Advanced PyTorch & Deep Learning Tools

10.01.2024

Topics

- Convolution-like operations
- Tensors in memory
- Data loading
- Hooks
- Training vs Inference

- Reproducibility
- Saving & Loading models
- External Tools and libraries
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - PyTorch Lightning

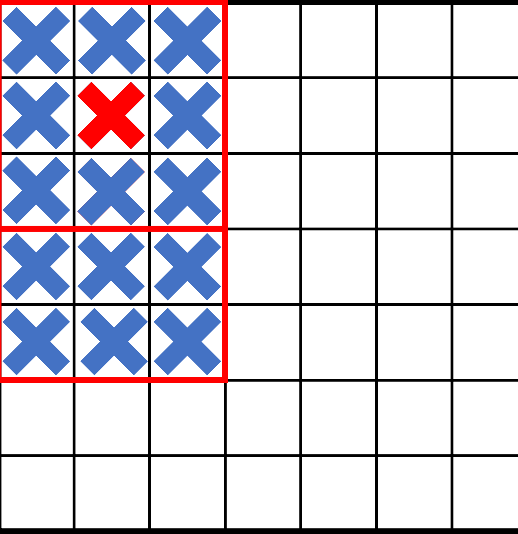
Topics

- **Convolution-like operations**
- Tensors in memory
- Data loading
- Hooks
- Training vs Inference
- Reproducibility
- Saving & Loading models
- External Tools and libraries
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - PyTorch Lightning

Quick Terminology Recap

- `kernel_size`
  - Number of elements in each block.

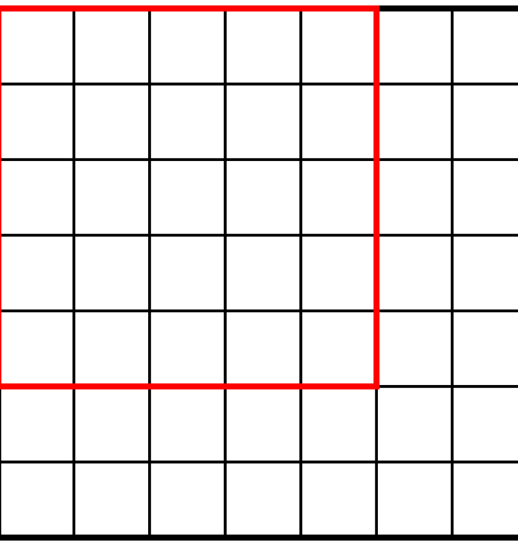
`kernel_size = (5,3)`



Quick Terminology Recap

- `kernel_size`
  - Number of elements in each block.
- `stride`
  - Distance between adjacent blocks.

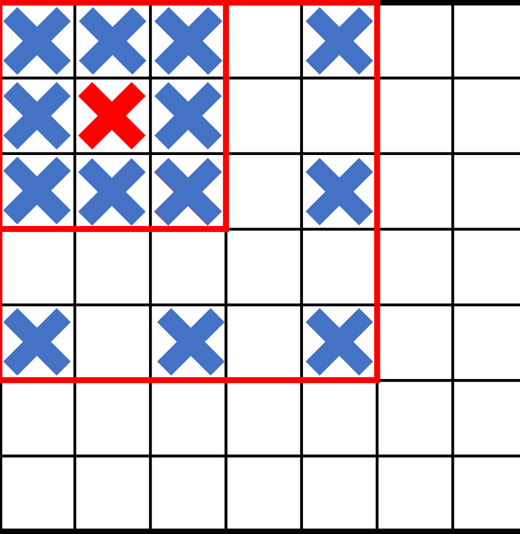
`stride = (2,2)`



Quick Terminology Recap

- `kernel_size`
  - Number of elements in each block.
- `stride`
  - Distance between adjacent blocks.
- `dilation`
  - Distance between adjacent elements in a block.

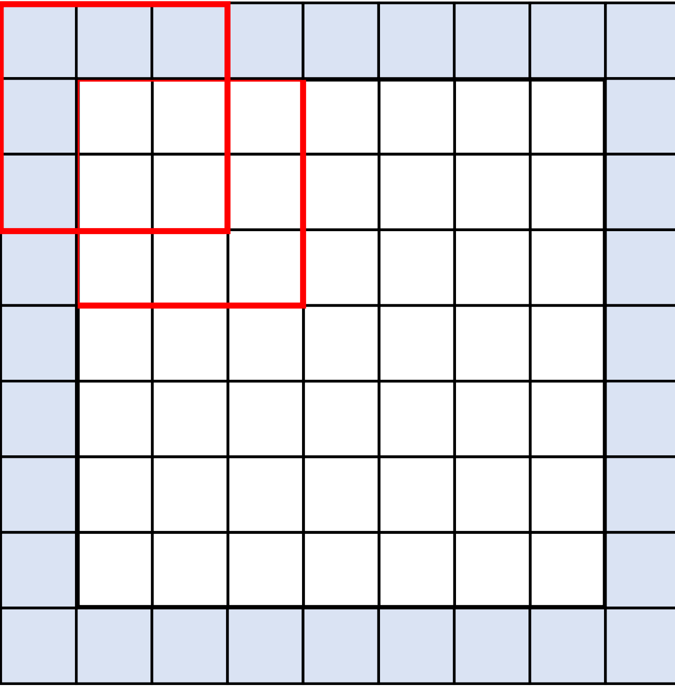
`dilation = (2,2)`



Quick Recap

- `kernel_size`
  - Number of elements in each block.
- `stride`
  - Distance between adjacent blocks.
- `dilation`
  - Distance between adjacent elements in a block.
- `padding`
  - Number of elements added at the borders of the image.

`padding = (1,1)`



Example

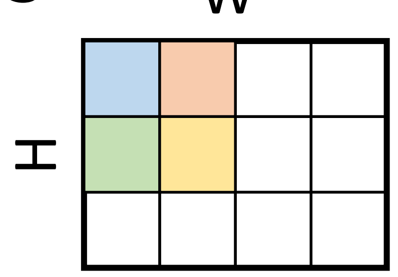
```
# Simple Conv2d layer
conv = nn.Conv2d(..., kernel_size=3)

# With stride
conv = nn.Conv2d(..., kernel_size=3, stride=2)

# With padding
conv = nn.Conv2d(..., kernel_size=3, padding=1)

# Different (H, W) dimensions
conv = nn.Conv2d(..., kernel_size=(3, 1), stride=(2, 1),
padding=(1, 0))

# Simple MaxPool2d layer
pool = nn.MaxPool2d(kernel_size=2, stride=2, dilation=2)
```

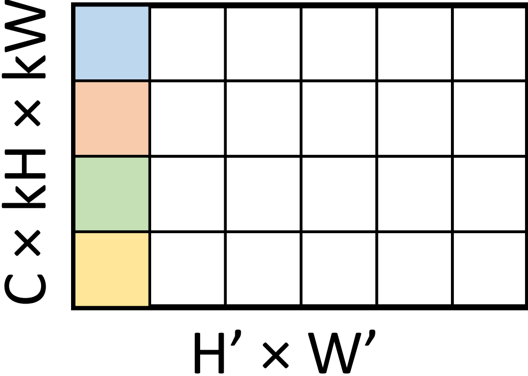
Implementation Convolution-like Ops

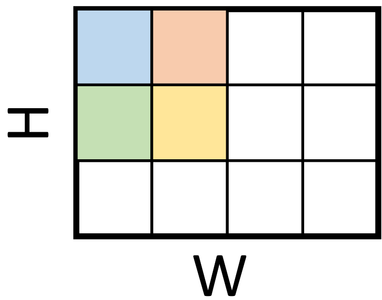
- Extract blocks from an image:
  - `F.unfold` (also called "im2col").

- Apply an operation on each block:
  - Linear, max pooling, etc.

- Combine blocks into an image:
  - `F.fold` (also called "col2im").

