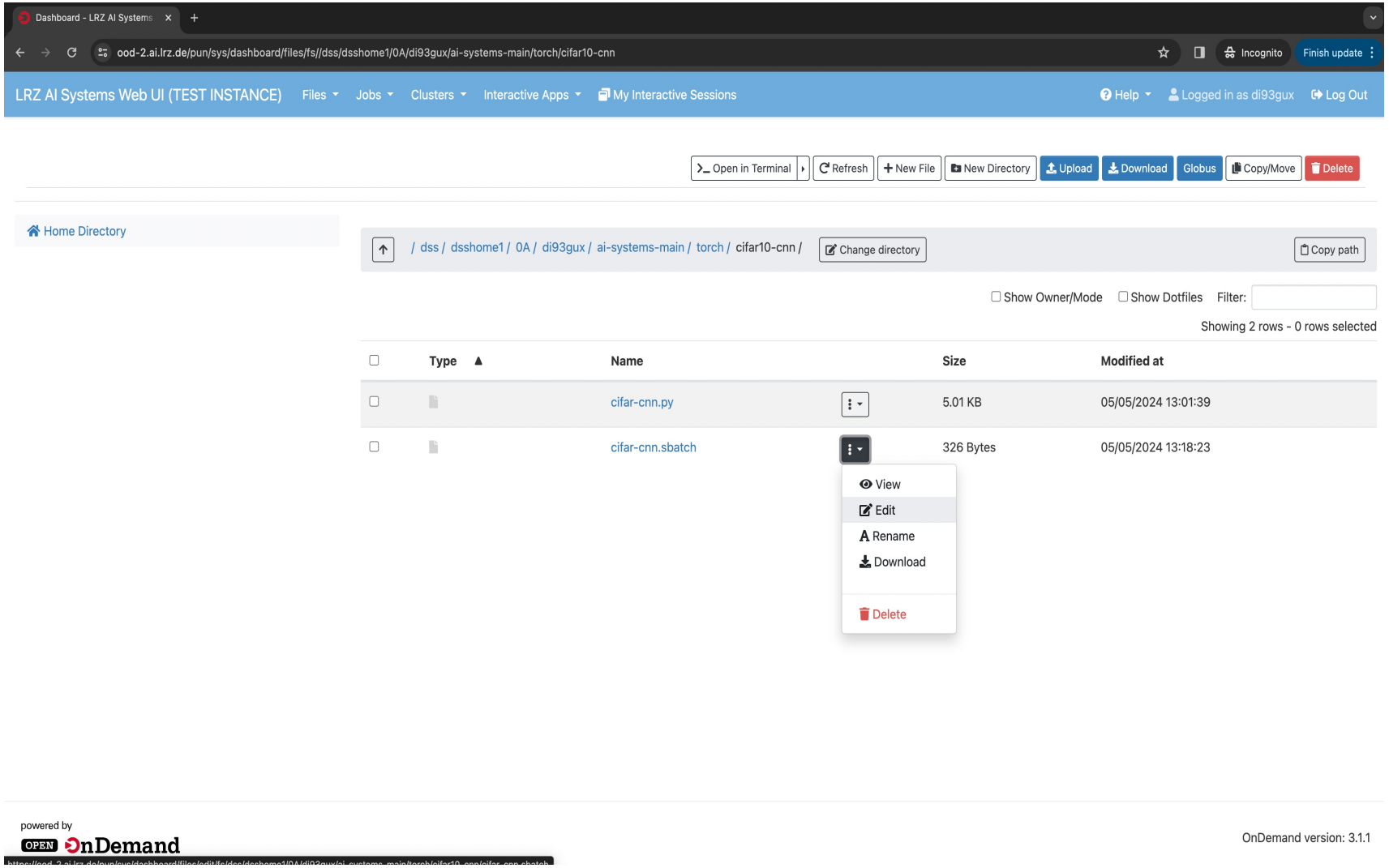
Exercise: Run CNN exercise in Pytorch



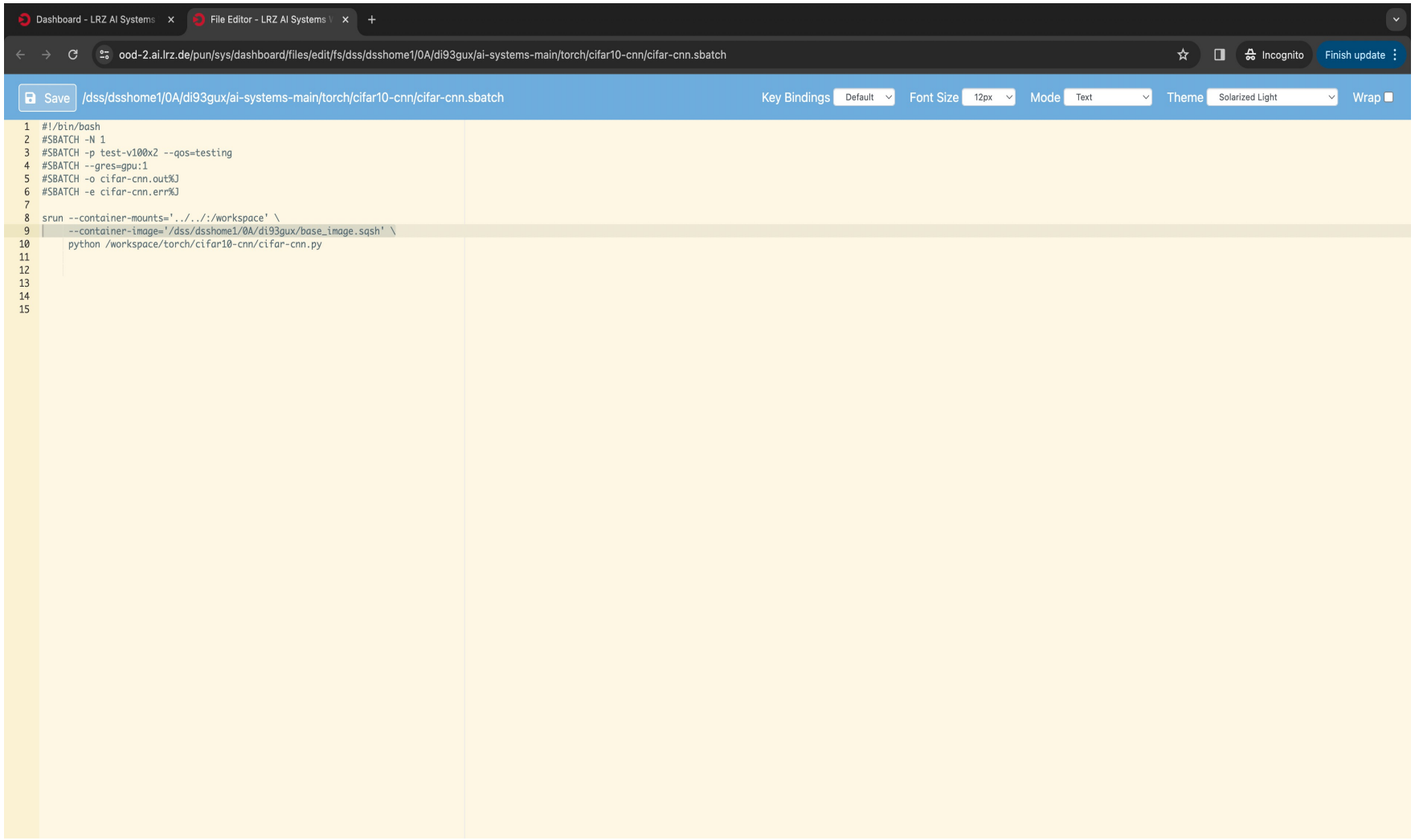
Exercise: Run CNN exercise in Pytorch



Exercise: Run CNN exercise in Pytorch



Exercise: Run CNN exercise in Pytorch



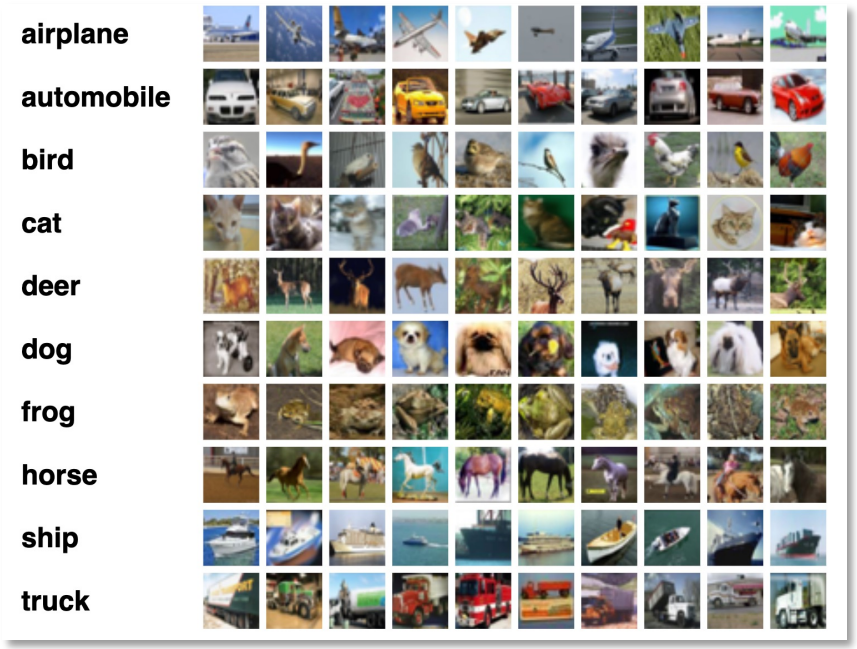
Exercise: Run CNN exercise in Pytorch

- Run exercises cifar-cnn.sbatch as batch jobs.

```
di93gux@login-03:~/ai-systems-main/torch/cifar10-cnn$ sbatch cifar-cnn.sbatch
Submitted batch job 308338
```

CIFAR10 Data

```
# download the data
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, transform=transforms.ToTensor(), download=True)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, transform=transforms.ToTensor(), download=True)

# transform to DataLoader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```



Input size: 32x32x3

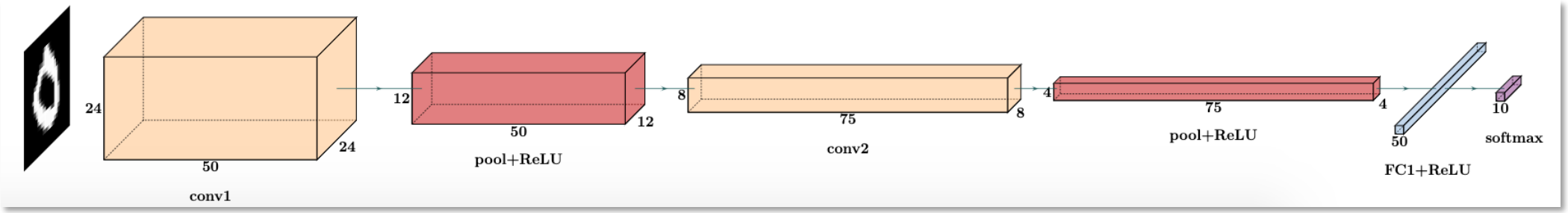
https://www.cs.toronto.edu/~kriz/cifar.html

Simple Convolutional Neural Network

```
class Model(nn.Module):
    def __init__(self, image_width, image_channels, num_classes):
        super().__init__()
        self.conv1 = nn.Conv2d(image_channels, conv1_out_channels, kernel_size)
        self.conv2 = nn.Conv2d(conv1_out_channels, conv2_out_channels, kernel_size)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(pool_size)
        self.fc1 = nn.Linear(conv2_out_channels*(dim_2**2), fc1_out_channels)
        self.fc2 = nn.Linear(fc1_out_channels, num_classes)

    def forward(self, x):
        x = self.relu(self.pool(self.conv1(x)))
        x = self.relu(self.pool(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
# Hyper-parameters
image_width = 28
image_channels = 1
conv1_out_channels = 50
conv2_out_channels = 75
kernel_size = 5
pool_size = 2
fc1_out_channels = 50
num_classes = 10
num_epochs = 5
batch_size = 100
learning_rate = 0.001
```

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```



Checkpointing

- Save and load a checkpoint for inference or resuming training.
- When saving a general checkpoint, you must save:
  - model's state_dict
  - optimizer's state_dict
  - optional: last epoch, training loss, etc.
- Steps:
  1. Import all necessary libraries
  2. Define and initialize the neural network
  3. Initialize the optimizer
  4. Save the general checkpoint
  5. Load the general checkpoint
- To save multiple checkpoints, you must organize them in a dictionary.

```
### 4. Save the general checkpoint

EPOCH = 5
PATH = "model.pt"

torch.save({
    'epoch': EPOCH,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    }, PATH)
```

```
### 5. Load the general checkpoint

checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']

# resume training
model.train()
```

https://pytorch.org/tutorials/recipes/recipes/saving_and_loading_a_general_checkpoint.html

Which filesystem to use for checkpointing ?

- Home directory
  - Not suitable for heavy and/or high-frequency I/O operations

- DSS (Data Science Storage)
  - high-bandwidth
  - low latency I/O operations

How to modify to the path for saving check point to DSS?

cifar-cnn.sbatch

Original Code

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -p test-v100x2 --qos=testing
#SBATCH --gres=gpu:1
#SBATCH -o cifar-cnn.out%J
#SBATCH -e cifar-cnn.err%J

srun --container-mounts='../../:/workspace' \
     --container-image='/dss/dsshome1/0A/di93gux/base_image.sqsh' \
     python /workspace/torch/cifar10-cnn/cifar-cnn.py
```

How to modify to the path for saving check point to DSS?

cifar-cnn.sbatch

Modified Code

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -p test-v100x2 --qos=testing
#SBATCH --gres=gpu:1
#SBATCH -o cifar-cnn.out%J
#SBATCH -e cifar-cnn.err%J

srun --container-mounts='../../:/workspace',/dss/dssfs04/pn72ka/pn72ka-dss-0000/navdar:/dss_data \
     --container-image='/dss/dsshome1/0A/di93gux/base_image.sqsh' \
     python /workspace/torch/cifar10-cnn/cifar-cnn.py
```

How to modify to the path for saving check point to DSS?

cifar-cnn.py

Original Code

```
# Additional information for checkpointing
PATH = "/workspace/torch/cifar10-cnn/cnn_model.pt"
```

Modified Code

```
# Define the directory outside of the container for saving files
save_dir = "/dss_data"

# Additional information for checkpointing
PATH = os.path.join(save_dir, "model_exercise1.pt")
```

PyTorch Lightning

- Wrapper library for PyTorch
- Helps organize raw PyTorch code & improves readability
- Provides maximum flexibility without sacrificing performance at scale

PyTorch Lightning

- Models → LightningModule - acts as a model "recipe" that specifies all training details

- Training → Lightning Trainer - handles all engineering to scale models

GPU Training

