Advanced

PyTorch

& Deep Learning Tools

YOU WERE HERE

YOU ARE HERE

YOU ARE GOING HERE

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

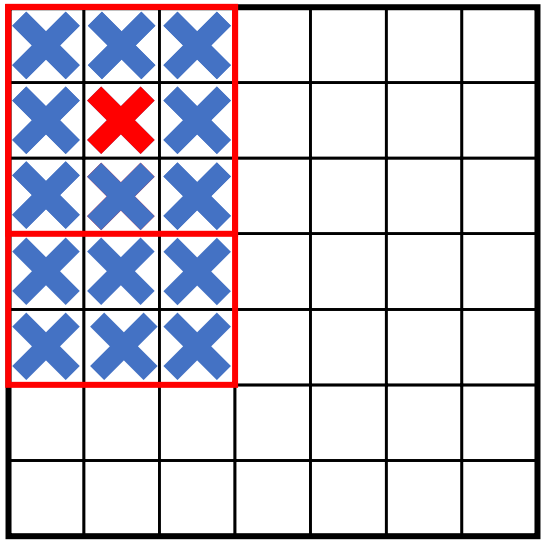
Topics

- **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 Recap

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

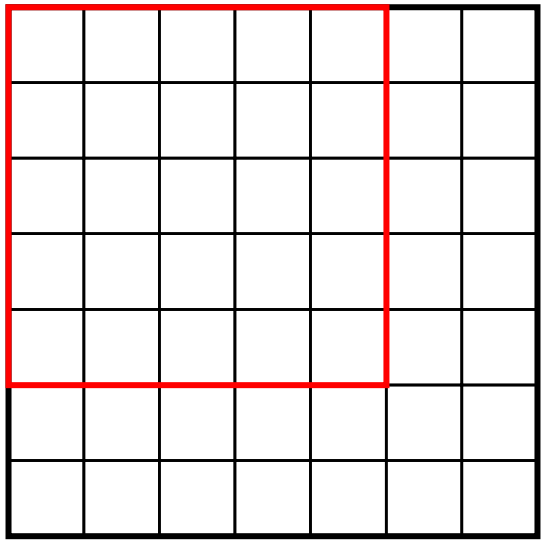
`kernel_size = (3,3)`



Quick Recap

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

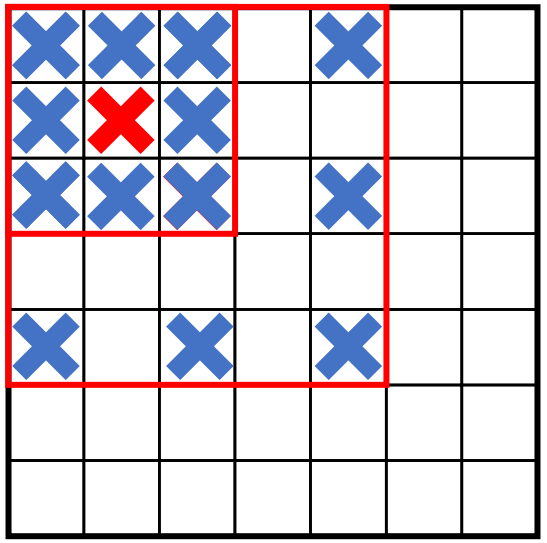
`stride = (2,2)`



Quick Recap

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

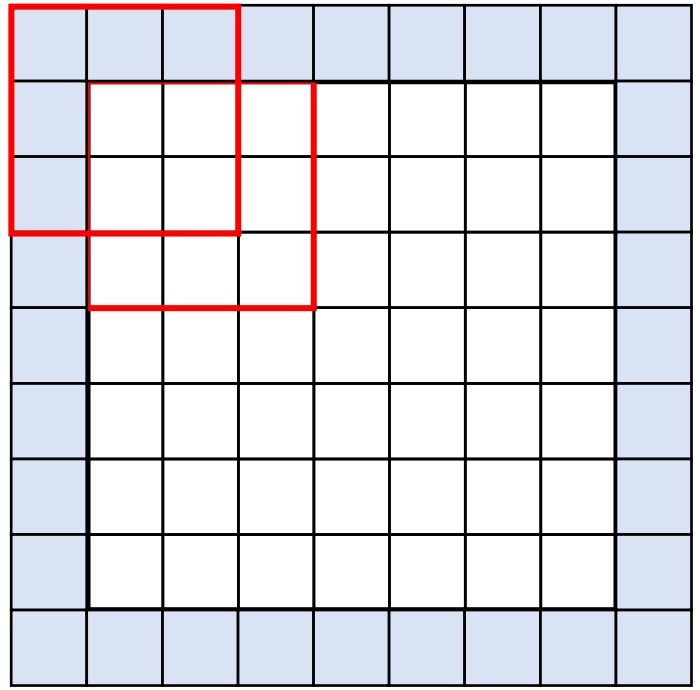
`dilation = (2,2)`



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)`

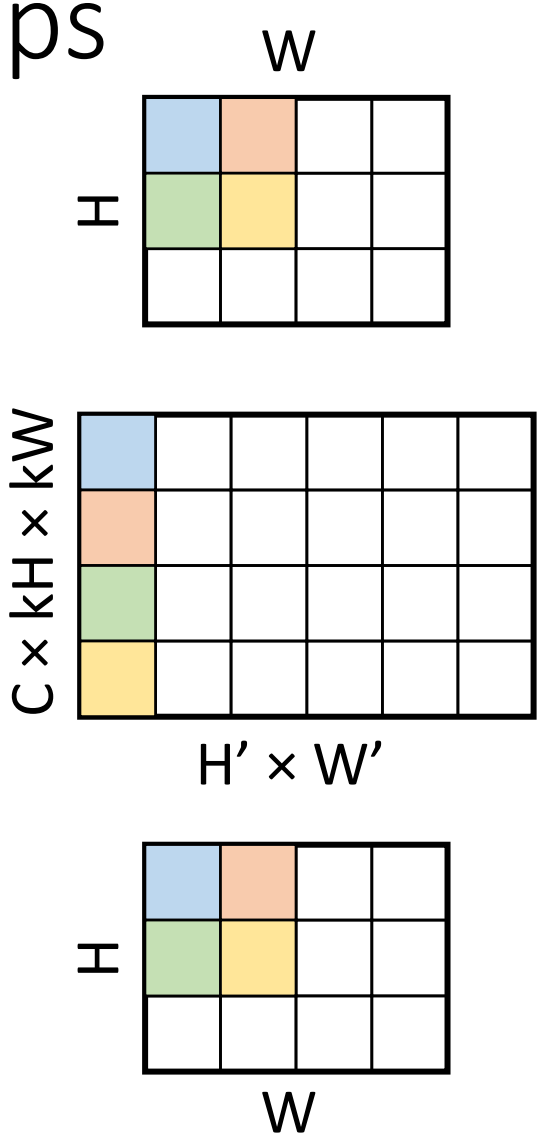


Example

```
# 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)
```

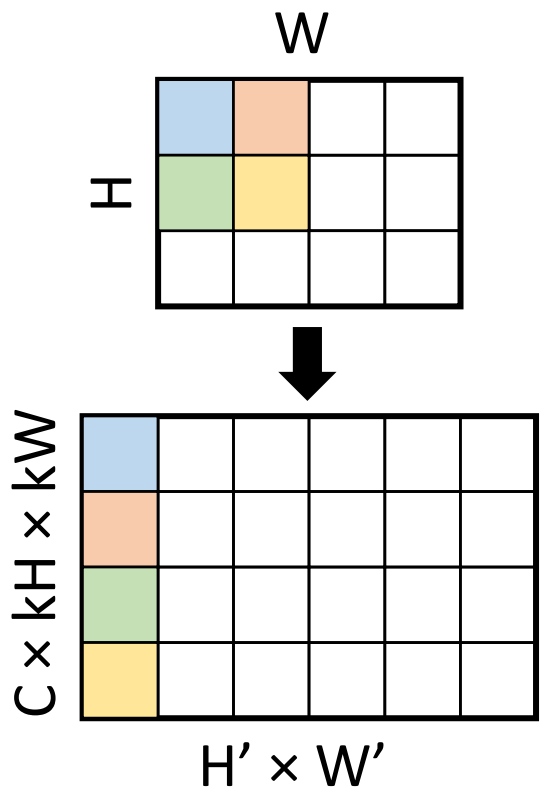
Implementation Convolution-like Ops

- 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”).