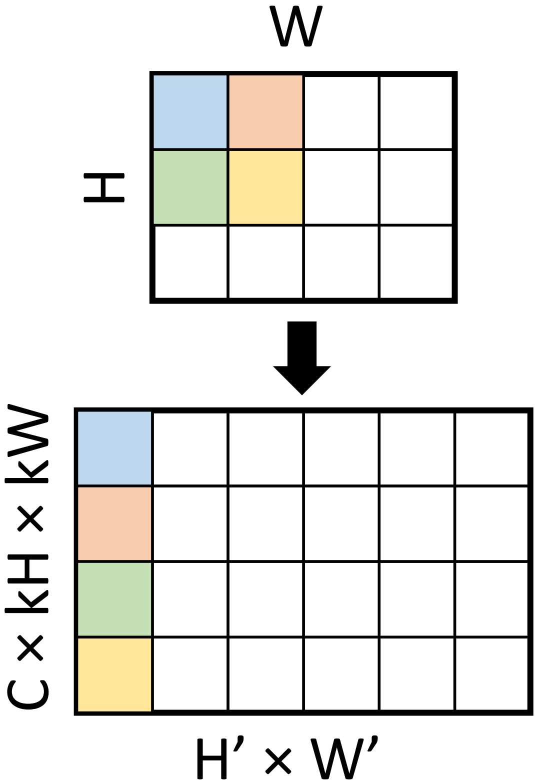






Using `F.unfold`

- Receives a batch of images:
  - Image shape: N, C, H, W
- Extract blocks:
  - Block size: C × kH × kW
  - Num of block per image: H’ × W’
- Return a batch of blocks:
  - Output shape: N, C × kH × kW, H’ × W’



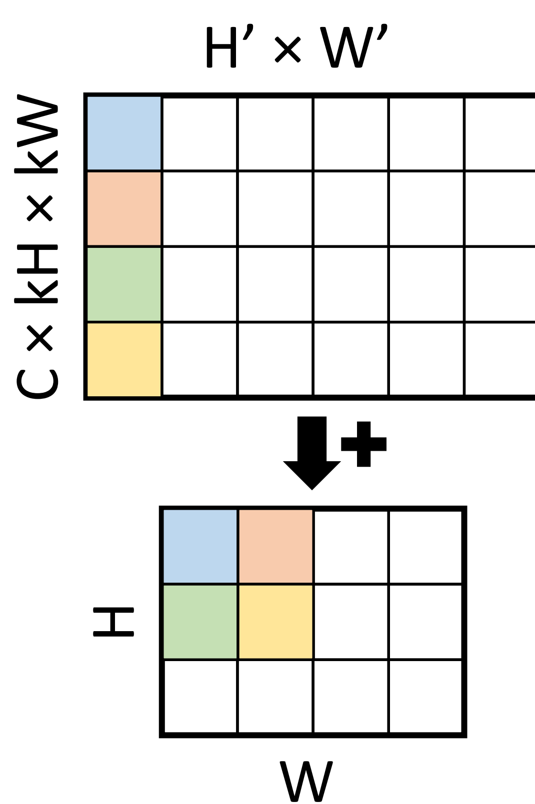
Example

```
image = torch.rand(size=(2, 3, 3, 4))

# Extract blocks
blocks = F.unfold(image, kernel_size=2)
# shape: (batch, block_size, num_blocks)
# blocks.shape == (2, 3 * 2 * 2, 2 * 3)
```


- What is affected by `kernel_size`?
- What is affected by: `stride`? `padding`?

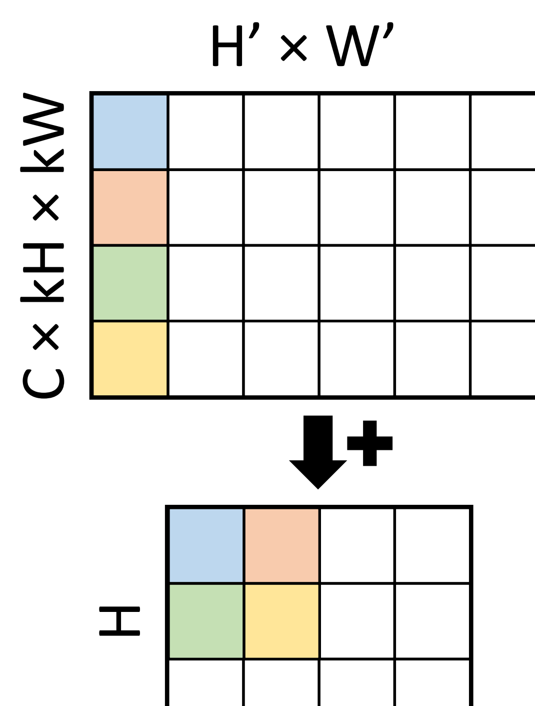
Using `F.fold`

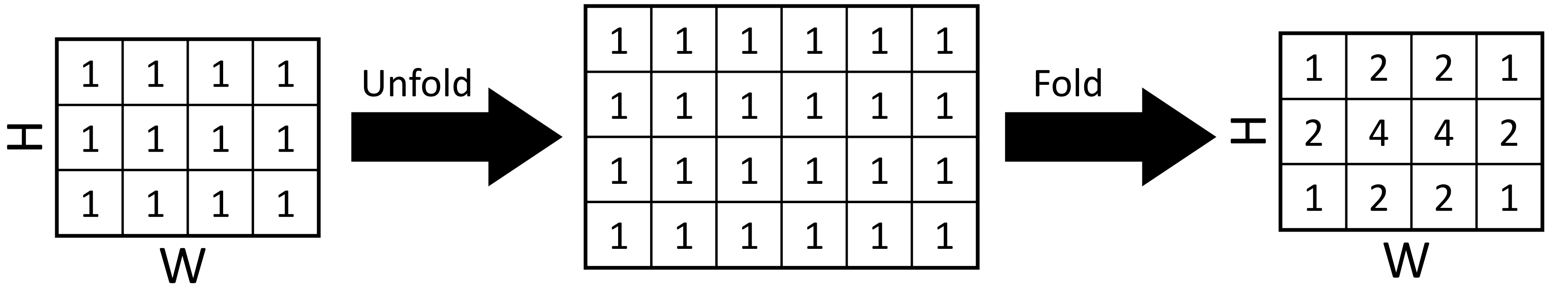
- The opposite of `F.unfold`.
- Receives `blocks`.
- Receives `output_size`.
  - Creates an image `output` of that size.
  - Initialize `output` with zeros.
- Iterates over the blocks.
  - Adds each element to its place in `output`.



Example

```
blocks = torch.rand(size=(2, 1 * 2 * 2, 2 * 3))
output_size = (3, 4)

# fold back into an image
output = F.fold(blocks, output_size, kernel_size=2)
```

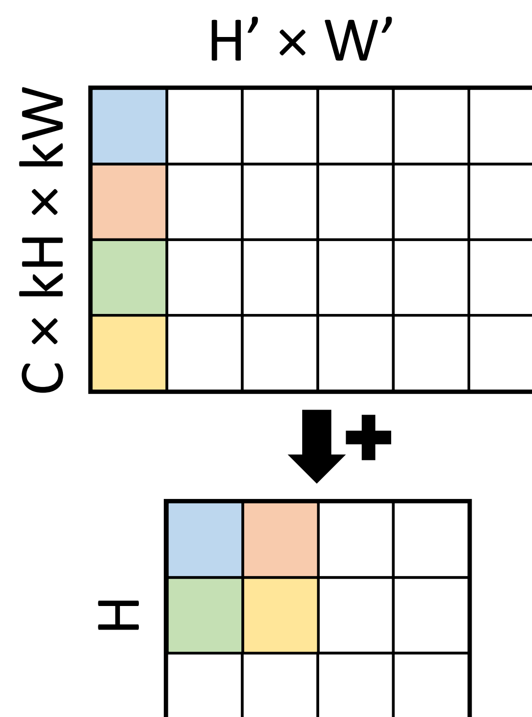


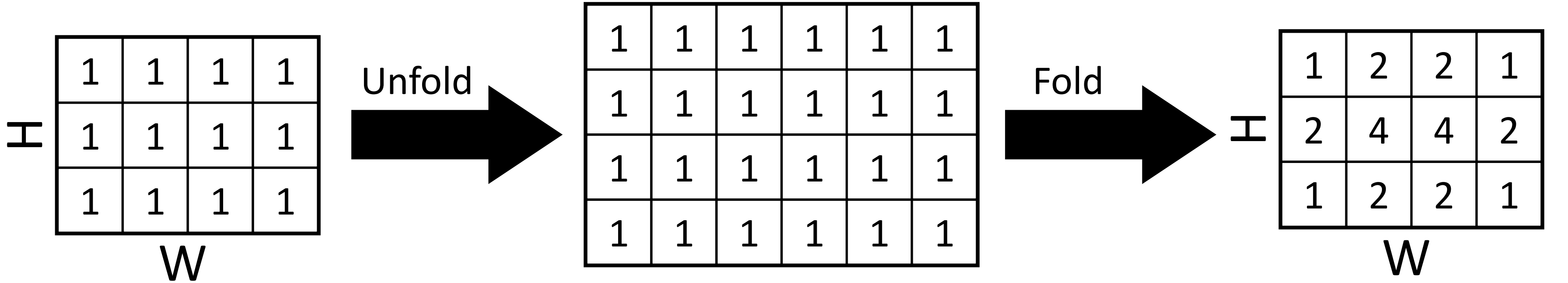


What is the value in `output[1,1]`?

Example

```
blocks = torch.rand(size=(2, 1 * 2 * 2, 2 * 3))
output_size = (3, 4)

# fold back into an image
output = F.fold(blocks, output_size, kernel_size=2)
```





What is the value in `output[1,1]`?