```
# Device configuration - cpu
device = torch.device('cpu')

# Device configuration - gpu
device = torch.device('cuda')

# Model to device
model = model.to(device)

# Data to device
images = images.to(device=device)
labels = labels.to(device=device)
```

PyTorch Lightning

```
# Device configuration - cpu
accelerator = 'cpu'

# Device configuration - gpu
accelerator = 'gpu'

# Trainer automatically handles all .to(device) calls
trainer = pl.Trainer(accelerator=accelerator)
```

Precision

```
# model to half precision
model = model.to(device).half()

# Data to half precision
images = images.to(device=device).half()
labels = labels.to(device=device)
```

PyTorch Lightning

```
# 32-bit precision
precision = '32'

# 16-bit precision
precision = '16'

# Trainer automatically handles all .half() calls
trainer = pl.Trainer(precision= precision)
```

Checkpointing - PyTorch Lightning

```
### 4. Save the general checkpoint

EPOCH = 5
PATH = "model.pt”

torch.save({
    'epoch': EPOCH,
    'model_state_dict’: model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    }, PATH)
```

```
### 5. Load the general checkpoint

checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch’]

# resume training
model.train()
```

```
### Save the general checkpoint

checkpoint_callback = ModelCheckpoint(dirpath='/path/to/checkpoint’,
                                      filename=‘model-{epoch}’,)

trainer = Trainer(callbacks=[checkpoint_callback])

# Load model from checkpoint
model = MyLightningModule.load_from_checkpoint("/path/to/checkpoint/model.ckpt")

# resume training from checkpoint
trainer = pl.Trainer()
trainer.fit(model, ckpt_path=/path/to/checkpoint/model.ckpt)
```

exercise1.py: Simple Convolutional Neural Network on a CPU vs on a GPU (191k params)

```
Time for training on cpu:  129.9463222026825
```

Test metric	DataLoader 0
test_acc_epoch	0.6310999989509583
test_loss_epoch	1.0603944063186646

```
Time for training on gpu:  52.2851619720459
```

Test metric	DataLoader 0
test_acc_epoch	0.6330000162124634
test_loss_epoch	1.0520167350769043

exercise1_CNN.py: Pretrained ResNet34 on CIFAR10 data on a CPU vs on a GPU (21.8M parameters)

```
Time for training resnet on cpu:  509.6669270992279
```

Test metric	DataLoader 0
Test Acc	0.6664999723434448
Test loss	1.0169105529785156

```
Time for training resnet on gpu:  116.25382375717163
```

Test metric	DataLoader 0
Test Acc	0.6812999844551086
Test loss	0.9379943609237671

exercise1_CNN.py: Pretrained VGG19 on CIFAR10 data on a CPU vs on a GPU
(143.7M parameters)

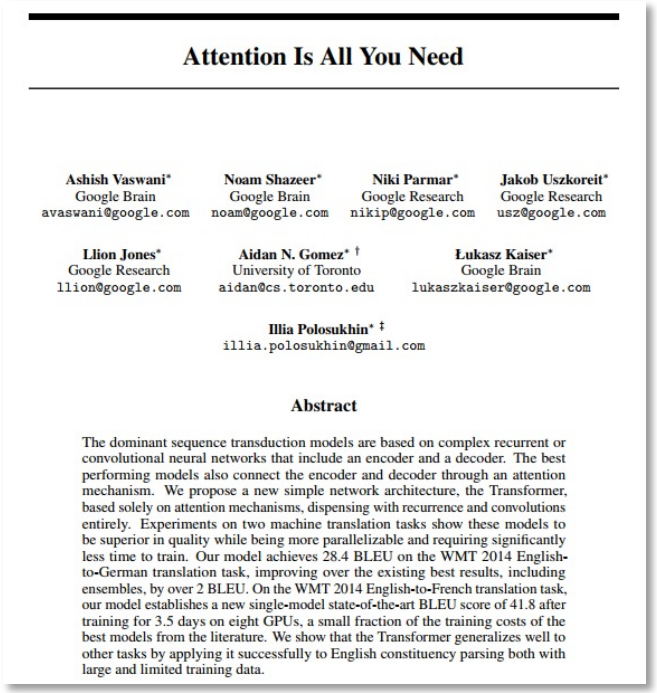
```
Time for training vgg19 on cpu:  2580.444151163101
```

Test metric	DataLoader 0
Test Acc	0.10000000149011612
Test loss	2.3030905723571777

```
Time for training vgg19 on gpu:  127.65016007423401
```

Test metric	DataLoader 0
Test Acc	0.10000000149011612
Test loss	2.303698778152466

Transformers

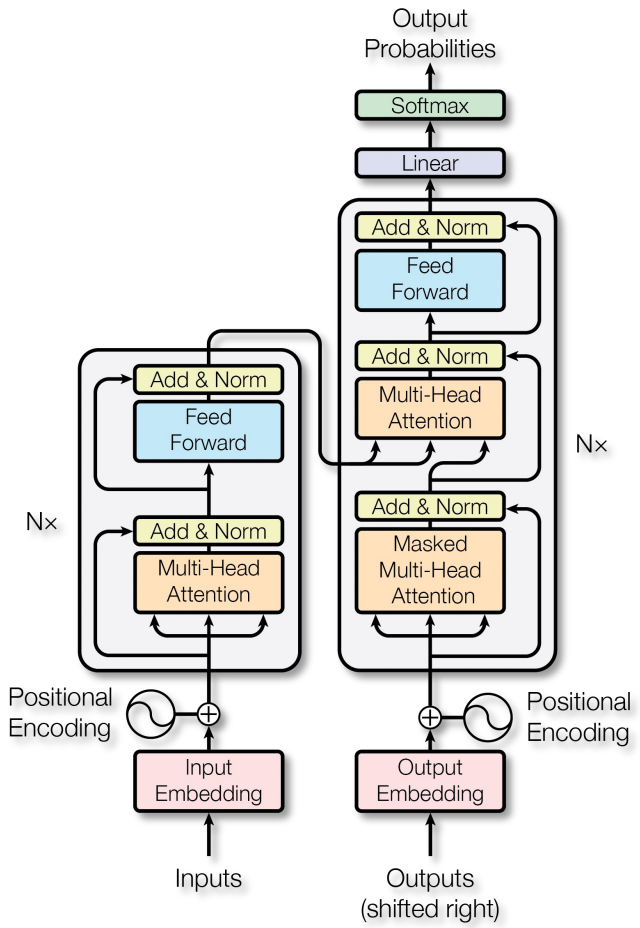


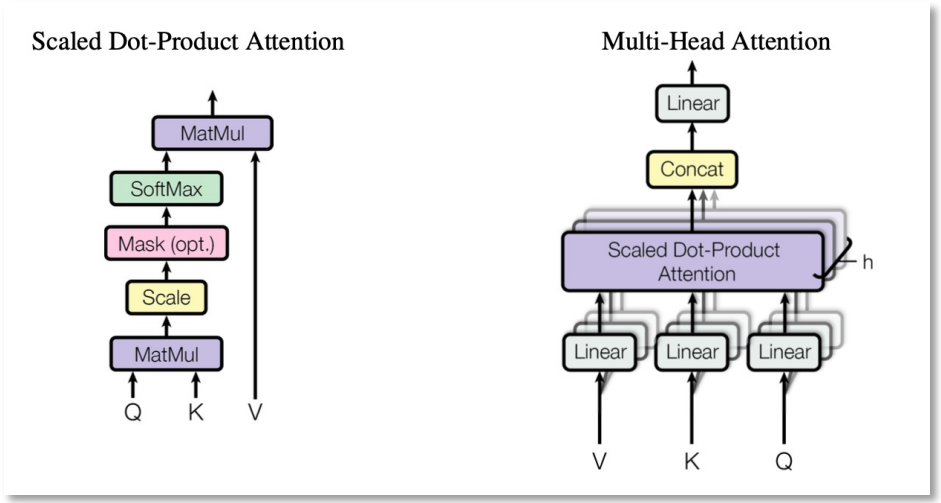
Attention Is All You Need

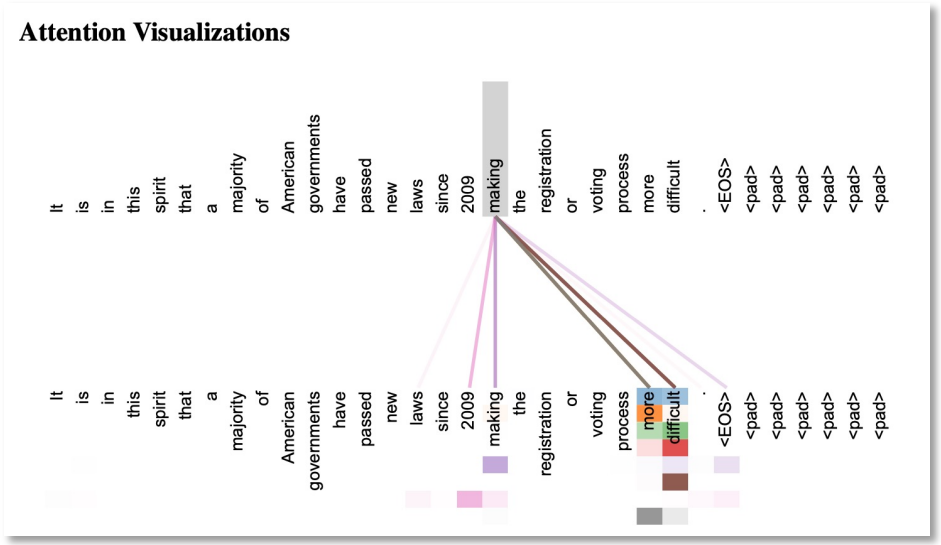
Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

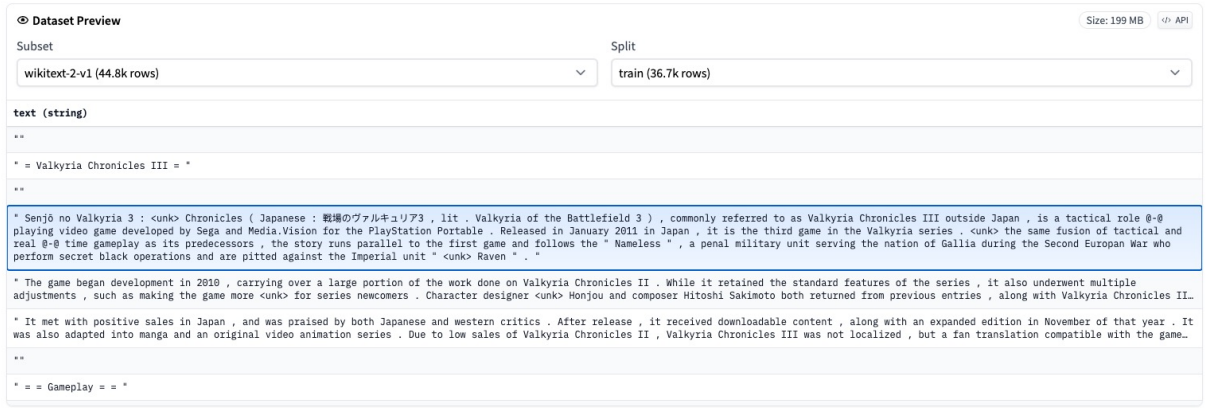


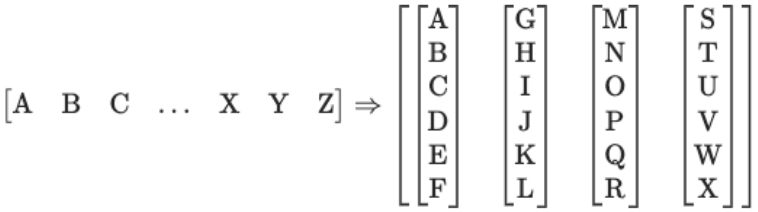


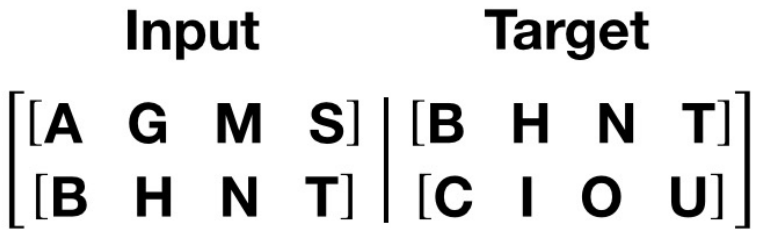


WikiText-2 Data

- torchtext development is stopped and it is not possible to download WikiText-2 dataset using torchtext library anymore, therefore we provide you with the raw data and data.py script to pre-process the data!
- WikiText language modelling dataset is a collection of tokens extracted from the set of verified good and featured Wikipedia articles.
- Number of lines per split:
  - train 36718, valid 3760, test 4358.
- Given a 1-D vector of sequential data, function **batchify()** arranges it into batch_size columns and trims to fit. Batching enables more parallelizable processing, however the model treats each column independently, e.g., the dependence of G and F on the example on the right.
- Function **get_batch()** generates a pair of input-target sequences for the transformer model. It subdivides the source data into chunks of length bptt, e.g., with bptt=2, we'd get the input and target as shown on the right.







https://pytorch.org/tutorials/beginner/transformer_tutorial.html

Transformer Exercise