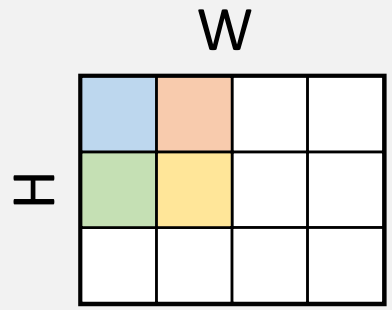
Using `F.unfold`

- 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

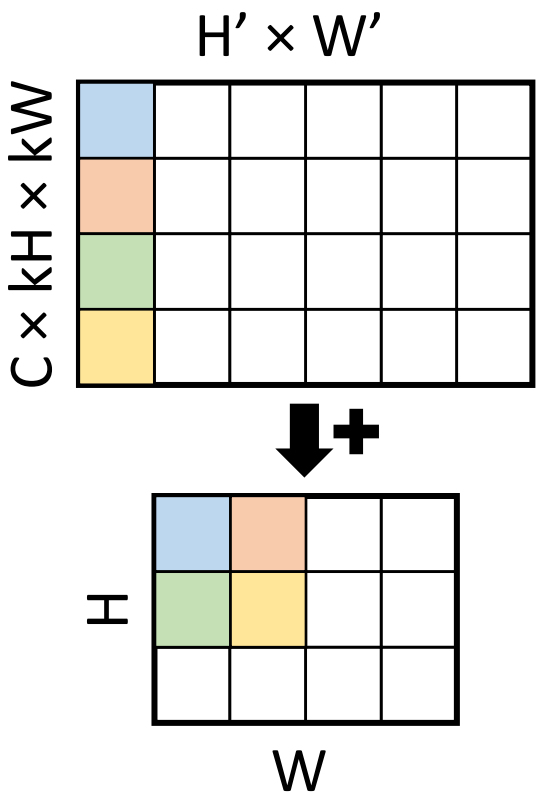
```
image = torch.rand(size=(2, 3, 3, 4))

# 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`?

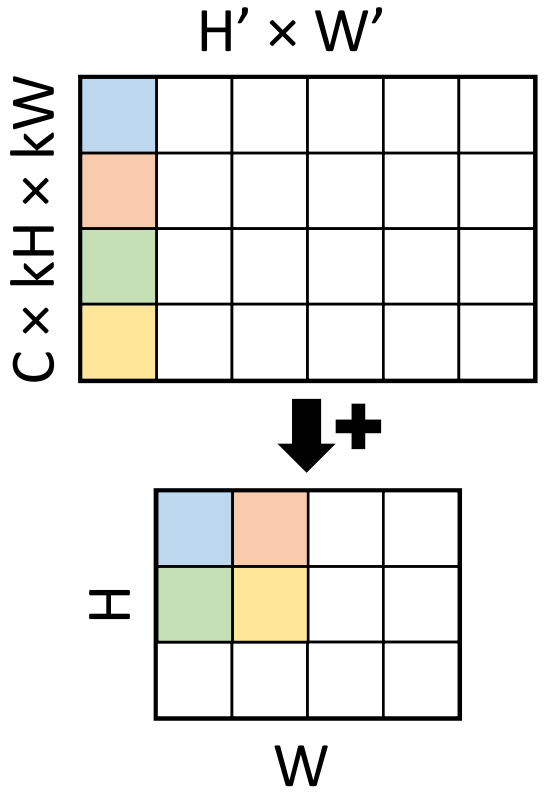
Using `F.fold`

- 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`.

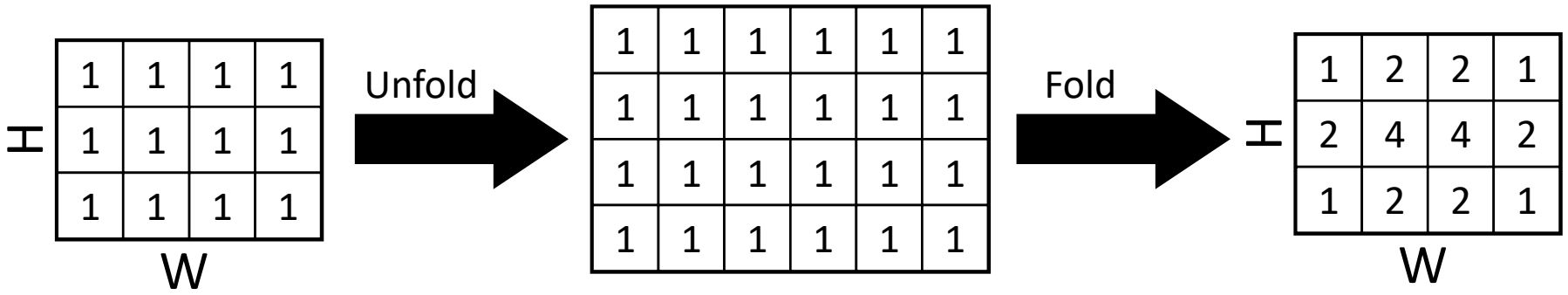


Example

```
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[0,0]`?
- What is the value in `output[1,1]`?



Topics

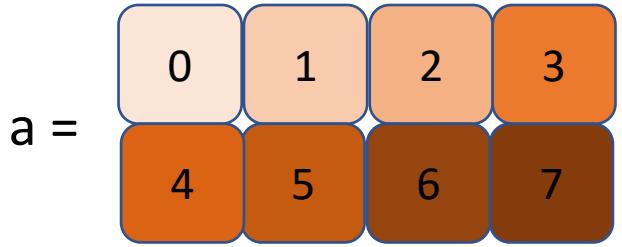
- Convolution-like operations
- **Tensors in memory**
- Data loading
- Hooks
- Training vs Inference

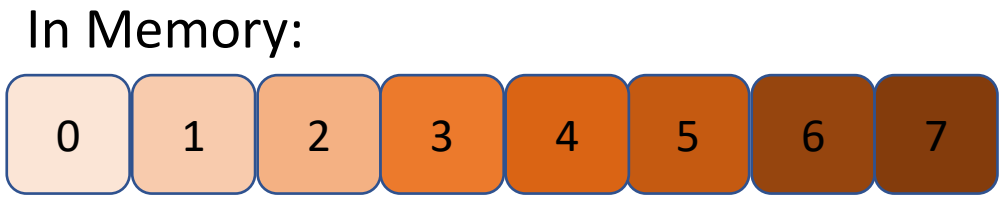
- Reproducibility
- Saving & Loading models
- External Tools and libraries
  - Using pre-trained models
  - Monitoring
  - Data augmentations
  - PyTorch Lightning

How are tensors kept in memory?

