PyTorch Introduction
Tensors et al.

David Rotermund & Udo Ernst

Recommendation

Open Book: Dive into Deep Learning

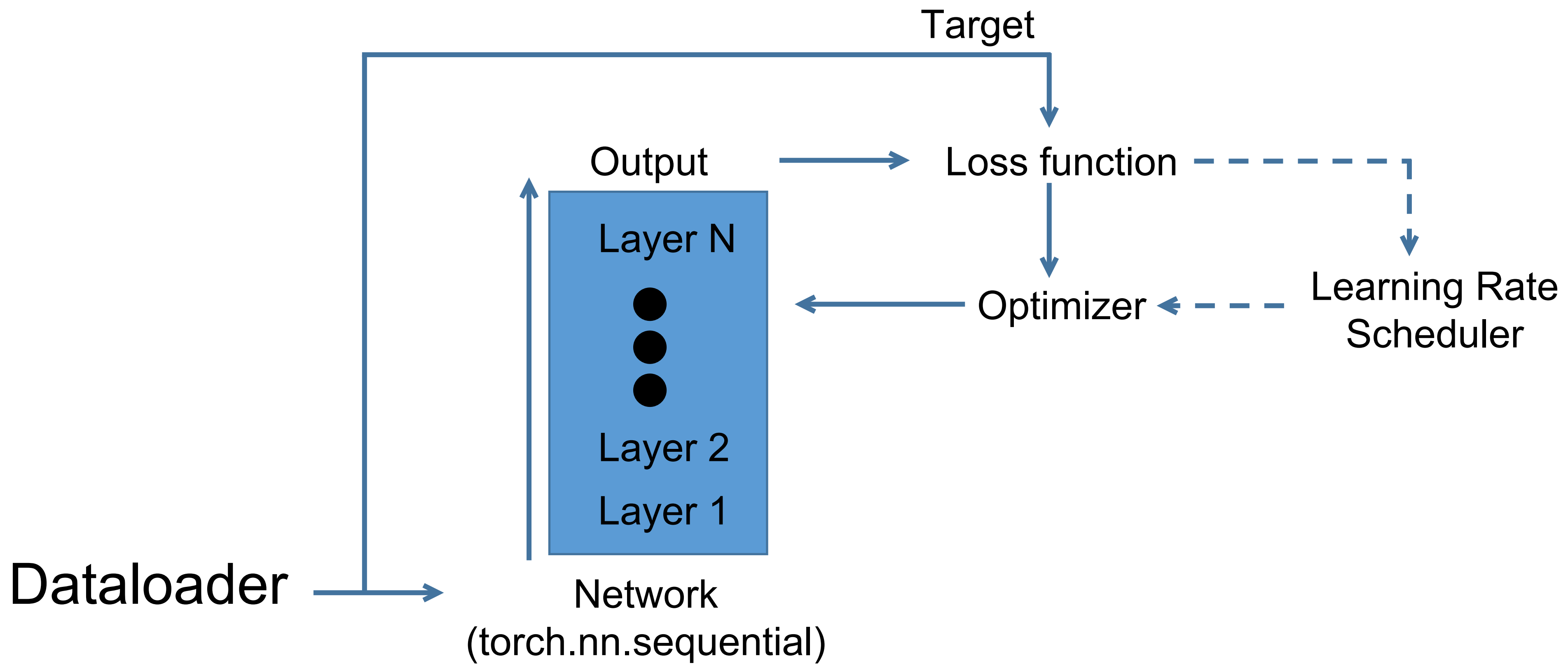
- ASTON ZHANG
- ZACHARY C. LIPTON
- MU LI
- ALEXANDER J. SMOLA

https://d2l.ai/d2l-en.pdf

Fig. 1.3.3 A donkey, a dog, a cat, and a rooster.

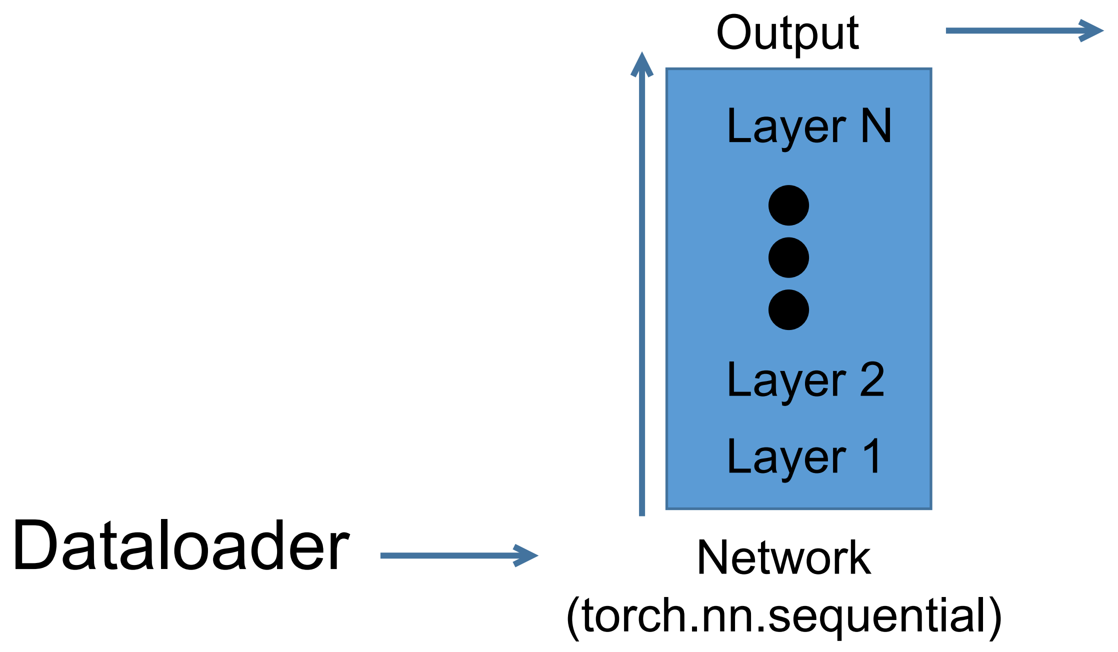
Anatomy of a PyTorch Network + Support

Training of the weights



Anatomy of a PyTorch Network + Support

Inference



torch.tensor is the numpy.ndarray of PyTorch

$\longrightarrow$ == torch.tensor

== stored data

Torch.tensor

Numpy to torch.tensor

```
torch.tensor(data, *, dtype=None, device=None, requires_grad=False,
pin_memory=False) → Tensor
```

```
import numpy as np
import torch
a_np: np.ndarray = np.zeros((10, 10))
a_torch: torch.Tensor = torch.tensor(a_np)
print(type(a_torch)) # <class 'torch.Tensor'>
```

! A torch tensor is of the class torch.Tensor
Be aware of the large ***T*** if you use type annotation.

Torch.Tensor to Numpy

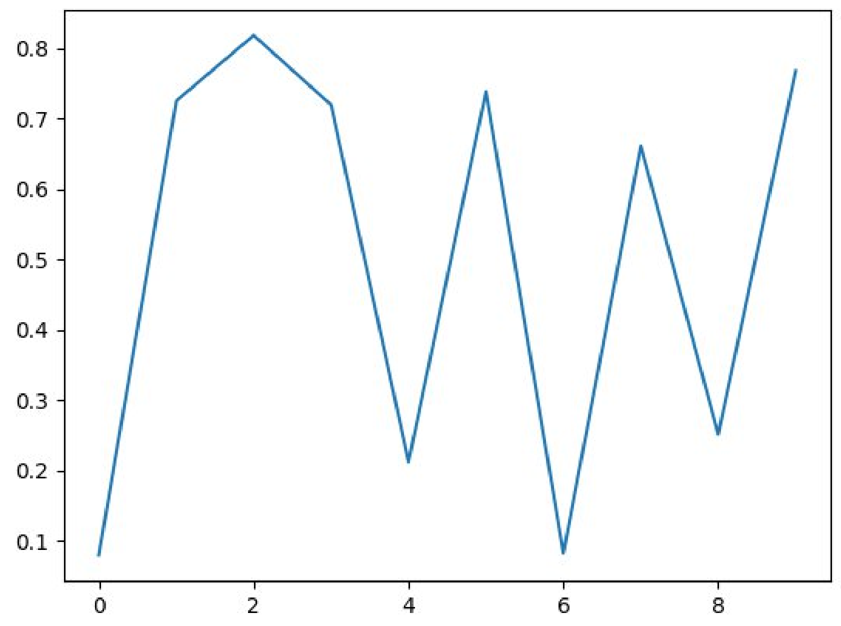
1. remove any gradient data if the tensor has some
   torch.Tensor.detach
2. bring the data to the CPU if it not already on the CPU.
   torch.Tensor.cpu
3. convert to numpy
   torch.Tensor.numpy

```
import numpy as np
import torch
a_torch: torch.Tensor = torch.zeros((10, 10))
a_np: np.ndarray = a_torch.detach().cpu().numpy()
print(type(a_np))  # <class 'numpy.ndarray'>
```

