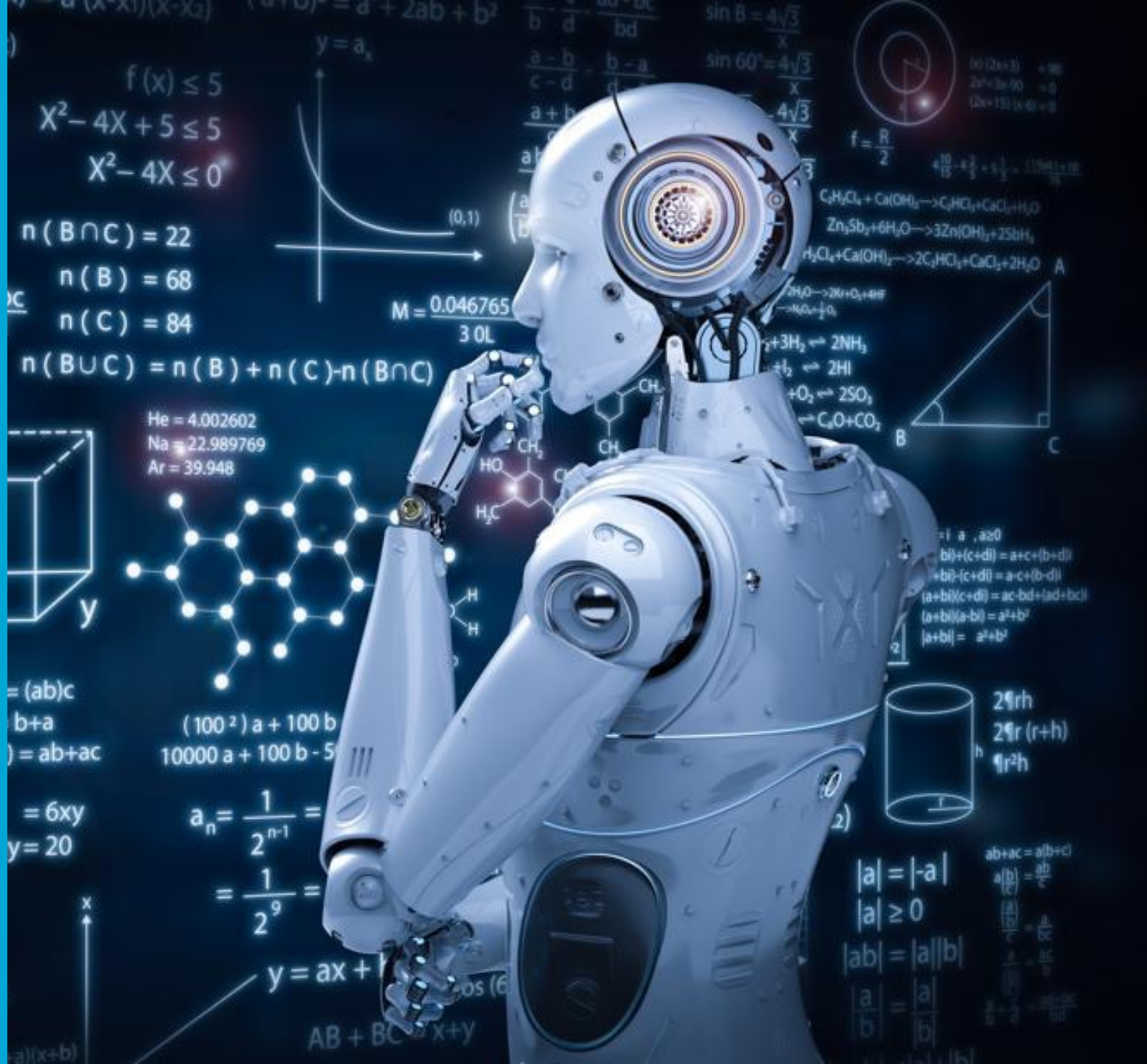
# Machine learning on GPUs

**Matthias Möller**

TU Delft

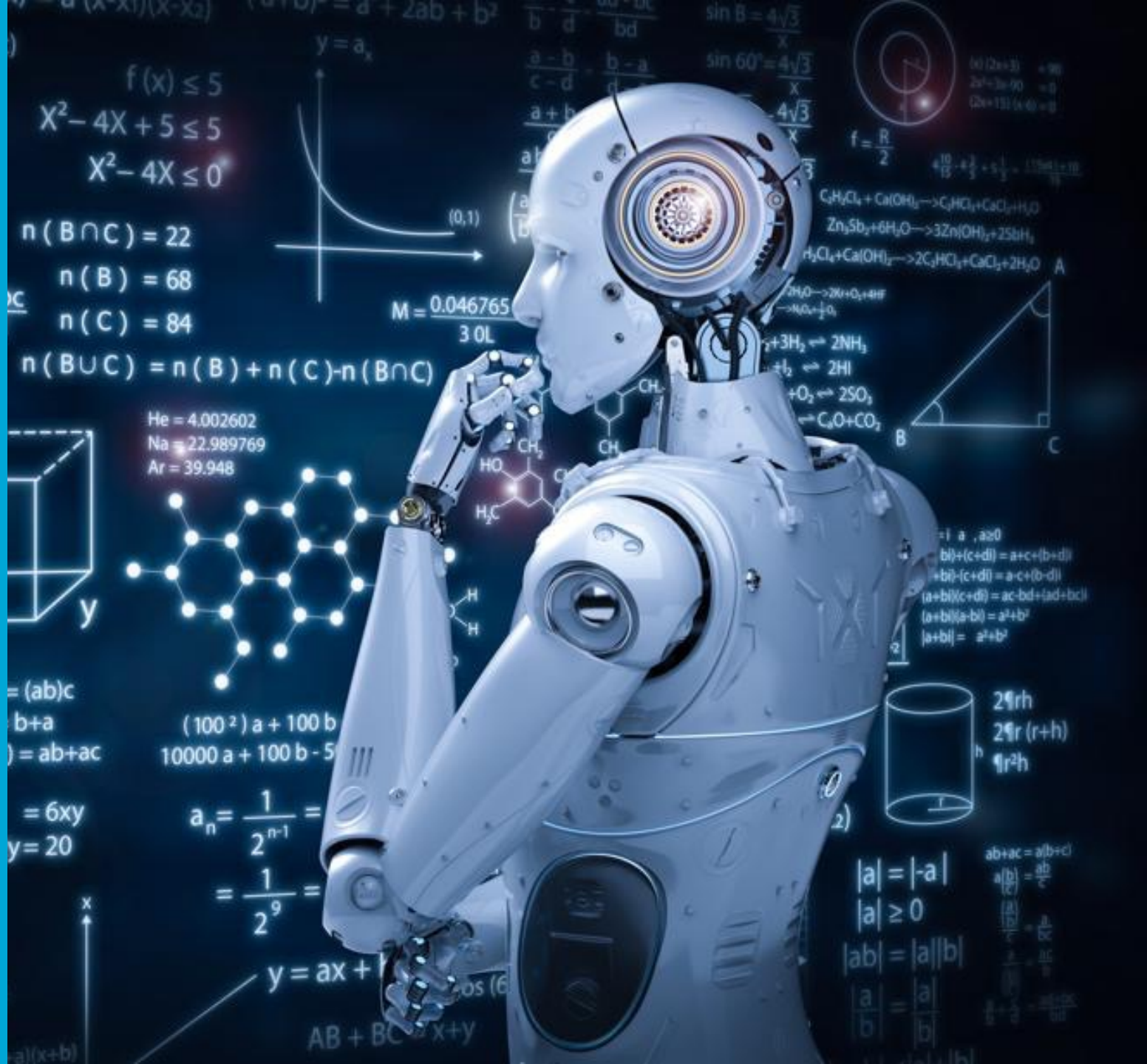


# Learning objectives

- Learn how to use TensorFlow and PyTorch on GPUs
- Learn about NVIDIA Tensor Cores and mixed-precision
- Learn about physics-informed machine learning

TensorFlow

TU Delft



# TensorFlow

- A free and open-source software library for machine learning developed by the Google Brain Team
- TensorFlow 1.x released in March 2018 (deprecated)
- **TensorFlow 2.x** release in September 2019, most current release 2.10.0 September 2022
- Supports CPUs, NVIDIA GPUs and tensor processing units (TPUs) on shared- and distributed-memory hardware
- Runs on Linux, Windows, and macOS (no GPU support)
- Easy installation via

```
conda install -c conda-forge cudatoolkit=11.2 cudnn=8.1.0
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CONDA_PREFIX/lib/
python3 -m pip install tensorflow
```

- For this course use

`singularity-tensorflow-gpu` (CPU-only version `singularity-tensorflow-cpu`)

TU Delft

# Getting started with TensorFlow

- Exercise: Start `python` and run the following script

```
import tensorflow as tf
print("Num CPUs available: ", len(tf.config.list_physical_devices("CPU")))
print("Num GPUs available: ", len(tf.config.list_physical_devices("GPU")))
```

- Expected output

```
Num CPUs available:  1
Num GPUs available:  1
```

- `tf.config.list_physical_devices(…)` retrieves information about the physical devices in your system

TU Delft

# Device naming conventions

- Short-hand notation

```
"/CPU:0", "/GPU:0", ...
```

- Regular notation

```
"/device:CPU:0", "/device:GPU:0", ...
```

- Fully qualified name notation

```
"/job:localhost/replica:0/task:0/device:CPU:0",
"/job:localhost/replica:0/task:0/device:GPU:0", ...
```

# Device placement conventions

- Many TensorFlow operations have both CPU and GPU implementations. By default, the GPU device is prioritized
- To find out which devices your operations and tensors are assigned to, put

```
tf.debugging.set_log_device_placement(True)
```

  as the first statement of your program

- Example

```
import tensorflow as tf
tf.debugging.set_log_device_placement(True)
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)

print(c)
```

# Device placement conventions, cont'd

- Expected output

```
2022-10-27 09:42:14.288855: I tensorflow/core/platform/cpu_feature_guard.cc:193] This
TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the
following CPU instructions in performance-critical operations:  AVX2 AVX512F FMA

To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

- This message is printed only once when the first TensorFlow operation is performed

# Device placement conventions, cont’d

- Expected output, cont'd

```
2022-10-27 09:42:17.397806: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1532] Created
device /job:localhost/replica:0/task:0/device:GPU:0 with 1533 MB memory:  -> device: 0, name:
Quadro P620, pci bus id: 0000:97:00.0, compute capability: 6.1

2022-10-27 09:42:17.504976: I tensorflow/core/common_runtime/eager/execute.cc:1323] Executing
op _EagerConst in device /job:localhost/replica:0/task:0/device:GPU:0
```

- The last message is printed for each TensorFlow operation

# Manual device placement

- Example

```
import tensorflow as tf
tf.debugging.set_log_device_placement(True)

with tf.device("/CPU:0"):
      a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
      b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
      c = tf.matmul(a, b)

      print(c)
```

- Expected output

```
2022-10-27 09:54:34.870856: I tensorflow/core/common_runtime/eager/execute.cc:1323] Executing
op MatMul in device /job:localhost/replica:0/task:0/device:CPU:0
```

TU Delft

# Manual device placement, cont'd

- Exercise: Try the following