```
class TransformerModel(nn.Module):
    def __init__(self, ntoken, d_model, nhead, d_hid, nlayers, dropout):
        super().__init__()
        self.embedding = nn.Embedding(ntoken, d_model)
        self.d_model = d_model
        self.pos_encoder = PositionalEncoding(d_model, dropout)
        encoder_layers = nn.TransformerEncoderLayer(d_model, nhead, d_hid, dropout)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layers, nlayers)
        self.linear = nn.Linear(d_model, ntoken)

    def forward(self, src, src_mask):
        """ src: Tensor, shape ``[seq_len, batch_size]``
            src_mask: Tensor, shape ``[seq_len, seq_len]``
            output: Tensor, shape ``[seq_len, batch_size, ntoken]`` """
        src = self.embedding(src) * math.sqrt(self.d_model)
        src = self.pos_encoder(src)
        src_mask = nn.Transformer.generate_square_subsequent_mask(len(src)).to(device)
        output = self.transformer_encoder(src, src_mask)
        output = self.linear(output)
```

- Training a sequence-to-sequence model based on **nn.Transformer**.
- The nn.Transformer module is modularized – a single component, like TransformerEncoder can be easily adapted / composed.
- We train a **nn.TransformerEncoder** to assign a probability for the likelihood of a given word (or a sequence of words) to follow a sequence of words.
- Square **attention mask** is required with the input sequence, because the self-attention layers are only allowed to see the earlier positions in the sequence.
- **PositionalEncoding** class is defined to use sine and cosine functions of different frequencies to inject the information on the position of the tokens in the input sequence.

exercise1_nlp.py: Transformer on a CPU vs on a GPU (77,879,406)

- Embedding dimension of 1024, hidden size of 1024, 4 attention heads and 3 transformer layers.

Output:

Device: cpu

| end of epoch 1 | time: 1618.58s | valid loss 5.75 | valid ppl 313.30
| end of epoch 2 | time: 1635.81s | valid loss 5.69 | valid ppl 296.32
| end of epoch 3 | time: 1493.84s | valid loss 5.57 | valid ppl 263.73
| End of training | test loss 5.48 | test ppl 240.33

(4748.23 s ≈ **80 min**)

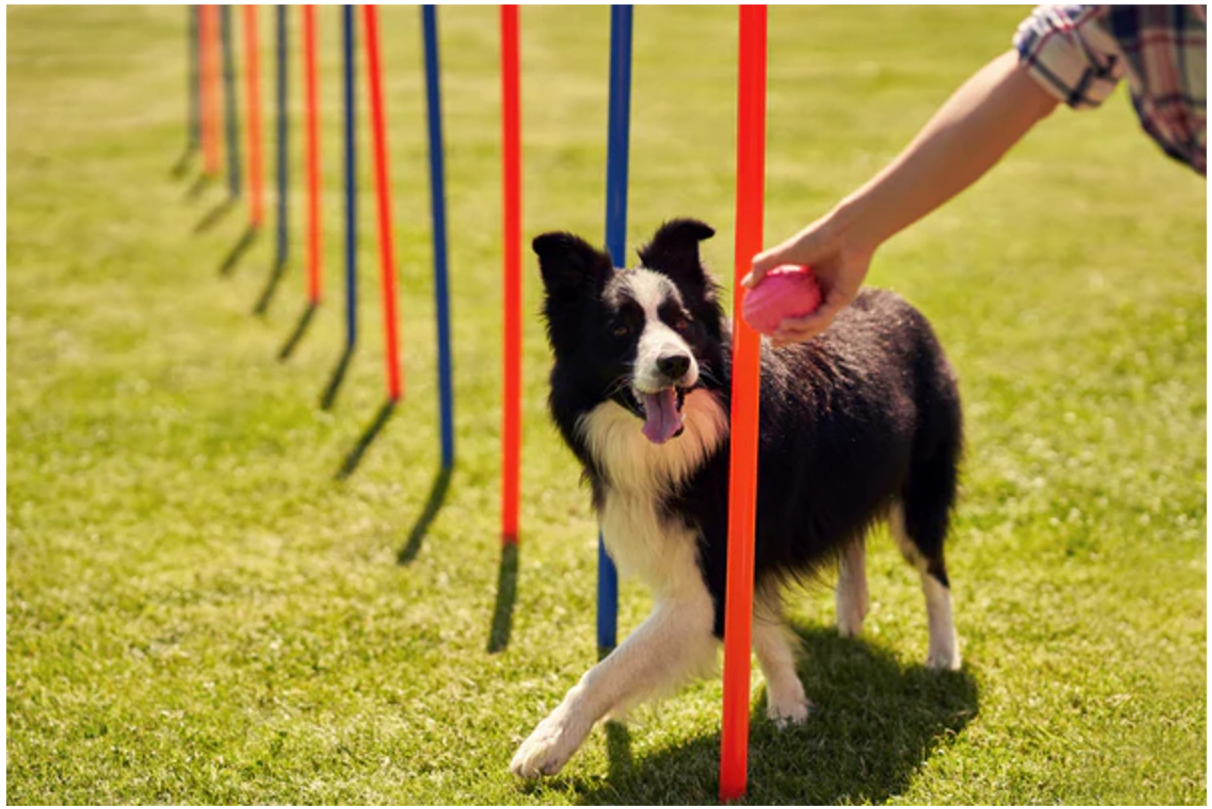
Output:

Device: cuda

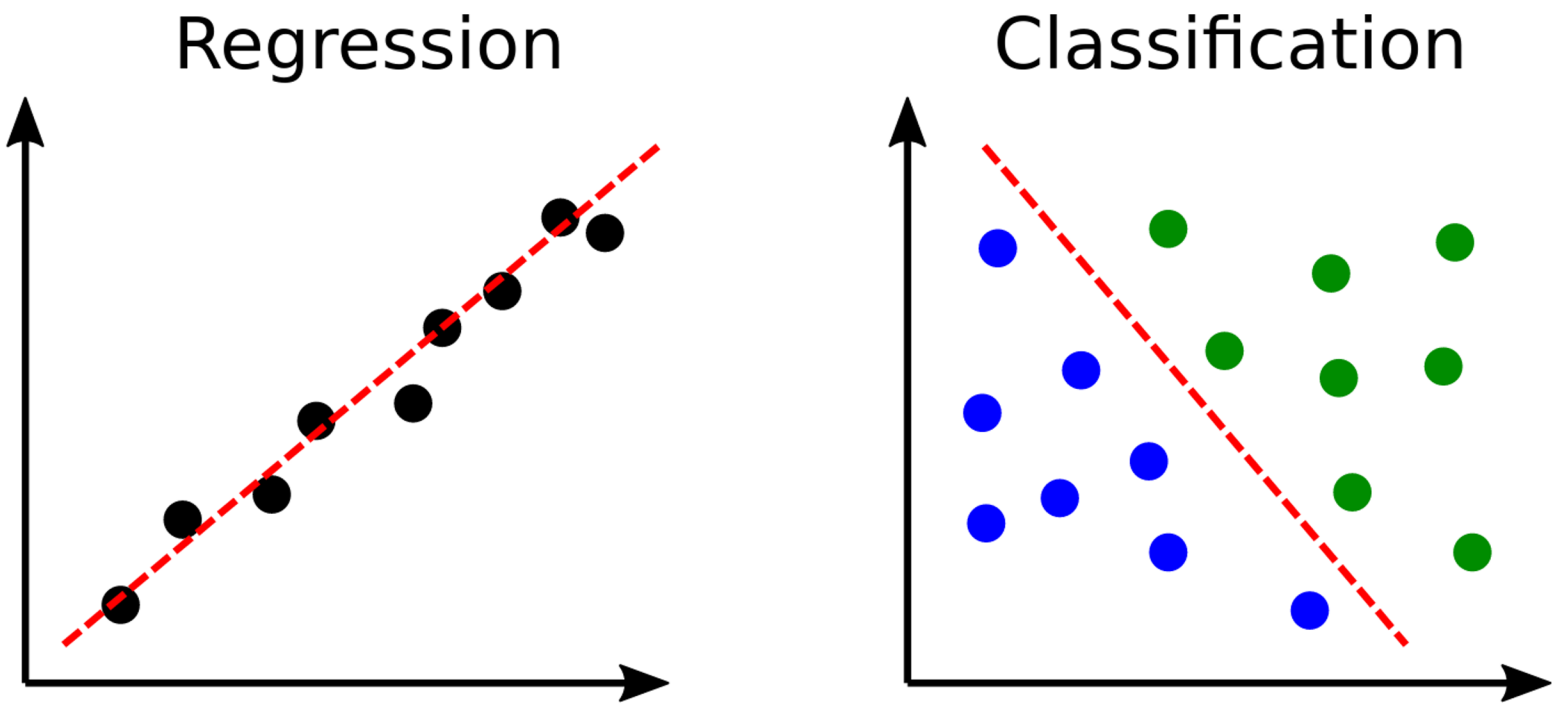
| end of epoch 1 | time: 79.06s | valid loss 5.75 | valid ppl 314.83
| end of epoch 2 | time: 70.25s | valid loss 5.58 | valid ppl 265.83
| end of epoch 3 | time: 69.92s | valid loss 5.49 | valid ppl 242.84
| End of training | test loss 5.40 | test ppl 221.61

(219.23 s ≈ **3.7 min**)