Torch.Tensor to Numpy

However, there are some best case scenarios...

```
import matplotlib.pyplot as plt
import torch
a_torch: torch.Tensor = torch.rand((10,))
plt.plot(a_torch)
plt.show()
```



Matrix generation

torch.zeros

```
torch.zeros(*size, *, out=None, dtype=None, layout=torch.strided, device=None,
requires_grad=False) → Tensor
```

torch.zeros_like

```
torch.zeros_like(input, *, dtype=None, layout=None, device=None,
requires_grad=False, memory_format=torch.preserve_format) → Tensor
```

torch.ones	torch.ones_like
torch.empty	torch.empty_like
torch.full	torch.full_like

Random sampling (examples)

seed	Sets the seed for generating random numbers to a non-deterministic random number on all devices.
bernoulli	Draws binary random numbers (0 or 1) from a Bernoulli distribution.
multinomial	Returns a tensor where each row contains num_samples indices sampled from the multinomial probability distribution located in the corresponding row of tensor input.
normal	Returns a tensor of random numbers drawn from separate normal distributions whose mean and standard deviation are given.
poisson	Returns a tensor of the same size as input with each element sampled from a Poisson distribution with rate parameter
rand	Returns a tensor filled with random numbers from a uniform distribution on the interval [0,1)
randint	Returns a tensor filled with random integers generated uniformly between low (inclusive) and high (exclusive).
randn	Returns a tensor with the same size as input that is filled with random numbers from a normal distribution with mean 0 and variance 1.
randperm	Returns a random permutation of integers from 0 to n - 1.

Math operations

Constants (inf, nan)
Pointwise Ops (abs, acos, exp, log, ...)
Reduction Ops (argmax, min, max, mean, nanmean, var, ...)
Comparison Ops (isfinite, sort, le, ge, ...)
~~Spectral Ops~~
Other Operations (clone, corrcoef, cumsum, diff, flip, histc, ...)
BLAS and LAPACK Operations (bmm, inverse, svd, trapezoid, ...)
~~Foreach Operations~~

At some point you want to scroll through the list...

Close to numpy.ndarray but not the same

Close to numpy.ndarray but not the same

Replace:
axis —> dim
keepdims —> keepdim

Example: max()

```
numpy.max(a, axis=None, out=None, keepdims=<no value>, initial=<no value>,
where=<no value>)
```

—>

```
torch.max(input, dim, keepdim=False, *, out=None)
```

Dimensions: remove

Squeeze

```
torch.squeeze(input: Tensor, dim: Optional[Union[int, List[int]]]) → Tensor
```

“Returns a tensor with all specified dimensions of input of size 1 removed.”

```
x = torch.zeros(2, 1, 2, 1, 2)
x.size() # torch.Size([2, 1, 2, 1, 2])
y = torch.squeeze(x)
y.size() # torch.Size([2, 2, 2])
```

Dimensions: add

Unsqueeze

```
torch.unsqueeze(input, dim) → Tensor
```

“Returns a new tensor with a dimension of size one inserted at the specified position.”

```
x = torch.tensor([1, 2, 3, 4])
torch.unsqueeze(x, 0) # tensor([[ 1,  2,  3,  4]])
torch.unsqueeze(x, 1) # tensor([[ 1], [ 2], [ 3], [ 4]])
```

Copy is clone

```
torch.clone(input, *, memory_format=torch.preserve_format) → Tensor
```

```
Tensor.clone(*, memory_format=torch.preserve_format) → Tensor
```

! Don’t forget to .detach() first and then .clone() otherwise you copy the gradients too.

Load / Save

```
torch.load(f, map_location=None, pickle_module=pickle, *, weights_only=True,
mmap=None, **pickle_load_args)
```

```
torch.save(obj, f, pickle_module=pickle, pickle_protocol=2,
_use_new_zipfile_serialization=True)
```

```
import torch
a_torch: torch.Tensor = torch.rand((10,))
torch.save(a_torch, "test_file.pt")
b_torch = torch.load("test_file.pt", weights_only=False)
print(type(b_torch)) # <class 'torch.Tensor'>
print(b_torch.shape) # torch.Size([10])
```

weights_only is a topic for saving networks. Here define whatever we want but we need to define something to make torch to be silent.

torch.mm (@) vs torch.bmm

```
torch.mm(input, mat2, *, out=None) → Tensor
```

"Performs a matrix multiplication of the matrices input and mat2.
If input is a (n×m) tensor, mat2 is a (m×p) tensor, out will be a (n×p) tensor."

```
torch.bmm(input, mat2, *, out=None) → Tensor
```

"Performs a batch matrix-matrix product of matrices stored in input and mat2.
input and mat2 must be 3-D tensors each containing the same number of matrices.
If input is a (b×n×m) tensor, mat2 is a (b×m×p) tensor, out will be a (b×n×p) tensor."

! Most tensors we encounter index the pattern within a batch at the dim=0.
This is denoted by b. Weights are typically different.

torch.einsum

Under numpy the einsum (einstein sum) was less important but due to the batch index we need it more often.

```
As = torch.randn(3, 2, 5)
Bs = torch.randn(3, 5, 4)
torch.einsum('bij,bjk->bik', As, Bs)
```

fft

```
torch.fft.rfft(input, n=None, dim=-1, norm=None, *, out=None) → Tensor
torch.fft.irfft(input, n=None, dim=-1, norm=None, *, out=None) → Tensor
torch.fft.rfft2(input, s=None, dim=(-2, -1), norm=None, *, out=None) → Tensor
torch.fft.irfft2(input, s=None, dim=(-2, -1), norm=None, *, out=None) → Tensor
torch.fft.rfftn(input, s=None, dim=None, norm=None, *, out=None) → Tensor
torch.fft.irfftn(input, s=None, dim=None, norm=None, *, out=None) → Tensor
torch.fft.rfftfreq(n, d=1.0, *, out=None, dtype=None, layout=torch.strided,
device=None, requires_grad=False) → Tensor
```

Torch fft is prepared for the batch dimension too.

! Don’t forget that you want to use rfft if all the inputs are real and not complex.
=> faster and uses less memory (important for the GPU)

torch.scatter_ for one hot encoding

```
Tensor.scatter_(dim, index, src, *, reduce=None) → Tensor
```

```
# Convert label into one hot
target_one_hot: torch.Tensor =
          torch.zeros( (labels.shape[0],number_of_output_neurons,),
          device=output.device, dtype=ouput.dtype,)
target_one_hot.scatter_(1, labels.to(h.device).unsqueeze(1),
          torch.ones( (labels.shape[0], 1), device=output.device,
          dtype=ouput.dtype,),)
```

Converting a vector of integer labels into a matrix of one hot encoded data.

Bandpass filter with filtfilt

```
torchaudio.functional.filtfilt(waveform: Tensor, a_coeffs: Tensor, b_coeffs: Tensor, clamp: bool = True) → Tensor
```

This works on batch data and can be used for processing data on the GPU

We can steal the coefficients from scipy

```
import scipy  # type: ignore
butter_b_np, butter_a_np = scipy.signal.butter(4, [low_frequency,
          high_frequency], btype="bandpass", output="ba", fs=fs)
butter_a = torch.tensor(butter_a_np, device=device, dtype=torch.float32)
butter_b = torch.tensor(butter_b_np, device=device, dtype=torch.float32)
```

dtype — data type

dtype

dtypes exist in numpy too but is normally ignored.

With PyTorch dtype is important because

- the “wrong” dtype can double the computational time
- and, even worse, may require too much memory which your GPU may not be able to provide

torch.Tensor

Most important data types:

Data type	dtype
32-bit floating point	torch.float32
64-bit floating point	torch.float64
8-bit integer (unsigned)	torch.uint8
16-bit integer (signed)	torch.int16
32-bit integer (signed)	torch.int32
64-bit integer (signed)	torch.int64
Boolean	torch.bool

most important dtype

Default dtype

torch.set_default_dtype

```
torch.set_default_dtype(d, /)
```

“Sets the default floating point dtype to d.

Supports floating point dtype as inputs.

Other dtypes will cause torch to raise an exception.”

torch.get_default_dtype

```
torch.get_default_dtype() → torch.dtype
```

“Get the current default floating point torch.dtype.”

Convert the dtype

torch.Tensor.type

```
Tensor.type(dtype=None, non_blocking=False, **kwargs) → str or Tensor
```

“Returns the type if dtype is not provided, else casts this object to the specified type.

If this is already of the correct type, no copy is performed and the original object is returned.”

torch.Tensor.type_as

```
Tensor.type_as(tensor) → Tensor
```

“Returns this tensor cast to the type of the given tensor.