- Tensors are saved as 1D-arrays in memory.
- A tensor is **contiguous** if
  - It is saved in a single non-broken 1D array
  - is it saved in the correct order

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

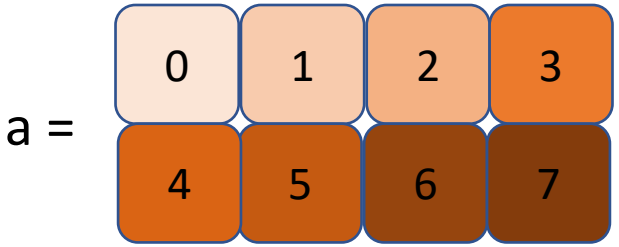




Tensor.view()

- We can change the shape of the tensor
  - `Tensor.view()`
  - Doesn't change real data (i.e. *O(1)*)

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

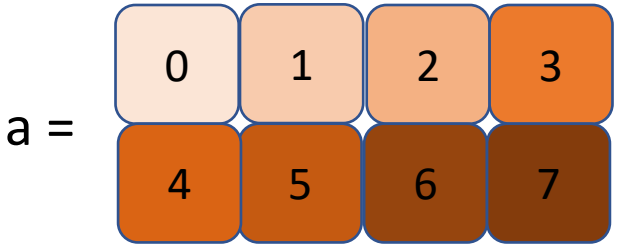


In Memory:

Tensor.transpose()

- We can switch tensor dimensions
  - `Tensor.transpose()`
  - `Tensor.permute()`
  - Doesn't change real data
  - Breaks contiguoancy

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



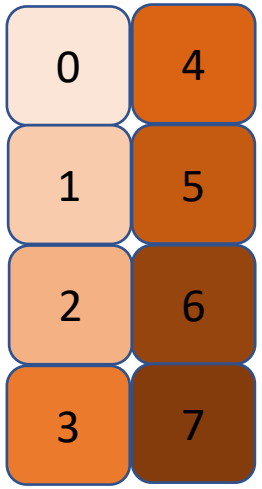
In Memory:

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
```

a =



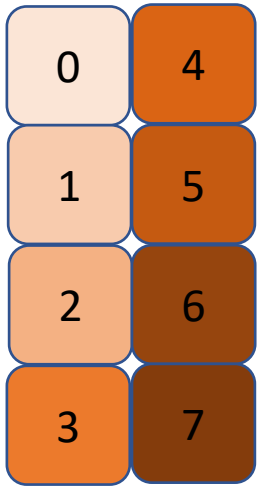
In Memory:

Tensor.reshape()

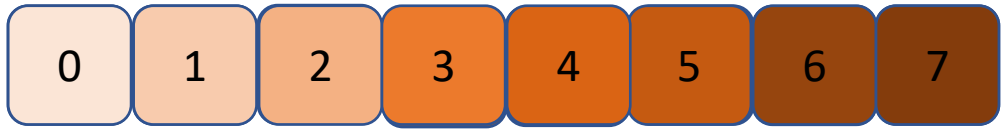
- `Tensor.reshape()` solves this by copying the tensor

```
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 `__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, 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“, … “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 loss

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 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

```python
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
```

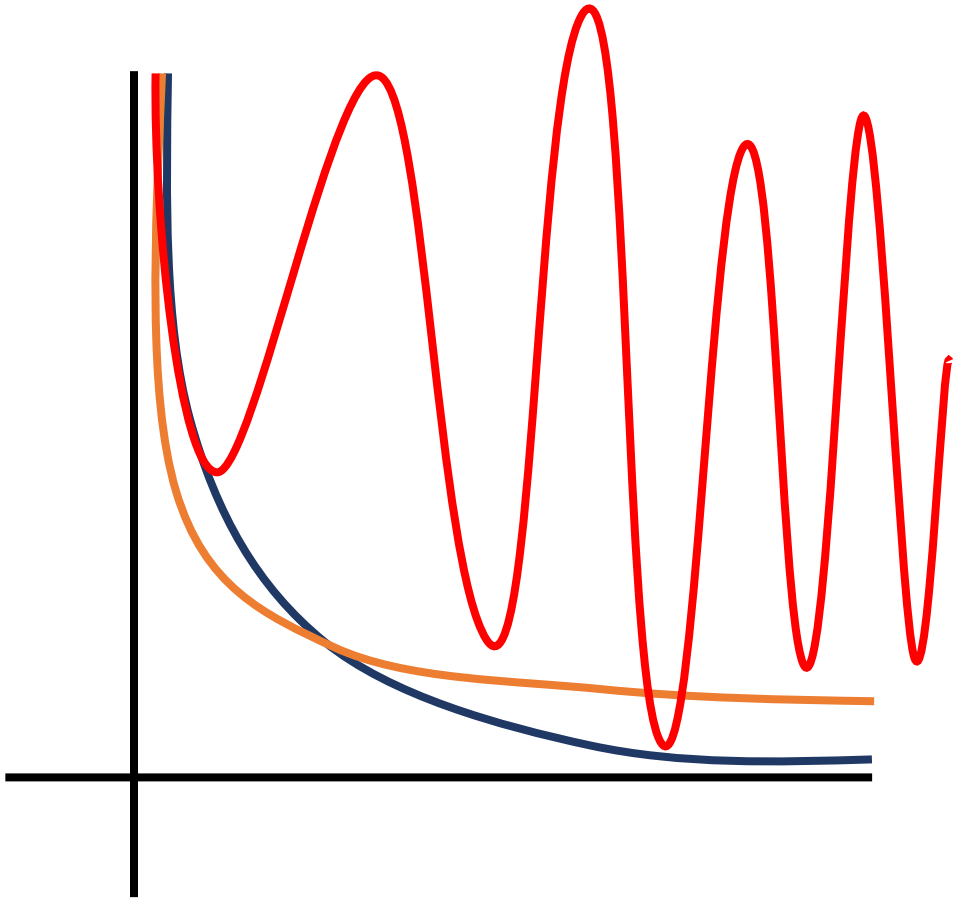
```python
def inference_loop(dataloader, model, criterion):
  size = len(dataloader.dataset)
  model.eval()
  running_loss, running_corrects = 0, 0

  # 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



Run 1

Run 2

Run 3

Limiting randomness

```
# Set random seeds
seed = 1337
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(seed)

# Use deterministic algorithms only
torch.use_deterministic_algorithms(True)

# Use deterministric 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

```
# 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,
    worker_init_fn=seed_worker, generator=g
)
```

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

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)
```

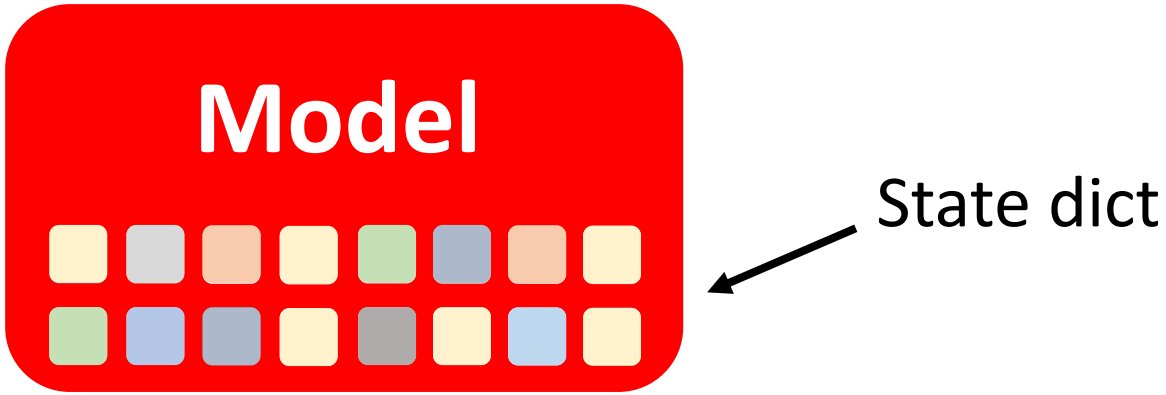
Pros:

Simple

Cons:

Can’t change the model class

Can’t continue training



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)

# load the state dict to a model
model.load_state_dict(sd)
```

Saving & Loading Models

```
# Initialize optimizer
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

```
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

Don’t Reinvent the Wheel!