Reinforcement Learning



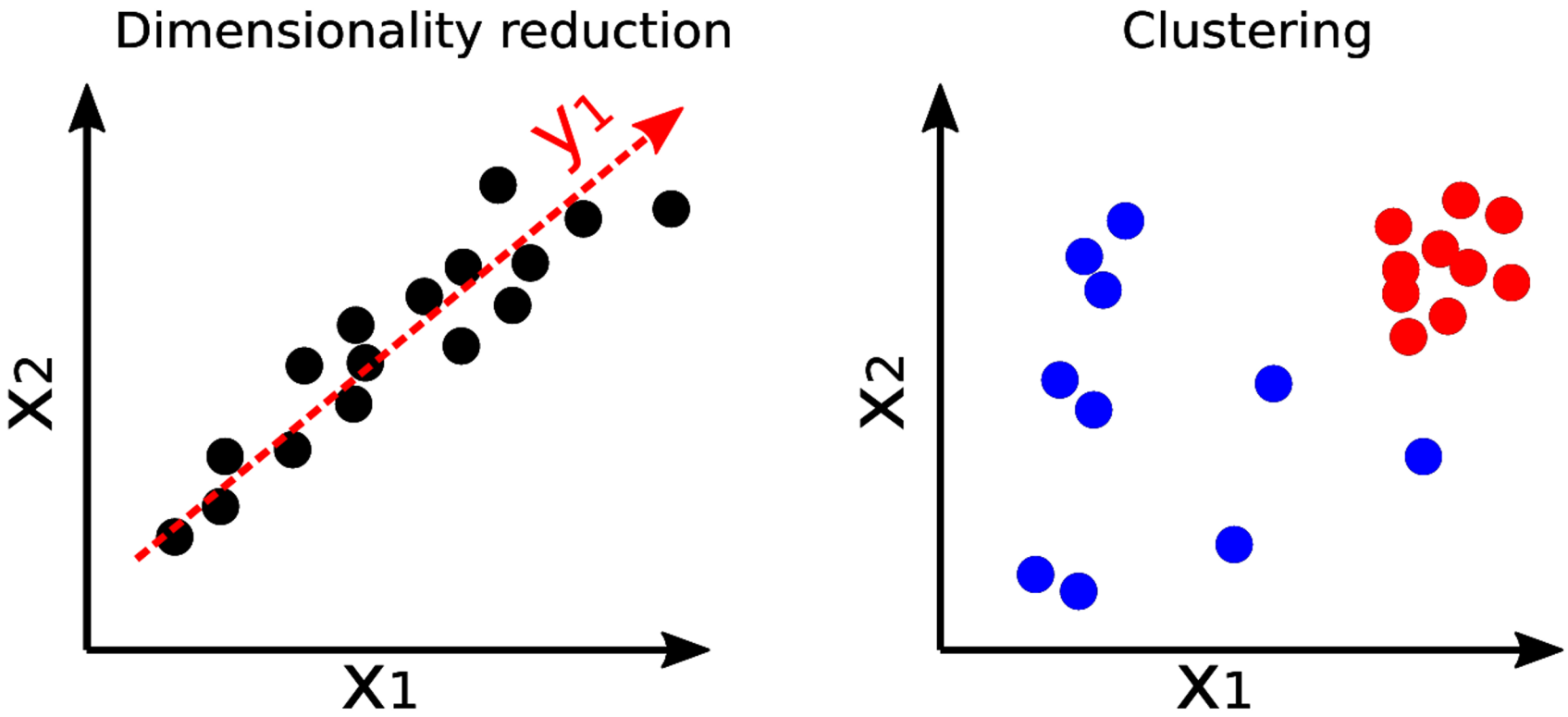
Supervised Learning



Creating mapping from features to labels based on data.

Source : https://andreweiner.github.io/reveal.js/of_conf_2021.html

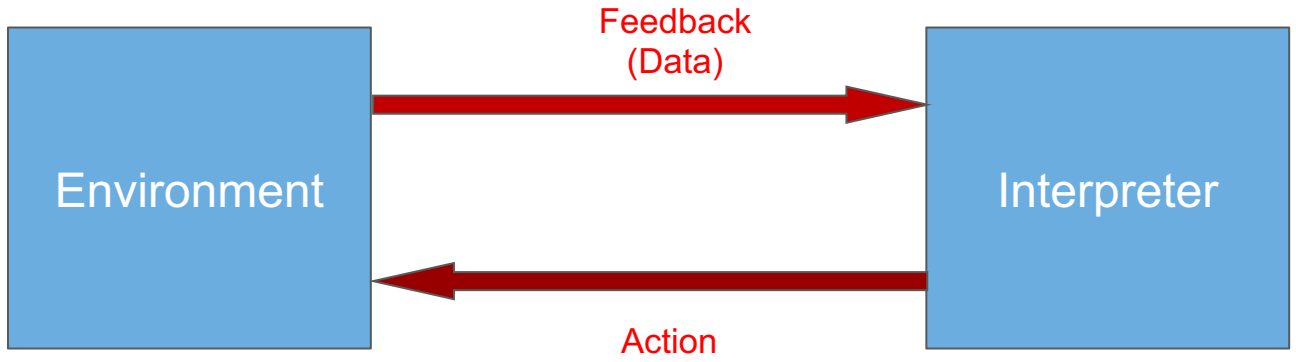
Unsupervised Learning

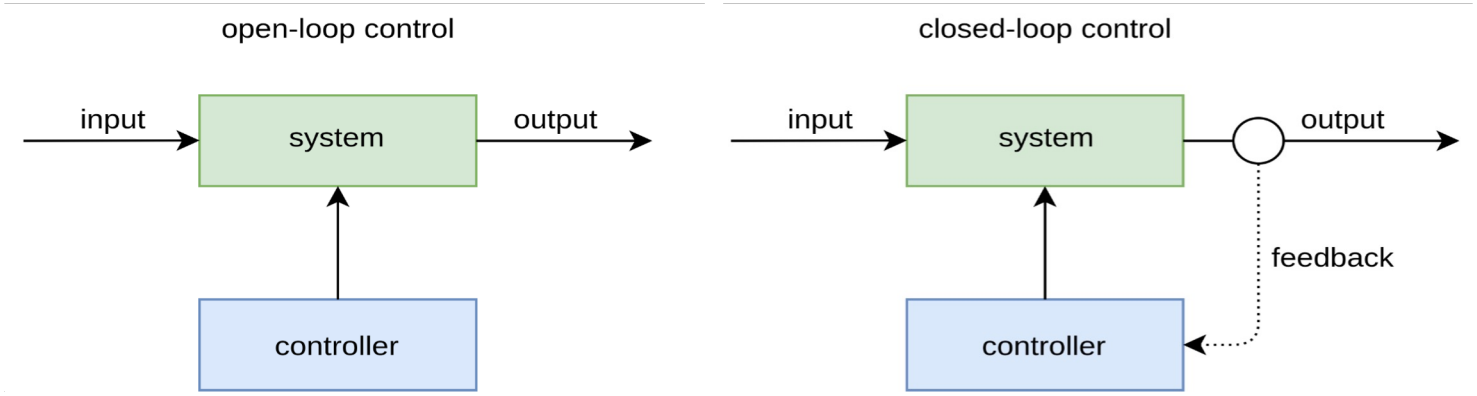


Finding patterns in **unlabelled** data

Source : https://andreweiner.github.io/reveal.js/of_conf_2021.html

2. Reinforcement Learning



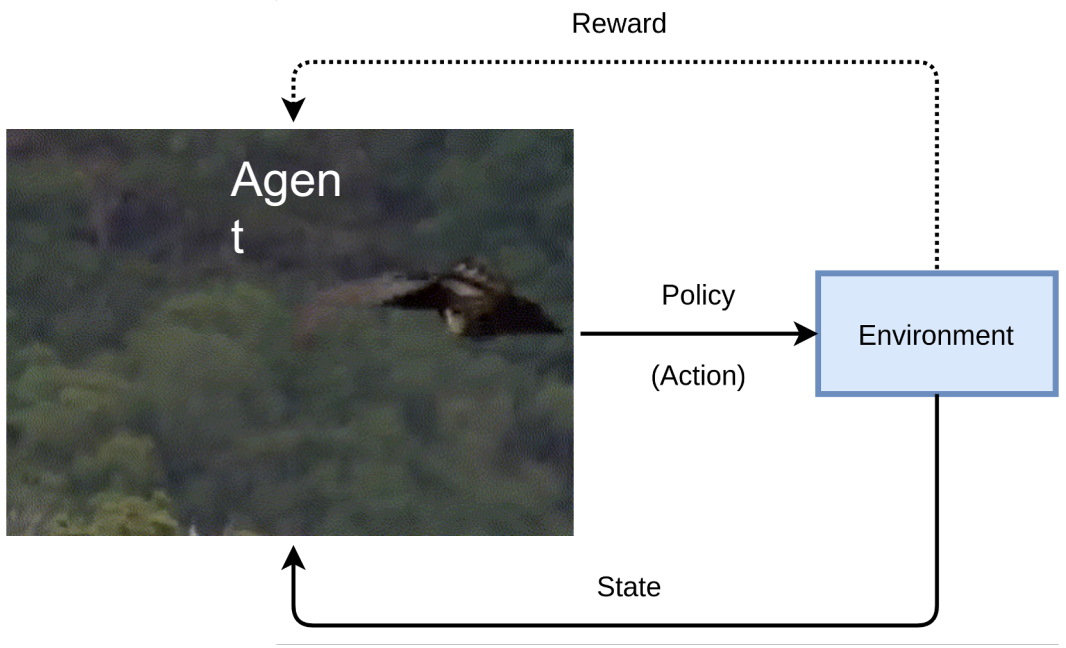


No feedback
Pre-defined
controller

Feedback
Control determined
by state

Source : https://doi.org/10.5281/zenodo.4897961

2. Reinforcement Learning

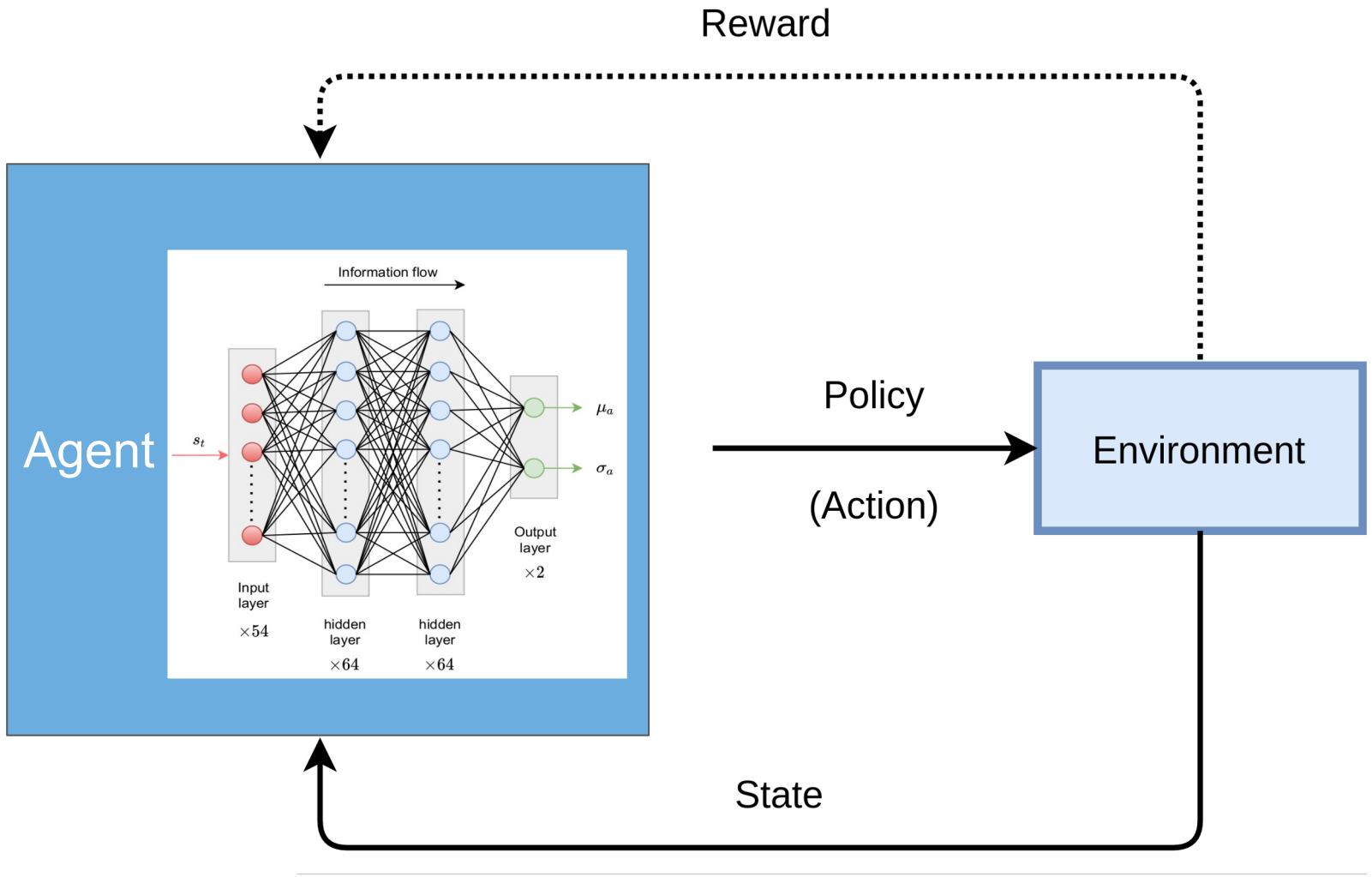


- Q learning
- Temporal-Difference prediction (TD)
- Dyna-Q
- Bandit Walk

- Random Walk
- FrozenLake
- Deep Reinforcement Learning
- …

Source : https://doi.org/10.5281/zenodo.4897961

What is Deep Reinforcement Learning (DRL) ?



Source : https://doi.org/10.5281/zenodo.4897961

DRL algorithms

- Deep Q-Networks (DQN)
- Deep Deterministic Policy Gradient (DDPG)
- **Proximal Policy Optimization (PPO)**
- **…**

- **Proximal Policy Optimization**

- **developed by John Schulman in 2017 at OpenAI.**
- **Over the period proved very simple, efficient and gives high performance.**
  PPO-Clip instead of trust region constraints
- **Defeated many video games seem complex to human beings**
  e.g. Chess, Atari

PPO Algorithm

Pseudocode

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

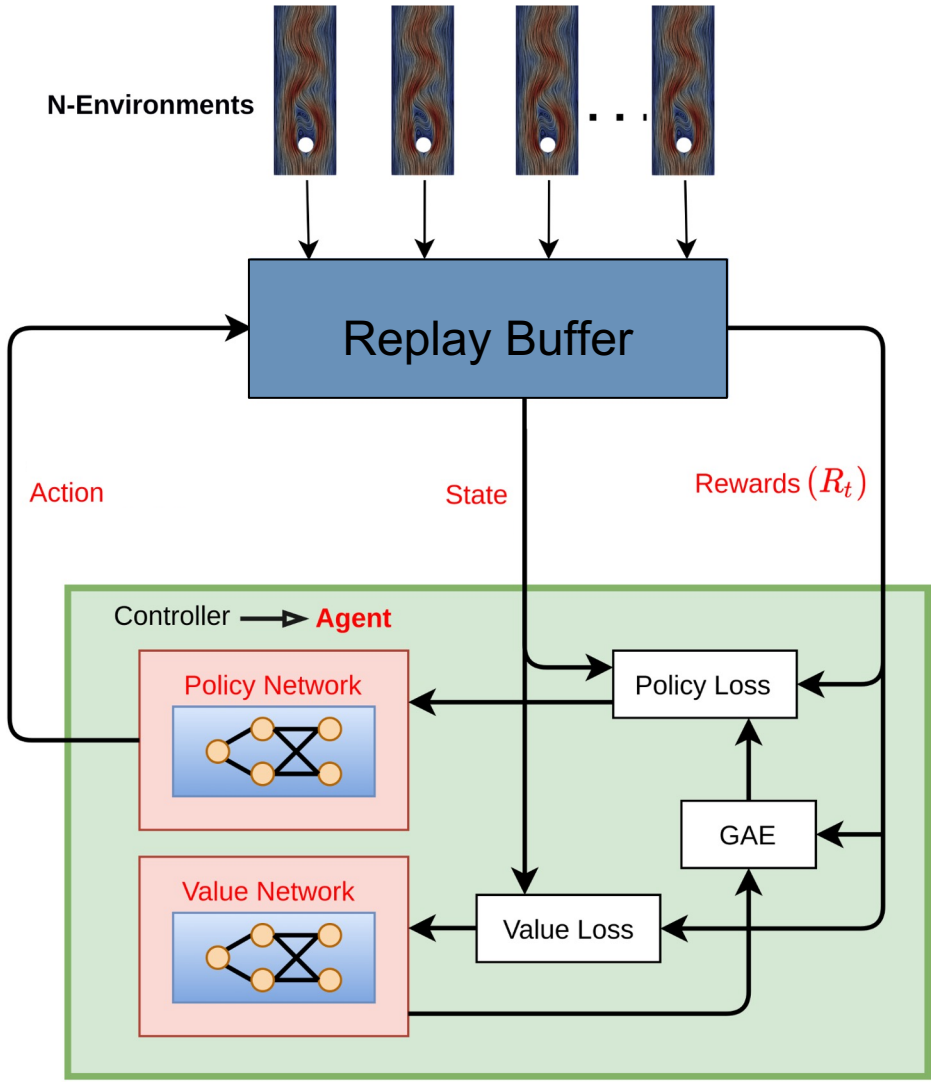
$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Source : https://spinningup.openai.com/en/latest/algorithms/ppo.html

2. Reinforcement Learning

PPO Algorithm

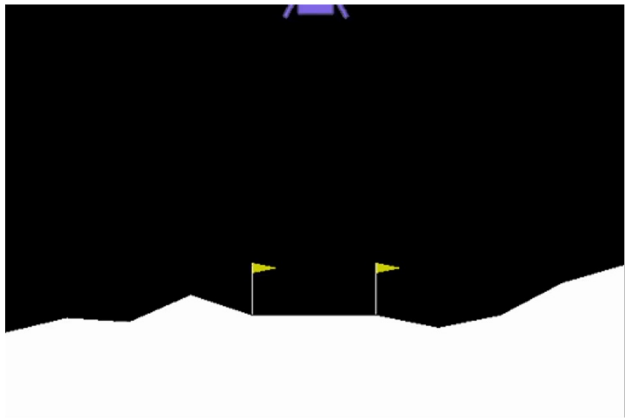


Source : https://doi.org/10.5281/zenodo.4897961

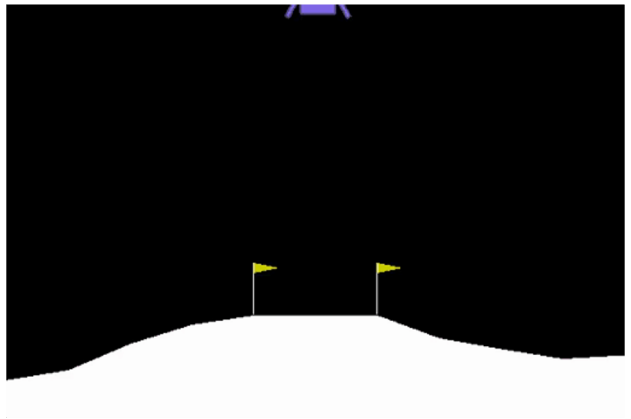
2. Reinforcement Learning

PPO Algorithm with Gym Environment

Lunar Lander example



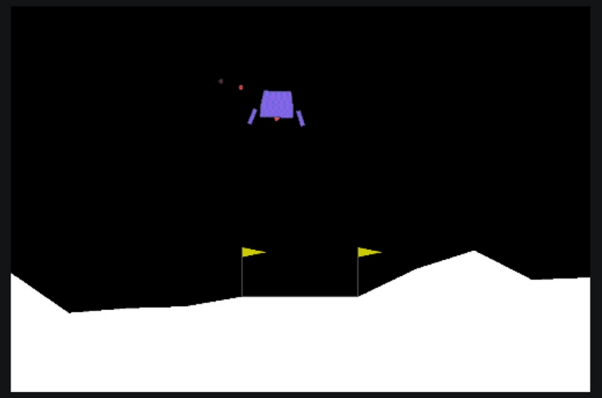
Before Training



After Training

Gym Documentation

Gym is a standard API for reinforcement learning, and a diverse collection of reference environments



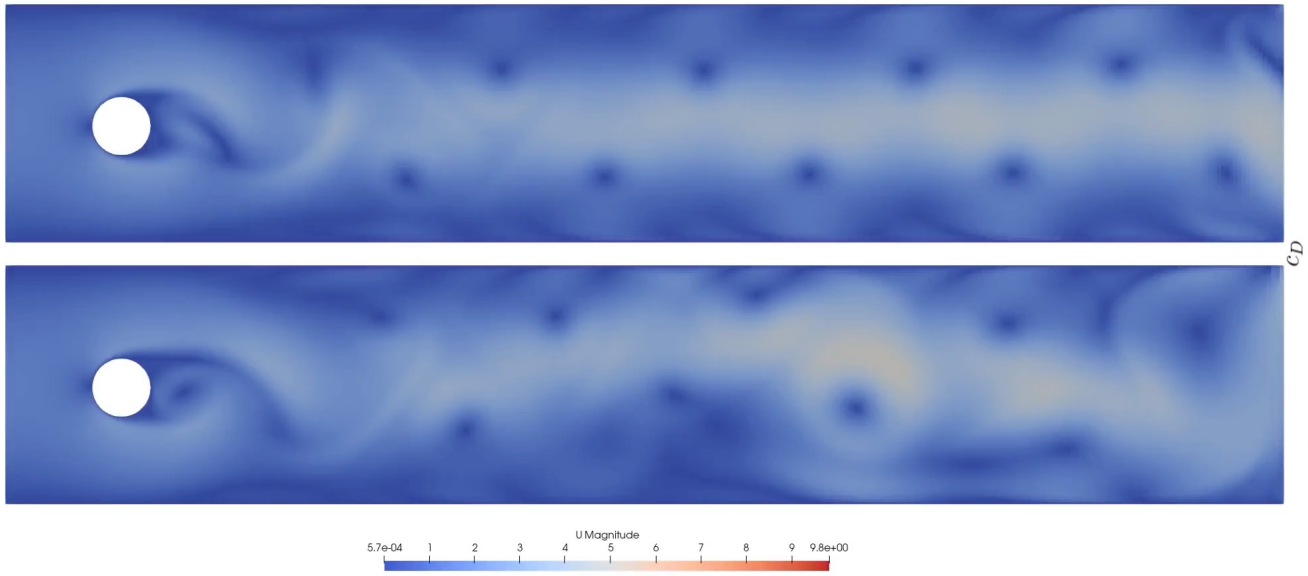
The Gym interface is simple, pythonic, and capable of representing general RL problems:

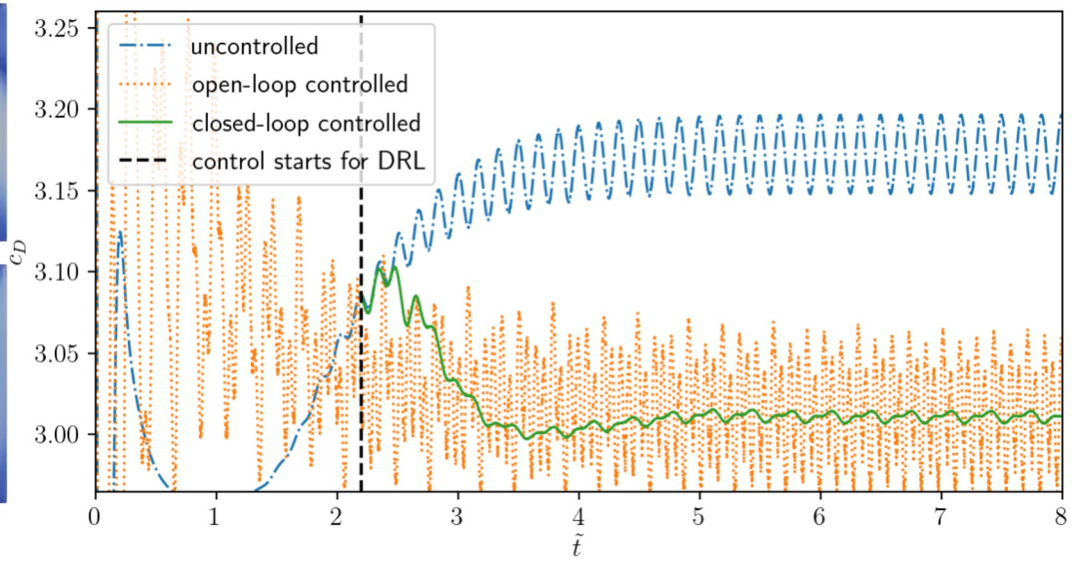
```
import gym
env = gym.make("LunarLander-v2", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(1000):
   action = policy(observation)  # User-defined policy function
   observation, reward, terminated, truncated, info = env.step(action)