This is a no-op if the tensor is already of the correct type.”

Get the dtype

torch.dtype

```
float_tensor = torch.ones(1, dtype=torch.float)
float_tensor.dtype # torch.float32
double_tensor = torch.ones(1, dtype=torch.double)
double_tensor.dtype # torch.float64
(float_tensor + double_tensor).dtype # torch.float64
```

device — CPU or GPU?

Windows / Linux driver for nvidia GPUs

If you have a nvidia GPU, you need a PyTorch version
Check here for a installation configurator.

with CUDA support AND nvidia cuda drivers. I.e. these two drivers:

CUDA Toolkit

https://developer.nvidia.com/cuda-downloads

cuDNN

https://developer.nvidia.com/cudnn-downloads

I have a GPU. Is it ready for PyTorch?

```
import torch
torch.cuda.is_available() # True
torch.backends.cuda.is_built() # True
```

Is the GPU found by PyTorch?

```
import torch
my_cuda_device = torch.device('cuda:0')
print(torch.cuda.get_device_properties(my_cuda_device))
# _CudaDeviceProperties(name='NVIDIA GeForce RTX 3060', major=8, minor=6,
total_memory=12011MB, multi_processor_count=28)
```

I have a GPU. Is it ready for PyTorch?

Is cuDNN ready?

```
import torch
torch.backends.cudnn.enabled # True
torch.backends.cudnn.version() # 8904
```

Determining if a GPU is there...

These are the two typical lines you put in your PyTorch program

```
device: torch.device = ( torch.device("cuda:0") if torch.cuda.is_available()
          else torch.device("cpu") )
torch.set_default_dtype(torch.float32)
```

Now device contains the device you want to use...

...if there are no other conditions (e.g. GPU memory) you need to take into consideration.

device

```
torch.cuda.device(device)
```

“Context-manager that changes the selected device.”

On which device is my tensor?

torch.Tensor.device

```
import torch a = torch.zeros((3,3))
a.device # device(type='cpu')
b = torch.zeros((3,3),device=torch.device("cuda:0"))
b.device # device(type='cuda', index=0)
```

Example

```
a: torch.Tensor = torch.zeros((5, 5), device=torch.device("cuda:0"))
b: torch.Tensor = torch.zeros((5, 5), device=a.device)
```

Moving the tensor around

torch.Tensor.cpu

```
Tensor.cpu(memory_format=torch.preserve_format) → Tensor
```

“Returns a copy of this object in CPU memory.

If this object is already in CPU memory and on the correct device, then no copy is performed and the original object is returned.”

torch.Tensor.cuda

```
Tensor.cuda(device=None, non_blocking=False,
memory_format=torch.preserve_format) → Tensor
```

“Returns a copy of this object in CUDA memory.

If this object is already in CUDA memory and on the correct device, then no copy is performed and the original object is returned.”

Moving the tensor around

torch.Tensor.to

```
Tensor.to(*args, **kwargs) → Tensor
```

“Performs Tensor dtype and/or device conversion. A torch.dtype and torch.device are inferred from the arguments of self.to(*args, **kwargs).”

torch.cuda.get_device_properties

```
torch.cuda.get_device_properties(device=None)
```

“Get the properties of a device.”

Example

```
_CudaDeviceProperties(name='NVIDIA GeForce GTX 1080 Ti', major=6, minor=1,
total_memory=11162MB, multi_processor_count=28, uuid=56e29e01-678c-89df-
b384-1f709f903a19, L2_cache_size=2MB)
```

```
cuda_total_memory: int = torch.cuda.get_device_properties(device).total_memory

print(cuda_total_memory) # 11704532992
```

How much memory can I use?

```
if device != torch.device("cpu"):
    torch.cuda.empty_cache()
    cuda_total_memory: int = torch.cuda.get_device_properties(device.index).total_memory
    free_mem = cuda_total_memory - max(
            [torch.cuda.memory_reserved(device), torch.cuda.memory_allocated(device)]
)
print(f"CUDA memory: {free_mem // 1024} MByte")
```

torch.cuda.memory_reserved

```
torch.cuda.memory_reserved(device=None)
```

“Return the current GPU memory managed by the caching allocator in bytes for a given device.”

torch.cuda.memory_allocated

```
torch.cuda.memory_allocated(device=None)
```

“Return the current GPU memory occupied by tensors in bytes for a given device.”

Other functions

torch.cuda.manual_seed_all(seed)

“Set the seed for generating random numbers on all GPUs.
It’s safe to call this function if CUDA is not available; in that case, it is silently ignored.”

torch.cuda.empty_cache()

“empty_cache() doesn’t increase the amount of GPU memory available for PyTorch.
However, it may help reduce fragmentation of GPU memory in certain cases.”

torch.memory_format

Mainly relevant for developing C++ extensions

pybind11

pybind11 — Seamless operability between C++11 and Python

torch.memory_format

torch.contiguous_format	Tensor is or will be allocated in dense non-overlapping memory. Strides represented by values in decreasing order.
~~torch.channels_last~~	
~~torch.channels_last_3d~~	
torch.preserve_format	

For a tensor X with shape [batch_size, channels, height, width] and a specific element at position [b, c, h, w], the memory offset (in number of elements) from the start of the tensor would be:

$$\text{offset} = b * (\text{channels} * \text{height} * \text{width}) + c * (\text{height} * \text{width}) + h * \text{width} + w$$

torch.Tensor.data_ptr

“Returns the address of the first element of self tensor.”

torch.Tensor.contiguous

```
Tensor.contiguous(memory_format=torch.contiguous_format) → Tensor
```

“Returns a contiguous in memory tensor containing the same data as self tensor. If self tensor is already in the specified memory format, this function returns the self tensor.”

torch.Tensor.is_contiguous

```
Tensor.is_contiguous(memory_format=torch.contiguous_format) → bool
```

“Returns True if self tensor is contiguous in memory in the order specified by memory format.“