Use Existing Tools!

Pretrained Models

PyTorch HUB

timm

Hugging Face

TorchVision

Torchvision

```
import torchvision

dir(torchvision.models)
# printed: ['alextnet', 'convnext', ...]
# ~100 models

# load model
model = torchvision.models.resnet18(pretrained=True)
model.eval()

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

# output.shape is 1x1000
```

Timm

```
!pip install timm
import timm
import torch

# list of available models in PyTorch repo.
timm.list_models(pretrained=True)

# printed: ['adv_inception_v3', 'cait_m36_384', 'cait_m48_448', ...]
# ~450 models

# load model
model = timm.create_model('resnet18', pretrained=True)
model.eval()

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

# output.shape is 1x1000
```

Torch Hub

```
import torch

# list of available models in pyTorch's repo.
torch.hub.list('pytorch/vision')

# printed: ['alexnet', 'deeplabv3_mobilenet_v3_large',
'deeplabv3_resnet101', ...]

# load model
model = torch.hub.load('pytorch/vision', 'resnet18', pretrained=True)
model.eval()

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

# output.shape is 1x1000
```

Hugging Face

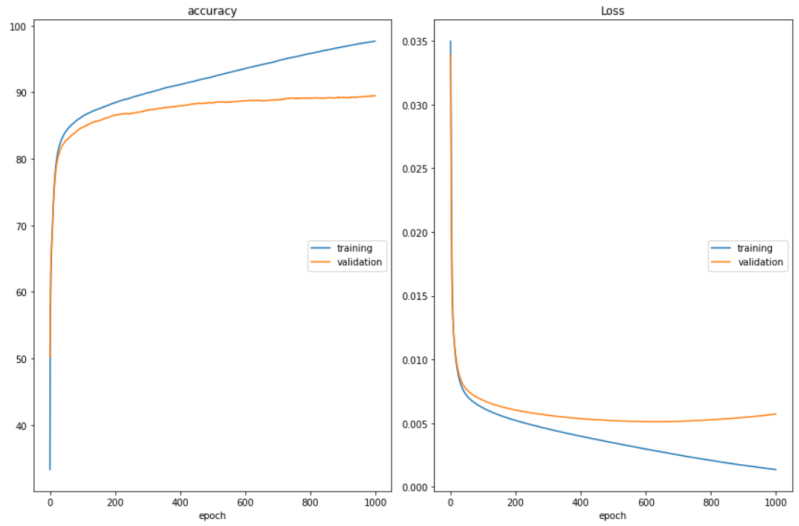
- The most updated model hub
  - 1-day resolution
- Models
- Demos
- Datasets

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

TensorBoard

`live loss plot`



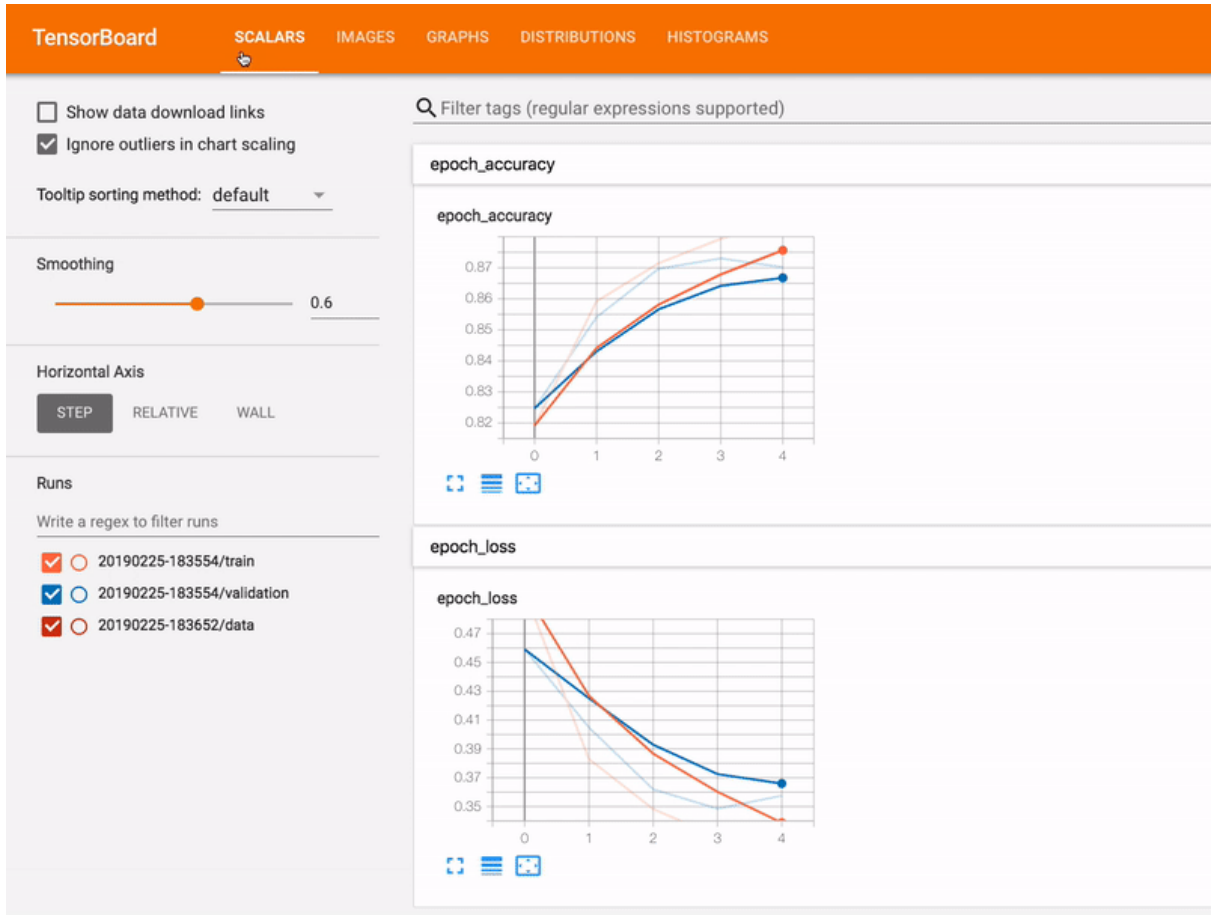


Image credit to Tensorflow

TensorBoard

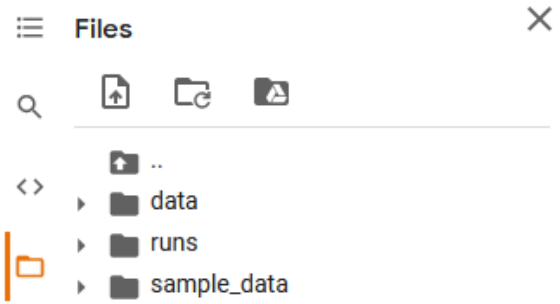
Preparations

```
# install tensorboard
!pip install tensorboard

%load_ext tensorboard

...

# run the user interface
%tensorboard --logdir .
```