```
import tensorflow as tf
tf.debugging.set_log_device_placement(True)

with tf.device("/CPU:0"):
      a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
      b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)

print(c)
```

- Obviously, the tensors `a` and `b` are created in the CPU but because the multiplication is performed on the GPU, duplicates of `a` and `b` need to be copied from the CPU to the GPU. That is very inefficient as we will see.

# Exercise: Performance measurement

- With

  ```
  tf.ones([n,m], tf.dtypes.float32)
  ```

  you create a new tensor with n-x-m single-precision values.
- Write a small program that generates tensors `a` and `b` and performs the multiplication `c = tf.matmul(a, b)`.
- Test your program both on CPU and GPU. For which problem sizes does the GPU outperform the CPU?
- You can measure the time in Python as follows

  ```
  import time
  tic = time.perf_counter()
  ...
  toc = time.perf_counter()
  print(f"Time spend {toc-tic:.2e} seconds")
  ```

- Also try other datatypes, e.g., `tf.dtypes.float16, tf.dtypes.float64`

# Monitoring and controlling memory consumption

- If you create too large matrices (e.g., 1000x1000 double-precision) TensorFlow will stop with an error.
- You can monitor the amount of memory consumed on a GPU device by

```
tf.config.experimental.get_memory_usage("GPU:0")
```

- You can also set an upper bound for the maximum amount of memory consumed on a GPU device. To do so, we first need to create a **virtual GPU** prior(!!!) to calling any GPU-related operation (do it after the imports)

```
gpus = tf.config.list_physical_devices("GPU")
tf.config.set_logical_device_configuration(
      gpus[0],
      [tf.config.LogicalDeviceConfiguration(memory_limit=1024)])
```

- Exercise: Test your previous program with a maximum of 1GB memory

TU Delft

# Virtual GPUs

- If you have more than one GPU in your system, TensorFlow selects the GPU with the lowest ID by default. You can explicitly select a particular GPU by `tf.device("/device:GPU:1")`.
- Our systems here only have one GPU. If you want to develop code for a multi-GPU setup, you can create multiple **virtual GPUs** (e.g., 2 GPUs with 768MB memory each) as follows

```
gpus = tf.config.list_physical_devices("GPU")
tf.config.set_logical_device_configuration(
      gpus[0],
      [tf.config.LogicalDeviceConfiguration(memory_limit=768),
       tf.config.LogicalDeviceConfiguration(memory_limit=768)])
```

- You can list all virtual GPUs as follows

```
tf.config.list_logical_devices("GPU")
```

- Expected output

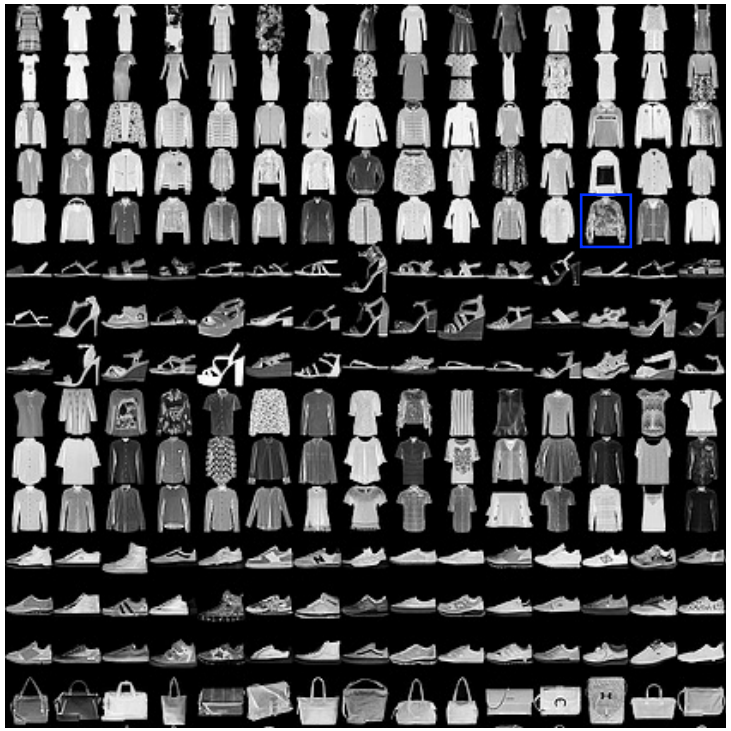
```
[LogicalDevice(name='/device:GPU:0', device_type='GPU'),
 LogicalDevice(name='/device:GPU:1', device_type='GPU')]
```

Try it yourself

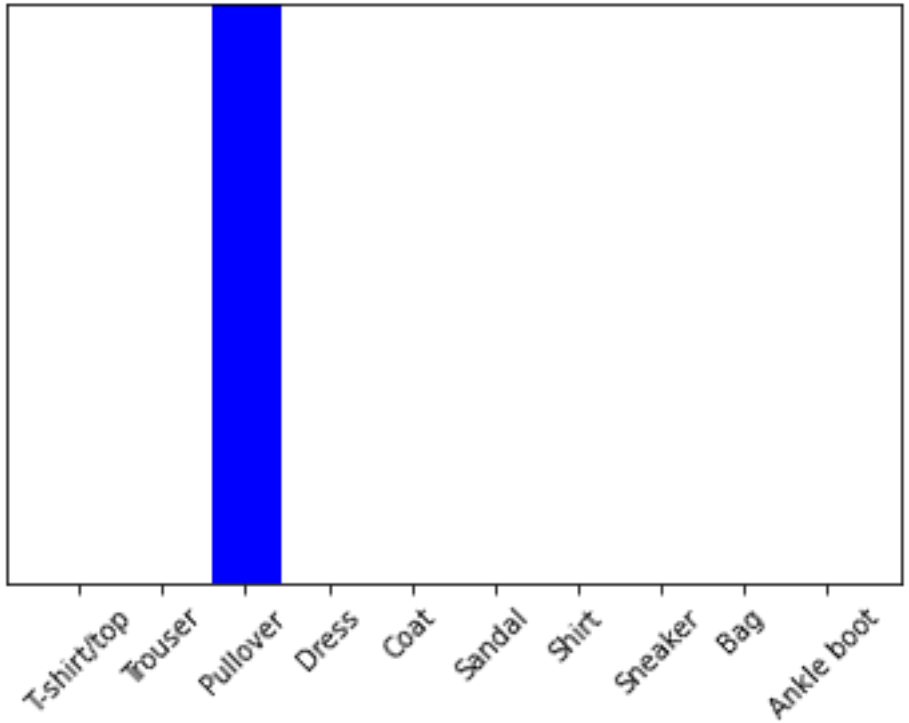
TU Delft

# Basic image classifier

- Fashion MNIST data base



FFNN




TU Delft

# Importing the Fashion MNIST dataset

- TensorFlow provides a straightforward way to import the 70,000-item fashion MNIST dataset

```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

- The so-loaded data is already separated into 60,000 images/labels for training and 10,000 images/labels for testing
- Let us assign human-readable class labels

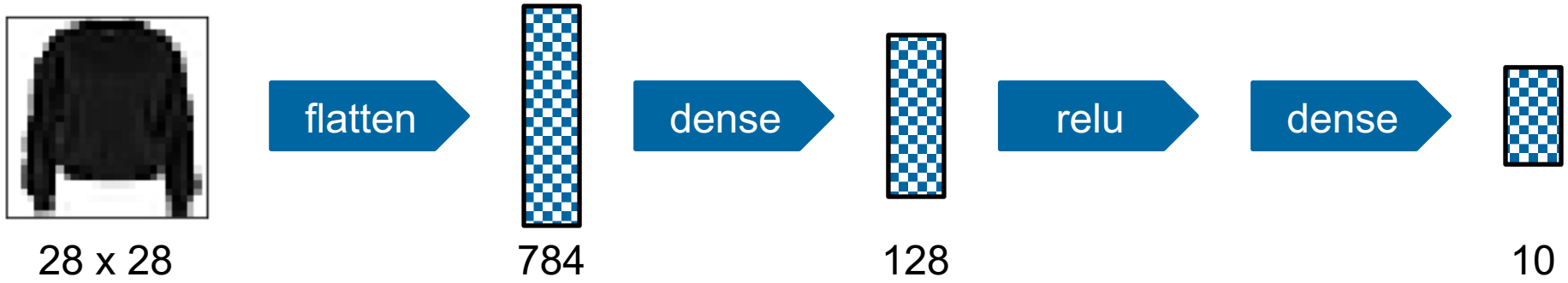
```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

- We finally need to rescale the pixel values to the range 0 to 1 (instead of 0 to 255)

```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

# Creating the model

- Feed-forward neural network architecture




```
model = tf.keras.Sequential([
            tf.keras.layers.Flatten(input_shape=(28, 28)),
            tf.keras.layers.Dense(128, activation='relu'),
            tf.keras.layers.Dense(10)
        ])
```

# Compiling and training the model

- Before the model is ready for training, it needs a few more settings:
  - Loss function
  - Optimizer
  - Metrics
- For the basic image classifier let us go with the following settings

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

- Now we are ready to train the model

```
model.fit(train_images, train_labels, epochs=10)
```

# Evaluating the model's accuracy

- The first step after training is to evaluate the model's accuracy

```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print('\nTest accuracy:', test_acc)
```

- Let us use the model to predict the labels of the test data

```
probability_model = tf.keras.Sequential([model,
                                         tf.keras.layers.Softmax()])
predictions = probability_model.predict(test_images)
predictions[0]
```

- Expected output

```
array([1.3835326e-08, 2.7011181e-11, 2.6019606e-10, 5.6872784e-11,
       1.2070331e-08, 4.1874609e-04, 1.1151612e-08, 5.7000564e-03,
       8.1178889e-08, 9.9388099e-01], dtype=float32)
```

TU Delft

# Exercise: Basic image classifier

- Open the file

  `tensorflow/02_basic_classifier/basic_classifier.py`

  and experiment with the different settings. You can change the following:

  - Loss function
  - Optimizer
  - Metrics
  - CPU vs. GPU
  - Increase/decrease the number of epochs
  - Add more hidden layers, change the number of neurons per layer, test different activation functions
  - …

# Performance profiling

- TensorFlow provides a graphical tool – **TensorBoard** – for inspecting your model and analyzing its performance, i.e.
  - tracking and visualizing metrics such as loss and accuracy
  - visualizing the model graph (ops and layers)
  - viewing histograms of weights, biases, or other tensors as they change over time
  - projecting embeddings to a lower dimensional space
  - displaying images, text, and audio data
  - profiling TensorFlow programs
- Installation

```
python3 -m pip install tensorboard tensorboard_plugin_profile
```

- For installation issues see

```
https://www.tensorflow.org/guide/profiler#install_the_profiler_and_gpu_prerequisites
```