Data augmentation

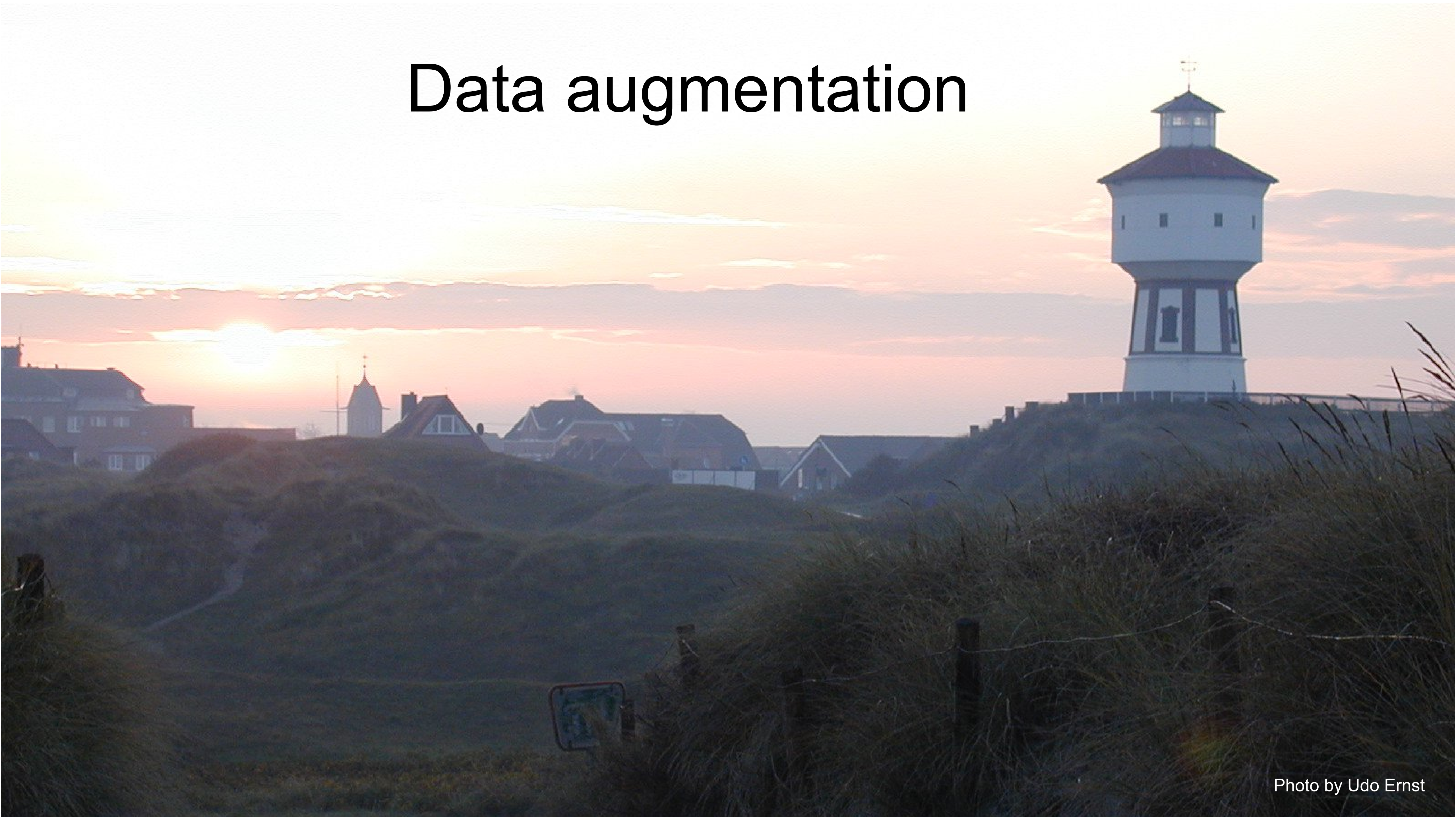


Photo by Udo Ernst

Data augmentation

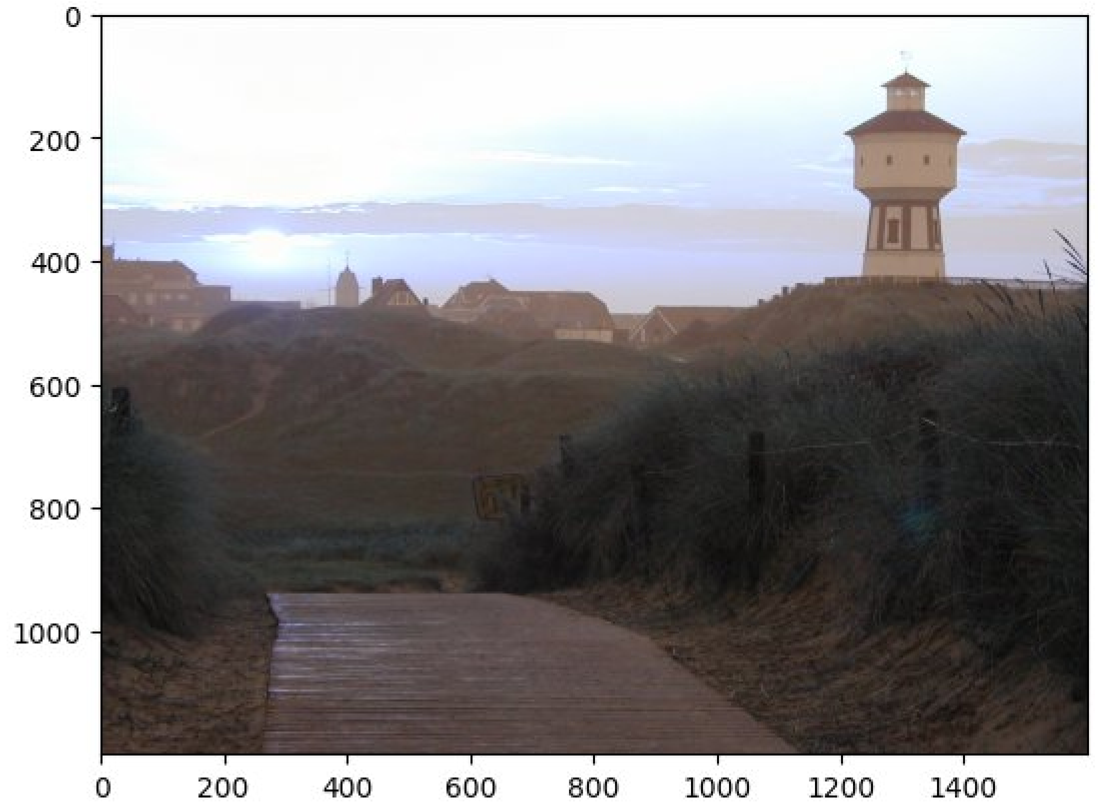
The goal is to automatically produce variants of an input pattern.

Thus increasing the amount of training data several fold.

Loading an example image
(with opencv2)

Load it via [cv2.imread(filename[, flags]) -> retval]

```python
import cv2
import matplotlib.pyplot as plt
filename: str = "data_augmentation_test_image.jpg"
original_image = cv2.imread(filename)
plt.imshow(original_image)
plt.show()
```



As you can see (not very well, I might add) is that the color channels are wrong.

Loading an example image

```
import numpy as np
original_image = cv2.imread(filename, cv2.IMREAD_COLOR)
# "Convert" from BlueGreenRed (BGR) to RGB (RedGreenBlue)
# This is a flip in the third dimension.
original_image = np.flip(original_image, axis=2)
plt.imshow(original_image)
plt.show()
```

Loading an example image in grayscale

```
original_image = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)
plt.imshow(original_image, cmap="gray")
plt.show()
```

But may be we want no color anyway
Other options can be found here

torchvision

“The torchvision package consists of popular datasets, model architectures, and common image transformations for computer vision.”

V1 or V2? Which one should I use?

“**TL;DR** We recommending using the torchvision.transforms.v2 transforms instead of those in torchvision.transforms. They’re faster and they can do more things. Just change the import and you should be good to go. Moving forward, new features and improvements will only be considered for the v2 transforms.”

Test setup

```
import torch
import cv2
import matplotlib.pyplot as plt
from torchvision.transforms import v2  # type: ignore

filename: str = "data_augmentation_test_image.jpg"
original_image = cv2.imread(filename, cv2.IMREAD_COLOR)
torch_image = (
    (torch.tensor(original_image).flip(dims=(2,)).type(dtype=torch.float32) / 255.0)
    .movedim(-1, 0)
    .unsqueeze(0))

print(torch_image.shape)  # torch.Size([1, 3, 1200, 1600])

transforms = v2.Compose([ SOME TRANSFORMATION])

plt.imshow(transforms(torch_image)[0, ...].movedim(0, -1).detach().numpy())
plt.show()
```

Compose

```
torchvision.transforms.v2.Compose(transforms: Sequence[Callable])
```

We use compose to chain transformations together like:

```
v2.Compose([
    v2.CenterCrop(10),
    v2.PILToTensor(),
    v2.ConvertImageDtype(torch.float),])
```

As an alternative you may want to use torch.nn.Sequential

RandomApply

```
torchvision.transforms.v2.RandomApply(transforms: Union[Sequence[Callable],
ModuleList], p: float = 0.5)
```

“Apply randomly a list of transformations with a given probability.”

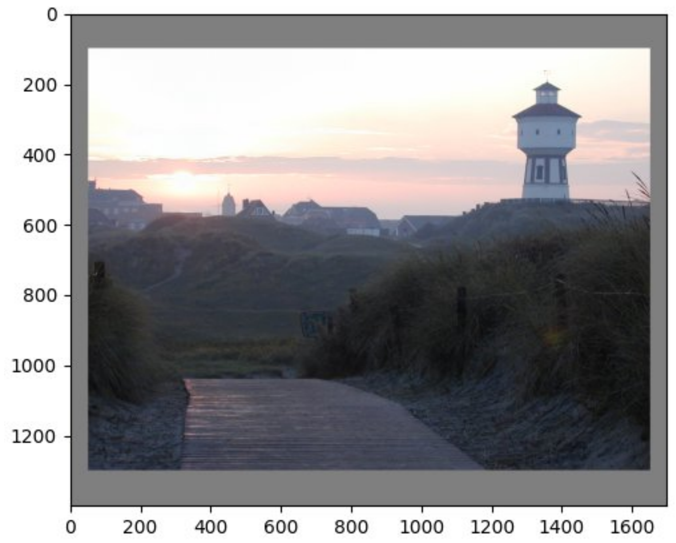
```
v2.RandomApply([
    v2.CenterCrop(10),
    v2.PILToTensor(),
    v2.ConvertImageDtype(torch.float),])
```

Note: It randomly applies the whole list of transformation or none.

Example from V2: Pad

```
torchvision.transforms.v2.Pad(padding: Union[int, Sequence[int]], fill: Union[int, float, Sequence[int], Sequence[float], None, Dict[Union[Type, str], Optional[Union[int, float, Sequence[int], Sequence[float]]]]] = 0, padding_mode: Literal['constant', 'edge', 'reflect', 'symmetric'] = 'constant')
```

```
transforms = v2.Compose([v2.Pad(padding=(50, 100), fill=0.5)])
```



Example from V2: Flip

Horizontal Vertical

Horizontally flip the input with a given probability.

```
torchvision.transforms.v2.RandomHorizontalFlip(p: float = 0.5)
```

Vertically flip the input with a given probability.