TensorBoard

Summary Writer

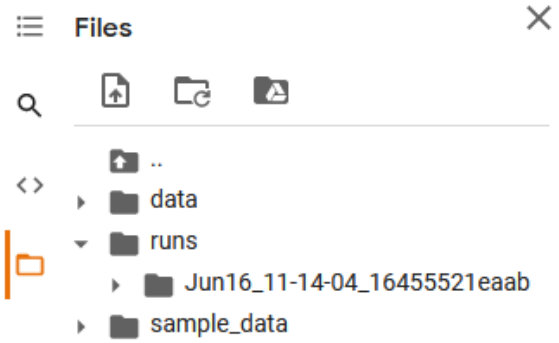
```
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter()
```



TensorBoard

Summary Writer

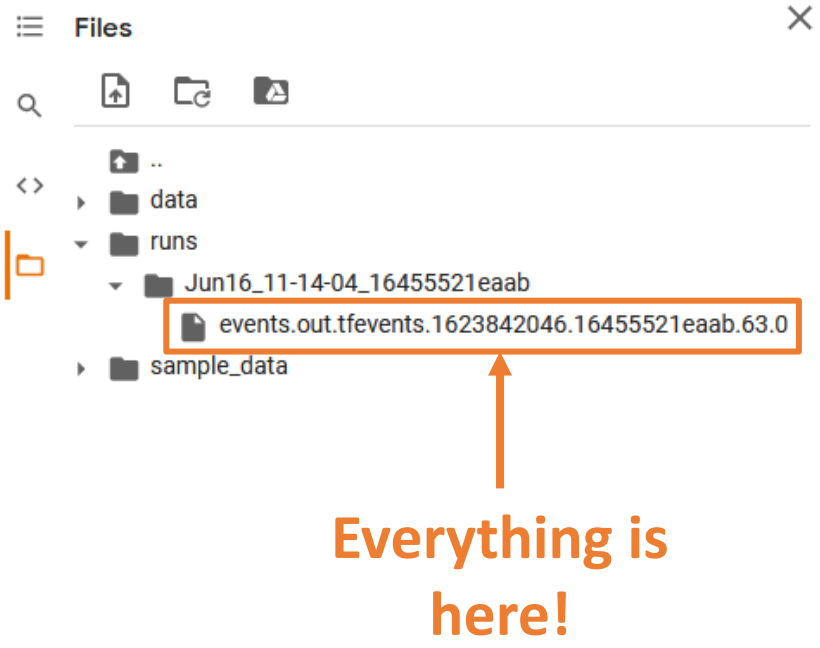
```
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter()
```



TensorBoard

Summary Writer

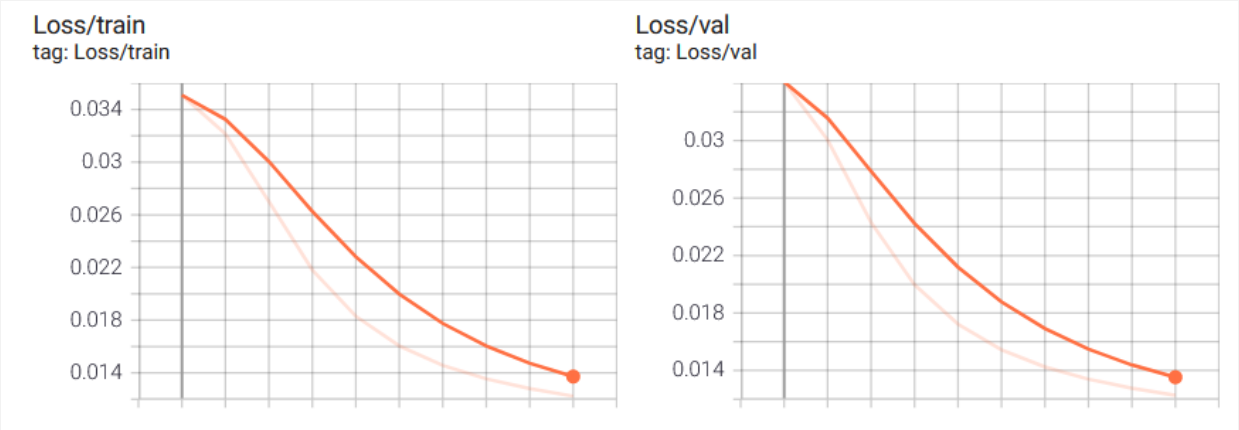
```
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter()
```



TensorBoard

Summary Writer

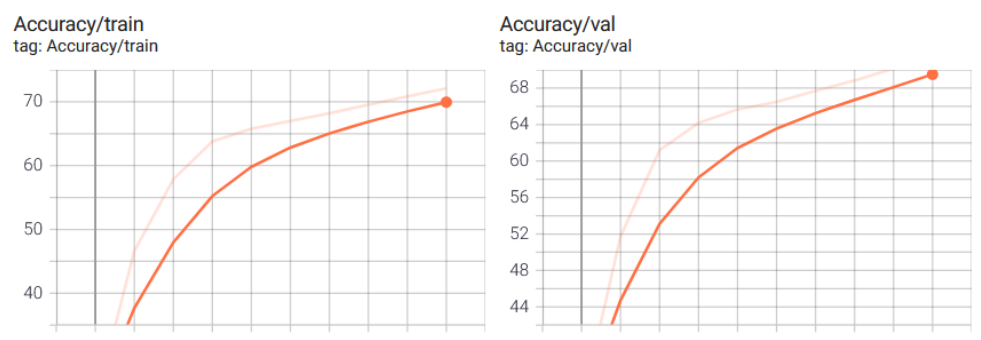
```
# train
for epoch in range(EPOCHS):
   ...
   writer.add_scalar("loss/train", loss, epoch)
   writer.add_scalar("loss/val", val_loss, epoch)
   ...
```

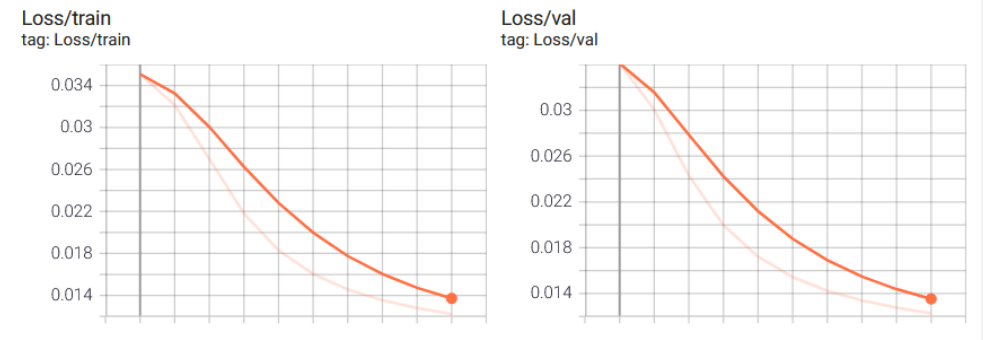


TensorBoard

Summary Writer

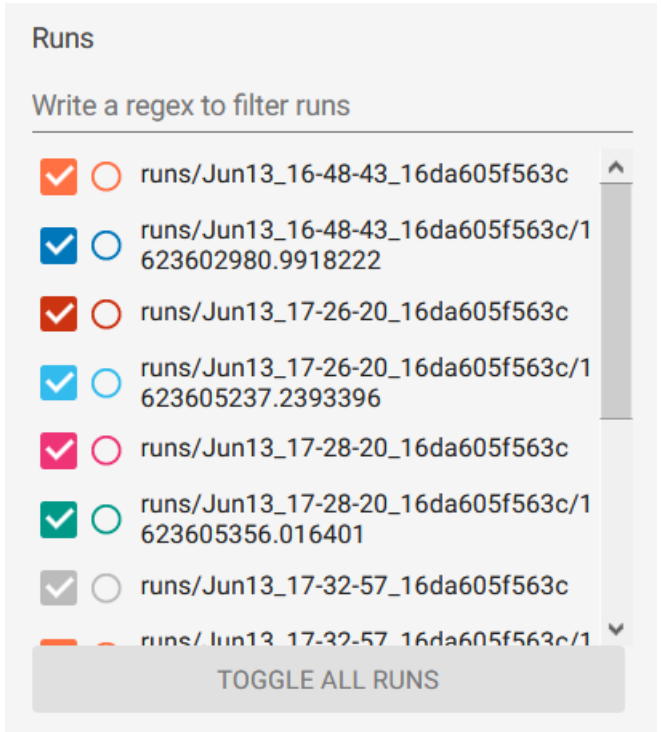
```
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter()

# train
for epoch in range(EPOCHS):
  ...
  writer.add_scalar("loss/train", loss, epoch)
  writer.add_scalar("loss/val", val_loss, epoch)
  writer.add_scalar("accuracy/train", acc, epoch)
  writer.add_scalar("accuracy/val", val_acc, epoch)
```