# Instructing your code to generate profiling data

- Create a TensorBoard callback

```
from datetime import datetime
…
logs = "logs/" + datetime.now().strftime("%Y%m%d-%H%M%S")
tboard_callback = tf.keras.callbacks.TensorBoard(log_dir = logs,
                                                 histogram_freq = 1,
                                                 profile_batch = '500,520')
```

- Inject callback into model fitting

```
model.fit(train_images, train_labels, epochs=10, callbacks = [tboard_callback])
```

- Rerun your model and open TensorBoard

```
tensorboard --logdir logs
```

TU Delft

# TensorBoard

- Live demo

  `tensorflow/02_basic_classifier/basic_classifier_profiler.py`

TU Delft

# Optimizing the data loading

- Installation

```
python3 -m pip install tensorflow_datasets
```

- Modify your code

```
import tensorflow_datasets as tfds
tfds.disable_progress_bar()

(ds_train, ds_test), ds_info =
tfds.load('fashion_mnist',
          split=['train', 'test'],
          shuffle_files=True,
          as_supervised=True,
          with_info=True )
```

```
~~fashion_mnist = tf.keras.datasets.fashion_mnist~~
~~(train_images, train_labels),~~
      ~~(test_images, test_labels) =~~
      ~~fashion_mnist.load_data()~~
```

TU Delft

# Optimizing the data loading, cont'd

- Modify your code

```
def normalize_img(image, label):
      return tf.cast(image, tf.float32) / 255., label
```

- Set the batch size and enable caching and data prefetching

```
ds_train = ds_train.map(normalize_img)
ds_train = ds_train.batch(128)
ds_train = ds_train.cache()
ds_train = ds_train.prefetch(tf.data.experimental.AUTOTUNE)

ds_test = ds_test.map(normalize_img)
ds_test = ds_test.batch(128)
ds_test = ds_test.cache()
ds_test = ds_test.prefetch(tf.data.experimental.AUTOTUNE)
```

TU Delft

# Exercise: Basic image classifier, cont'd

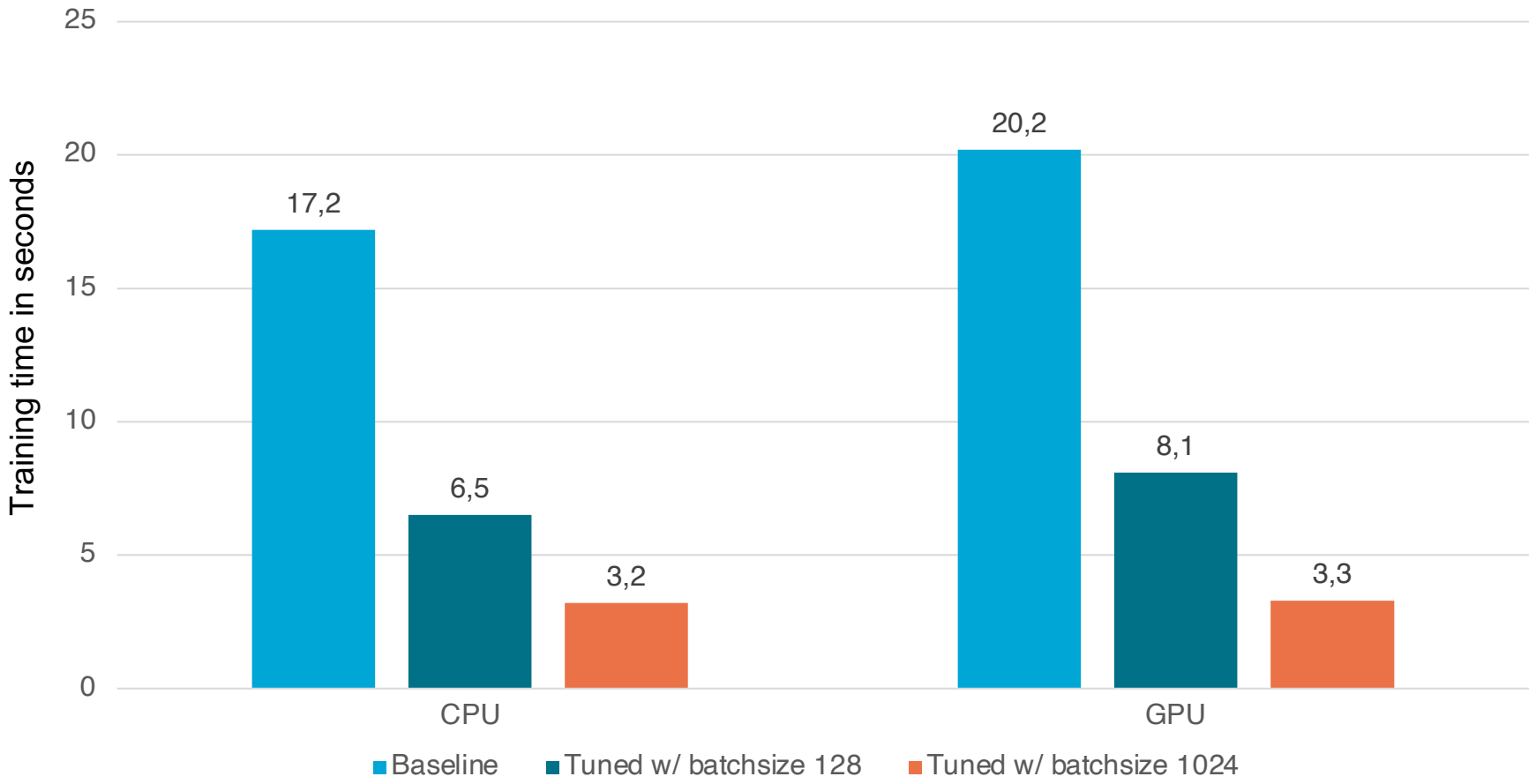
- Open the file

  `tensorflow/02_basic_classifier/basic_classifier_dataloader.py`

  and experiment with the different settings. You can change the following:

  - Batch size
  - Caching
  - Prefetching
  - CPU vs. GPU
  - …

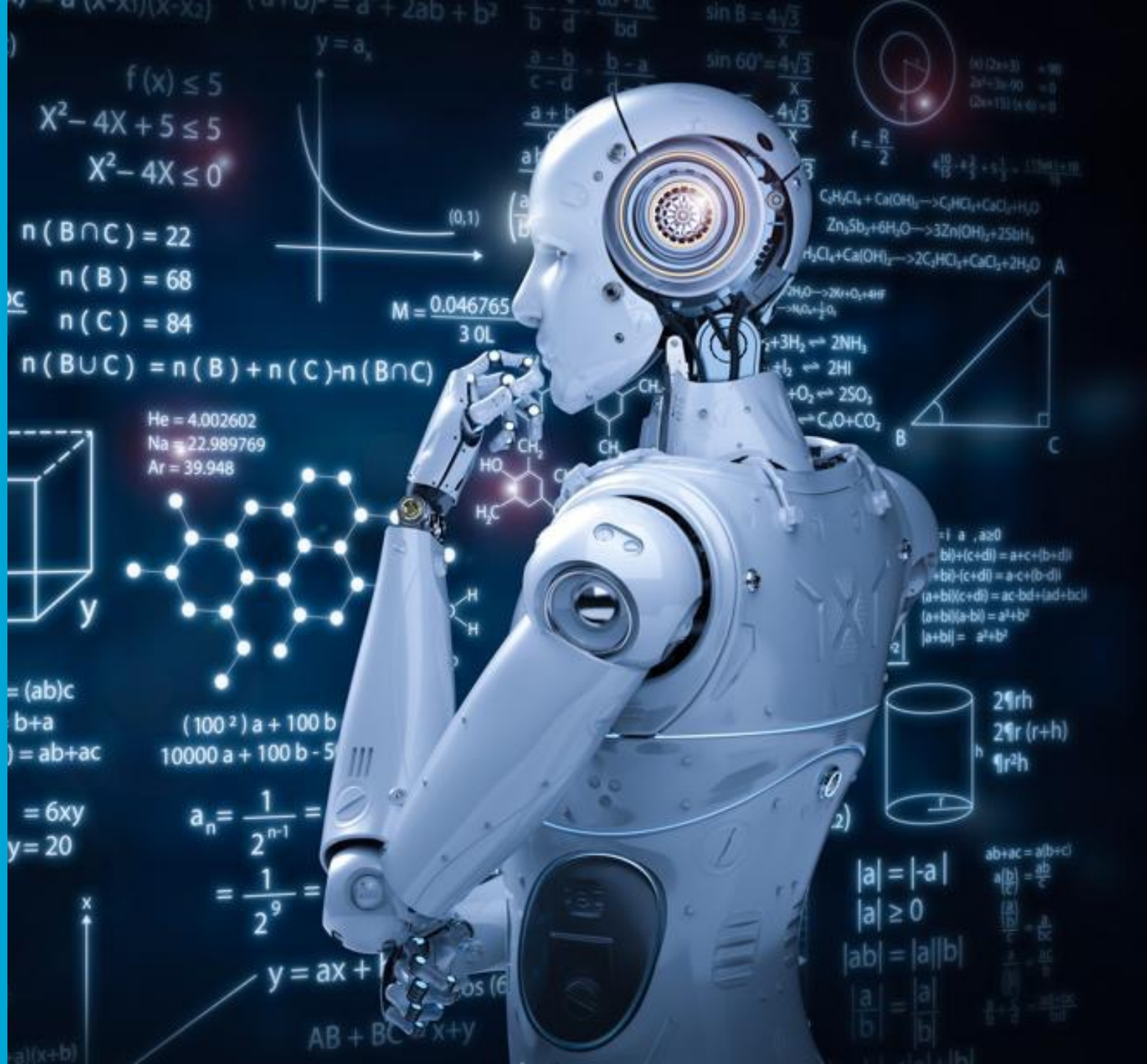
# Exercise: Basic image classifier, cont'd




TU Delft

PyTorch

TU Delft



# PyTorch

- A free and open-source software library for machine learning developed by Meta AI, now part of the Linux Foundation
- **PyTorch 1.13.0** released October 28th, 2022
- Supports CPUs, AMD/NVIDIA GPUs and tensor processing units (TPUs) on shared- and distributed-memory hardware
- Runs on Linux, Windows, and macOS
- Provides a Python and a C++ API
- For this course we install it using (`exit` singularity before!!!)

```
pip install --user torch torchvision
```

TU Delft

# Getting started with PyTorch

- Exercise: Start `python` and run the following script

```
import torch
print("CUDA device available: ",      torch.cuda.is_available())
print("Num CUDA devices available: ", torch.cuda.device_count())
```

- Expected output

```
CUDA device available:  True
Num CUDA devices available:  1
```