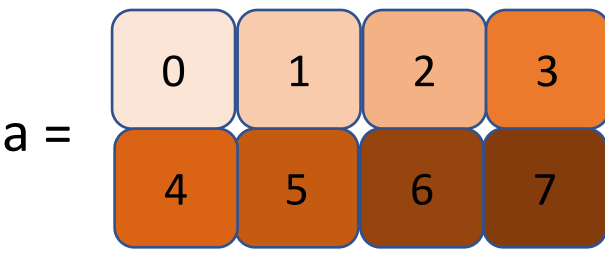
Topics

- Convolution-like operations
- **Tensors in memory**
- Data loading
- Hooks
- Training vs Inference
- Reproducibility
- Saving & Loading models
- External Tools and libraries
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - PyTorch Lightning

How are tensors kept in memory?

- Tensors are saved as 1D-arrays in memory.
- A tensor is (**C**) **contiguous** if
  - It is saved in a single non-broken 1D array
  - It is saved in the correct order
    - 1 byte to go to the next element in the row
    - len(row) bytes to go to the next row

```
a = torch.tensor([[0,1,2,3],
                  [4,5,6,7]])
# shape: 2,4
a.is_contiguous()
# True
```



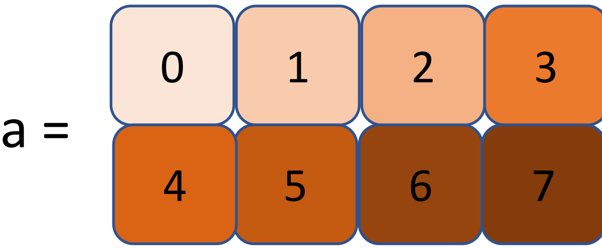
In Memory:

0	1	2	3	4	5	6	7

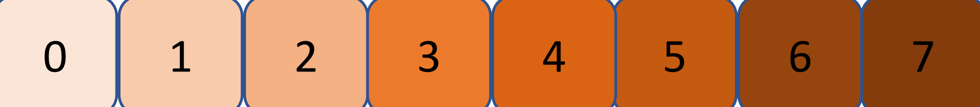
Tensor.view()

- We can change the shape of the tensor
  - `Tensor.view()`
  - Doesn't create a copy of the tensor

```
a = torch.tensor([[0,1,2,3],
                  [4,5,6,7]])
a = a.view(4,2)
# shape: 4,2
a.is_contiguous()
# True
```



In Memory:

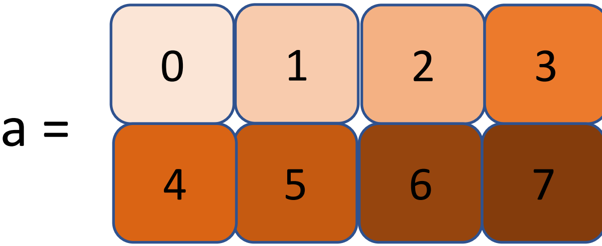


Tensor.transpose()

- We can switch tensor dimensions
  - `Tensor.transpose()`
  - `Tensor.permute()`
  - Doesn’t change memory layout of data
  - Breaks contiguity

```
a = torch.tensor([[0,1,2,3],
                  [4,5,6,7]])
a = a.transpose(0, 1)
# shape: 4,2
a.is_contiguous()
# False
```

a =



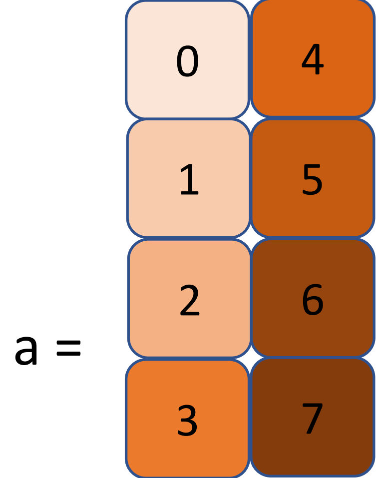
In Memory:

0	1	2	3	4	5	6	7

Back to Tensor.view()

- `Tensor.view()` operates in-place It is not always possible

```
a = torch.tensor([[0,1,2,3],[4,5,6,7]])
a = a.transpose(0, 1)
a = a.view(8)
# RuntimeError
```

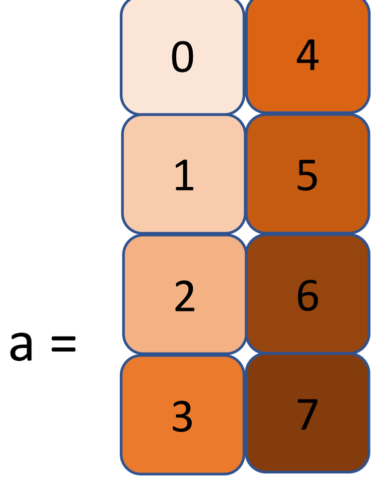


Tensor.reshape()

- `Tensor.reshape()` solves this by copying the tensor

```
a = torch.tensor([[0,1,2,3],[4,5,6,7]])
a = a.transpose(0, 1)
a = a.reshape(8)
```

a =



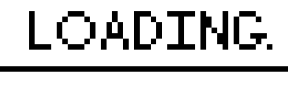
In Memory:

Reshape vs View

- `Tensor.reshape()`
  - Use if you just want to reshape
- `Tensor.view()`
  - Use if you have memory concerns
  - Use if you want both tensors to share the data in memory

Topics

- Convolution-like operations
- Tensors in memory
- **Data loading**
- Hooks
- Training vs Inference
- Reproducibility
- Saving & Loading models
- External Tools and libraries
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - PyTorch Lightning



Datasets and Dataloaders

- Datasets are collection of data samples.
  - Collection of images and labels.
  - Collection of pairs of images.

- Dataloaders help loading data from Datasets:
  - Create batches.
  - Support multi-processing.

Datasets

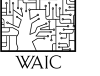
- Should implement `__init__`, `__len__` and `__getitem__`
- Usually returns a tensor, a tuple of tensors or dict of tensors.

```
class MyDataset(torch.utils.data.Dataset):
  def __init__(self, img_paths: List[str], transform):
    self.img_paths = img_paths
    self.transform = transform

  def __len__(self):                  # == len(dataset)
    return len(self.img_paths)

  def __getitem__(self, index):   # == dataset[index]
    img = load_image(self.img_paths[index])
    img = self.transform(img)
    return {"img": img}
```

Dataloaders

- Receives a dataset and batch size.
- Returns an iterator.

```
transform = torchvision.transforms.Compose([
  torchvision.transforms.CenterCrop(256),
  torchvision.transforms.ToTensor()
])
dataset = MyDataset(img_paths=["a.jpg", "b.jpg�even", … “zzz.jpg”],
                    transform=transform)

dataloader = torch.utils.data.DataLoader(dataset,
                                         batch_size=32)
for batch in dataloader:
  img = batch["img"]
  # shape: 32,3,256,256
  # do something
```



Topics

- Convolution-like operations
- Tensors in memory
- Data loading
- **Hooks**
- Training vs Inference
- Reproducibility
- Saving & Loading models
- External Tools and libraries
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - PyTorch Lightning

Accessing Intermediate Results

- Why would one access intermediate results?
  - Feature extraction
  - Weight gradient extraction
    - GradCAM
  - Modifying intermediate outputs
    - Regularization
  - Special losses

Hooks

- A hook is function registered to a module / tensor
- Forward hooks
  - Module only
  - `Module.register_forward_hook()`
    - Called **after** the forward call
  - `Module.register_forward_pre_hook()`
    - Called **before** the forward call

Hooks

- A hook is function registered to a module / tensor
- Backward hooks
  - `Module.register_full_backward_hook()`
    - Called (every time) after a gradient w.r.t to the module output is computed
    - Can modify gradients by returning new value
  - `Tensor.register_hook()`
    - Called (every time) after a gradient w.r.t to the tensor is computed
    - Tensor must have `require_grad=True`

Register a Hook

```
def my_hook(module, input, output):
  # module: the module being hooked
  # input: a tuple of inputs
  # output: a tuple of outputs
  print("hook!", input[0].shape, output[0].shape)

# Register the hook
net.conv1.register_forward_hook(my_hook)

# Trigger the hook
y = net(x)
# printed: "hook! torch.Size([...]) torch.Size([...])"
```

Remove a Hook

- A remove handle is returned during the registration

```
# Register the hook
handle = net.conv1.register_forward_hook(my_hook)

# Remove the hook
handle.remove()

# Trigger the hook
y = net(x)
# nothing is printed
```

Topics

- Convolution-like operations
- Tensors in memory
- Data loading
- Hooks
- **Training vs Inference**
- Reproducibility
- Saving & Loading models
- External Tools and libraries
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - PyTorch Lightning