   if terminated or truncated:
      observation, info = env.reset()
env.close()
```

https://www.gymlibrary.dev/

2. Reinforcement Learning

Active Flow control with PPO

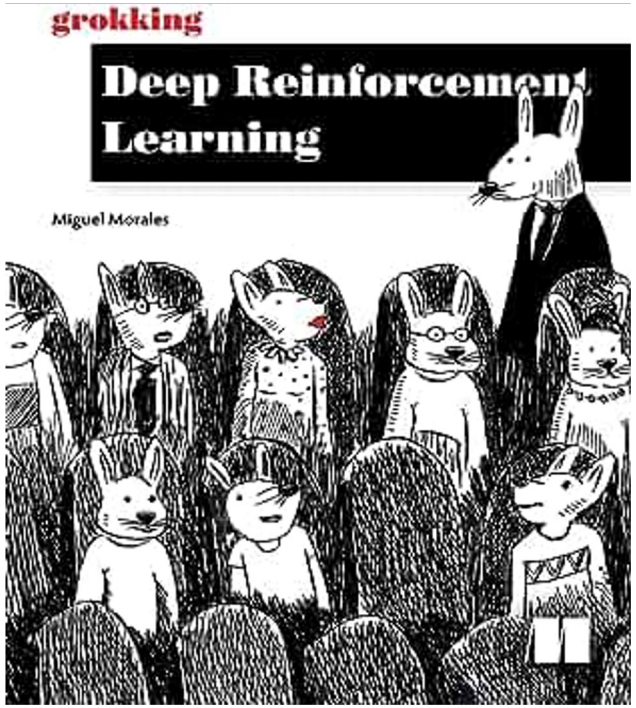


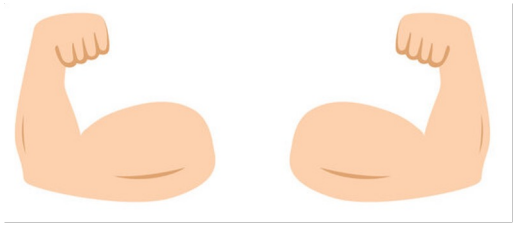


Source : https://doi.org/10.5281/zenodo.4897961

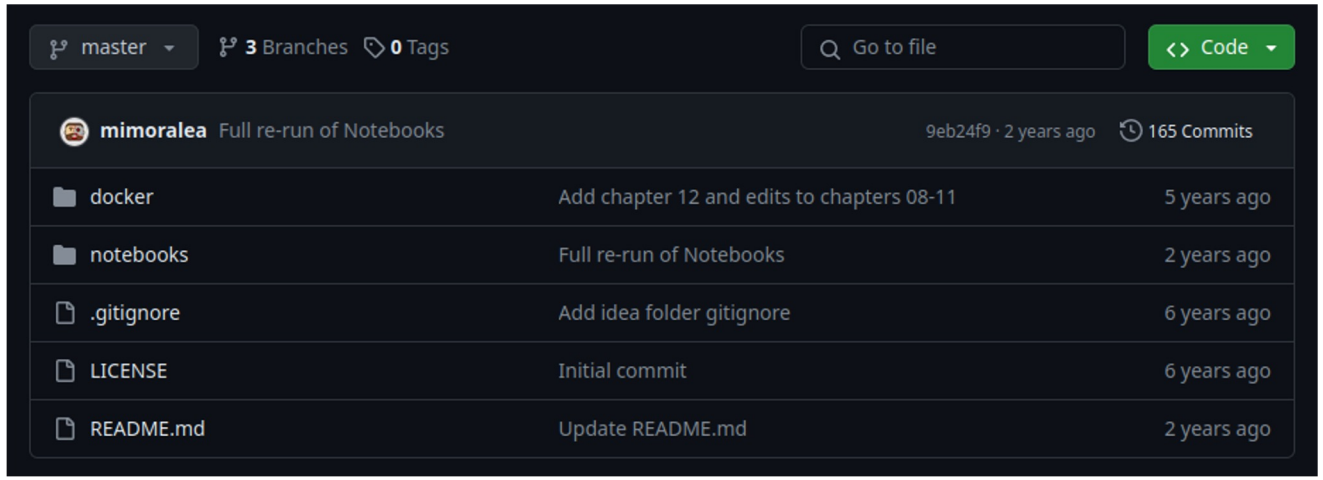
2. Reinforcement Learning

Run PPO algorithm on AI system



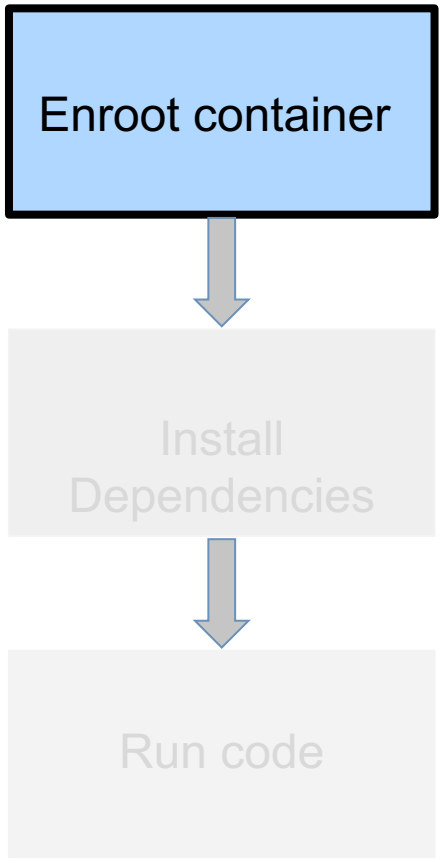


https://github.com/mimoralea/gdrl



2. Reinforcement Learning

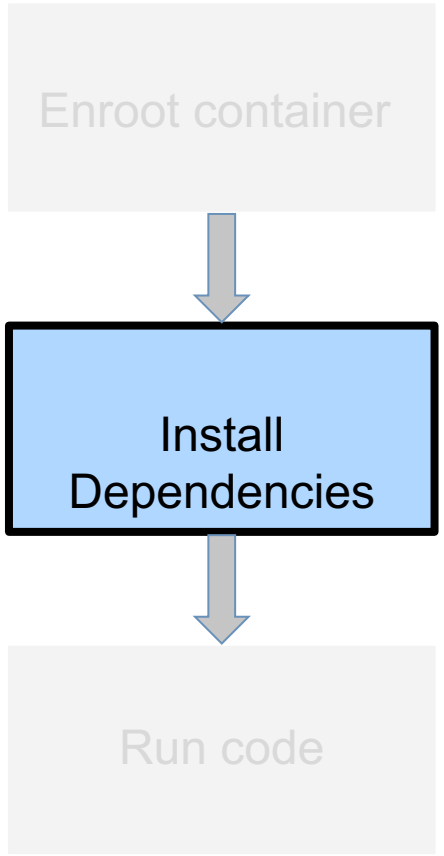
Run PPO algorithm on AI system




```
# login to AI system - login node
❯❯ ssh USERNAME@login.ai.lrz.de
user@login-05:~$

# login to compute node
user@login-05:~$ salloc -N 1 -p PartitionName --qos=Quota -t 00:30:00 --
gres=gpu:1
gpu:1
salloc: Pending job allocation 307659
salloc: job 307659 queued and waiting for resources
salloc: job 307659 has been allocated resources
salloc: Granted job allocation 307659
user@login-05:~$ srun --pty bash
user@gpu-004:~$

# enroot pull ubuntu base image
user@gpu-004:~$ enroot import docker://ubuntu:latest
# spin up the enroot container
user@gpu-004:~$ enroot create --name MyContainerName ubuntu+latest.sqsh
user@gpu-004:~$ enroot start --root MyContainerName
root@gpu-004:/#
root@gpu-004:/# Add Nvidia driver environment variables
See doku :
https://collab.dvb.bayern/display/LRZPUBLICDRAFT/9.+Creating+and+Reusing+a+
Custom+Enroot+Container+Image
```

2. Reinforcement Learning

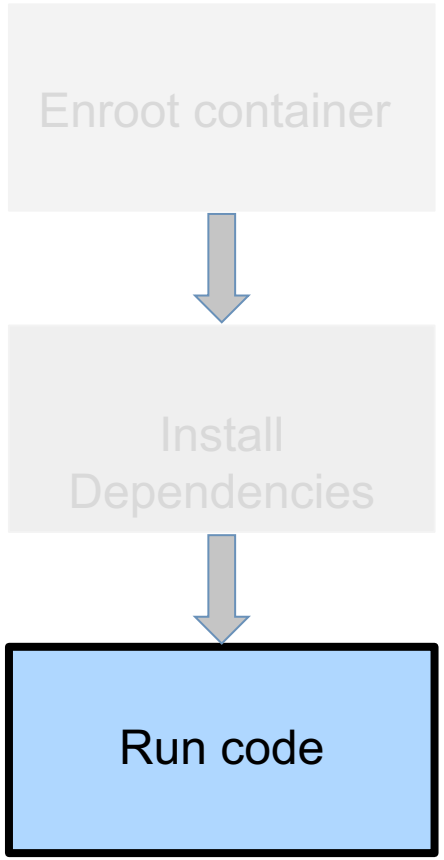
Run PPO algorithm on AI system




```
root@gpu-004:/# Install conda
root@gpu-004:/# Install pip
root@gpu-004:/# Install all required packages

# Once each dependency is installed, export the container in ".sqsh" file
root@gpu-004:/# exit
user@gpu-004:~$ enroot export --output myPreparedContainer.sqsh
MyContainerName
```

2. Reinforcement Learning

Run PPO algorithm on AI system



(1) Run code with container in Interactive session

```
user@login-05:~$ sbatch bachscript.sh
```

bashscript.sh

```
#!/bin/bash
#SBATCH -p lrz-v100x2
#SBATCH --gres=gpu:1
#SBATCH -o enroot_test.out
#SBATCH -e enroot_test.err