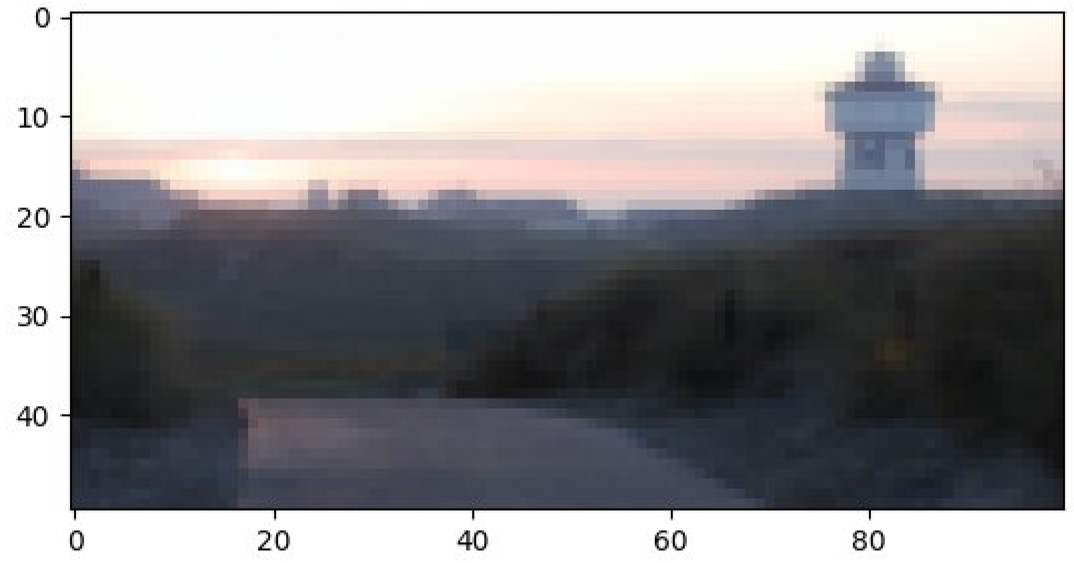
```
torchvision.transforms.v2.RandomVerticalFlip(p: float = 0.5)
```

Example from V2: Resize

```
torchvision.transforms.v2.Resize(size: Optional[Union[int, Sequence[int]]], interpolation: Union[InterpolationMode, int] = InterpolationMode.BILINEAR, max_size: Optional[int] = None, antialias: Optional[bool] = True)
```

Resize the input to the given size.

```
transforms = v2.Compose([v2.Resize(size=(50, 100))])
```

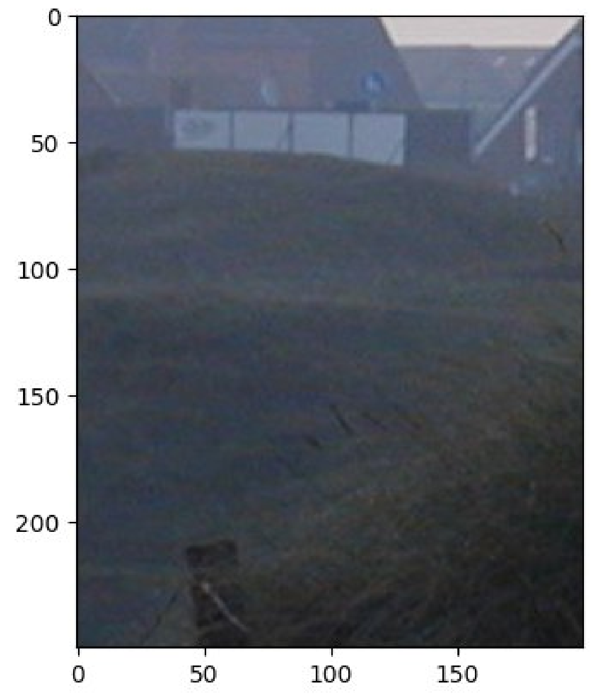


Example from V2: CenterCrop

```
torchvision.transforms.v2.CenterCrop(size: Union[int, Sequence[int]])
```

Crop the input at the center.

```
transforms = v2.Compose([v2.CenterCrop(size=(250, 200))])
```



Example from V2: FiveCrop

```
torchvision.transforms.v2.FiveCrop(size: Union[int, Sequence[int]])
```

Crop the image or video into four corners and the central crop.

```
print(torch_image.shape)  # torch.Size([1, 3, 1200, 1600])

transforms = v2.Compose([v2.FiveCrop(size=(250, 200))])
output = transforms(torch_image)

print(len(output)) # 5
print(output[0].shape) # torch.Size([1, 3, 250, 200])
```

Example from V2: TenCrop

```
torchvision.transforms.v2.TenCrop(size: Union[int, Sequence[int]], vertical_flip: bool = False)
```

Crop the image or video into four corners and the central crop plus the flipped version of these (horizontal flipping is used by default).

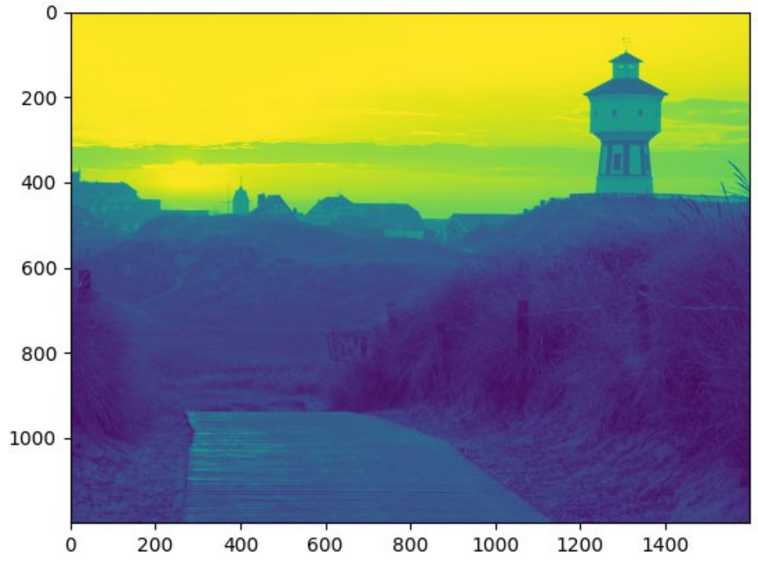
```
transforms = v2.Compose([v2.TenCrop(size=(250, 200))])

output = transforms(torch_image)

print(len(output))  # 10
print(output[0].shape)  # torch.Size([1, 3, 250, 200])
```

Example from V2: Grayscale

```
torchvision.transforms.v2.Grayscale(num_output_channels: int = 1)
```

Crop the image or video into four corners and the central crop plus the flipped version of these (horizontal flipping is used by default).

```
transforms = v2.Compose([v2.Grayscale()])
print(transforms(torch_image).shape)  # torch.Size([1, 1, 1200, 1600])
```



Example from V2

RandomGrayscale

```
torchvision.transforms.v2.RandomGrayscale(p: float = 0.1)
```

Randomly convert image or videos to grayscale with a probability of p (default 0.1).

RandomInvert

```
torchvision.transforms.v2.RandomInvert(p: float = 0.5)
```

Inverts the colors of the given image or video with a given probability.

RandomEqualize

```
torchvision.transforms.v2.RandomEqualize(p: float = 0.5)
```

Equalize the histogram of the given image or video with a given probability.

Example from V2

Normalize

```
torchvision.transforms.v2.Normalize(mean: Sequence[float], std:
Sequence[float], inplace: bool = False)
```

Normalize a tensor image or video with mean and standard deviation.

ColorJitter

```
torchvision.transforms.v2.ColorJitter(
        brightness: Optional[Union[float, Sequence[float]]] = None,
        contrast: Optional[Union[float, Sequence[float]]] = None,
        saturation: Optional[Union[float, Sequence[float]]] = None,
        hue: Optional[Union[float, Sequence[float]]] = None)
```

Randomly change the brightness, contrast, saturation and hue of an image or video.

Example from V2

GaussianBlur

```
torchvision.transforms.v2.GaussianBlur(kernel_size: Union[int, Sequence[int]],
sigma: Union[int, float, Sequence[float]] = (0.1, 2.0))
```

Blurs image with randomly chosen Gaussian blur kernel.

GaussianNoise

```
torchvision.transforms.v2.GaussianNoise(mean: float = 0.0, sigma: float = 0.1,
clip=True)
```

Add gaussian noise to images or videos.

Example from V2

RandomPerspective

```
torchvision.transforms.v2.RandomPerspective(distortion_scale: float = 0.5, p:
float = 0.5, interpolation: Union[InterpolationMode, int] =
InterpolationMode.BILINEAR, fill: Union[int, float, Sequence[int], Sequence[float],
None, Dict[Union[Type, str], Optional[Union[int, float, Sequence[int],
Sequence[float]]]]] = 0)
```

Perform a random perspective transformation of the input with a given probability.

RandomRotation

```
torchvision.transforms.v2.RandomRotation(degrees: Union[Number, Sequence],
interpolation: Union[InterpolationMode, int] = InterpolationMode.NEAREST,
expand: bool = False, center: Optional[List[float]] = None, fill: Union[int, float,
Sequence[int], Sequence[float], None, Dict[Union[Type, str], Optional[Union[int,
float, Sequence[int], Sequence[float]]]]] = 0)
```

Rotate the input by angle.