Training vs Inference

- Important – always set your model to the correct “mode”
- Affects various modules
  - Batch normalization
    - And other normalization layers
  - Dropout

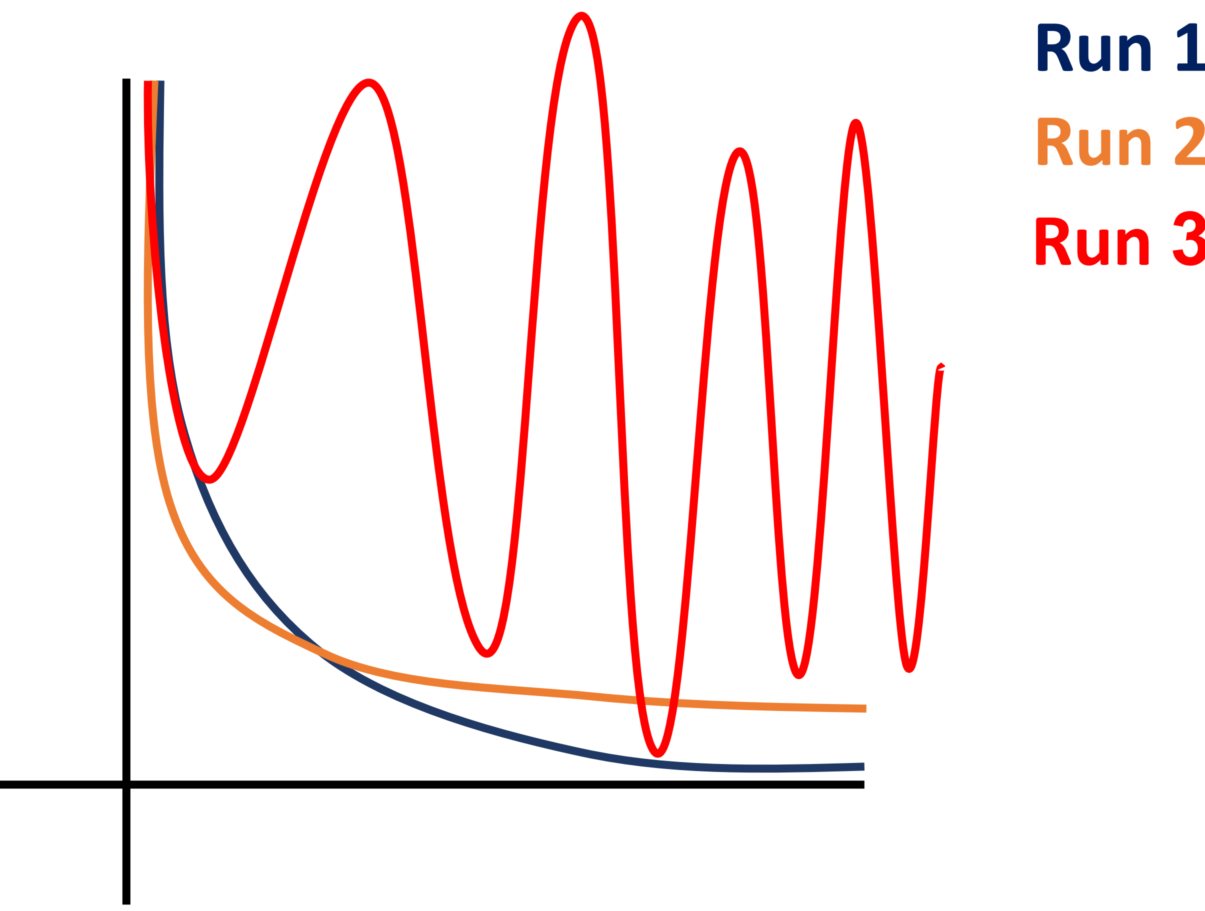
```
def train_loop(dataloader, model, criterion, optimizer):
  size = len(dataloader.dataset)
  model.train()
  running_loss, running_corrects = 0, 0

  # iterate through all batches
  for batch, (X, y) in enumerate(dataloader):
    # move data to device
    X, y = X.to(device), y.to(device)
    # forward pass
    pred = model(X)
    loss = criterion(pred, y)
    # new gradients per batch
    optimizer.zero_grad()
    # backward pass
    loss.backward()
    # update gradients
    optimizer.step()

    running_loss += loss.item()
    running_corrects += (pred.argmax(1) == y).type(torch.float).sum().item()

  epoch_loss = running_loss / size
  epoch_accuracy = 100 * running_corrects / size
  return epoch_loss, epoch_accuracy
```

```
def inference_loop(dataloader, model, criterion):
  size = len(dataloader.dataset)
  model.eval()
  running_loss, running_corrects = 0, 0

  # disregard gradients when not training
  with torch.no_grad():
    # iterate through all batches
    for X, y in dataloader:
      # move data to device
      X, y = X.to(device), y.to(device)
      # forward pass
      pred = model(X)
      # save data for evaluation measures (loss & accuracy)
      running_loss += criterion(pred, y).item()
      running_corrects += (pred.argmax(1) == y).type(torch.float).sum().item()

  epoch_loss = running_loss / size
  epoch_accuracy = 100 * running_corrects / size
  return epoch_loss, epoch_accuracy
```

Topics

- Convolution-like operations
- Tensors in memory
- Data loading
- Hooks
- Training vs Inference
- **Reproducibility**
- Saving & Loading models
- External Tools and libraries
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - PyTorch Lightning

Reproducibility



Limiting randomness

```
# Set random seeds
seed = 42
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(seed)

# Use deterministic algorithms only
torch.use_deterministic_algorithms(True)

# Use deterministic convolution algorithm in CUDA
torch.backends.cudnn.benchmark = False
torch.backends.cudnn.deterministic = True
```

Based on:
https://pytorch.org/docs/stable/notes/randomness.html

Limiting randomness

```python
# Fix workers randomness
def seed_worker(worker_id):
    worker_seed = torch.initial_seed() % 2**32
    numpy.random.seed(worker_seed)
    random.seed(worker_seed)

g = torch.Generator()
g.manual_seed(seed)

DataLoader(train_dataset, batch_size, num_workers,
    generator=g
    worker_init_fn=seed_worker,
    )
```

Topics

- Convolution-like operations
- Tensors in memory
- Data loading
- Hooks
- Training vs Inference
- Reproducibility
- **Saving & Loading models**
- External Tools and libraries
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - PyTorch Lightning

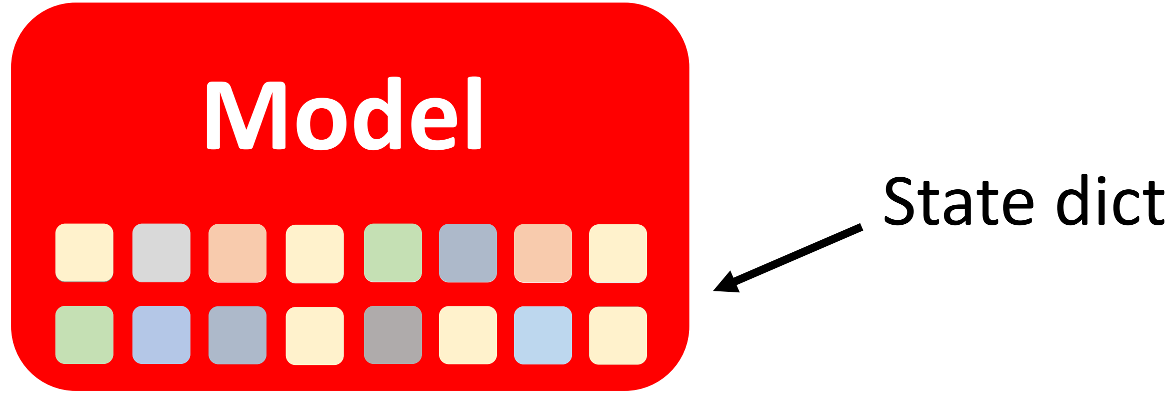
Saving & Loading Models

Serialize entire model

```
torch.save(model, "my_model.pth")
...
model = torch.load("my_model.pth")
```

```
# save an object to disk
torch.save(object, path)

# load an object from disk
object = torch.load(path)
```

Pros:	**Cons:**
Simple	Can’t change the model class



Saving & Loading Models

```
# Define model
class ModelClass(nn.Module):
    def __init__(self):
        super(TheModelClass, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        ...

# Initialize model
model = ModelClass()
```

```
Model's state_dict:
conv1.weight torch.Size([6,3,5,5])
conv1.bias   torch.Size([6])
conv2.weight torch.Size([16,6,5,5])
conv2.bias   torch.Size([16])
fc1.weight   torch.Size([120, 400])
fc1.bias     torch.Size([120])
fc2.weight   torch.Size([84, 120])
fc2.bias     torch.Size([84])
fc3.weight   torch.Size([10, 84])
fc3.bias     torch.Size([10])
```

