UNIVERSITY
OF OSLO



IN4310 Deep Learning for Image Analysis

ghadia@uio.no



PyTorch Library

PyTorch is a Python library that allows you to build deep learning models from different levels.

PyTorch allows you to write and perform linear algebra operations in a numpy-like way.

PyTorch provides GPU implementations for efficient and fast runs.

PyTorch Tensors

A tensor is a (multi-dimensional) array that stores numeric values (image, audio stream, text encoding...) akin to a NumPy array.

A tensor can be created in several ways:

- From list (of lists): `torch.Tensor([0,1,2])`
- From a Numpy array: `torch.from_numpy(a)`
- From a random number generator: `torch.rand(2, 3)`
- From a fixed value: `torch.full((2, 3), 3.141592)`
- ⋮

Tensor Properties

A tensor has three main properties:

- size: important for broadcasting rules.
- device: all tensors in an operation must be on the same device.
- data type: all tensors in an operation must be of the same data type. Some operations require specific data types for their input.

To change the device or data type of a tensor use `x.to()`. To change the shape use `x.reshape(new_shape)`. Nvidia GPUs are typically called `"cuda:GPU_index"`

Broadcasting

Broadcasting is a built-in functionality that allows one to perform mathematical operations even when tensor shapes are mismatched.

For example:

```
Tensor(2) * Tensor([3,4])
Tensor([[1, 2, 3, 4]]) * Tensor([[1], [2]])
             (1,4)                (2,1)
```

However, there are rules to it:

- Corresponding shape entries must be equal, or one of them should be equal to 1.
- The shape of the tensor with fewer dimensions can be appended by 1's from the left