Srun --container-image='/path/myPreparedContainer.sqsh' \
          python /path/script.py --epochs 55 --batch-size 512
```

(2) Run code with container in Web-Interactive session

See doku :
https://collab.dvb.bayern/display/LRZPUBLICDRAFT/10.+Interactive+Web+Servers+on+the+LRZ+AI+Systems

Agenda

1. Introduction to the LRZ AI Systems

- ✓ Overview of the LRZ AI Systems
- ✓ Access to the LRZ AI Systems
- ✓ NVIDIA NGC Cloud
- ✓ Introduction to Enroot Containers
- ✓ Interactive and Batch Jobs
- ✓ Open on Demand
- ✓ Exercise: Run a job with an Enroot container

2. Fundamentals of Deep Learning

- ✓ Introduction to Convolutional Neural Networks
- ✓ Exercises: Train CNNs on a GPU
- ✓ Introduction to Transformers
- ✓ Exercise: Train a Transformer on a GPU
- ✓ Introduction to Reinforcement Learning
- ✓ Exercise: Reinforcement Learning

3. Distributed Training of Neural Networks

- ❑ Data Parallel Training
- ❑ Exercise: DP Training of CNN on 2 GPUs
- ❑ Model Parallel Training: Pipeline Parallel and Tensor Parallel
- ❑ Exercise: PP Training of Transformer on 2 GPUs

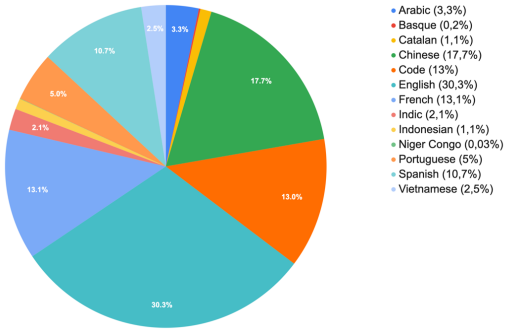
Increasing Amount of Data Available for Deep Learning

ImageNet

14,197,122 images
(150 GB)

https://paperswithcode.com/dataset/imagenet

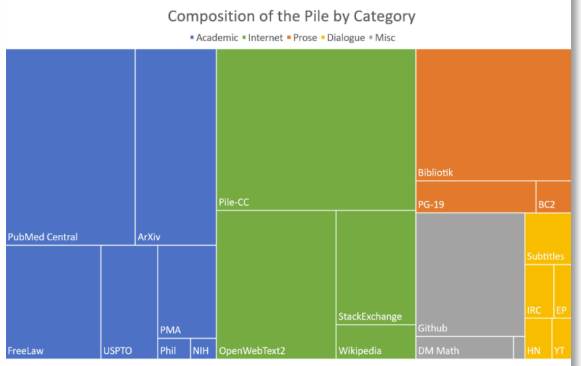
BigScience Multilingual Dataset for Language Modeling



350 billion tokens
(1.5 TB)
46 languages

https://bigscience.huggingface.co/blog/building-a-tb-scale-multilingual-dataset-for-language-modeling

Pile: Dataset of Diverse Text for Language Modeling



800GB

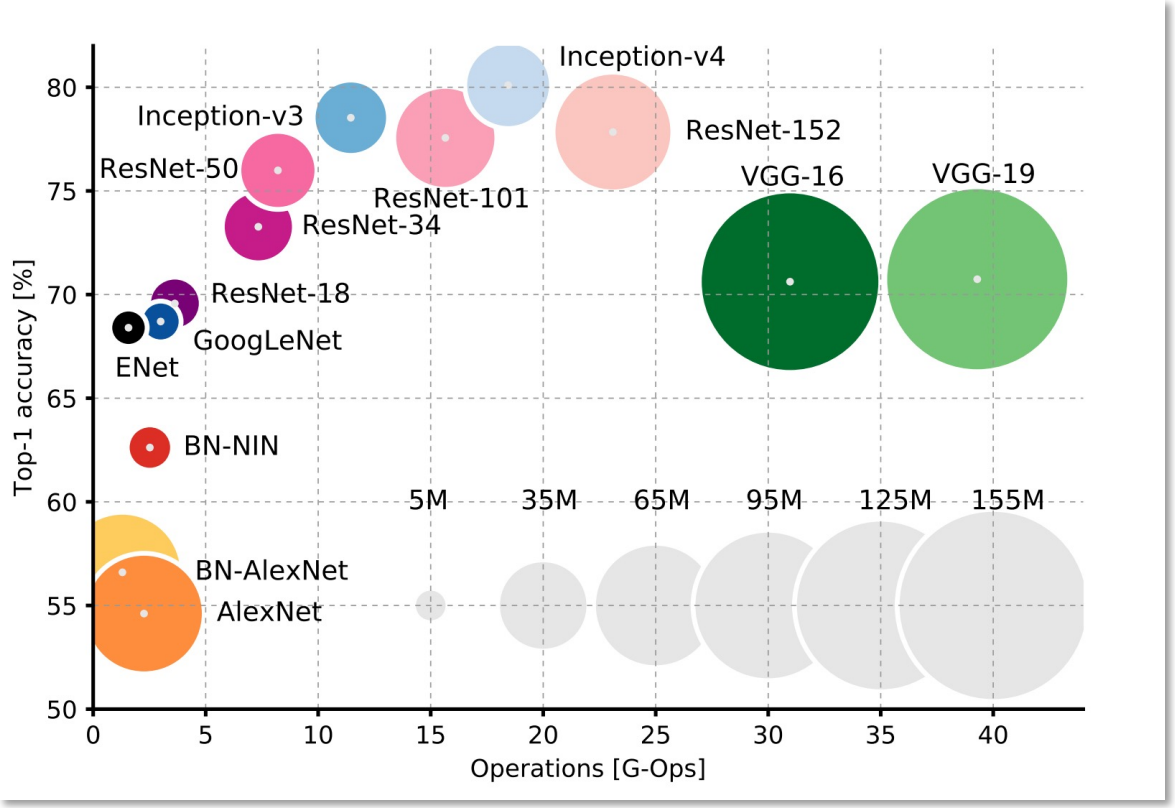
https://arxiv.org/abs/2101.00027

Common Crawl

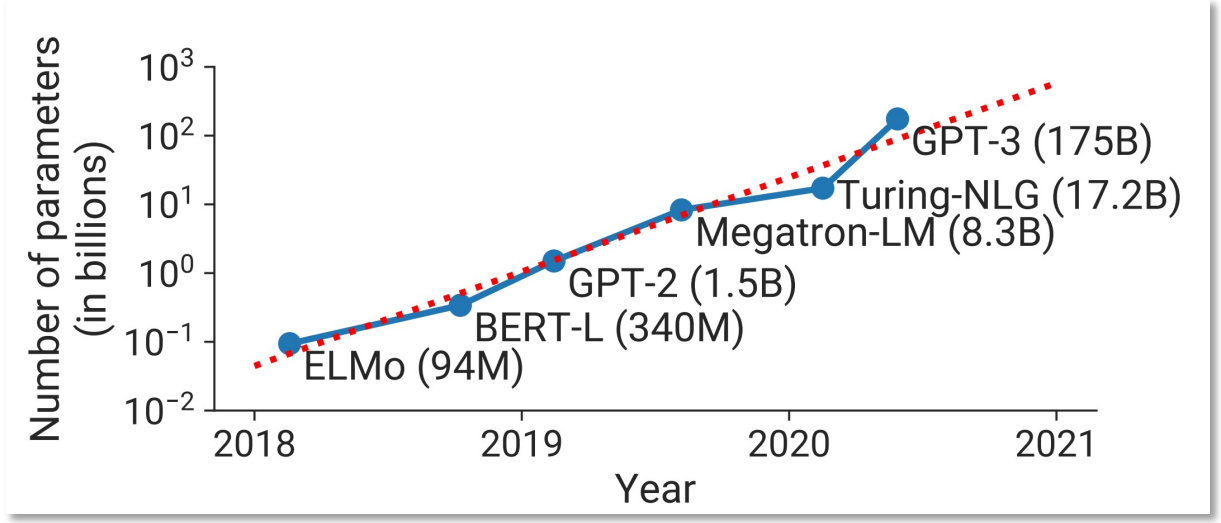
2.95 billion web pages
(351.844 TB)

https://commoncrawl.org/

Increasing Complexity of Deep Learning Models



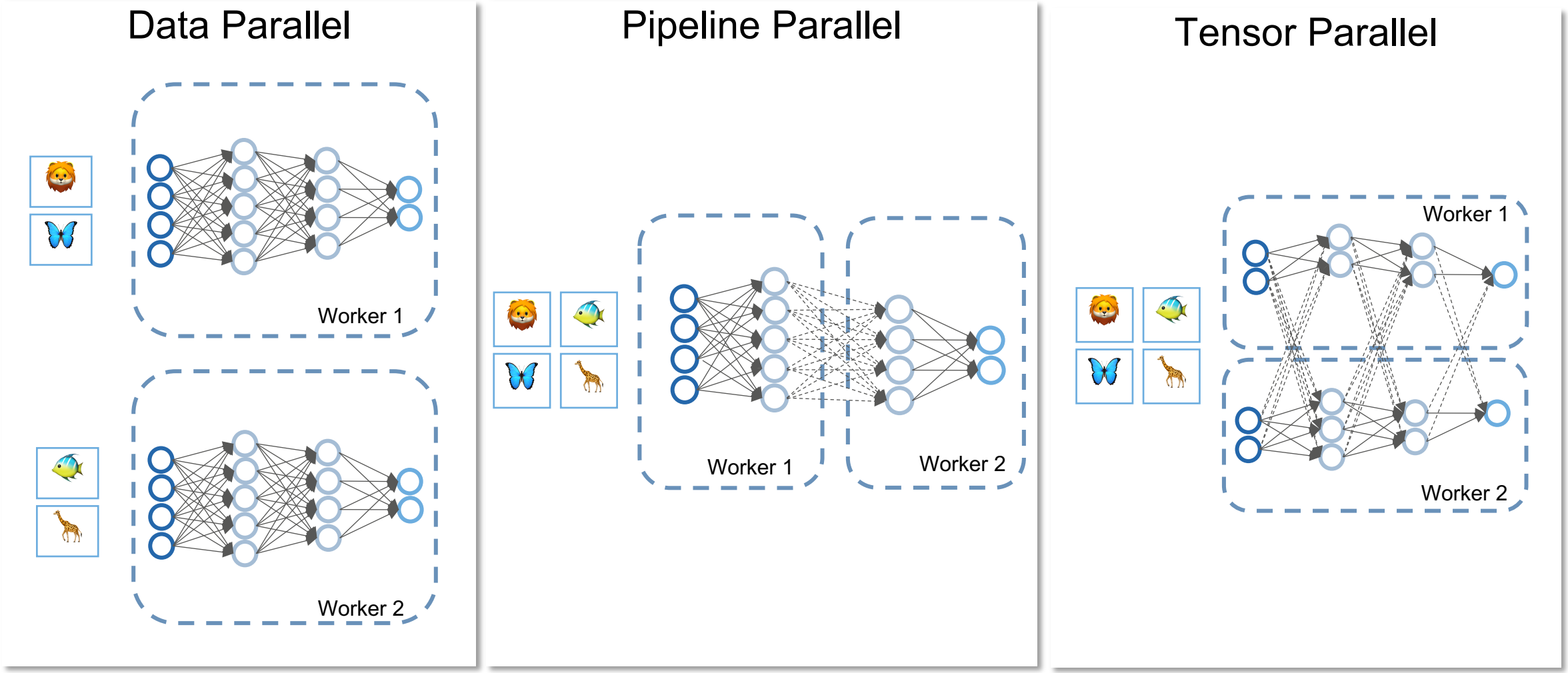
Source: Canziani et al., 2017, arXiv:1605.07678



Source: Narayanan et al., 2021, arXiv:2104.04473

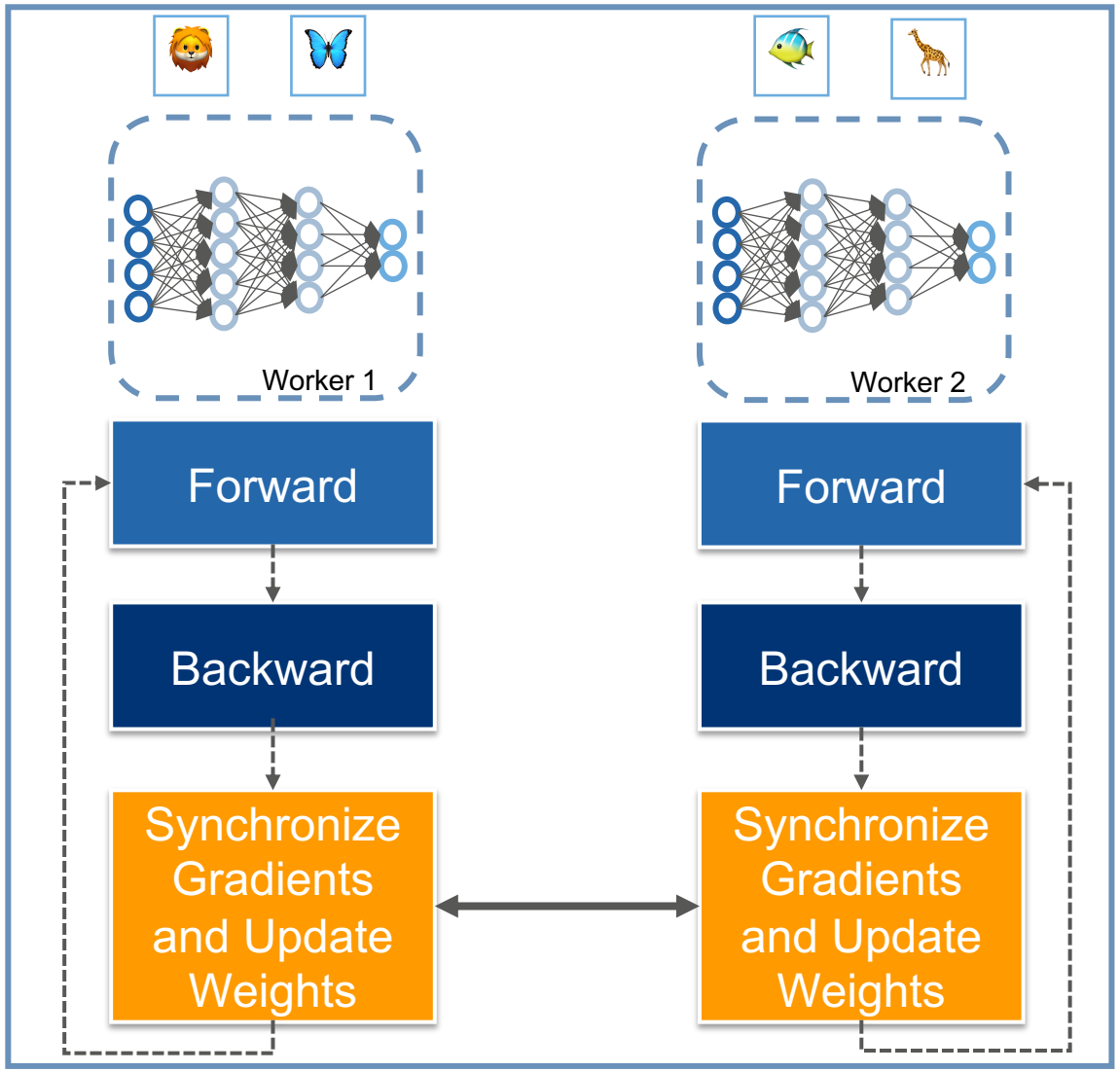
BigScience's BLOOM – 176B
Google's PaLM – 540B
GPT-4 – reportedly 1T

Overview of the Techniques for Distributed Training of NNs



Data Parallel Training of Neural Networks

- Every worker has a copy of the whole model and a subset of the data.
- Each worker generates its own weights, biases, gradients and optimizer states.
- At the end of every iteration, workers aggregate the gradients across all workers, average gradients, and then apply those averaged gradients (using e.g. *allreduce)*.
- **Collective (all-to-all)** communication of weight gradients.



Horovod

- Distributed open-source deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet.
- Runs across *multiple GPUs* and across *multiple machines/nodes*.
- *Fast* – scales up to hundreds of GPUs with upwards of 90% scaling efficiency.
- *Easy* – a few lines of codes.
- *Portable* – different frameworks.

#1 initialization
import horovod.torch as hvd
hvd.init()

#2 pin each GPU to a single process
if torch.cuda.is_available():
    torch.cuda.set_device(**hvd.local_rank()**)

#3 distributed optimizer
opt = #chose your optimizer of preference
opt = **hvd.DistributedOptimizer(opt)**

#4 broadcast the initial variable states from rank 0 to all other processes
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
hvd.broadcast_optimizer_state(optimizer, root_rank=0)

#5 differentiate among different workers (e.g. check pointing on worker 0)
if **hvd.rank()** == 0:
    checkpoint.save(checkpoint_dir)

run training with 2 GPUs on a single machine
$ **horovodrun -np 2** python train.py
# run training with 2 GPUs on two machines (1 GPUs each)
$ **horovodrun -np 2 -H hostname1:1,hostname2:1** python train.py

Sergeev and Del Balso, 2018., arXiv:1802.05799
https://horovod.readthedocs.io/en/stable/pytorch.html#

exercise2.py: Data Parallel Training of ResNet34 on 2 GPUs with Horovod