TU Delft

# Selecting devices

- PyTorch has a different philosophy for specifying devices. By default, the CPU is used.
- Create a device object

```
cuda  = torch.device('cuda')        # Default CUDA device
```

and specify the device during tensor creation

```
a = torch.ones(100, 100, device=cuda)
```

or transfer the tensor from CPU to GPU (which is less efficient)

```
a = torch.ones(100, 100).cuda()
```

or

```
a = torch.ones(100, 100).to(cuda)
```

# Selecting devices, cont'd

- PyTorch has a different philosophy for specifying devices. By default, the CPU is used.
- Create a device object

```
cuda0 = torch.device('cuda:0')      # CUDA device no.0
```

and specify the device during tensor creation

```
a = torch.ones(100, 100, device=cuda0)
```

or transfer the tensor from CPU to GPU (which is less efficient)

```
a = torch.ones(100, 100).cuda(cuda0)
```

or

```
a = torch.ones(100, 100).to(cuda0)
```

# Selecting devices, cont'd

- While `'cuda:0'` selects a specific device, `'cuda'` just selects the default CUDA device. PyTorch allows you to create a context that specifies the default CUDA device

```
cuda  = torch.device('cuda')        # Default CUDA device
cuda0 = torch.device('cuda:0')      # CUDA device no.0

with toch.cuda.device(1):

      a = torch.ones(100, 100, device=cuda)
      b = torch.ones(100, 100).cuda(cuda)     # all three will end up on GPU1
      c = torch.ones(100, 100).to(cuda)

      e = torch.ones(100, 100, device=cuda0)
      f = torch.ones(100, 100).cuda(cuda0)    # all three will end up on GPU0
      g = torch.ones(100, 100).to(cuda0)
```

# Selecting the data type

- The default data type can be specified globally using

```
torch.set_default_type(torch.float32)   # or torch.float16, torch.float64, …
```

- The data type per tensor can be specified during creation or via conversion

```
a = torch.ones(100, 100, dtype=torch.float64)
b = a.to(torch.float32)
```

- Exercise: Reimplement the 'matmul example' in PyTorch.
  - The command for retrieving the amount of memory allocated currently on the GPU is `torch.cuda.memory_allocated()`
  - The command for retrieving the amount of maximally allocated memory on the GPU is `torch.cuda.max_memory_allocated()`

# Basic image classifier, revisited

- A word of caution, PyTorch is much more low level than TensorFlow. This can be a curse and a blessing at the same time. We have full control over what we do, but we have to do it manually. Anyway, this is a course on GPU programming, not on PyTorch. So, we take the following source code 'as is'.
- Let's get started

```
# Torch and TorchVision
import torch
import torchvision
import time

# Hyperparmeters
batch_size = 128
```

# Basic image classifier, revisited

- Data loader

```
fashion_mnist = torchvision.datasets.FashionMNIST(
      root='~/pytorch_datasets',
      train=True,
      transform=torchvision.transforms.ToTensor(),
      download=True)

ds_train, ds_val = torch.utils.data.random_split(fashion_mnist, [50000, 10000])
ds_test = torchvision.datasets.FashionMNIST(
      root='~/pytorch_datasets',
      train=False,
      transform=torchvision.transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(ds_train, batch_size, shuffle=True)
val_loader   = torch.utils.data.DataLoader(ds_val, batch_size*2)
test_loader  = torch.utils.data.DataLoader(ds_test, batch_size*2)
```

# Basic image classifier, revisited

- Model

```
class FMnistModel(torch.nn.Module):
      def __init__(self):
            super().__init__()
            self.linear = torch.nn.Linear(28*28, 10)

      def forward(self, xb):
            xb = xb.reshape(-1, 28*28)
            out = self.linear(xb)
            return out

      def training_step(self, batch):
            images, labels = batch
            out = self(images)                                    # Generate predictions
            loss = torch.nn.functional.cross_entropy(out, labels) # Calculate loss
            return loss
```

TU Delft

# Basic image classifier, revisited

- Model, cont'd

```
class FMnistModel(torch.nn.Module):
      def validation_step(self, batch):
            images, labels = batch
            out = self(images)                                    # Generate predictions
            loss = torch.nn.functional.cross_entropy(out, labels)   # Calculate loss
            acc = accuracy(out, labels)                           # Calculate accuracy
            return {'val_loss': loss.detach(), 'val_acc': acc.detach()}

      def validation_epoch_end(self, outputs):
            batch_losses = [x['val_loss'] for x in outputs]
            epoch_loss = torch.stack(batch_losses).mean()         # Combine losses
            batch_accs = [x['val_acc'] for x in outputs]
            epoch_acc = torch.stack(batch_accs).mean()            # Combine accuracies
            return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}
```

# Basic image classifier, revisited

- Model, cont'd

```
class FMnistModel(torch.nn.Module):
      def epoch_end(self, epoch, result):
            print("Epoch [{}], val_loss: {:.4f},
           val_acc: {:.4f}".format(epoch, result['val_loss'], result['val_acc']))
```

- Create the model

```
model = FMnistModel()
```

TU Delft

# Basic image classifier, revisited

- Accuracy and evaluator function

```
def accuracy(outputs, labels):
      _, preds = torch.max(outputs, dim=1)
      return torch.tensor(torch.sum(preds == labels).item() / len(preds))

def evaluate(model, val_loader):
      outputs = [model.validation_step(batch) for batch in val_loader]
      return model.validation_epoch_end(outputs)
```

# Basic image classifier, revisited

- Fitting function

```
def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.SGD):
      history = []
      optimizer = opt_func(model.parameters(), lr)
      for epoch in range(epochs):
            # Training phase
            for batch in train_loader:
                  loss = model.training_step(batch)
                  loss.backward()
                  optimizer.step()
                  optimizer.zero_grad()
            # Validation phase
            result = evaluate(model, val_loader)
            model.epoch_end(epoch, result)
            history.append(result)
      return history
```

# Basic image classifier, revisited

- And here we go …

```
epochs = 10
lr = 0.1
tic = time.perf_counter()
history = fit(epochs, lr, model, train_loader, val_loader)
toc = time.perf_counter()
print(f"Spent {(toc-tic)} seconds on training")
```

- Full source code in file

```
pytorch/02_basic_classifier/basic_classifier_cpu.py
```

TU Delft

# Basic image classifier on the GPU

- Create a device

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

- Move data onto device

```
class FMnistModel(torch.nn.Module):

    def training_step(self, batch):
        images, labels = batch
        images = images.to(device), labels = labels.to(device)
        out = self(images)                                  # Generate predictions
        loss = torch.nn.functional.cross_entropy(out, labels) # Calculate loss
        return loss
```

TU Delft

# Basic image classifier on the GPU

- Move data onto device, cont'd

```
class FMnistModel(torch.nn.Module):

    def validation_step(self, batch):
        images, labels = batch
        images = images.to(device), labels = labels.to(device)
        out = self(images)                                    # Generate predictions
        loss = torch.nn.functional.cross_entropy(out, labels)   # Calculate loss
        acc = accuracy(out, labels)                           # Calculate accuracy
        return {'val_loss': loss.detach(), 'val_acc': acc.detach()}
```

- Move model onto device

```
model = FMnistModel().to(device)
```

TU Delft

# Basic image classifier on the GPU

- This code will run on the GPU, but it will not be faster than the CPU version.
- Let's tune the data loader

```
train_loader = torch.utils.data.DataLoader(ds_train, batch_size, shuffle=True,
                    pin_memory=True, num_workers = 8, prefetch_factor=8)
```

  - `pin_memory` enables fast memory copies between host and device memory
  - `num_workers` sets the number of parallel data loading threads
  - `prefetch_factor` sets the number of batches to be prefetched
- Full source code in file

```
pytorch/02_basic_classifier/basic_classifier_gpu.py
```

TU Delft

# Exercise: Basic image classifier

- Open the file(s)

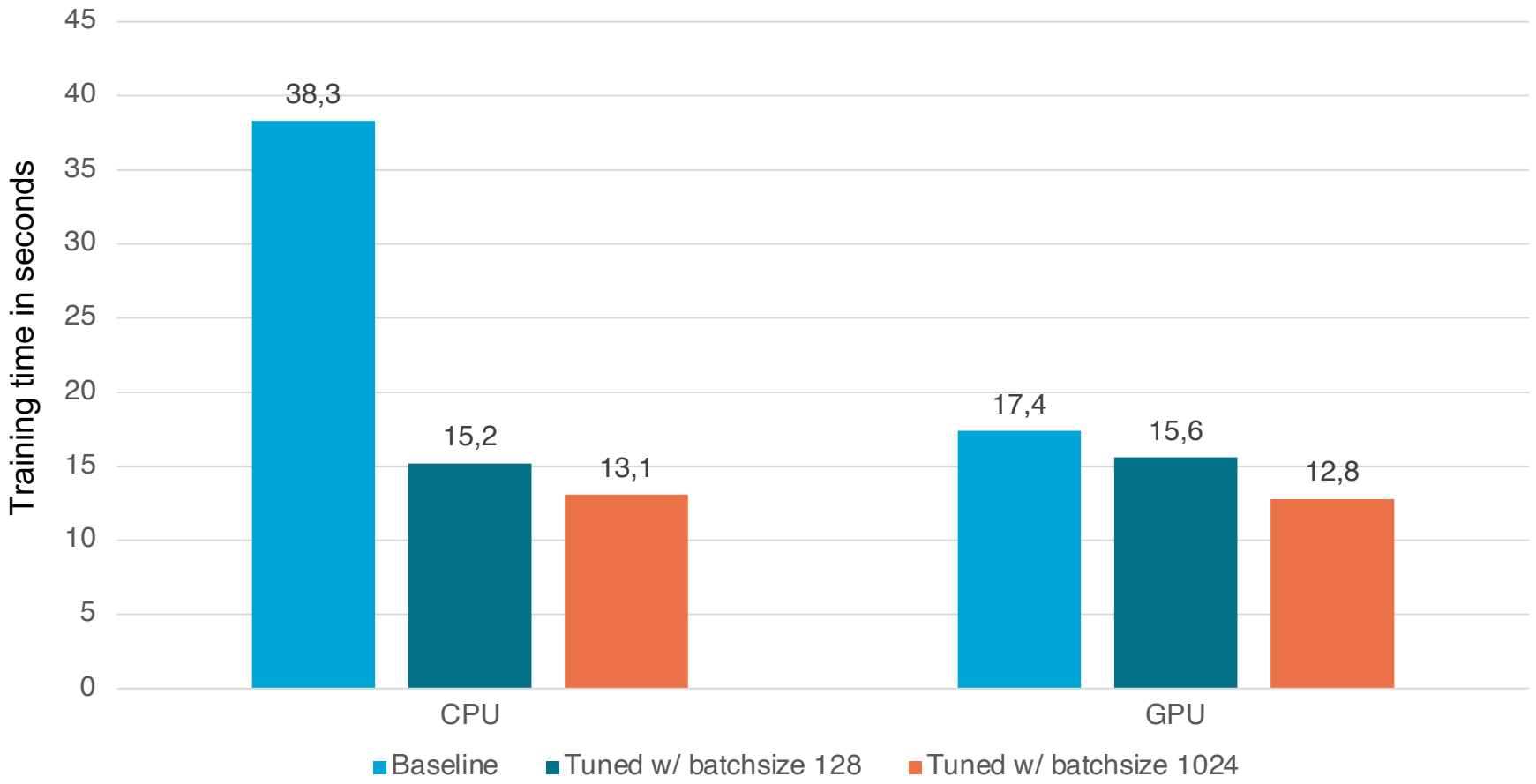
  `pytorch/02_basic_classifier/basic_classifier_cpu.py` (for CPU)
  `pytorch/02_basic_classifier/basic_classifier_gpu.py` (for GPU)

  and experiment with the different settings. You can change the following:

  - Loss function
  - Optimizer
  - Metrics
  - CPU vs. GPU, `pin_memory, num_workers, prefetch_factor` (try the latter two for CPU as well!!!)
  - Increase/decrease the number of epochs
  - Add more hidden layers, change the number of neurons per layer, test different activation functions
  - …

TU Delft

# Exercise: Basic image classifier




TU Delft

# PyTorch profiler

- PyTorch provides a simple profiler API

```
with torch.profiler.profile(
    activities=[torch.profiler.ProfilerActivity.CPU,
                torch.profiler.ProfilerActivity.CUDA],
    record_shapes=True,  # record shapes of operator inputs
    profile_memory=True, # report amount of memory consumed
    use_cuda=True        # measure execution time of CUDA kernels
    ) as prof:
    with torch.profiler.record_function("model_inference"):
           history = fit(epochs, lr, model, train_loader, val_loader)
```

- It is advisable to **not** turn on all measurements at the same time because the output will be very busy

# PyTorch profiler, cont'd

- Let's print some stats about the execution time

```
print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=10))
```

- Expected output (some columns removed)

```
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------------
                                                 Name    Self CPU %      Self CPU   CPU total %     CPU total  CPU time avg
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------------
                                        model_inference        32.93%        6.905s        94.52%       19.822s       19.822s
enumerate(DataLoader)#_MultiProcessingDataLoaderIter...        43.69%        9.163s        44.08%        9.244s       2.135ms
                                             aten::to         0.18%      37.257ms         5.15%        1.080s      78.612us
                                        aten::_to_copy         0.39%      80.947ms         5.05%        1.059s     117.264us
                                          aten::linear         0.13%      27.341ms         4.63%     971.712ms     225.455us
                                           aten::addmm         1.40%     292.880ms         4.23%     886.854ms     205.767us
                                           aten::copy_         0.42%      88.002ms         4.18%     875.780ms      96.986us
                                cudaStreamSynchronize          3.14%     659.071ms         3.15%     659.756ms      73.063us
                                     cudaLaunchKernel          2.97%     622.524ms         2.97%     623.152ms       9.475us
                                Optimizer.step#SGD.step         1.67%     349.641ms         2.58%     540.053ms     138.121us
-------------------------------------------------------  ------------  ------------  ------------  ------------  ------------
Self CPU time total: 20.971s
Self CUDA time total: 1.067s
```

TU Delft

# PyTorch profiler, cont'd

- Let's print some stats about the execution time

```
print(prof.key_averages().table(sort_by="cpu_time_total", row_limit=10))
```

- Expected output (some columns removed)

```
-----------  -----------  -----------  -----------  -----------  -----------  -----------  -----------  -----------
Self CUDA   Self CUDA %    CUDA total  CUDA time avg      CPU Mem  Self CPU Mem      CUDA Mem  Self CUDA Mem    # of Calls
-----------  -----------  -----------  -----------  -----------  -----------  -----------  -----------  -----------
     0.000us         0.00%        1.016s        1.016s          -4 b      -3.82 Mb      31.50 Kb      -1.82 Gb            1
     0.000us         0.00%       5.358ms       1.237us     -16.91 Kb      -4.76 Mb           0 b           0 b         4330
     0.000us         0.00%     627.680ms      45.683us          40 b           4 b       1.76 Gb      19.12 Mb        13740
     0.000us         0.00%     633.519ms      70.157us          40 b           0 b       1.76 Gb    -785.00 Kb         9030
     0.000us         0.00%     238.468ms      55.329us           0 b           0 b      22.90 Mb       1.01 Mb         4310
   183.202ms        17.17%     238.732ms      55.390us           0 b           0 b      22.90 Mb      -4.18 Gb         4310
   607.987ms        56.97%     635.022ms      70.324us           0 b           0 b           0 b           0 b         9030
    14.127ms         1.32%      14.393ms       1.594us           0 b           0 b           0 b           0 b         9030
    91.527ms         8.58%      91.687ms       1.394us           0 b           0 b           0 b           0 b        65768
     0.000us         0.00%      28.341ms       7.248us     -15.27 Kb    -781.48 Kb           0 b           0 b         3910
-----------  -----------  -----------  -----------  -----------  -----------  -----------  -----------  -----------
Self CPU time total: 20.971s
Self CUDA time total: 1.067s
```

TU Delft

# PyTorch profiler, cont'd

- Let's print some stats about the execution time

```
print(prof.key_averages(group_by_input_shape=True).
      table(sort_by="cpu_time_total", row_limit=10))
```

- Expected output (some columns removed)

```
------------------------------------------------------- ------------------------------------------------------
                                                  Name Input Shapes
------------------------------------------------------- ------------------------------------------------------
                                       model_inference []
enumerate(DataLoader)#_MultiProcessingDataLoaderIter... []
                                         aten::linear [[128, 784], [10, 784], [10]]
                                          aten::addmm [[10], [128, 784], [784, 10], [], []]
                                             aten::to [[128, 1, 28, 28], [], [], [], [], [], [], []]
                                       aten::_to_copy [[128, 1, 28, 28], [], [], [], [], [], []]
                                cudaStreamSynchronize []
                                          aten::copy_ [[128, 1, 28, 28], [128, 1, 28, 28], []]
                                    cudaLaunchKernel []
                               Optimizer.step#SGD.step []
------------------------------------------------------- ------------------------------------------------------
Self CPU time total: 20.971s
Self CUDA time total: 1.067s
```

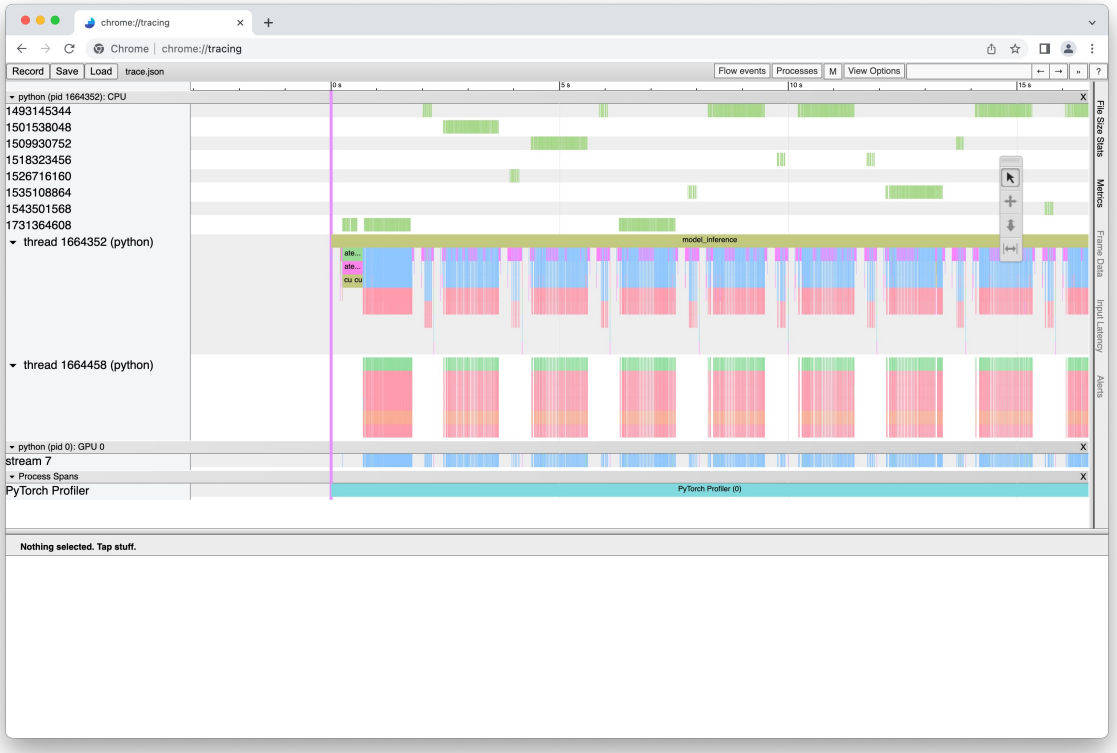
TU Delft

# PyTorch profiler, cont'd

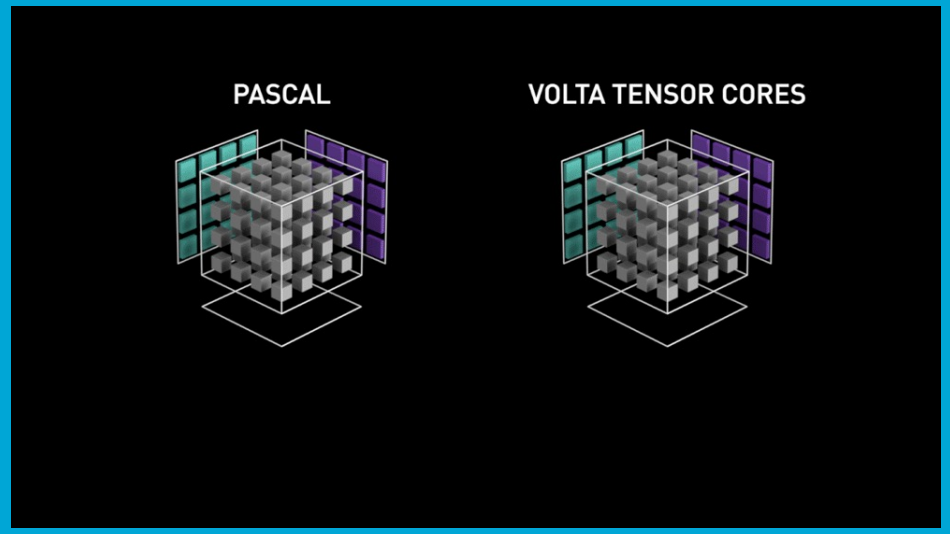
- You can also generate a trace file

```
prof.export_chrome_trace("trace.json")
```

and inspect it with the Chrome trace viewer (`chrome://tracing`)




TU Delft




TU Delft

# Tensor Cores

- Modern GPUs (Volta microarchitecture or better) have specialized Tensor Cores that enable very fast mixed-precision fused-multiply-add operations for small-size matrices, i.e.

$$\begin{pmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{pmatrix} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \cdot \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32