Common Linear Algebra Operations

Dot product: `torch.dot(a,b)`

Matrix multiplication (no broadcasting): `torch.mm(a, b)`

Matrix multiplication (broadcasting): `torch.matmul(a, b)`

Matrix Inverse: `torch.linalg.inv(x)`

Matrix/vector norm: `torch.linalg.norm(x)`

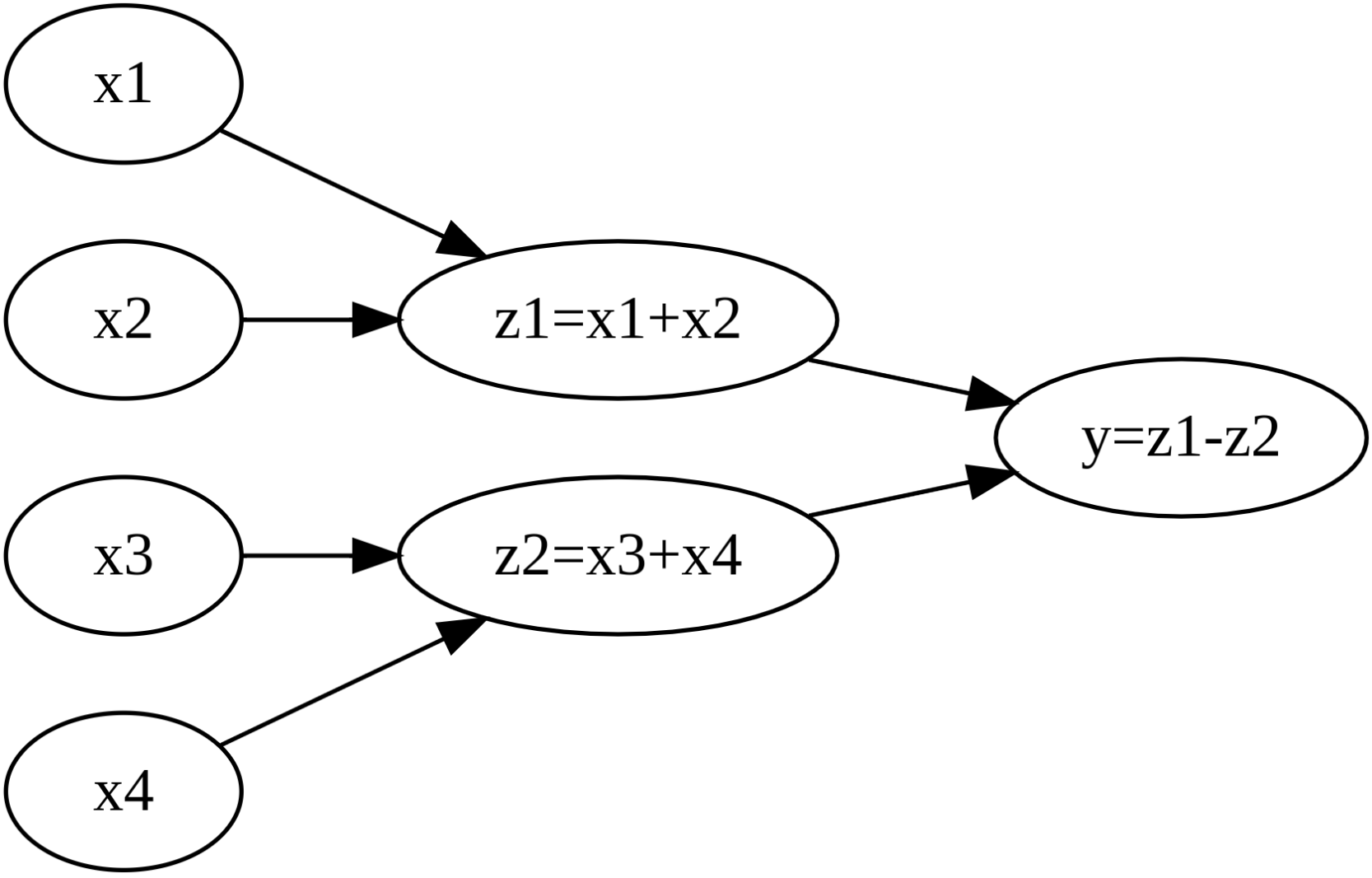
Autograd

In deep learning, we typically need to calculate the gradients to train a model.

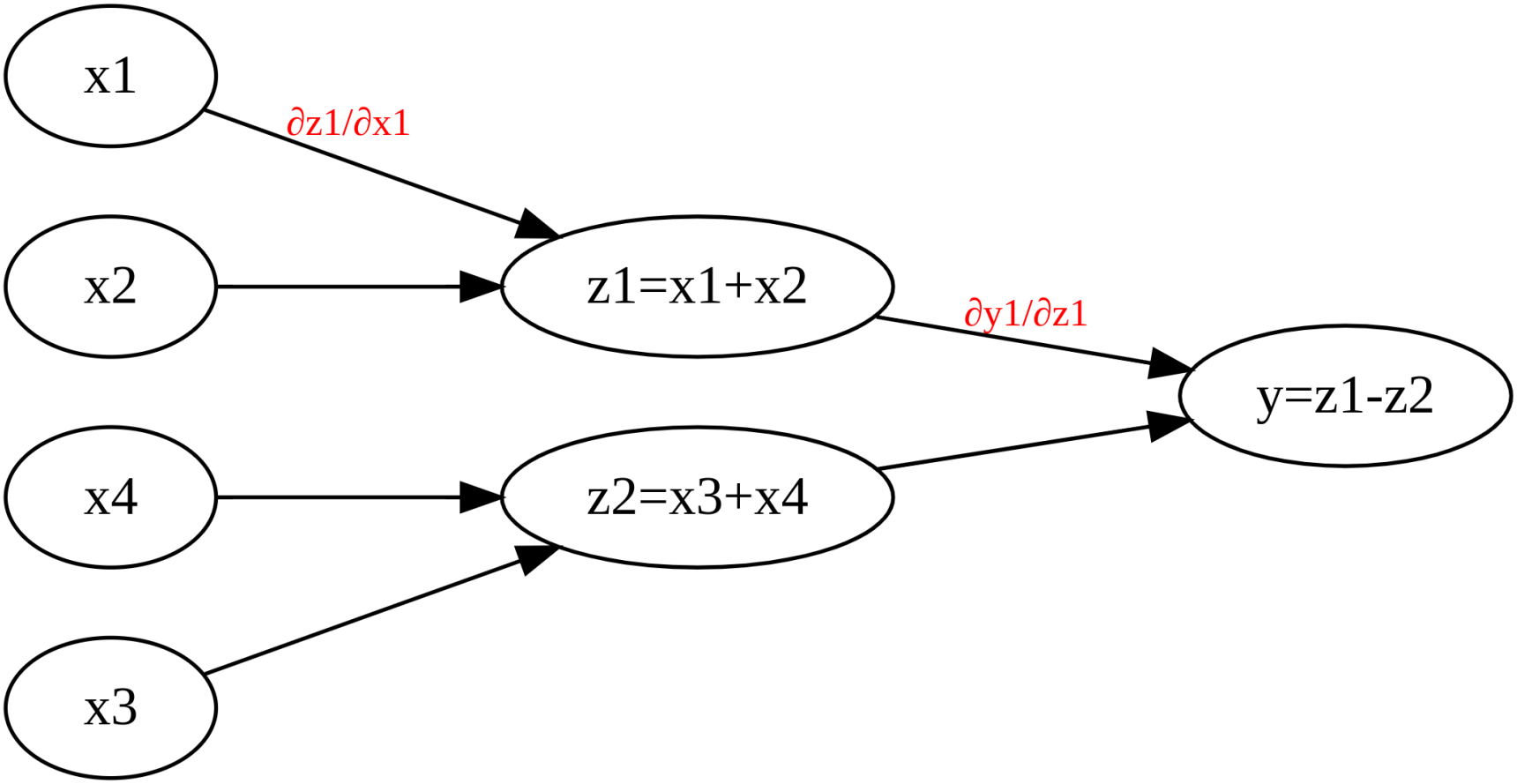
Any sequence of mathematical operations can be represented as a computational graph.