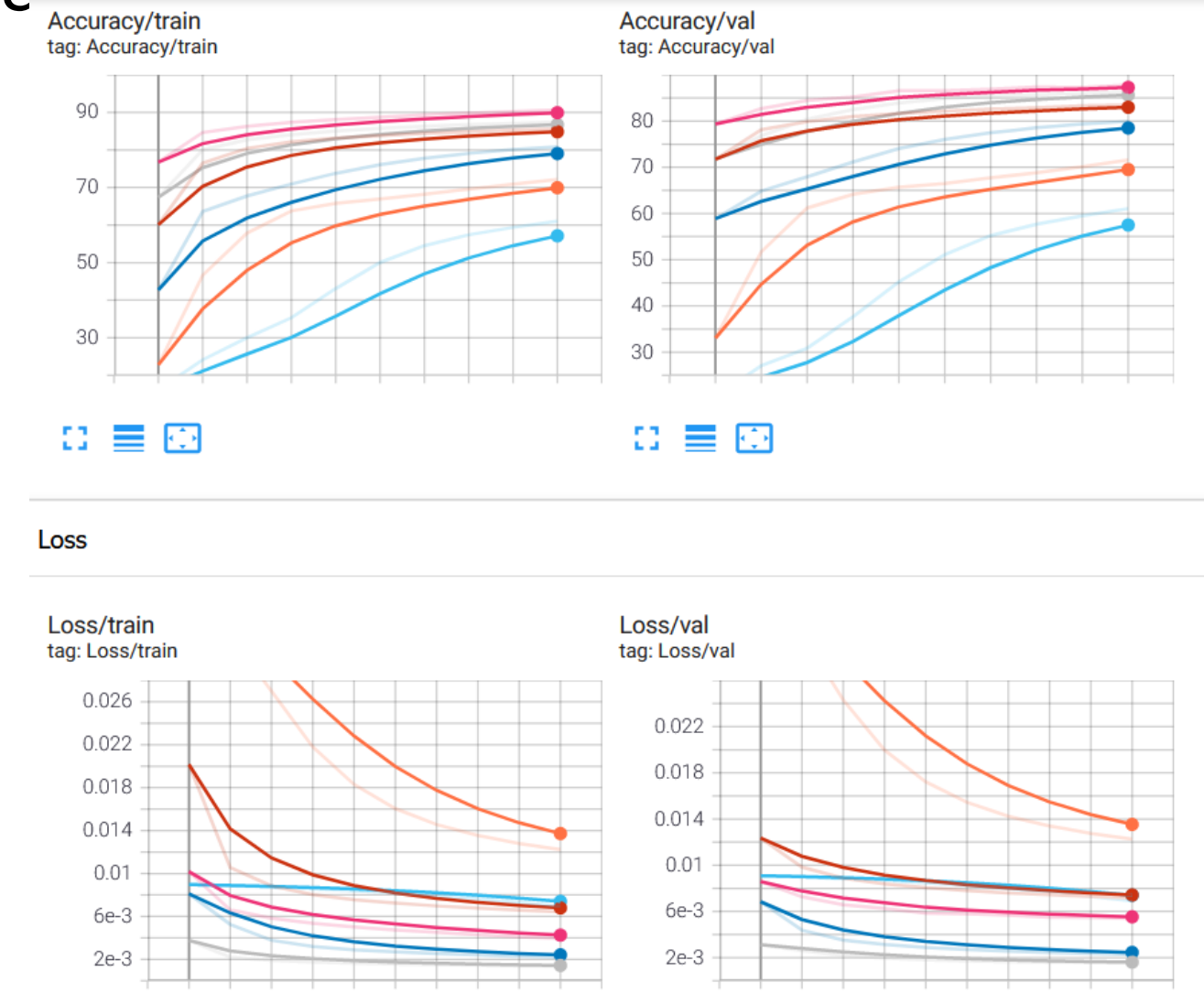




TensorBoard

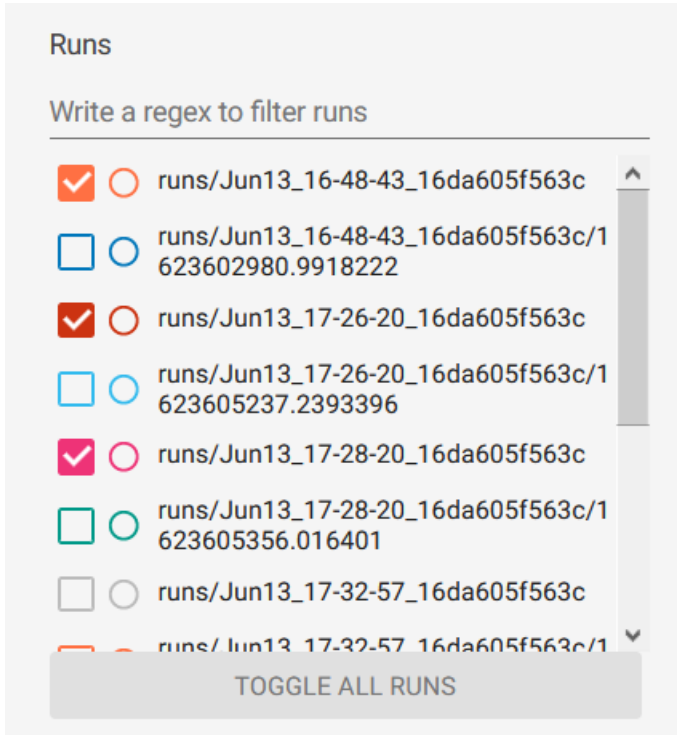
Summary Writer

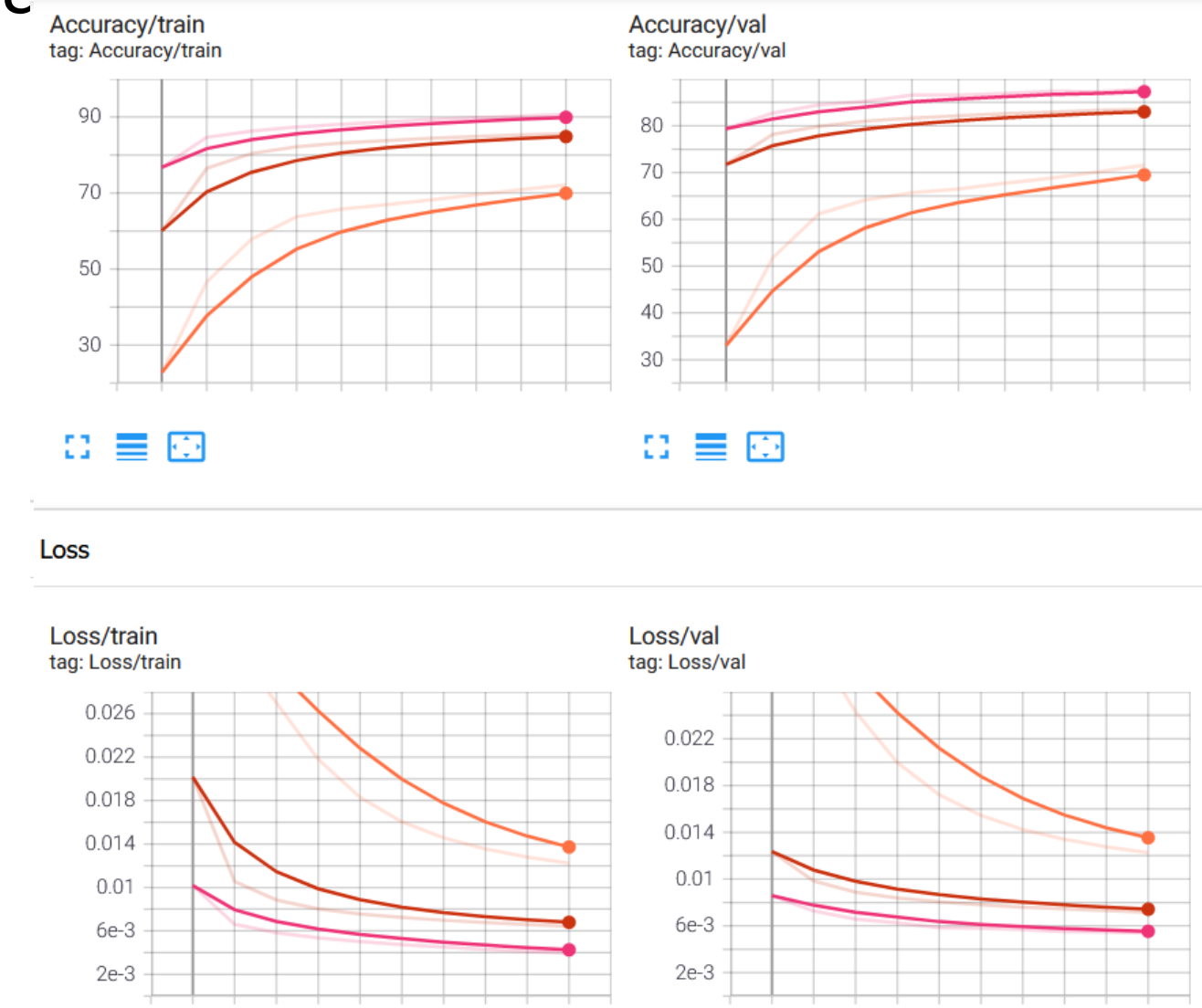




TensorBoard

Summary Writer





TensorBoard

Log Images

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