Saving & Loading Models

Saving the better way

```
# save the model’s state dict
torch.save(model.state_dict(), "my_model.pth")

...

# create and load the model’s state dict
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load("my_model.pth"))
```

```
# save an object to disk
torch.save(object, path)

# load an object from disk
torch.load(path)
```

Saving & Loading Models

```
# Initialize optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
Optimizer's state_dict:
state    {}
param_groups     [{'lr': 0.001, 'momentum': 0.9, 'weight_decay': 0, ...]
```

Saving & Loading Models

Saving for training

```
checkpoint = torch.save({'epoch': epoch,
                         'model_sd': model.state_dict(),
                         'opt_sd': optimizer.state_dict(),
                         'loss': loss,
                         ...}, 'checkpoint.pth')
```

```
model = TheModelClass(*args, **kwargs)
optimizer = TheOptimizerClass(*args, **kwargs)

checkpoint = torch.load('checkpoint.pth')
model.load_state_dict(checkpoint['model_sd'])
optimizer.load_state_dict(checkpoint['opt_sd'])
epoch = checkpoint['epoch']
loss = checkpoint['loss’]
# continue training
```

Topics

- Convolution-like operations
- Tensors in memory
- Data loading
- Hooks
- Training vs Inference
- Reproducibility
- Saving & Loading models
- **External Tools and libraries**
  - **Using pre-trained models**
  - Monitoring
  - Data augmentations
  - PyTorch Lightning

Pretrained Models

TorchVision

Torch hub, Torchvision, Timm

```
import torch
import torchvision

# load models

torch.hub.list('pytorch/vision’) # ~100 models
torchhub_model = torch.hub.load('pytorch/vision', 'resnet18’,
                                weights='ResNet18_Weights.DEFAULT’)

dir(torchvision.models) # ~200 models
torchvision_model = torchvision.models.resnet18(pretrained=True)

!pip install timm
import timm
timm.list_models(pretrained=True) # ~1300 models
timm_model = timm.create_model('resnet18', pretrained=True)

model.eval()
input = torch.randn(1, 3, 224, 224)
output =  model(input)
```

Hugging Face

- The most updated model hub
- Models
- Demos
- Datasets
- Inspiration for projects (?)

Topics

- Convolution-like operations
- Tensors in memory
- Data loading
- Hooks
- Training vs Inference
- Reproducibility
- Saving & Loading models
- **External Tools and libraries**
  - Using pre-trained models
  - **Monitoring**
  - Data augmentations
  - PyTorch Lightning

Weights & Biases

live loss plot

accuracy

Loss

training
validation

epoch

carey > Projects > pytorch-sweep

Runs (211)

Search panels

Create report

Charts 13

Name (211 visualized)

brisk-sweep-210
solar-sweep-210
ethereal-sweep-209
blooming-sweep-208
floral-sweep-207
olive-sweep-206
zesty-sweep-205

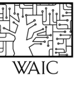
batch_size
dropout
accuracy

Accuracy Sandal
Showing first 10 runs

Step

Weights & Biases

```python
import random
import wandb

total_runs = 5 # Launch 5 simulated experiments
for run in range(total_runs):
    wandb.init( # Start a new run to track this script
      project="basic-intro",
      name=f"experiment_{run}",
      config={"learning_rate": 0.02,
              "architecture": "CNN",
              "dataset": "CIFAR-100",
              "epochs": 10,}
      )
    epochs = 10 # This simple block simulates a training loop logging metrics
    offset = random.random() / 5
    for epoch in range(2, epochs):
        acc = 1 - 2 ** -epoch - random.random() / epoch - offset
        loss = 2 ** -epoch + random.random() / epoch + offset
        wandb.log({"acc": acc, "loss": loss})

    wandb.finish() # Mark the run as finished
```



Weights & Biases

Runs (5)

Search runs

Name (5 visualized)

experiment_4

experiment_3

experiment_2

experiment_1

experiment_0

Search panels with regex

Charts 2

loss

experiment_4 experiment_3 experiment_2 experiment_1 experiment_0

acc

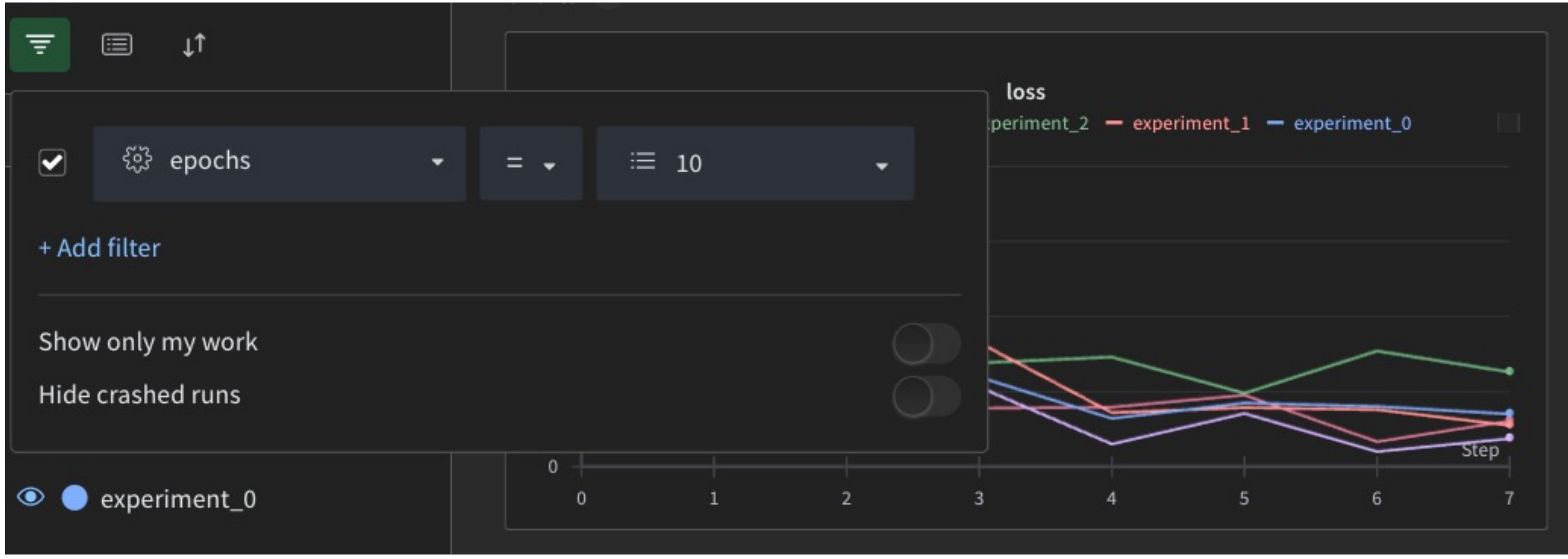
experiment_4 experiment_3 experiment_2 experiment_1 experiment_0