- Fused-multiply-add operations play an important role in machine learning (but also in other applications)

$$y = \sigma(Wx + b)$$

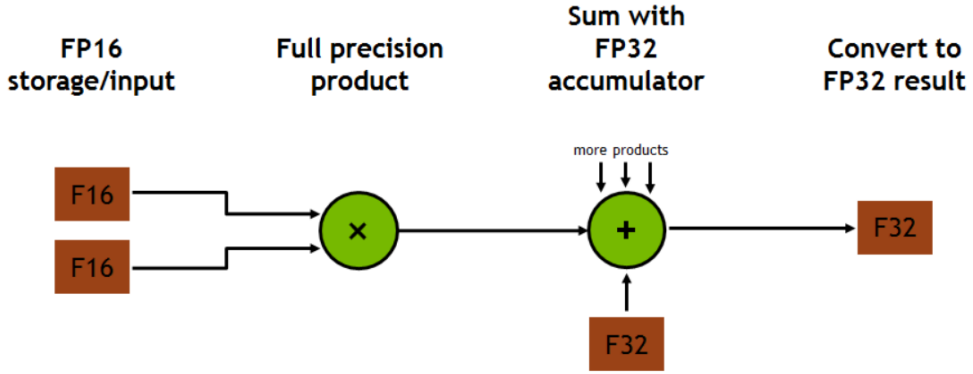
TU Delft

# Tensor Cores, cont'd

- Modern GPUs (Volta microarchitecture or better) have specialized Tensor Cores that enable very fast mixed-precision fused-multiply-add operations for small-size matrices, i.e.

$$\begin{pmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{pmatrix} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \cdot \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32




TU Delft

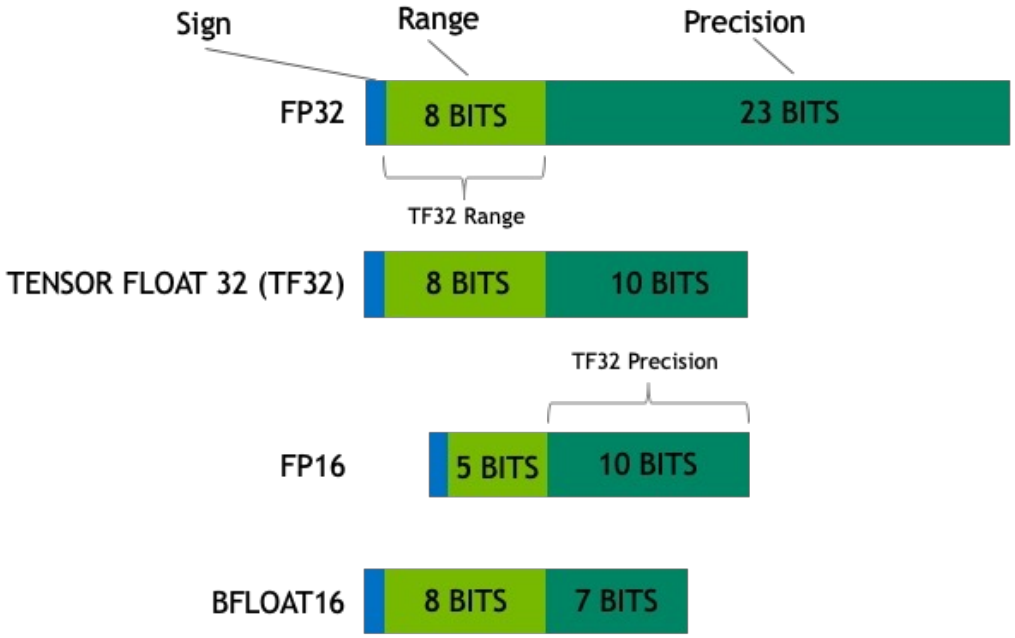
# Tensor Cores, cont'd

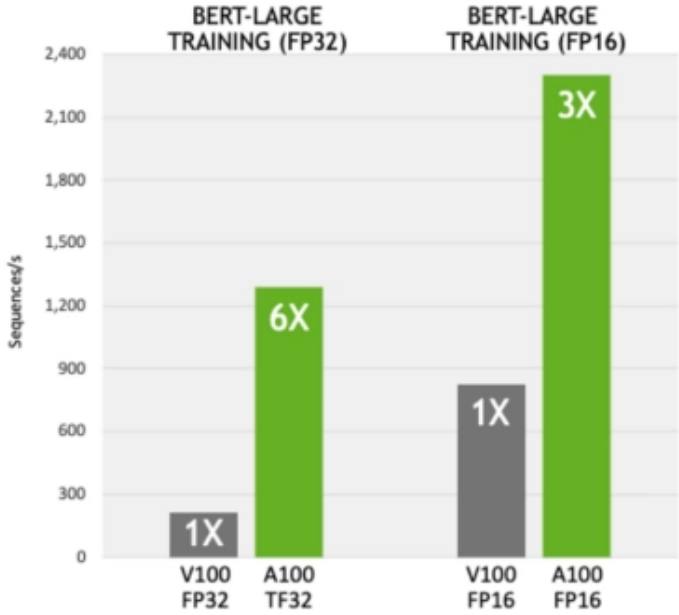
- With each new microarchitecture, NVIDIA increases the number of Tensor Cores and their capabilities

| | Supported CUDA Core precisions | | | | | | | | | Supported Tensor Core precisions | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | FP8 | FP16 | FP32 | FP64 | INT1 | INT4 | INT8 | TF32 | BF16 | FP8 | FP16 | FP32 | FP64 | INT1 | INT4 | INT8 | TF32 | BF16 |
| P4 | No | No | Yes | Yes | No | No | Yes | No | No | No | No | No | No | No | No | No | No | No |
| P100 | No | Yes | Yes | Yes | No | No | No | No | No | No | No | No | No | No | No | No | No | No |
| Volta | No | Yes | Yes | Yes | No | No | Yes | No | No | No | Yes | No | No | No | No | No | No | No |
| Turing | No | Yes | Yes | Yes | No | No | Yes | No | No | No | Yes | No | No | Yes | Yes | Yes | No | No |
| A100 | No | Yes | Yes | Yes | No | No | Yes | No | Yes | No | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes |
| H100 | No | Yes | Yes | Yes | No | No | Yes | No | Yes | Yes | Yes | No | Yes | No | No | Yes | Yes | Yes |

TU Delft

# New data formats TF32 and BF16



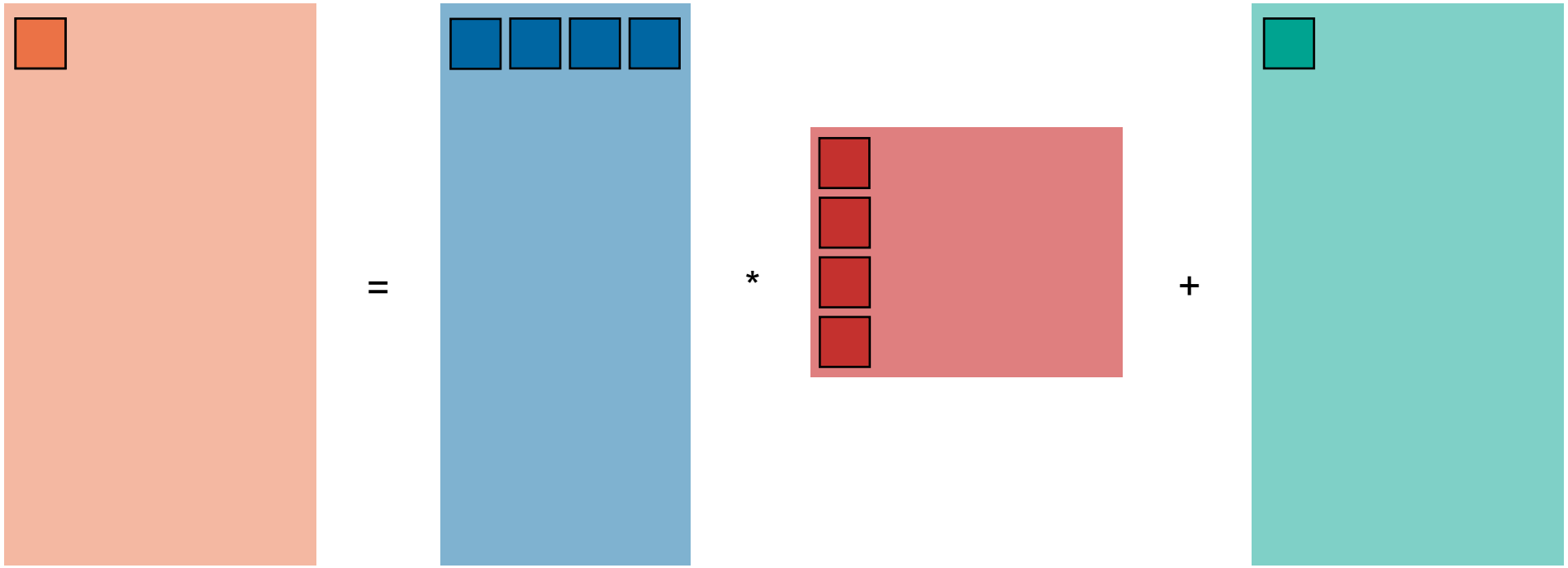




*BERT Large Training (FP32 & FP16) measures Pre-Training phase, uses PyTorch including (2/3) Phase1 with Seq Len 128 and (1/3) Phase 2 with Seq Len 512, V100 is DGX1 Server with 8xV100, A100 is DGX A100 Server with 8xA100, A100 uses TF32 Tensor Core for FP32 training*

TU Delft

# Utilization of Tensor Cores in practice


- Consider a large fused-multiply add problem with matrix sizes multiples of 16 (warp size)



# Warp Matrix Multiply Accumulation API

- Since CUDA 9, NVIDIA provides the Warp Matrix Multiply Accumulation (WMMA) API in the header file `mma.h`. All functions are in the namespace `nvcuda::wmma` but it is saver to keep `wmma` explicit throughout the code.
- The following example is taken from https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/
- Let's get started

```
#include <mma.h>
using namespace nvcuda;
```

- Strategy: Compute one 16x16 tile of the output matrix by a single warp

```
const int WMMA_M = 16;
const int WMMA_N = 16;
const int WMMA_K = 16;
```

# Warp Matrix Multiply Accumulation API, cont'd

- Declarations and initialization

```
__global__ void wmma_example(half *a, half *b, float *c,
                             int M, int N, int K,
                             float alpha, float beta)
{
    // Leading dimensions. Packed with no transpositions.
    int lda = M;
    int ldb = K;
    int ldc = M;

    // Tile using a 2D grid
    int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
    int warpN = (blockIdx.y * blockDim.y + threadIdx.y);
```

# Warp Matrix Multiply Accumulation API, cont'd

- Declarations and initialization

```
// Declare the fragments
wmma::fragment<wmma::matrix_a,    WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
wmma::fragment<wmma::matrix_b,    WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag

// Fill accumulator fragment with zeros
wmma::fill_fragment(acc_frag, 0.0f);
```

# Warp Matrix Multiply Accumulation API, cont'd

- Inner Loop

```
for (int i = 0; i < K; i += WMMA_K) {
  int aRow = warpM * WMMA_M;
  int aCol = i;
  int bRow = i;
  int bCol = warpN * WMMA_N;

  // Bounds checking
  if (aRow < M && aCol < K && bRow < K && bCol < N) {
      // Load the inputs
      wmma::load_matrix_sync(a_frag, a + aRow + aCol * lda, lda);
      wmma::load_matrix_sync(b_frag, b + bRow + bCol * ldb, ldb);

      // Perform the matrix multiplication
      wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
  }
}
```