# create an image from all the images
grid = torchvision.utils.make_grid(X)

# add to summary
writer.add_image("images", grid)
```

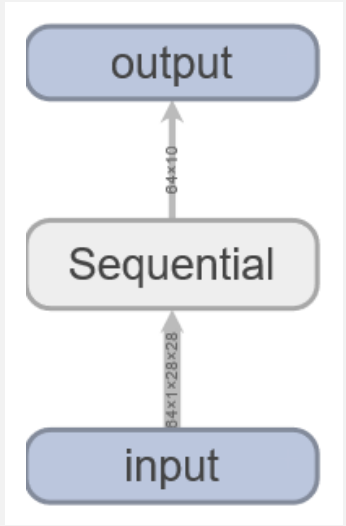
TensorBoard

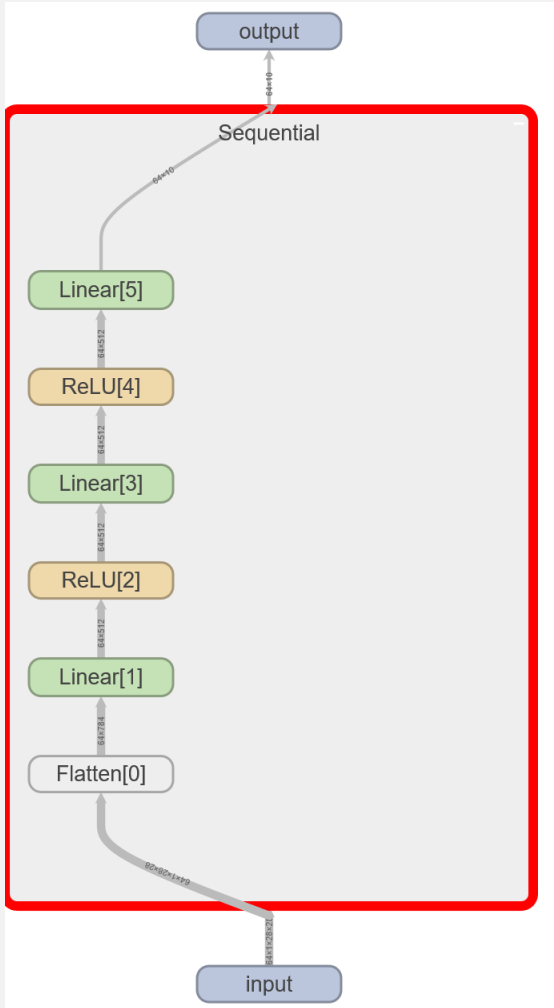
Log Figures

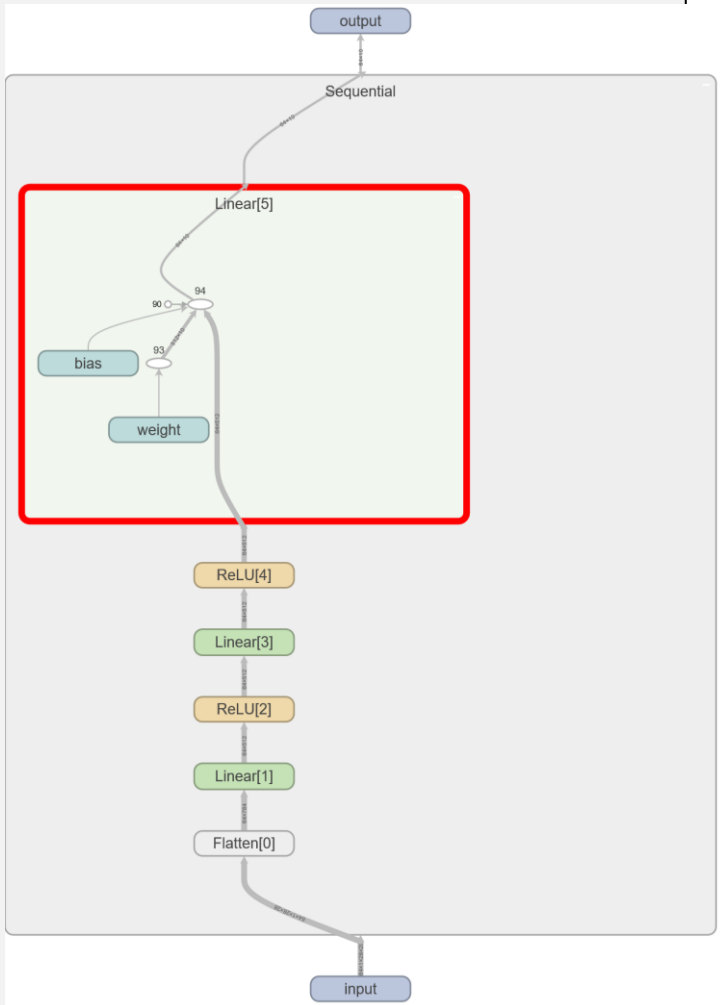
```
# create matplotlib figure
figure = ...