Step

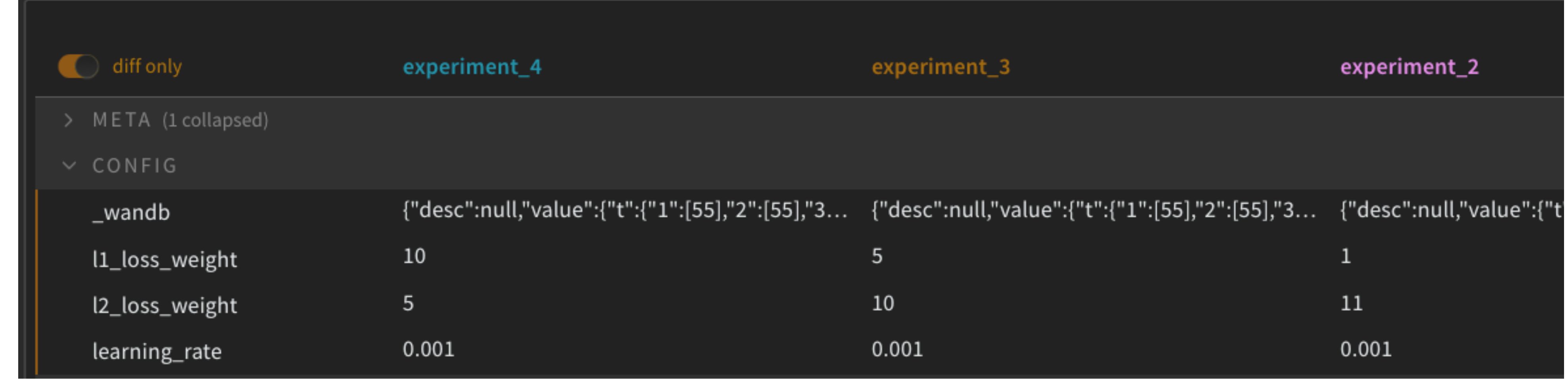
System 11

Weights & Biases

- Filter by hyperparameters



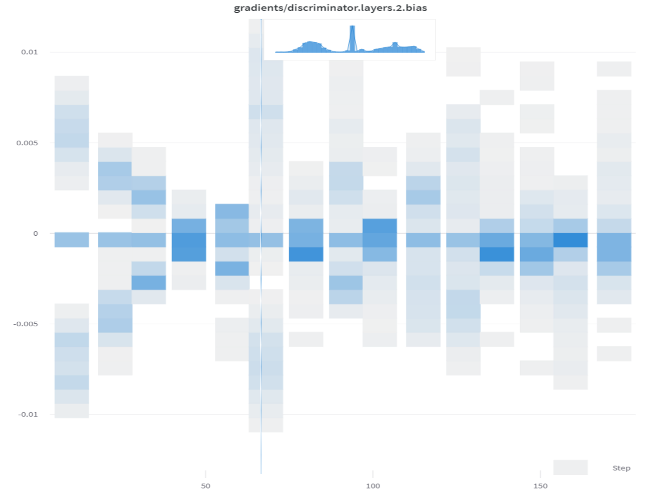
- Compare run configurations

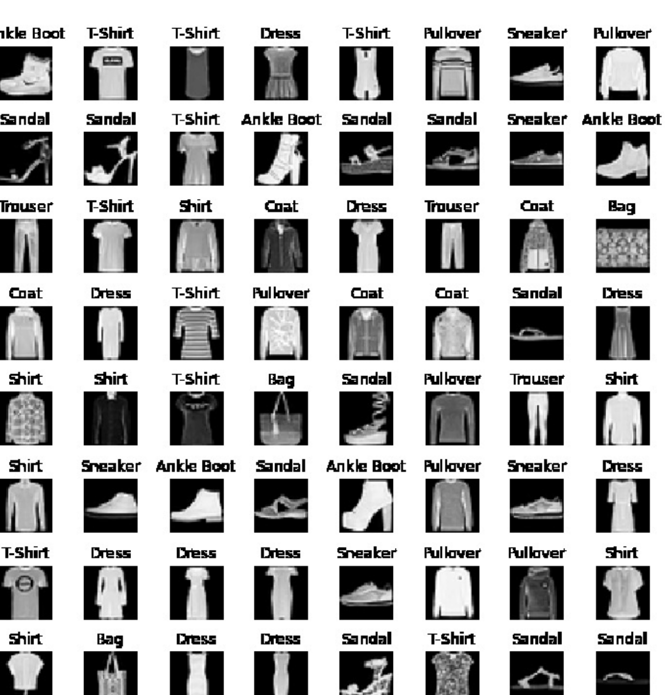


```
wandb.log({”gradients": wandb.Histogram(grads})
```

```
# get a batch of data
X, y = next(iter(train_dataloader))

# create a grid from all images
grid = torchvision.utils.make_grid(X)
wandb.log({”image_grid": wandb.Image(grid})
```

```
wandb.log({”video":
            wandb.Video(video_array, fps=4, format="gif")
            })
```



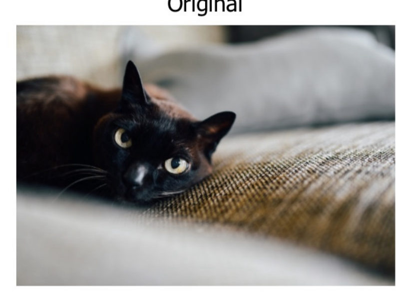
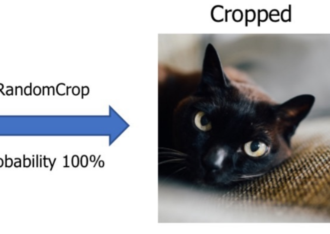
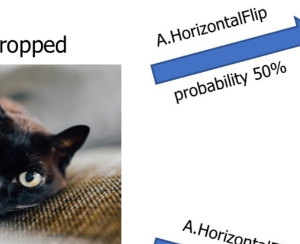
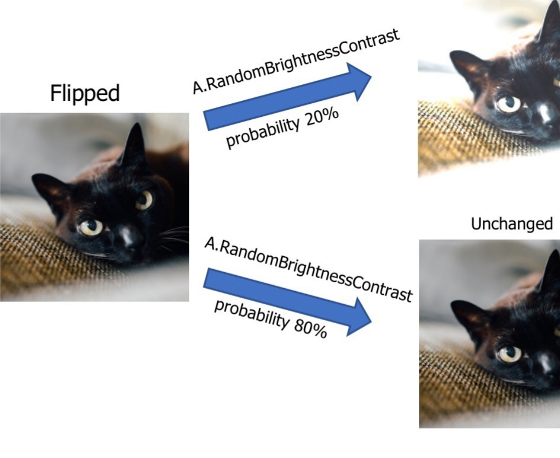
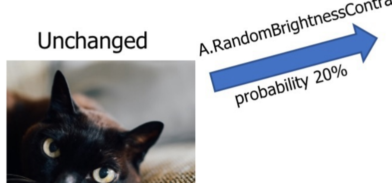
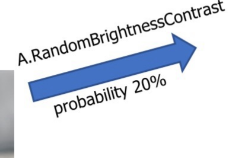
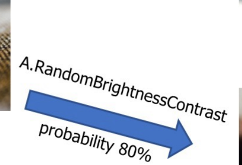
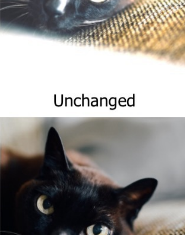


Topics

- Convolution-like operations
- Tensors in memory
- Data loading
- Hooks
- Training vs Inference
- Reproducibility
- Saving & Loading models
- **External Tools and libraries**
  - Using pre-trained models
  - Monitoring
  - **Data augmentations**
  - PyTorch Lightning

Albumentations

- Data Augmentations for various tasks
  - Classification / Representation Learning

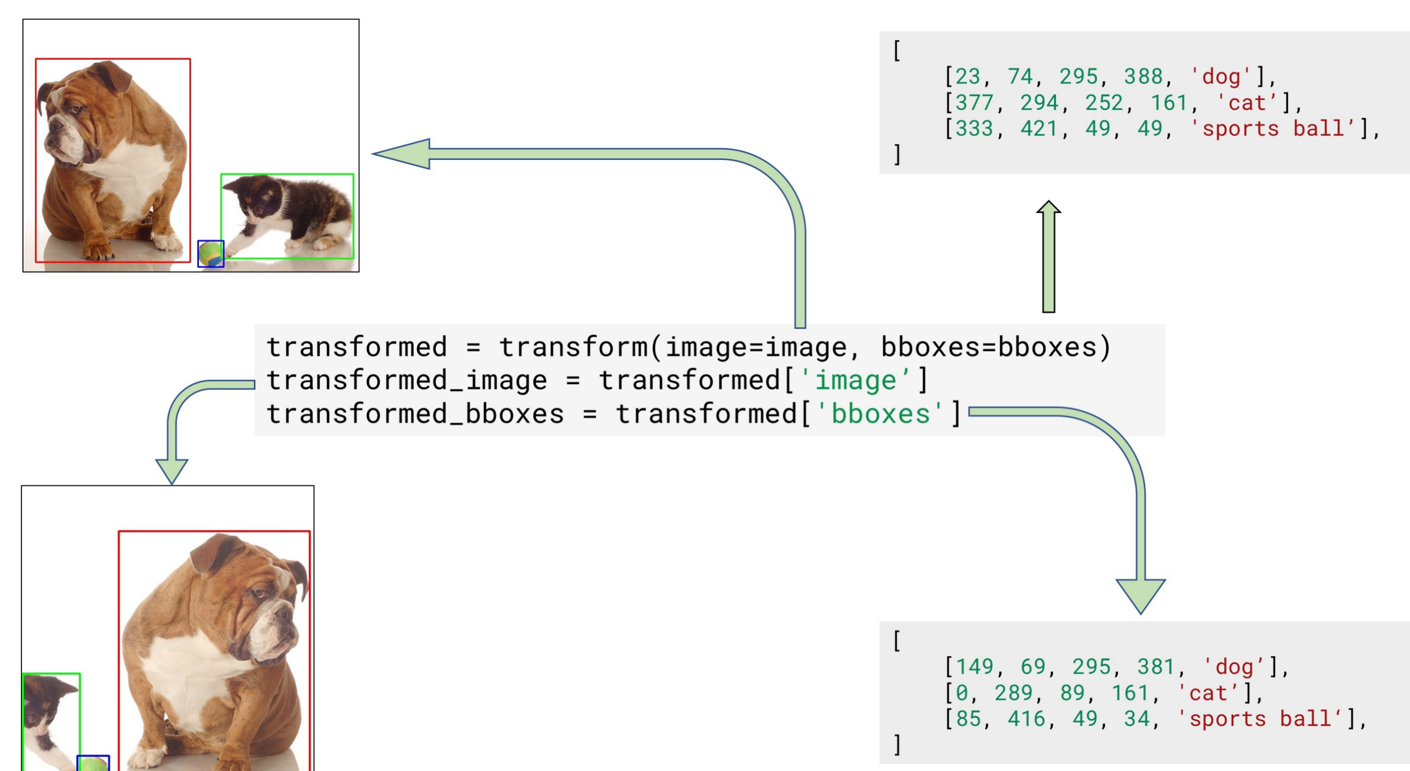









```
import albumentations as A

transform = A.Compose([
    A.RandomCrop(width=256, height=256),
    A.HorizontalFlip(p=0.5),
    A.RandomBrightnessContrast(p=0.2),
])
```