```
Import torch.utils.data.distributed.DistributedSampler as DistributedSampler

# Initialize Horovod and pin each GPU to a single process
hvd.init()
if torch.cuda.is_available():
    torch.cuda.set_device(hvd.local_rank())

# Partition datasets among workers using DistributedSampler
sampler = DistributedSampler(data, num_replicas=hvd.size(), rank=hvd.rank())
loader = torch.utils.data.DataLoader(data, batch_size, sampler)

# Use distributed optimizer
optimizer = hvd.DistributedOptimizer(optimizer, model.named_parameters())

# Horovod: average metric values across workers.
def metric_average(val, name):
    tensor = val.clone().detach()
    avg_tensor = hvd.allreduce(tensor, name=name)
    return avg_tensor.item()
```

- Install in Pytorch container with: HOROVOD_GPU_OPERATIONS=NCCL pip install --no-cache-dir horovod

```
$ horovodrun -np 2 python exercise2.py
```

Output:

```
[1,1]<stdout>:Epoch [2/10], Step [24960/25000], Loss: 0.5788
[1,0]<stdout>:Epoch [2/10], Step [24960/25000], Loss: 0.4635
[1,1]<stdout>:Epoch [4/10], Step [24960/25000], Loss: 0.3478
[1,0]<stdout>:Epoch [4/10], Step [24960/25000], Loss: 0.2552
[1,1]<stdout>:Epoch [6/10], Step [24960/25000], Loss: 0.2159
[1,0]<stdout>:Epoch [6/10], Step [24960/25000], Loss: 0.1682
[1,1]<stdout>:Epoch [8/10], Step [24960/25000], Loss: 0.1184
[1,0]<stdout>:Epoch [8/10], Step [24960/25000], Loss: 0.0652
[1,0]<stdout>:Epoch [10/10], Step [24960/25000], Loss: 0.1116
[1,1]<stdout>:Epoch [10/10], Step [24960/25000], Loss: 0.0584

[1,0]<stdout>:Test set: Accuracy: 81.08%
[1,0]<stdout>:Elapsed time: 126.31 s ≈ 2 min
```

Pytorch Distributed Data Parallel

- Based on package *torch.distributed for* synchronizing gradients.
- DDP registers a hook for each parameter that fires when the corresponding gradient is computed in the backward pass. That signal is used to trigger gradient synchronization across processes.

- Runs across *multiple GPUs* and across *multiple machines/nodes.*
- Near-linear scalability using 256 GPUs.

Li et al., 2020, arXiv:2006.15704

https://pytorch.org/docs/master/notes/ddp.html

https://pytorch.org/tutorials/intermediate/ddp_tutorial.html#basic-use-case

```
import torch.distributed as dist
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP

def example(rank, world_size):
        # create default process group – nccl for building with cuda
        dist.init_process_group(backend="nccl", rank=rank,
                                                world_size=world_size)
        # create a model and wrap it with DDP
        model = nn.Linear(10, 10).to(rank)
        ddp_model = DDP(model, device_ids=[rank])
        # define loss function and optimizer
        loss_fn = nn.MSELoss()
        optimizer = torch.optim.SGD(ddp_model.parameters(), lr=0.01)
        # forward pass
        outputs = ddp_model(torch.randn(20, 10).to(rank))
        labels = torch.randn(20, 10).to(rank)
        # backward pass and update parameters
        loss_fn(outputs, labels).backward()
        optimizer.step()
        # cleanup
        dist.destroy_process_group()

if __name__=="__main__":
        # environment variables for using torch.distributed
        os.environ["MASTER_ADDR"] = "localhost"
        os.environ["MASTER_PORT"] = "29500"
        world_size = 2   # world_size = gpus * nodes
        mp.spawn(example, args=(world_size,), nprocs=world_size)
```

exercise3.py: Data Parallel Training of VGG19 on 2 GPUs with Pytorch DDP

```
# environment variables for using torch.distributed
world_size = gpus * nodes
os.environ['MASTER_ADDR'] = 'localhost'
os.environ['MASTER_PORT'] = '8888'
mp.spawn(train_and_test, gpus, args=(gpus, nodes, nr, world_size))

def train_and_test(gpu, gpus, nodes, nr, world_size):
    rank = nr * gpus + gpu  # nr is ranking within the nodes
    dist.init_process_group(backend='nccl', init_method='env://', world_size, rank)
    # Set the seed to initiate the model with the same weights on each worker
    torch.manual_seed(0)
    # Set the model and loss to GPU
    model.cuda()
    criterion = nn.CrossEntropyLoss().cuda()
    # Wrap the model
    model = nn.parallel.DistributedDataParallel(model, device_ids=[gpu])
    # Partition datasets among workers using DistributedSampler
    sampler = DistributedSampler(data, num_replicas=world_size, rank=rank)
    loader = torch.utils.data.DataLoader(data, batch_size, sampler)
```

```
$ python exercise3.py
```

Output:

```
Epoch [2/10], Step [24960/25000], Loss: 1.5240
Epoch [2/10], Step [24960/25000], Loss: 1.4402
Epoch [4/10], Step [24960/25000], Loss: 0.7013
Epoch [4/10], Step [24960/25000], Loss: 0.7268
Epoch [6/10], Step [24960/25000], Loss: 0.5393
Epoch [6/10], Step [24960/25000], Loss: 0.4175
Epoch [8/10], Step [24960/25000], Loss: 0.3391
Epoch [8/10], Step [24960/25000], Loss: 0.4450
Epoch [10/10], Step [24960/25000], Loss: 0.3889
Epoch [10/10], Step [24960/25000], Loss: 0.3106

Test set: Accuracy: 81.76%
Elapsed time:  234.65 s  ≈ 4 min
```

Pytorch Lightning Distributed Data Parallel

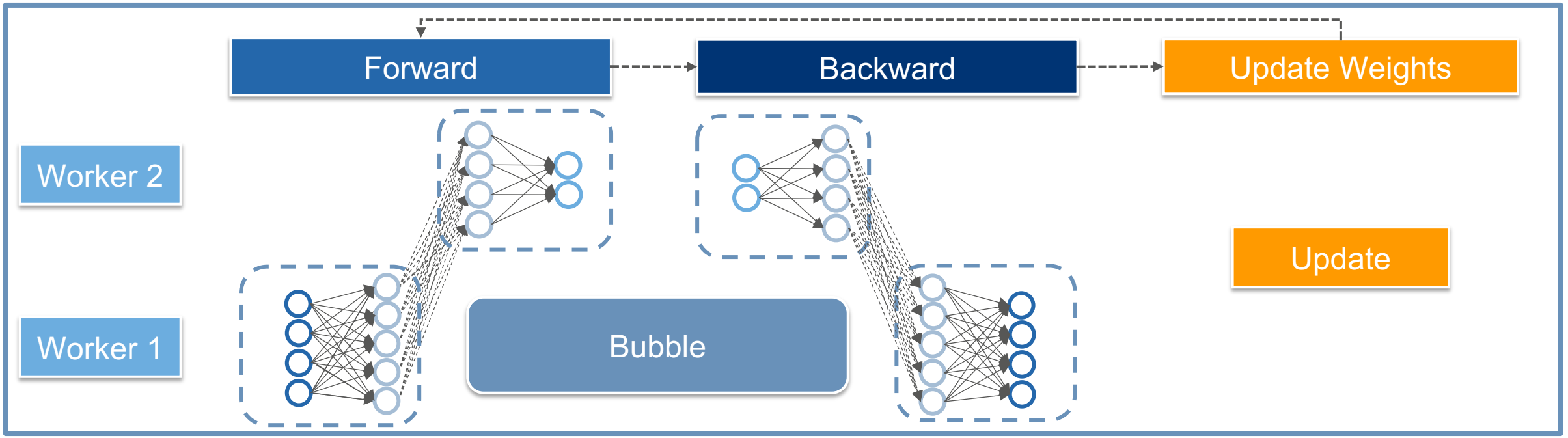
```
accelerator = 'gpu’
devices = 2 # or more

trainer = pl.Trainer(devices=devices,
                     accelerator=accelerator,
                     strategy='ddp’,)
```

5 epochs

```
Time for training vgg19 on gpu:  81.21439981460571
```

Pipeline Parallel Training of Neural Networks

- Each worker has one or more layers, and the corresponding parameters of the neural network. There is a single copy of weight parameters shared between the workers.
- Goal: Make feasible models, that are larger than the memory capacity of a single GPU.
- **Point-to-point** communication of intermediate activations and gradients.



Pipeline Parallel Training of Neural Networks - Different Schemes

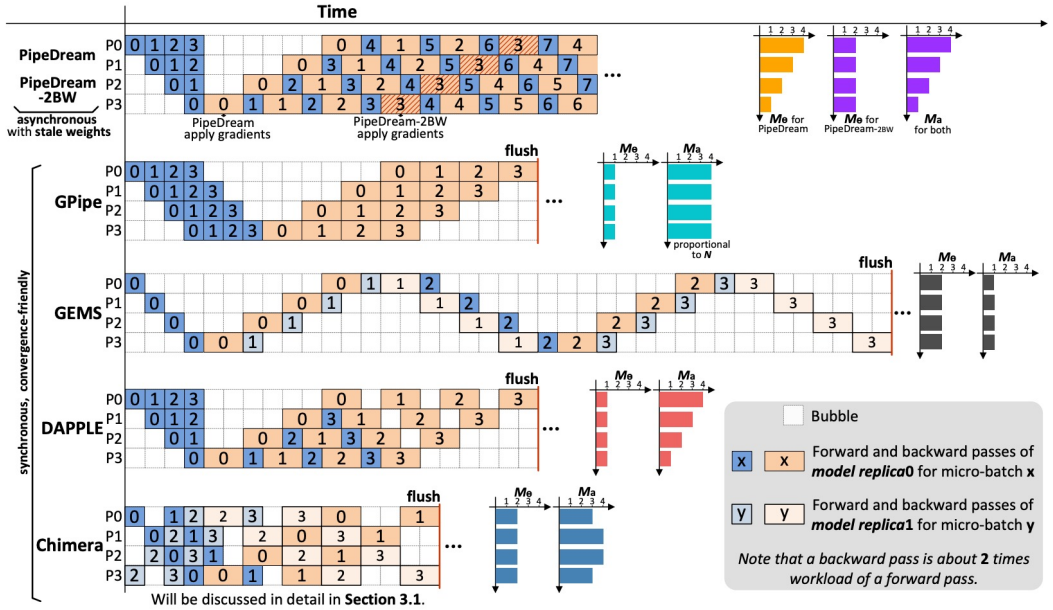


Figure 2: Pipeline parallelism schemes, with four pipeline stages (D=4) and four micro-batches (N=4) within a training iteration, except that PipeDream updates the model after each backward pass on a micro-batch.

Table 1: The list of symbols frequently used in the paper.

D	The number of pipeline stages (*depth*)
W	The number of replicated pipelines (*width*) for data parallelism [1]
P	The number of workers ($= W * D$)
B	Micro-batch size
N	The number of micro-batches executed by each worker within a training iteration
$\hat{B}$	Mini-batch size ($= B * N * W$)
M_θ	Memory consumption for the weights of one stage
M_a	Memory consumption for the activations of one stage

[1] This paper considers the cases where all pipeline stages have balanced workload, and therefore are equally replicated to combine with data parallelism.

- Goal: Minimizing bubbles in the pipeline.

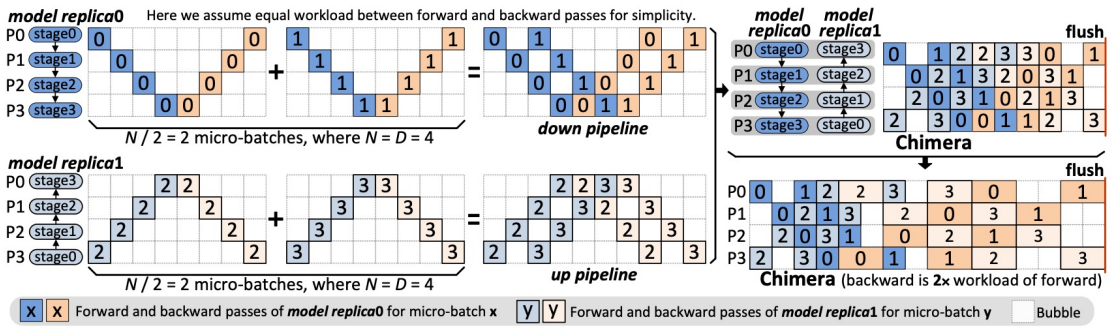


Figure 3: Model replicas and bidirectional pipelines scheduling of Chimera.

Li and Hoefler, 2021,
http://htor.inf.ethz.ch/publications/img/shigang-chimera.pdf

Pipeline Parallelism with Pytorch Pipe

- Micro-batch pipeline parallelism based on Gpipe scheme.
- Depends on the Remote Procedure Call (RPC) framework from *torch.distributed* for communication.
- Currently runs only on a *single machine/node.*
- *Experimental and subject to change! Not yet fully compatible with CUDA!*

https://pytorch.org/docs/stable/pipeline.html

Kim et al., 2020, https://arxiv.org/abs/2004.09910

```
from torch.distributed import rpc
from torch.distributed.pipeline.sync import Pipe

# initialize Remote Procedure Call (RPC) framework
os.environ['MASTER_ADDR'] = “localhost”
os.environ['MASTER_PORT'] = “29500”
rpc.init_rpc(name='worker', rank=0, world_size=1)

# build the pipe with each layer on different GPU
fc1 = nn.Linear(16, 8).cuda(0)
fc2 = nn.Linear(8, 4).cuda(1)
model = nn.Sequential(fc1, fc2)
model = Pipe(model, chunks=8)
input = torch.rand(16, 16).cuda(0)

# rref is a reference to a value of a tensor on a remote worker
output_rref = model(input)
outputs = output_rref.local_value()
loss = criterion(outputs, targets.cuda(1))
```

Exercise: Pipeline Parallel Training of a Transformer on 2 GPUs

```
nlayers = 12
num_gpus = 2
partition_len = ((nlayers - 1) // num_gpus) + 1
# Add encoder in the beginning.
tmp_list = [Encoder(ntokens, emsize, dropout).cuda(0)]
module_list = []
# Add all the necessary transformer blocks.
for i in range(nlayers):
    transformer_block = TransformerEncoderLayer(emsize, nhead, nhid, dropout)
    if i != 0 and i % (partition_len) == 0:
        module_list.append(nn.Sequential(*tmp_list))
        tmp_list = []
    device = i // (partition_len)
    tmp_list.append(transformer_block.to(device))
# Add decoder in the end.
tmp_list.append(Decoder(ntokens, emsize).cuda(num_gpus - 1))
module_list.append(nn.Sequential(*tmp_list))

from torch.distributed.pipeline.sync import Pipe
# Build the pipeline.
chunks = 8
model = Pipe(torch.nn.Sequential(*module_list), chunks = chunks)
```

- Using an embedding dimension of 4096, hidden size of 4096, 16 attention heads and 12 transformer layers.
- Model with ~1.4 billion parameters.
- Split the model such that half of the nn.TransformerEncoderLayer is on 1st GPU and the other half is on the 2nd.