# Warp Matrix Multiply Accumulation API, cont'd

- Finishing up

```
    // Load in current value of c, scale by beta, and add to result scaled by alpha
    int cRow = warpM * WMMA_M;
    int cCol = warpN * WMMA_N;
    if (cRow < M && cCol < N) {
        wmma::load_matrix_sync(c_frag, c + cRow + cCol * ldc, ldc, wmma::mem_col_major);
        for(int i=0; i < c_frag.num_elements; i++) {
            c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
        }
        // Store the output
        wmma::store_matrix_sync(c + cRow + cCol * ldc, c_frag, ldc, wmma::mem_col_major);
    }
}
```

# Warp Matrix Multiply Accumulation API, cont'd

- The NVIDIA Quadro P620 cards installed in our systems do not provide Tensor Cores ☹
- Live demo on DelftBlue with V100 GPU

# Mixed-precision

- By default, Tensorflow adopts 32-bit floating-point types (`tf.dtypes.float32`) in a model
- You can override this behavior globally by

```
tf.keras.backend.set_floatx('float64')
```

- The current default data type is returned by

```
tf.keras.backend.floatx()
```

- You can also control the data type per layer as follows

```
model = tf.keras.Sequential([
                    tf.keras.layers.Flatten(input_shape=(28, 28)),
                    tf.keras.layers.Dense(128, activation='relu', dtype=tf.float16),
                    tf.keras.layers.Dense(10, dtype=tf.float64)
                ])
```

# Mixed-precision, cond't

- A good way to inspect your models is by exporting it as JSON string and dumping it to the screen

```
import json
print(json.dumps(json.loads(model.to_json()), indent=2))
```

- Expected output

```
{
  "class_name": "Sequential",
  "config": {
    "name": "sequential_2",
    "layers": [
      {
        "class_name": "InputLayer",
        "config": {
          "batch_input_shape": [
            null,
            28,
            28
          ],
          "dtype": "float32",
          "sparse": false,
          "ragged": false,
          "name": "flatten_2_input"
        }
      },
```

```
"class_name": "Dense",
"config": {
  "name": "dense_4",
  "trainable": true,
  "dtype": "float16",
  "units": 128,
  "activation": "relu",
  "use_bias": true,
  "kernel_initializer": {
    "class_name": "GlorotUniform",
    "config": {
      "seed": null
    }
  },
  ...
```

```
"class_name": "Dense",
"config": {
  "name": "dense_5",
  "trainable": true,
  "dtype": "float64",
  "units": 10,
  "activation": "linear",
  "use_bias": true,
  "kernel_initializer": {
    "class_name": "GlorotUniform",
    "config": {
      "seed": null
    }
  },
  ...
```

TU Delft

# Mixed-precision , cont'd

- When you change from `float64` to `float32` to `float16` to … you
  - most likely reduce the accuracy of computations ☹
  - reduce the memory consumption by factor 2 each time ☺
  - speed-up the data transfer by factor 2 each time ☺
  - speed-up computations by factor 2 each time ☺

# Mixed-precision , cont'd

- Tensorflow provides an easy way to enable meaningful mixed-precision throughout your model

```
policy = tf.keras.mixed_precision.Policy('mixed_float16')
tf.keras.mixed_precision.set_global_policy(policy)
```

- With this policy each layer will use float16 for faster computations and float32 for variables for numeric stability

```
print('Compute dtype: %s' % policy.compute_dtype)
print('Variable dtype: %s' % policy.variable_dtype)
```

- Expected output

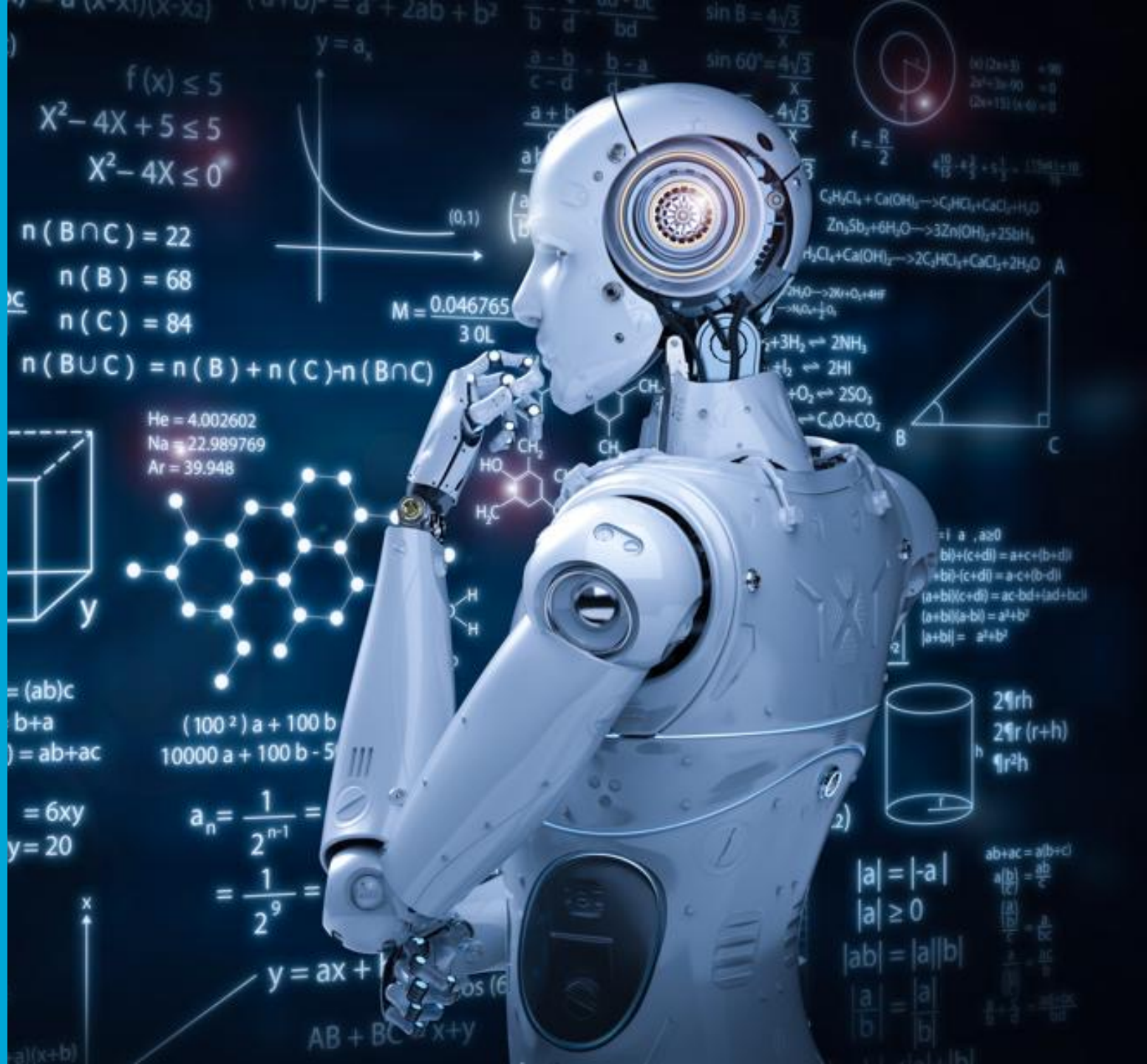
```
Compute dtype: float16
Variable dtype: float32
```

- On CPUs, mixed-precision will run significantly slower. On GPUs mixed-precision requires compute capability 7.0 or better. Otherwise, it is emulated and, again, your code will run significantly slower.