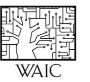
Images from albumentations tutorial:
https://albumentations.ai/docs/getting_started/image_augmentation/

Albumentations

- Data Augmentations for various tasks
  - Classification / Representation Learning
  - Object Detection

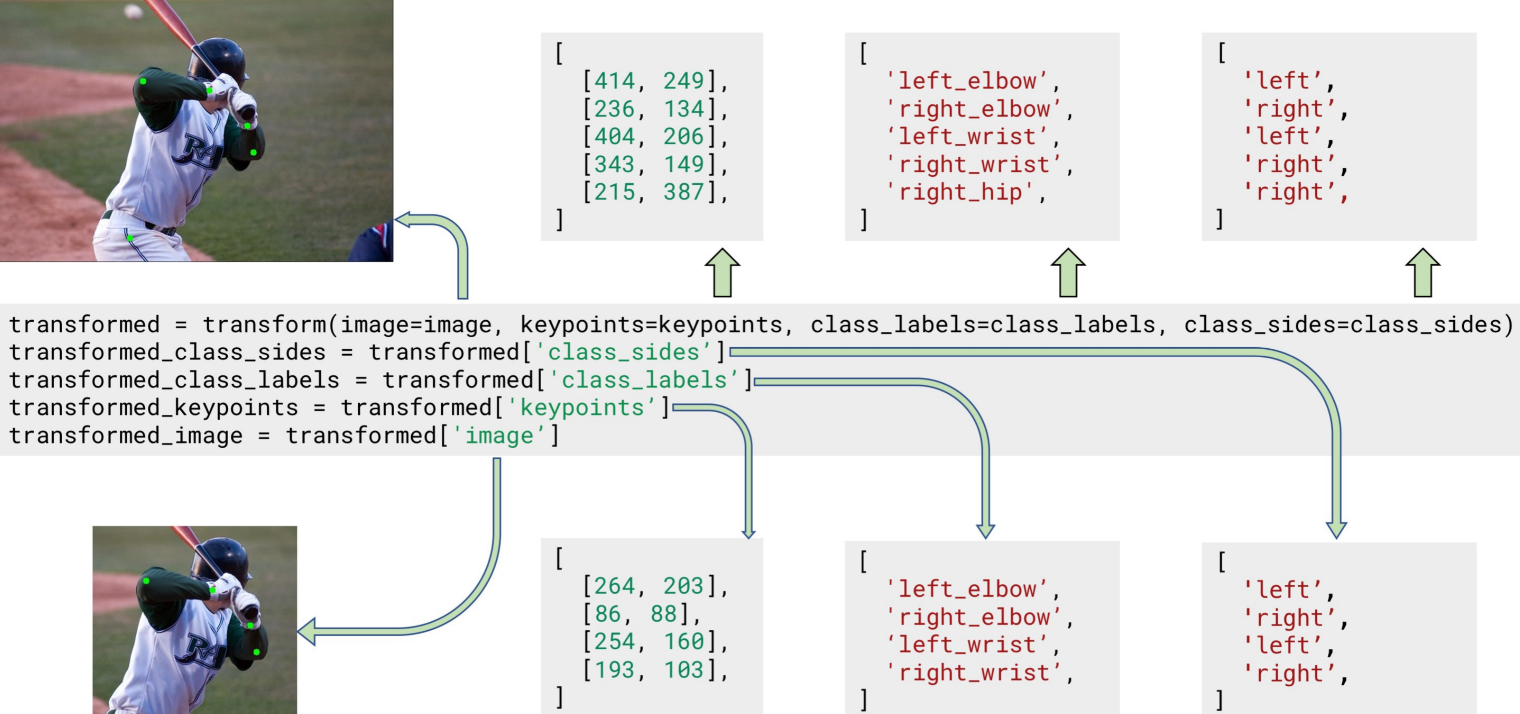


Images from albumentations tutorial:
https://albumentations.ai/docs/getting_started/bounding_boxes_augmentation/



Albumentations

- Data Augmentations for various tasks
  - Classification / Representation Learning
  - Object Detection
  - Keypoint Detection



Images from albumentations tutorial:
https://albumentations.ai/docs/getting_started/keypoints_augmentation/

Albumentations

- Data Augmentations for various tasks
  - Classification / Representation Learning
  - Object Detection
  - Keypoint Detection
  - Mask Segmentation

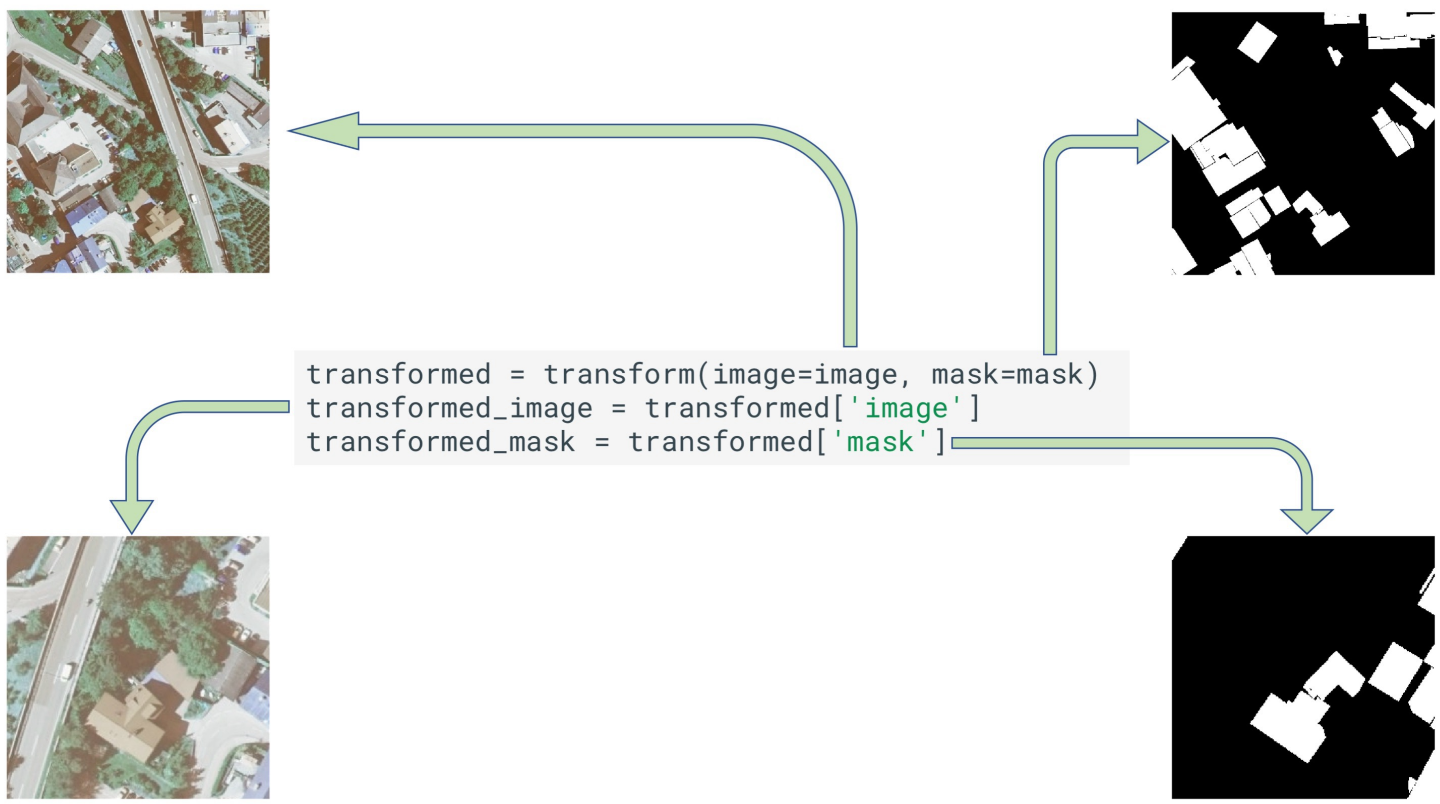
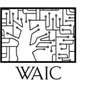