```
$ python exercise3.py
```

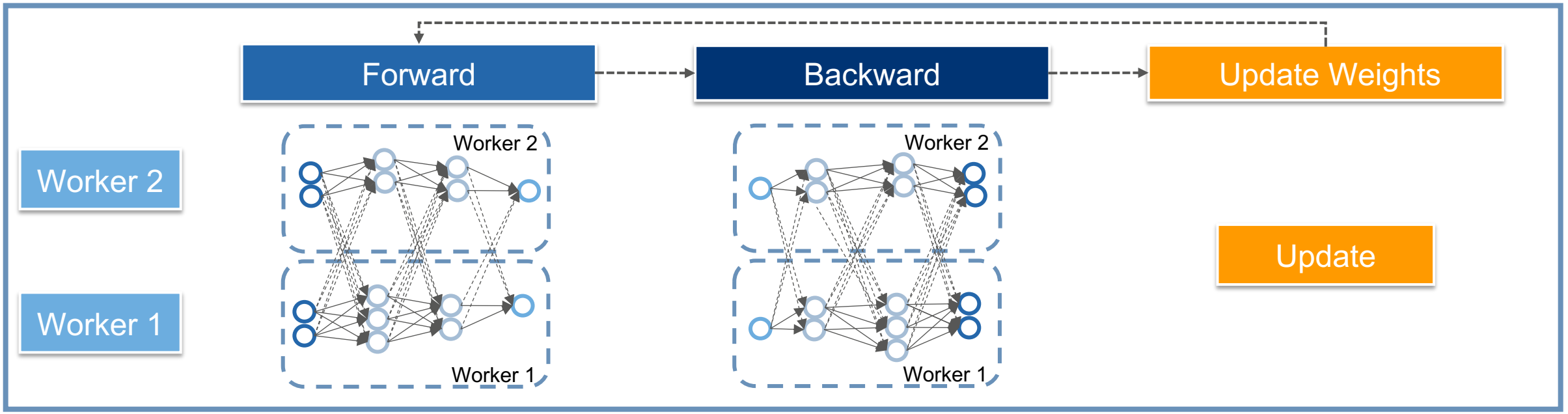
Output:

```
Total parameters in model: 1,444,261,998
end of epoch 1 | time: 28.42s | valid loss 1.22
end of epoch 2 | time: 27.36s | valid loss 0.25
end of epoch 3 | time: 27.49s | valid loss 0.23
End of training | test loss 0.20
```

https://pytorch.org/tutorials/intermediate/pipeline_tutorial.html

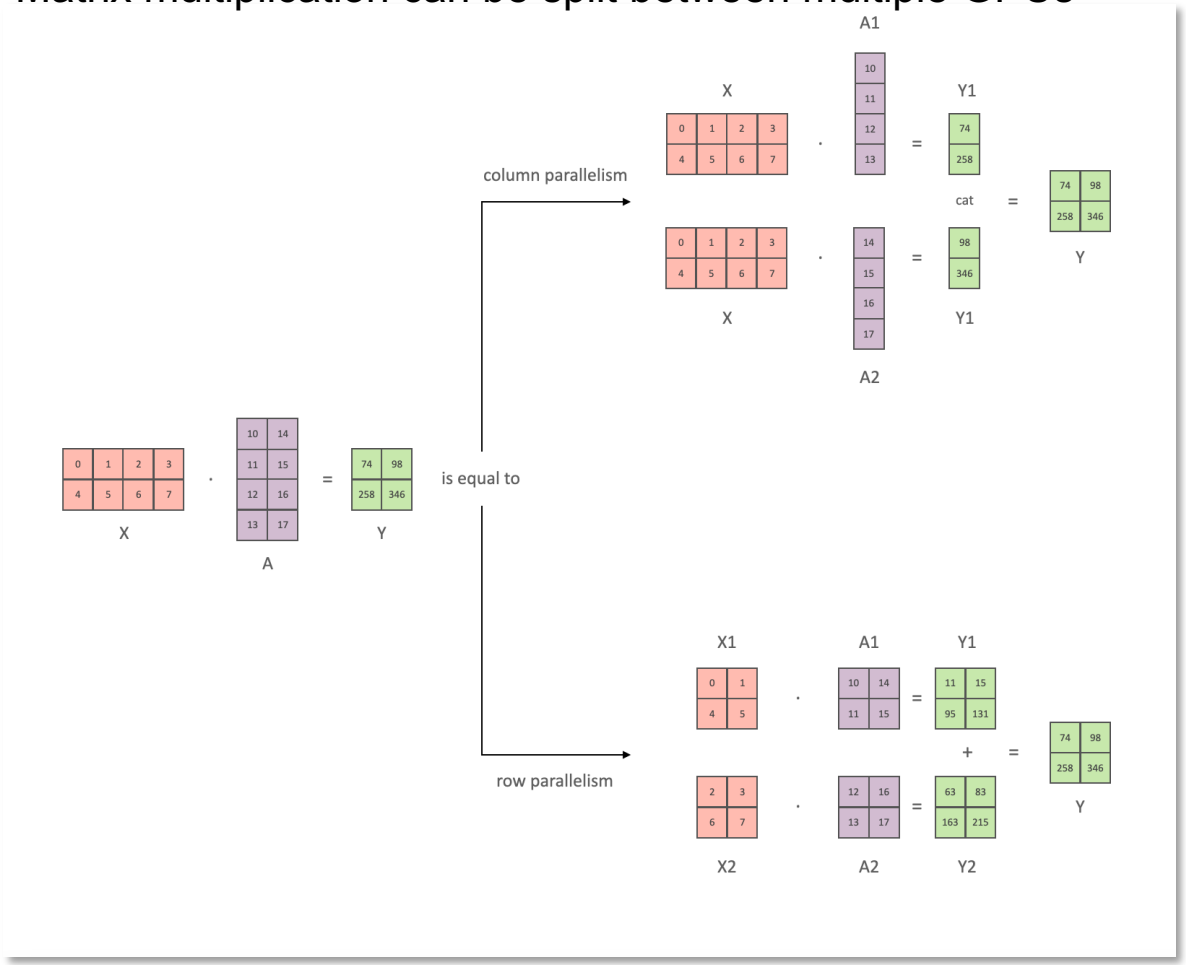
Tensor Parallel Training of Neural Networks

- Each operator or layer of the neural network split between the workers.
- There is a single copy of weight parameters shared between the workers.
- Goal: Make feasible models that are larger than the memory capacity of a single GPU.
- **Collective (all-to-all)** communication of <u>partial</u> activations and gradients of each worker.

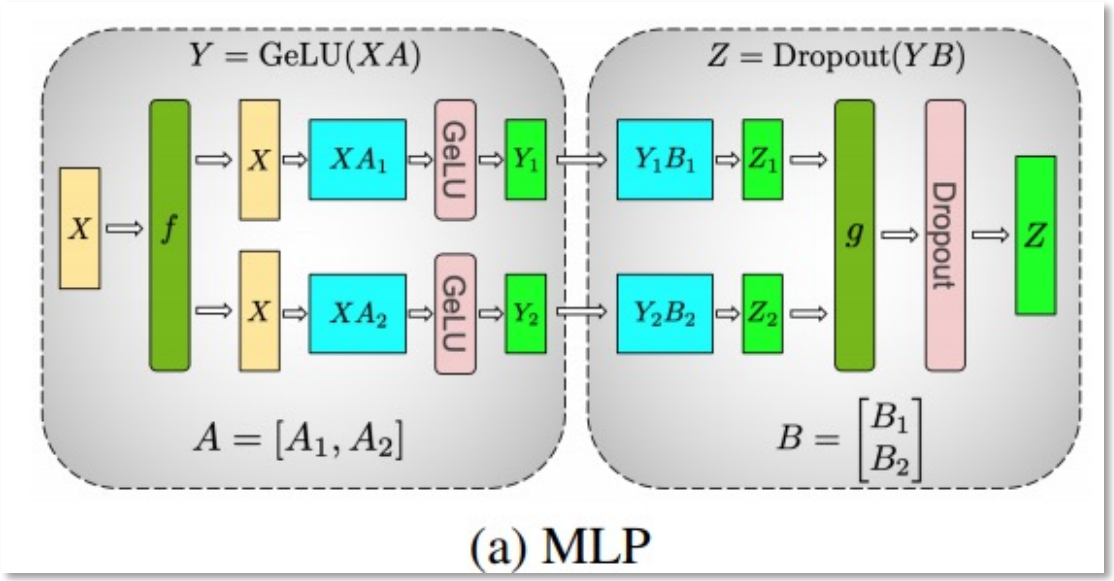


Tensor Parallel Training of Neural Networks

Matrix multiplication can be split between multiple GPUs



Source: https://huggingface.co/docs/transformers/parallelism#naive-model-parallelism-vertical-and-pipeline-parallelism



Source: Narayanan et al., 2021, arXiv:2104.04473

Combination of Data, Pipeline and Tensor Parallelism – Megatron-LM

How do we navigate this configuration space?

- Degree of pipeline, tensor, and data parallelism
- Pipelining schedule
- Global batch size
- Microbatch size

→ Each of these choices influence amount of communication, size of pipeline bubble, memory footprint

Tensor, pipeline, and data parallelism can be composed to scale with a trillion parameters to thousands of GPUs.

Source: Narayanan et al., 2021, arXiv:2104.04473
https://github.com/NVIDIA/Megatron-LM
https://www.youtube.com/watch?v=MO_CESBOXgo

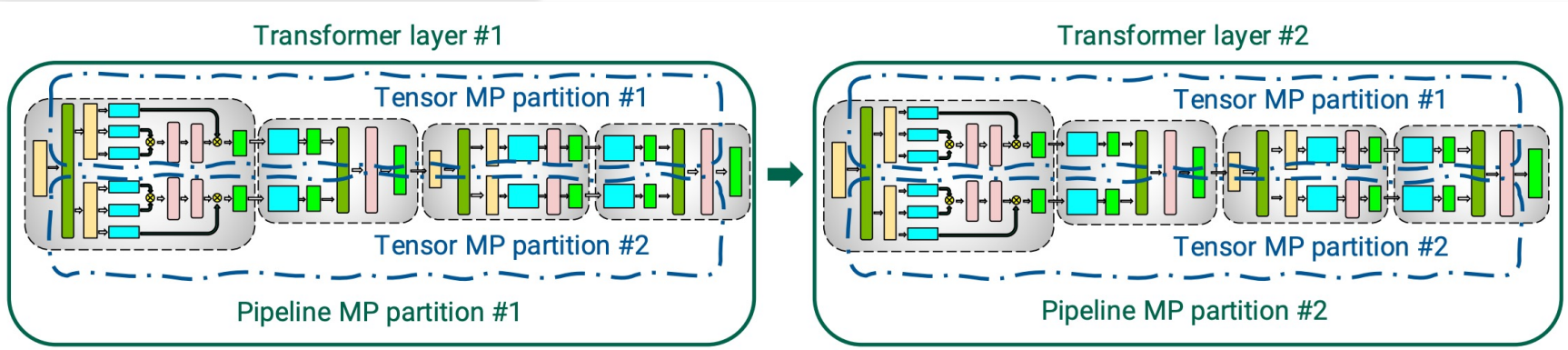


Figure 2: Combination of tensor and pipeline model parallelism (MP) used in this work for transformer-based models.

Agenda

1. Introduction to the LRZ AI Systems

- ✓ Overview of the LRZ AI Systems
- ✓ Access to the LRZ AI Systems
- ✓ NVIDIA NGC Cloud
- ✓ Introduction to Enroot Containers
- ✓ Interactive and Batch Jobs
- ✓ Open on Demand
- ✓ Exercise: Run a job with an Enroot container

2. Fundamentals of Deep Learning

- ✓ Introduction to Convolutional Neural Networks
- ✓ Exercises: Train CNNs on a GPU
- ✓ Introduction to Transformers
- ✓ Exercise: Train a Transformer on a GPU
- ✓ Introduction to Reinforcement Learning
- ✓ Exercise: Reinforcement Learning

3. Distributed Training of Neural Networks

- ❑ Data Parallel Training
- ❑ Exercise: DP Training of CNN on 2 GPUs
- ❑ Model Parallel Training: Pipeline Parallel and Tensor Parallel
- ❑ Exercise: PP Training of Transformer on 2 GPUs

Further Topics

- Pytorch tutorials on distributed training of transformer models:
  - Data and pipeline parallelism: https://pytorch.org/tutorials/advanced/ddp_pipeline.html
- Mesh TensorFlow data and model parallelism:
  - GitHub: https://github.com/tensorflow/mesh
  - Paper: Shazeer et al., 2018, https://arxiv.org/pdf/1811.02084.pdf
- DeepSpeed (PyTorch deep learning optimization library by Miscrosoft):
  - Pipeline parallelism: https://www.deepspeed.ai/tutorials/pipeline/
  - ZeRO data parallelism: https://deepspeed.readthedocs.io/en/latest/zero3.html
- FairScale (PyTorch library for high performance large scale training by Facebook Research):
  - Pipeline parallelism: https://fairscale.readthedocs.io/en/latest/tutorials/pipe.html
  - ZeRO data parallelism: https://github.com/facebookresearch/fairscale/#optimizer-state-sharding-zero
- Sequence parallelism, https://arxiv.org/pdf/2105.13120.pdf

Course Evaluation – Module 5: Introduction to the LRZ AI Systems

Please visit
https://survey.lrz.de/index.php/625212?lang=en
and rate this course.

Your feedback is highly appreciated!
Thank you!

Thank you for your attention!

Maja.Piskac@lrz.de
Ajay.Navilarekal@lrz.de
Darshan.Thummar@lrz.de
Navdar.Karabulut@lrz.de