PyTorch has a built-in engine that computes and keeps track of the gradients over such a graph anytime you execute a differentiable operation.

Computational Graph



Computational Graph - Gradients



Computing the gradients

```
z1 = x1 + x2

z2 = x3 + x4

y = z1 - z2

loss = my_loss(y, y_true)

loss.backwards()
```

Caution with modes and gradients

Just before running training code, use `model.train()`, and use `model.eval()` before running evaluation code. This is to ensure the right behavior of the model.

When running computations for evaluation, use the context manager `torch.no_grad()` to avoid computing gradients on validation and test sets:

```
with torch.no_grad():
    your code
```

Optimizers

PyTroch provides a set of built-in optimizers that one could use for gradient based optimization.

```
optimizer = optim.SGD(model.parameters(), **kwargs)
```

Other optimizers include `Adam`, `AdamW`, `NAdam`,...

Note: before running any computation, make sure that you reset the gradients from previous computations:

```
optimizer.zero_grad()
```

Optimizers

The typical sequence for running code, calculating a loss, and optimizing the parameters goes as follows:

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

Defining a model through PyTorch

PyTorch uses an OOP approach to build models, losses, datasets, or any related component.

To define a custom model, one inherits from `torch.nn.Module`, defines the attributes inside constructor, and the operations that the model should perform inside the forward method. The model can also contain any other helper methods.

To use the model, you instantiate an object, and call the object directly as a function, i.e., no need to use `.forward(args)`.

Example model

```
class SimpleCNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = torch.nn.Conv2d(*args, **kwargs)

    def forward(self, x):
        return self.layer(x)

    def helper_method(self, *args, **kwargs):
        some code
```

Defining a dataset

A dataset is also defined in an OOP fashion. One inherits from `torch.utils.data.Dataset`, and defines two essential methods, `__getitem__`, and `__len__`, in addition to the constructor.

`__getitem__` defines how to get an individual sample from the data (e.g. read data from disk, get the label of an example, preprocess the image...)

`__len__` specifies the length of the expected dataset. This is important to avoid problems when loading the data

Example dataset

```
from torch.utils.data import Dataset
class MyCustomDataset(Dataset):
    def __init__(self, csv_file, root_dir,
transform=None):
        self.metadata = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform
    def __len__(self):
        return len(self.metadata)
    def __getitem__(self, idx):
        code that reads an input example, performs
transforms, if needed, and returns the ready-to-use
sample.
```

Defining a data loader

As the name suggests, a data loader has the functionality to load data using the dataset. This includes ensuring that all samples are loaded properly, shuffling the data, parallelizing the loading across multiple workers...

The data loader class is available at `torch.utils.data.Dataloader`, and then an object is created by passing the dataset object, along other arguments.

For more info on data sets and data loaders, check this page

Example dataloader

```
from torch.utils.data import DataLoader

my_dataset = MyCustomDataset(*args, **kwargs)

dataloader = DataLoader(my_dataset, batch_size=4,
shuffle=True, num_workers=4)
```