Example from V2

RandomAffine

```
torchvision.transforms.v2.RandomAffine(degrees: Union[Number, Sequence],
translate: Optional[Sequence[float]] = None, scale: Optional[Sequence[float]] =
None, shear: Optional[Union[int, float, Sequence[float]]] = None, interpolation:
Union[InterpolationMode, int] = InterpolationMode.NEAREST, fill: Union[int,
float, Sequence[int], Sequence[float], None, Dict[Union[Type, str],
Optional[Union[int, float, Sequence[int], Sequence[float]]]]] = 0, center:
Optional[List[float]] = None)
```

Random affine transformation the input keeping center invariant.

Example from V2

RandomCrop

```
torchvision.transforms.v2.RandomCrop(size: Union[int, Sequence[int]], padding: Optional[Union[int, Sequence[int]]] = None, pad_if_needed: bool = False, fill: Union[int, float, Sequence[int], Sequence[float], None, Dict[Union[Type, str], Optional[Union[int, float, Sequence[int], Sequence[float]]]]] = 0, padding_mode: Literal['constant', 'edge', 'reflect', 'symmetric'] = 'constant')
```

Crop the input at a random location.

AND MANY MORE...

Auto-Augmentation

“AutoAugment is a common Data Augmentation technique that can improve the accuracy of Image Classification models. Though the data augmentation policies are directly linked to their trained dataset, empirical studies show that ImageNet policies provide significant improvements when applied to other datasets. In TorchVision we implemented 3 policies learned on the following datasets: ImageNet, CIFAR10 and SVHN. “

AutoAugment

```
torchvision.transforms.v2.AutoAugment(policy: AutoAugmentPolicy =
AutoAugmentPolicy.IMAGENET, interpolation: Union[InterpolationMode, int] =
InterpolationMode.NEAREST, fill: Union[int, float, Sequence[int], Sequence[float],
None, Dict[Union[Type, str], Optional[Union[int, float, Sequence[int],
Sequence[float]]]]] = None)
```

AutoAugment data augmentation method based on “AutoAugment: Learning Augmentation Strategies from Data”.

Available policies are IMAGENET, CIFAR10 and SVHN.

RandAugment

```
torchvision.transforms.v2.RandAugment(num_ops: int = 2, magnitude: int = 9,
num_magnitude_bins: int = 31, interpolation: Union[InterpolationMode, int] =
InterpolationMode.NEAREST, fill: Union[int, float, Sequence[int], Sequence[float],
None, Dict[Union[Type, str], Optional[Union[int, float, Sequence[int],
Sequence[float]]]]] = None)
```

RandAugment data augmentation method based on “RandAugment: Practical automated data augmentation with a reduced search space”.

Available policies are IMAGENET, CIFAR10 and SVHN.

What I use: MNIST

```
train_processing_chain = v2.Compose(
        transforms=[
                v2.RandomCrop((input_dim_x, input_dim_y))
        ],
)
```

```
test_processing_chain = v2.Compose(
        transforms=[
                v2.CenterCrop((input_dim_x, input_dim_y))
        ],
)
```

```
input_dim_x: int = 24
input_dim_y: int = 24
```

What I use: CIFAR10 (test)

```
test_processing_chain = v2.Compose(
        transforms=[
                v2.CenterCrop((input_dim_x, input_dim_y))
        ],
)
```

```
input_dim_x: int = 28
input_dim_y: int = 28
flip_p: float = 0.5
jitter_brightness: float = 0.5
jitter_contrast: float = 0.1
jitter_saturation: float = 0.1
jitter_hue: float = 0.15
```

What I use: CIFAR10 (training)

```
train_processing_chain = v2.Compose(
        transforms=[
                v2.RandomCrop((input_dim_x, input_dim_y)),
                v2.RandomHorizontalFlip(p=flip_p),
                v2.ColorJitter(
                        brightness=jitter_brightness,
                        contrast=jitter_contrast,
                        saturation=jitter_saturation,
                        hue=jitter_hue,
                ),
        ],
)
```

What I use: CIFAR10 (automatic)

```
train_processing_chain = v2.Compose(
    transforms=[
        v2.AutoAugment(policy=v2.AutoAugmentPolicy(v2.AutoAugmentPolicy.CIFAR10)),
        v2.CenterCrop((input_dim_x, input_dim_y)),
    ],
)
```

Building your own filter

```
from torchvision.transforms.v2 import Transform  # type: ignore
import torch
from typing import Any
class OnOffFilter(Transform):
        def __init__(self, p: float = 0.5) -> None:
                super().__init__()
                self.p: float = p
        def _transform(self, inpt: Any, params: dict[str, Any]) -> Any:
                return torch.cat(
                        (
                                torch.nn.functional.relu(self.p - inpt),
                                torch.nn.functional.relu(inpt - self.p),
                        ),
                        dim=1,
                )
        def __repr__(self):
                return self.__class__.__name__ + "(p={0})".format(self.p)
```

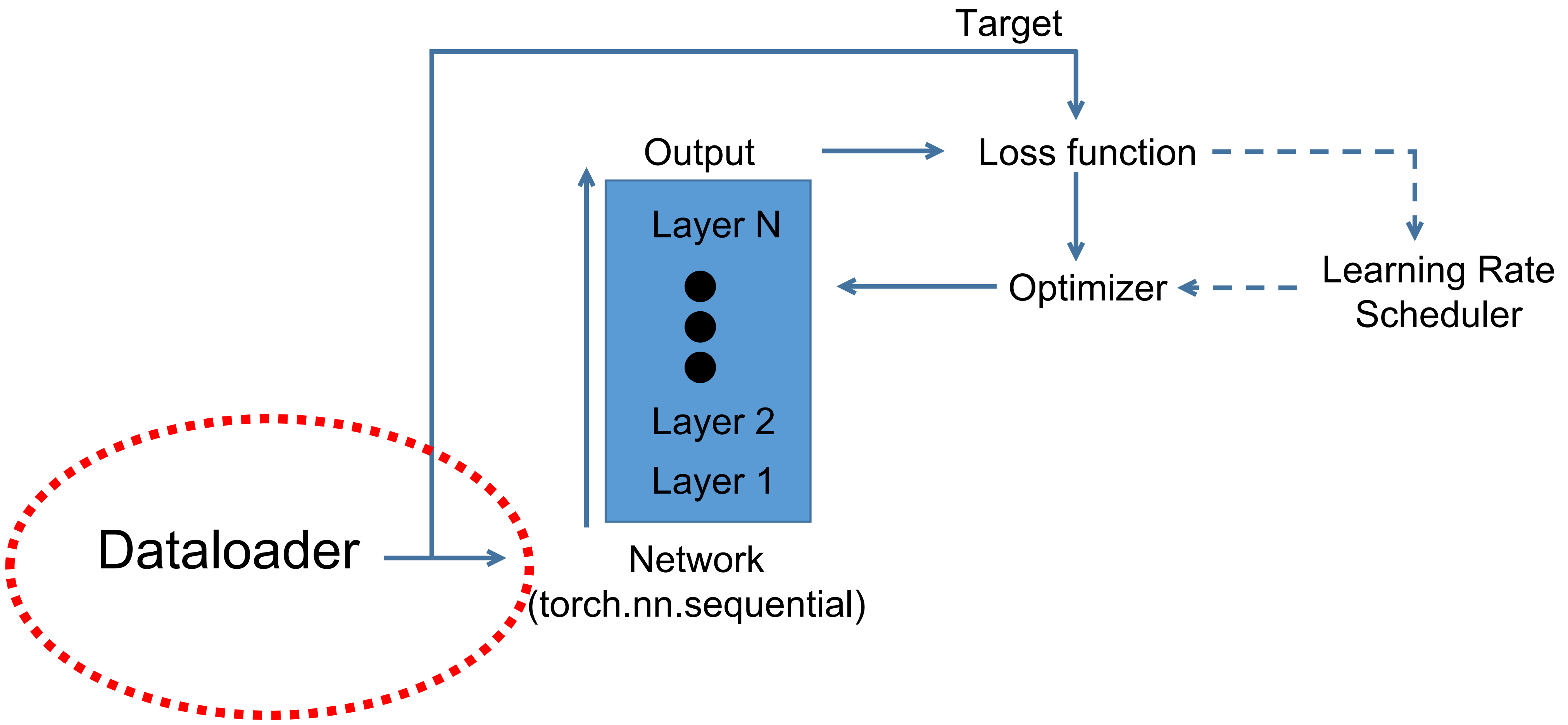
Building your own filter

```
print(torch_image.shape)  # torch.Size([1, 3, 1200, 1600])

transforms = v2.Compose([OnOffFilter()])

print(transforms(torch_image).shape)  # torch.Size([1, 6, 1200, 1600])
```