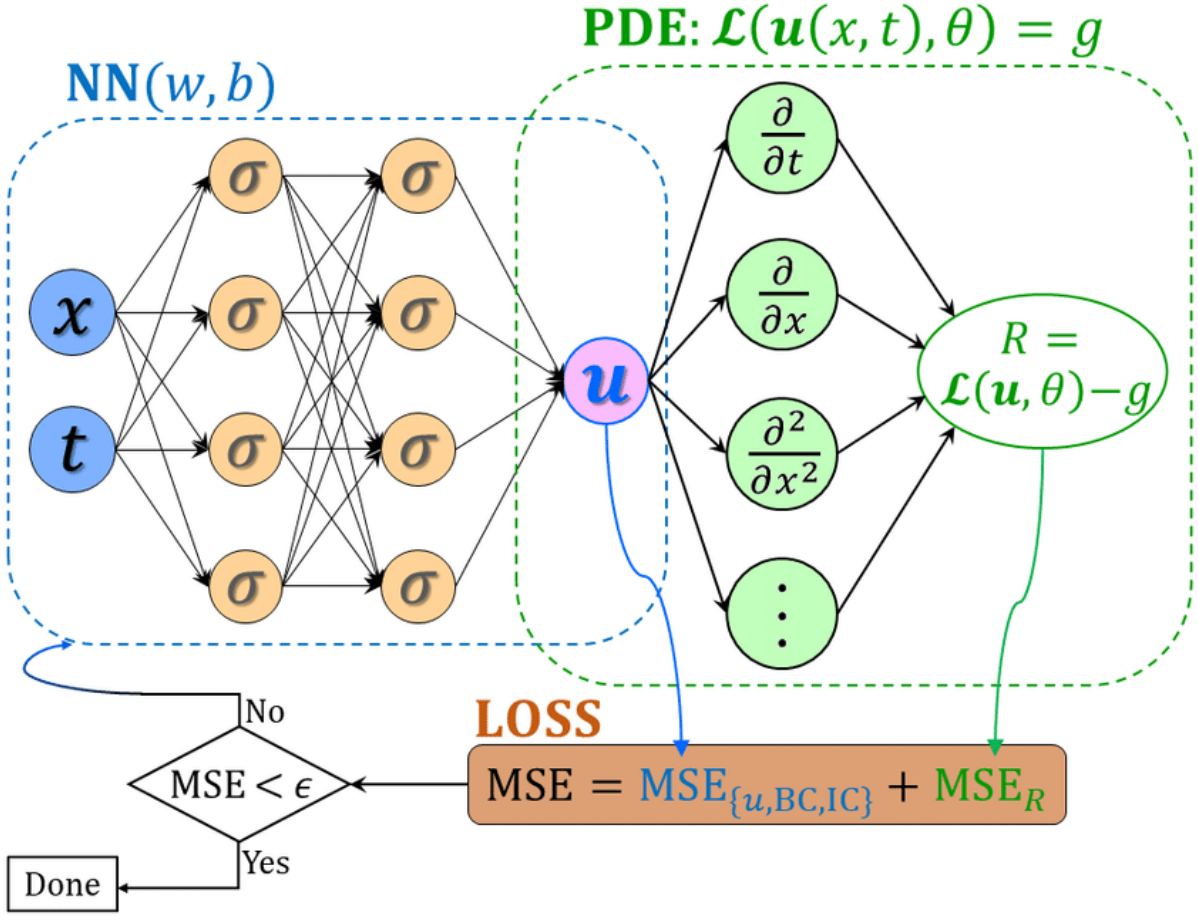
TU Delft

# Physics-informed machine learning

TU Delft

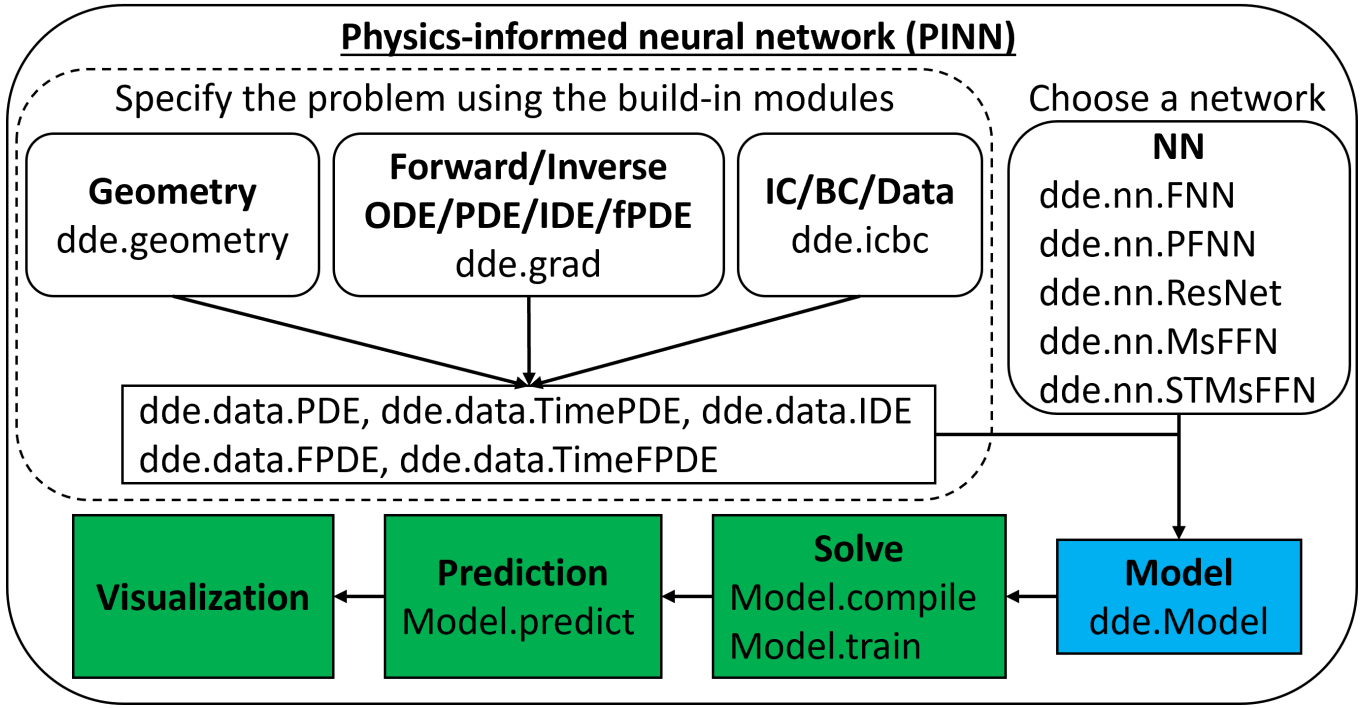


# Physics-informed neural networks (PINN) by Raissi et al. 2017

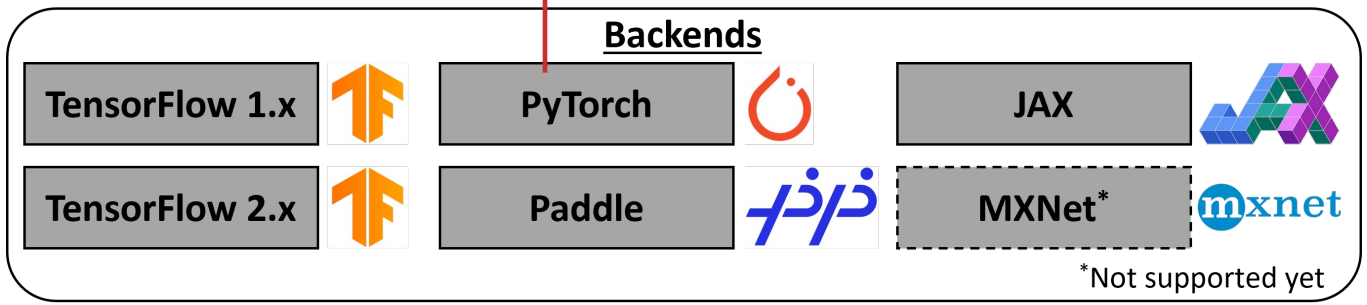



TU Delft

# DeepXDE




```
pip install --user deepxde
export DDE_BACKEND=pytorch
```




TU Delft

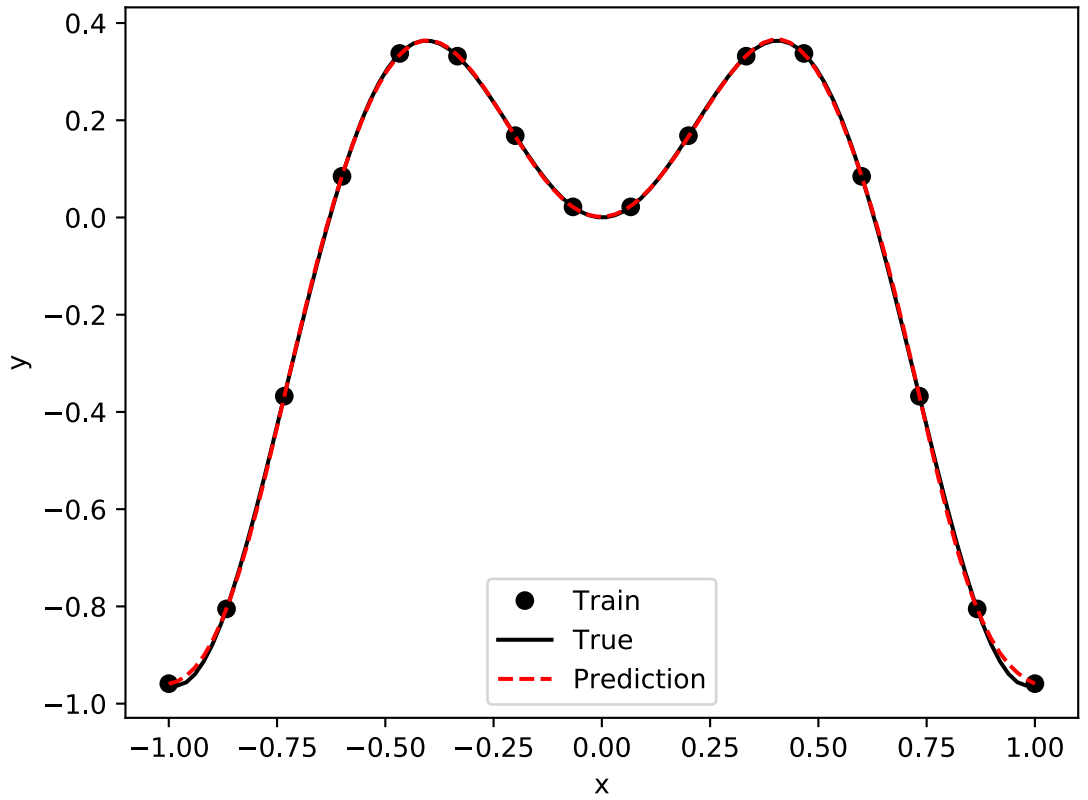
# Example: deepxde/01_function_approximation.py

```
import deepxde as dde
import numpy as np

def func(x):
    """
    x: array_like, N x D_in
    y: array_like, N x D_out
    """
    return x * np.sin(5.0 * x)


geom = dde.geometry.Interval(-1, 1)

num_train = 16
num_test = 100
```




TU Delft

# Example: deepxde/01_function_approximation.py

```
import deepxde as dde
import numpy as np

def func(x):
    """
    x: array_like, N x D_in
    y: array_like, N x D_out
    """
    return x * np.sin(5.0 * x)


geom = dde.geometry.Interval(-1, 1)

num_train = 16
num_test = 100
```

```
data = dde.data.Function(geom, func,
                                num_train, num_test)

activation = "tanh"
initializer = "Glorot uniform"
net = dde.nn.FNN([1] + [20] * 3 + [1],
                    activation, initializer)

model = dde.Model(data, net)
model.compile("adam", lr=0.001,
                  metrics=["l2 relative error"])
losshistory,
train_state = model.train(iterations=10000)

dde.saveplot(losshistory, train_state,
                issave=True, isplot=True)
```

TU Delft

# Example: deepxde/02_ode.py

```
def ode_system(x, y):

    """ODE system.
    dy1/dx = y2
    dy2/dx = -y1
    """
    y1, y2 = y[:, 0:1], y[:, 1:]
    dy1_x = dde.grad.jacobian(y, x, i=0)
    dy2_x = dde.grad.jacobian(y, x, i=1)

    return [dy1_x - y2, dy2_x + y1]

def boundary(_, on_initial):
    return on_initial

...
```

```
geom = dde.geometry.TimeDomain(0, 10)
ic1 = dde.icbc.IC(geom, lambda x: 0,
                  boundary, component=0)
ic2 = dde.icbc.IC(geom, lambda x: 1,
                  boundary, component=1)
data = dde.data.PDE(geom, ode_system, [ic1, ic2],
                    35, 2, solution=func, num_test=100)

layer_size = [1] + [50] * 3 + [2]
activation = "tanh"
initializer = "Glorot uniform"
net = dde.nn.FNN(layer_size, activation, initializer)

model = dde.Model(data, net)
model.compile("adam", lr=0.001,
              metrics=["l2 relative error"])

losshistory, train_state = model.train(iterations=20000)
```

TU Delft

# Example: deepxde/03_poisson_lshape.py

```
def pde(x, y):
    # Most backends
    dy_xx = dde.grad.hessian(y, x, i=0, j=0)
    dy_yy = dde.grad.hessian(y, x, i=1, j=1)
    return -dy_xx - dy_yy - 1

def boundary(_, on_boundary):
    return on_boundary

geom = dde.geometry.Polygon([[0, 0], [1, 0],
        [1, -1], [-1, -1], [-1, 1], [0, 1]])

bc = dde.icbc.DirichletBC(geom, lambda x: 0,
        boundary)
```

```
data = dde.data.PDE(geom, pde, bc, num_domain=1200,
                    num_boundary=120, num_test=1500)
net = dde.nn.FNN([2] + [50] * 4 + [1],
                 "tanh", "Glorot uniform")
model = dde.Model(data, net)

model.compile("adam", lr=0.001)
model.train(iterations=50000)
model.compile("L-BFGS")

losshistory, train_state = model.train()
dde.saveplot(losshistory, train_state,
             issave=True, isplot=True)
```

TU Delft