Image credit to albumentations tutorial:
https://albumentations.ai/docs/getting_started/mask_augmentation/



kornia

kornia

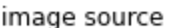
PyTorch

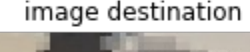
```
import torch
import kornia

frame: torch.Tensor = load_video_frame(...)

out: torch.Tensor = (
    kornia.rgb_to_grayscale(frame)
)
```

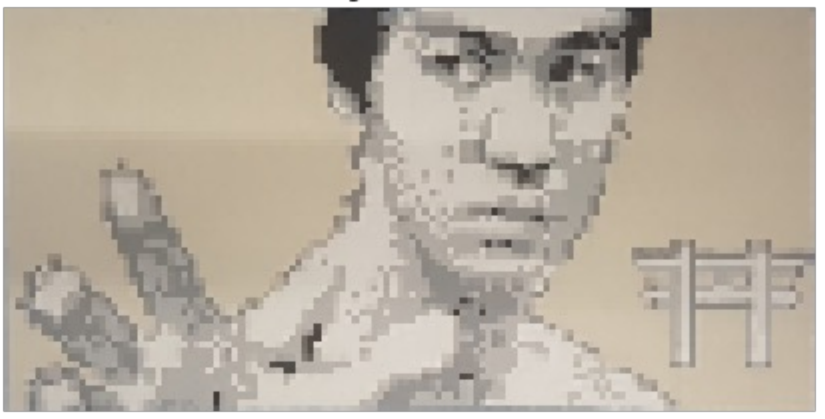
FULLY DIFFERENTIABLE

kornia

```
# compute perspective transform
M = K.get_perspective_transform(points_src, points_dst)

# warp the original image by the found transform
img_warp = K.warp_perspective(img.float(), M, dsize=(h, w))
```

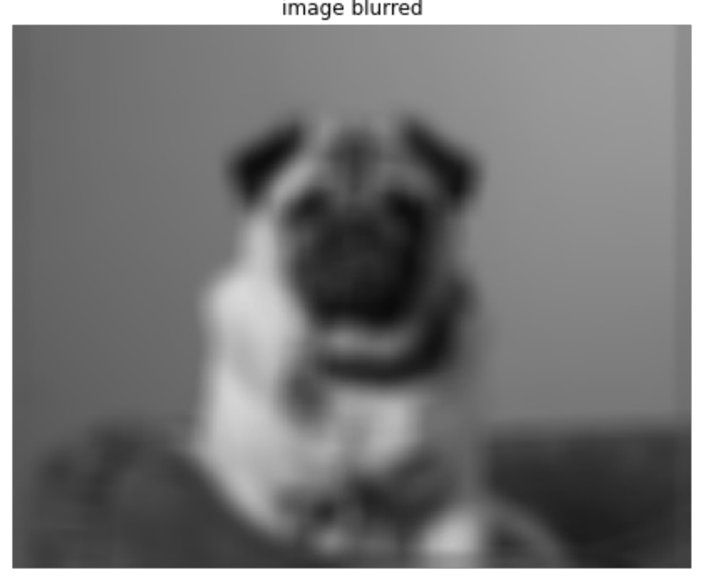






kornia

```
# create the operator
gauss = K.filters.GaussianBlur2d((11, 11), (10.5, 10.5))

# blur the image
x_blur = gauss(data.float())
```



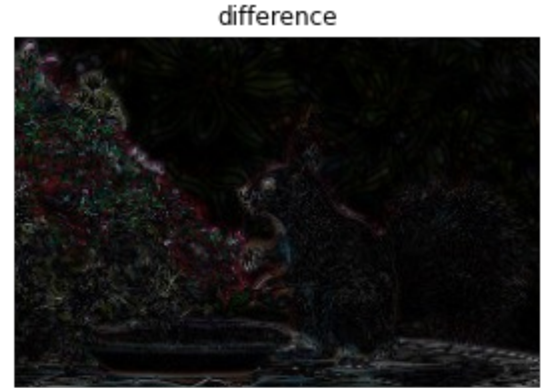
kornia

```
# define sharpening mask
sharpen = kornia.filters.UnsharpMask((9,9), (2.5,2.5))

# create the sharpened image
sharpened_tensor = sharpen(data)

# get difference between original and sharpened image
difference = (sharpened_tensor - data).abs()
```

image source

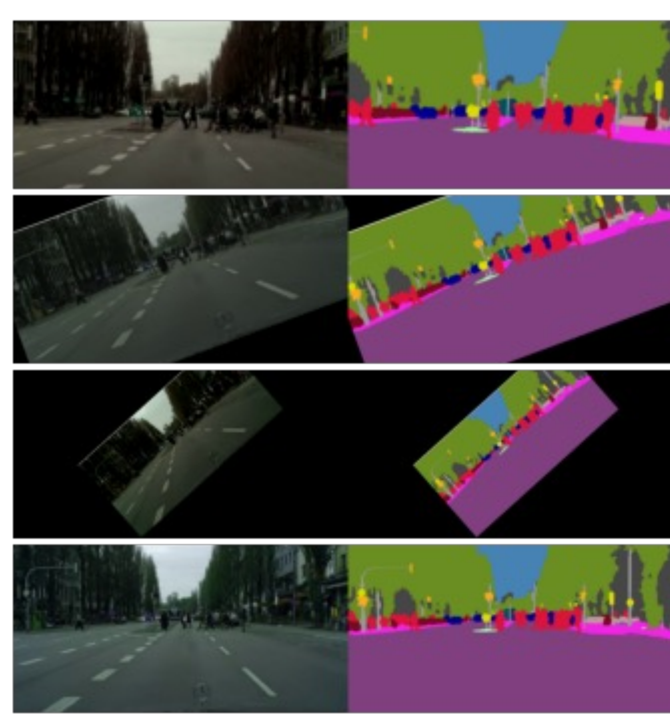
sharpened

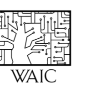
difference



kornia

```python
import torch
import torch.nn as nn
import kornia as K

img = load_image(...) #BxCxHxW

aug = nn.Sequential(
    K.augmentations.ColorJitter(0.15, 0.25,
                                0.25, 0.25),
    K.augmentation.RandomAffine([-45., 45.],
                                [0., 0.15],
                                [0.5, 1.5],
                                [0., 0.15])
)

out = aug(img) #BxCxHxW
```





e i n o p s

einops

einops

```python
from einops import rearrange

x = torch.randn(2, 32, 32, 3) # B H W C

# out1 = x.permute(0, 3, 1, 2)
out1 = rearrange(x, 'b h w c -> b c h w')

# out2 = x.permute(0, 3, 2, 1).reshape(2, 3, -1).unsqueeze(0)
out2 = rearrange(x, 'b h w c -> 1 b c (w h)')

# out3 = out2.reshape(2, 3, 32, 32).permute(0,1,3,2)
out3 = rearrange(out2, '1 b c (w h) -> b c h w', h=32, w=32)
```

einops

```
from einops import reduce, repeat

x = torch.randn(2, 3, 32, 32)

# out1 = x.mean(dim=[2,3])
out1 = reduce(x, 'b c h w -> b c', reduction='mean')

# repeat along specified dimension
out1 = repeat(x[0], 'c h w -> axis c h w', axis=5)
```

Topics

- Convolution-like operations
- Tensors in memory
- Data loading
- Hooks
- Training vs Inference
- Reproducibility
- Saving & Loading models
- **External Tools and libraries**
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - **PyTorch Lightning**

PyTorch Lightning

- A PyTorch research framework
- Designed to eliminate boilerplate code
    - Training loop
    - Simple device and Multi GPU management
    - Automatic (and customizable) checkpoints
    - And many more

Simple training

```
class LightningModel(L.LightningModule):
        def __init__(self, model):
                super().__init__()
                self.model = model
        def training_step(self, batch, batch_idx):
                x, y = batch
                y_hat = self.model(x)
                loss = F.mse_loss(y_hat, y)
                return loss
        def configure_optimizers(self):
                optimizer = torch.optim.Adam(self.parameters(), lr=1e-3)
                return optimizer

dataset = MNIST(os.getcwd(), download=True)
# model
autoencoder = LightningModel(Net())
# train model
trainer = L.Trainer(device=)
trainer.fit(model=autoencoder, train_dataloaders=DataLoader(dataset))
```

WAIC

Don’t Reinvent the Wheel!

Use Existing Tools!

Questions?

- Convolution-like operations
- Tensors in memory
- Data loading
- Hooks
- Training vs Inference
- Reproducibility
- Saving & Loading models
- External Tools and libraries
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - PyTorch Lightning

Next week:

Niv Haim

Adversarial Examples