Dataloader

Anatomy of a PyTorch Network + Support

Training of the weights



Data provider

torchvision.datasets

Here you find a collection of the many benchmark datasets

Torchvision data provider: MNIST

```
import torchvision  # type: ignore

tv_dataset_train = torchvision.datasets.MNIST(root="data", train=True, download=True)
tv_dataset_test = torchvision.datasets.MNIST(root="data", train=False,
            download=True)

print(len(tv_dataset_train))  # 60000
print(len(tv_dataset_test))  # 10000
print(tv_dataset_test[0][0]) # <PIL.Image.Image image mode=L size=28x28 at 0x7F876166D5E0>
print(tv_dataset_test[0][1]) # 7
```

Torchvision data provider: FashionMNIST

```
import torchvision  # type: ignore

tv_dataset_train = torchvision.datasets.FashionMNIST(
    root="data", train=True, download=True)
tv_dataset_test = torchvision.datasets.FashionMNIST(
    root="data", train=False, download=True)

print(len(tv_dataset_train))  # 60000
print(len(tv_dataset_test))  # 10000
print(
    tv_dataset_test[0][0]
)  # <PIL.Image.Image image mode=L size=28x28 at 0x7FC794E96C30>
print(tv_dataset_test[0][1])  # 9
```

Torchvision data provider: CIFAR10

```python
import torchvision  # type: ignore

tv_dataset_train = torchvision.datasets.CIFAR10(
    root="data", train=True, download=True)
tv_dataset_test = torchvision.datasets.CIFAT10(
    root="data", train=False, download=True)

print(len(tv_dataset_train))  # 50000
print(len(tv_dataset_test))  # 10000
print(
    tv_dataset_test[0][0]
)  # <PIL.Image.Image image mode=RGB size=32x32 at 0x7F2F03BE0F80>
print(tv_dataset_test[0][1])  # 3
```

Dealing with you own data

Dataset

There are different types of datasets available that are derived from Dataset

torch.utils.data.IterableDataset	An iterable Dataset.
torch.utils.data.TensorDataset	Dataset wrapping tensors.
torch.utils.data.StackDataset	Dataset as a stacking of multiple datasets.
torch.utils.data.ConcatDataset	Dataset as a concatenation of multiple datasets.
torch.utils.data.ChainDataset	Dataset for chaining multiple IterableDataset s.

torch.utils.data.TensorDataset(*tensors)

```
label_tensor = torch.randint(0, 9, (10,))
image_tensor = torch.rand((10, 1, 28, 28))

dataset = torch.utils.data.TensorDataset(image_tensor, label_tensor)
```

The order of image tensor and label vector you put in here, you will get out later.

torch.utils.data.Dataset

In the case we might not be able to load the fully dataset into memory, the **torch.utils.data.Dataset** is very helpful.

“All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite **__getitem__()**, supporting fetching a data sample for a given key. “

“Subclasses could also optionally overwrite **__len__()**, which is expected to return the size of the dataset by many Sampler implementations and the default options of DataLoader.”

Subclasses could also optionally implement **__getitems__()**, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

Example (__init__)

```
import numpy as np import torch

class MyDataset(torch.utils.data.Dataset):

     # Initialize
    def __init__(self, train: bool = False) -> None:
        super().__init__()
        if train is True:
            self.pattern_storage: np.ndarray = np.load("train_pattern_storage.npy")
            self.label_storage: np.ndarray = np.load("train_label_storage.npy")
        else:
            self.pattern_storage = np.load("test_pattern_storage.npy")
            self.label_storage = np.load("test_label_storage.npy")

        self.pattern_storage = self.pattern_storage.astype(np.float32)
        self.pattern_storage /= np.max(self.pattern_storage)
        # How many pattern are there?
        self.number_of_pattern: int = self.label_storage.shape[0]
```

Example (__len__ & __getitem__)

```
def __len__(self) -> int:
    return self.number_of_pattern     # Get one pattern at position index

def __getitem__(self, index: int) -> tuple[torch.Tensor, int]:
     image = torch.tensor(self.pattern_storage[index, ...])
     target = int(self.label_storage[index])
     return image, target
```

torch.utils.data.DataLoader

```
torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=None,
sampler=None, batch_sampler=None, num_workers=0, collate_fn=None,
pin_memory=False, drop_last=False, timeout=0, worker_init_fn=None,
multiprocessing_context=None, generator=None, *, prefetch_factor=None,
persistent_workers=False, pin_memory_device='', in_order=True)
```

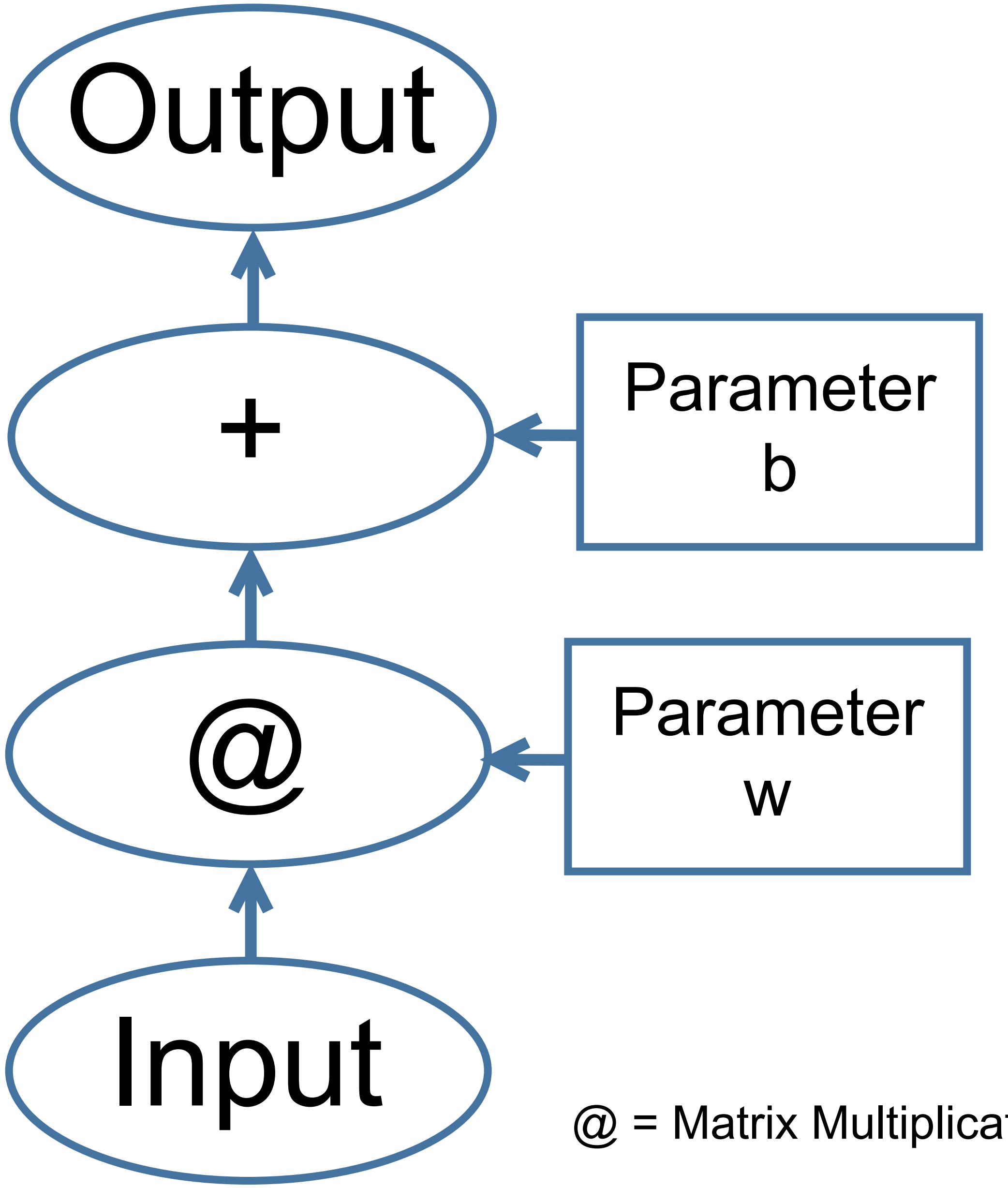
Important parameter

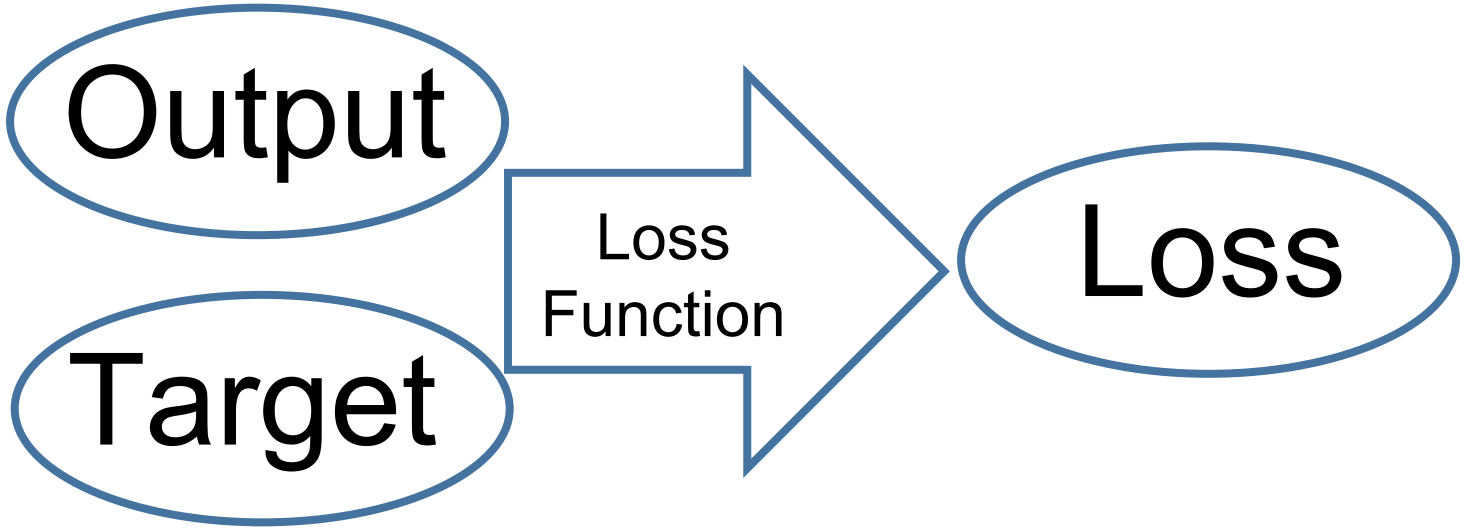
dataset (Dataset)	dataset from which to load the data.
batch_size (int, optional)	how many samples per batch to load (default: 1).
shuffle (bool, optional)	set to True to have the data reshuffled at every epoch (default: False).

Autograd

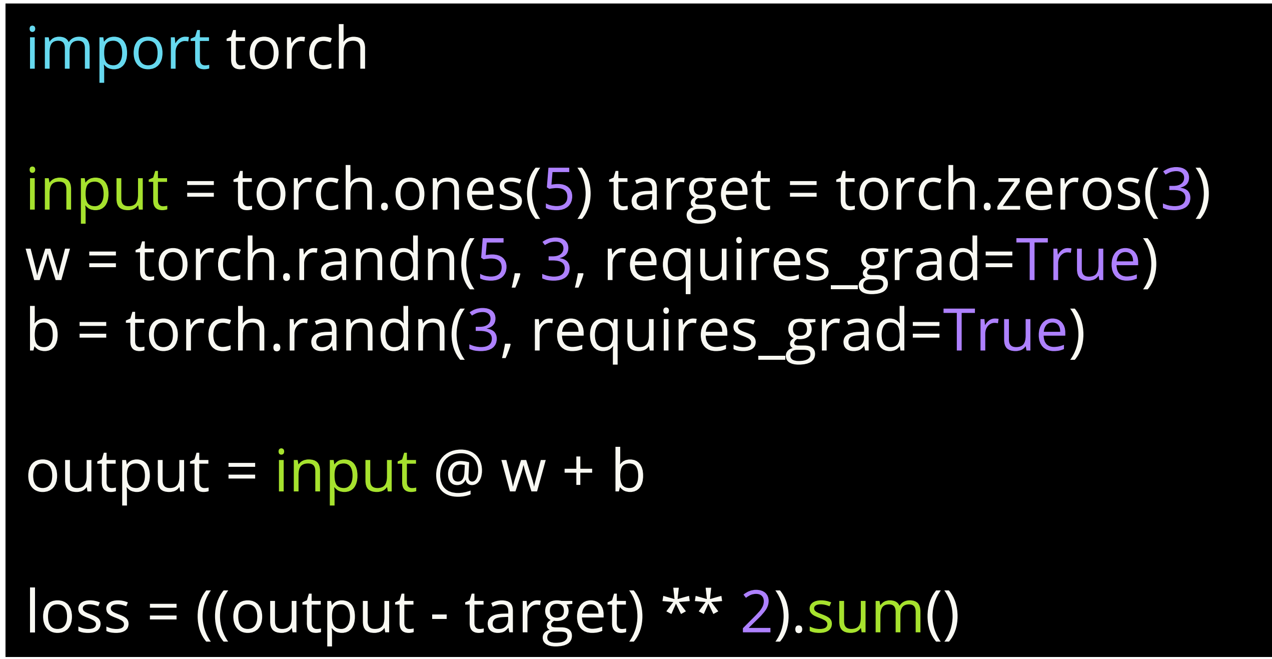
“torch.autograd is PyTorch’s automatic differentiation engine that powers neural network training”

Automatic Differentiation with torch.autograd





@ = Matrix Multiplication



```
import torch

input = torch.ones(5) target = torch.zeros(3)
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)

output = input @ w + b

loss = ((output - target) ** 2).sum()
```

“Parameters“

W and b are in this example parameters to be optimized.

Hence the requires_grad=True during their creation:

```
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
```

If not used during creation, it can be changed later via

torch.Tensor.requires_grad_

```
Tensor.requires_grad_(requires_grad=True) → Tensor
```

“Change if autograd should record operations on this tensor: sets this tensor’s requires_grad attribute **in-place**. Returns this tensor.”

Backprop — torch.Tensor.backward

```
Tensor.backward(gradient=None, retain_graph=None, create_graph=False,
inputs=None)
```

“Computes the gradient of current tensor wrt graph leaves.”

In our example:

```
print(w.grad) # None
print(b.grad) # None

loss.backward()

print(w.grad) # tensor([[1.6797, 2.3785, 7.0936], ...
print(b.grad) # tensor([1.6797, 2.3785, 7.0936])
```

Backprop

The other tensors don’t get a gradient

```
loss.backward()

print(input.grad)  # None
print(output.grad)  # None + Complains from PyTorch for even asking
print(target.grad)  # None
print(loss.grad)  # None + Complains from PyTorch for even asking
```

And don’t call backward more then once:

```
loss.backward()
loss.backward()
# RuntimeError: Trying to backward through the graph a second time (or
directly access saved tensors after they have already been freed). ...
```

Computational graph

If something is involved with requires_grad=True, a computational graph is created:

```
output = input @ w + b
loss = ((output - target) ** 2).sum()
print(output.grad_fn) # <AddBackward0 object at 0x7ff23f05d7e0>
print(loss.grad_fn) # <SumBackward0 object at 0x7ff23f05d7e0>
```

However this uses extra memory usage and computation cost. Sometimes we want prevent this behaviour.

[no_grad]

```
torch.no_grad(orig_func=None)
```

“Context-manager that disables gradient calculation.”

```
with torch.no_grad():
    output = input @ w + b
    loss = ((output - target) ** 2).sum()
print(output.grad_fn)  # None
print(loss.grad_fn)  # None
```

Walking the graph — For debugging

```
position = loss.grad_fn

while position is not None:
    print(position.next_functions)
    position = position.next_functions[0][0]
```

```
((<PowBackward0 object at 0x7f1b7fdc57e0>, 0),)
((<SubBackward0 object at 0x7f1b7fe26440>, 0),)
((<AddBackward0 object at 0x7f1b7fdc57e0>, 0), (None, 0))
((<SqueezeBackward4 object at 0x7f1b7fe26440>, 0),
          (<AccumulateGrad object at 0x7f1c64bed7e0>, 0))
((<MmBackward0 object at 0x7f1b7fdc57e0>, 0),)
((None, 0), (<AccumulateGrad object at 0x7f1b7fe26440>, 0))
```

Inplace operations are now a no-no

```
output = b
output += input @ w
# RuntimeError: a leaf Variable that requires grad is being used in an in-place operation.
```

Parameter

The intended way — Parameter

In reality we don’t do this:

```
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
```

we do this:

Parameter

```
torch.nn.parameter.Parameter(data=None, requires_grad=True)
```

“A kind of Tensor that is to be considered a module parameter.”

```
w = torch.nn.Parameter(torch.randn(5, 3))
b = torch.nn.Parameter(torch.randn(3))
```

A new pitfall

However, this is not allowed any more:

```
w = torch.nn.Parameter(torch.randn(5, 3))
w = w * 2
```

X

You need to access the data via .data

```
w = torch.nn.Parameter(torch.randn(5, 3))
w.data = w.data * 2
w.data = torch.randn(5, 3)
```

! You need to remember this if you want to change a parameter e.g. for initializing it.