# add to summary
writer.add_figure("batch_example",
                  figure)
```

TensorBoard Graphs

```
writer.add_graph(model, X)
```



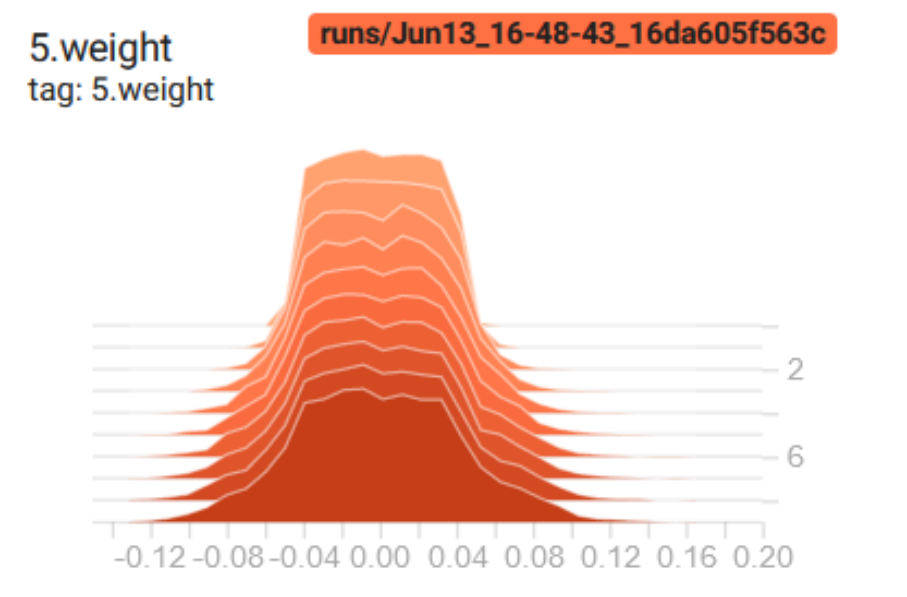


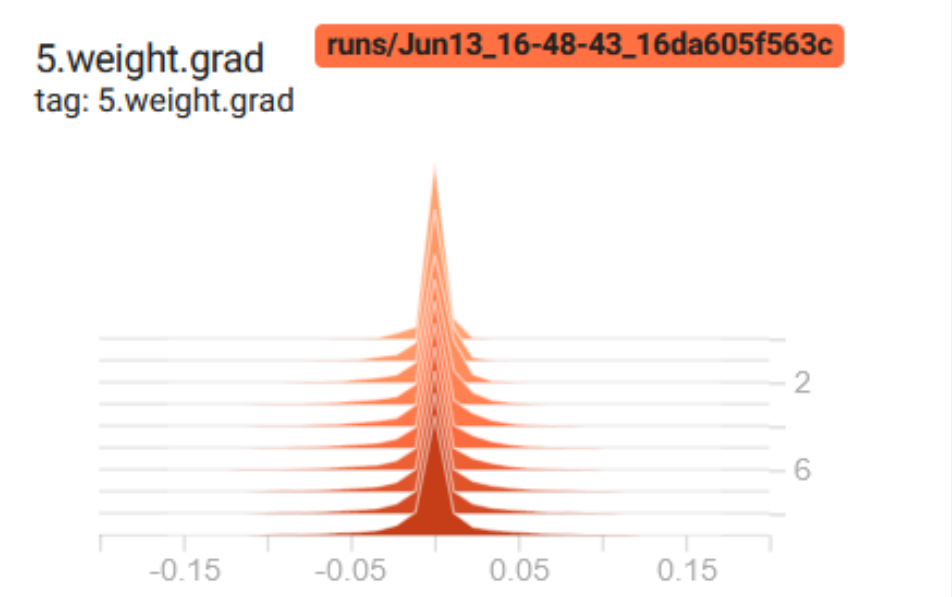


TensorBoard

Histograms

```
for name, weight in model.named_parameters():
  writer.add_histogram(name, weight, epoch)
  writer.add_histogram(f'{name}.grad', weight.grad, epoch)
```





TensorBoard

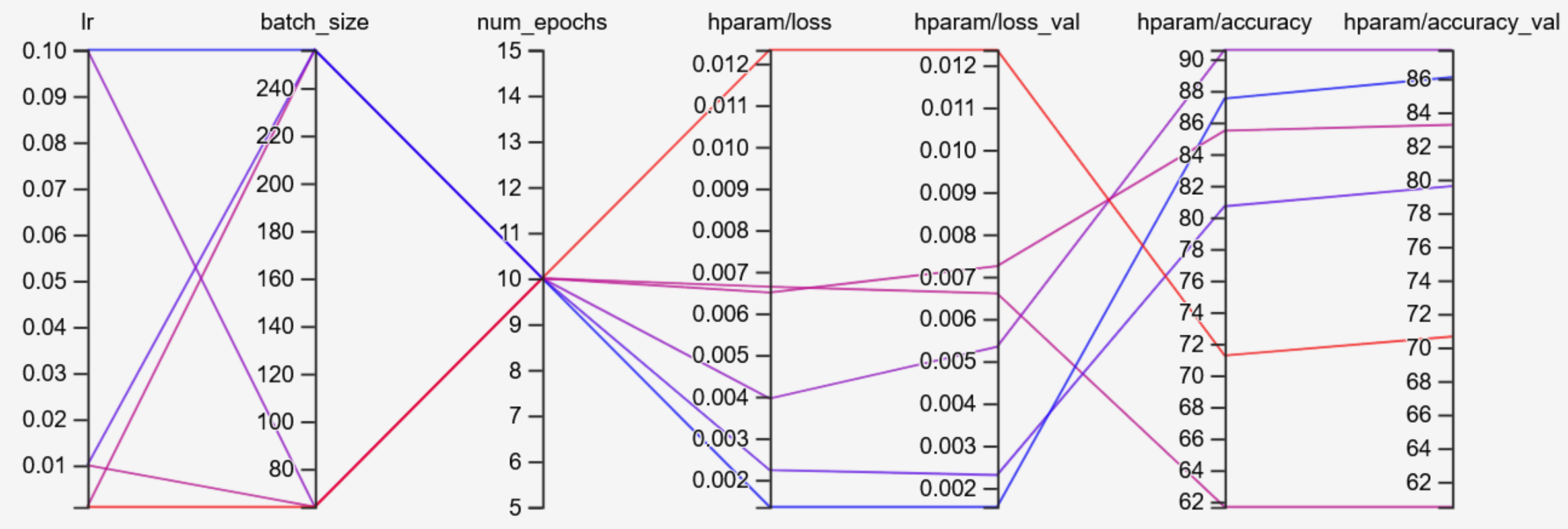
Hyper-parameters

```
writer.add_hparams({'lr': LR, 'batch_size': B,
                    'num_epochs': EPOCHS},
                   {'hparam/loss': train_loss,
                    'hparam/loss_val': val_loss,
                    'hparam/accuracy': train_acc,
                    'hparam/accuracy_val': val_acc})
```

Trial ID	Show Metrics	lr	batch_size	num_epochs	hparam/loss	hparam/loss_val	hparam/accuracy	hparam/accuracy_val
runs/Jun13_19-...	☐	0.0010000	256.00	10.000	0.0066344	0.0065976	61.670	60.500
runs/Jun13_19-...	☐	0.010000	256.00	10.000	0.0022308	0.0023018	80.687	79.600
runs/Jun13_19-...	☐	0.10000	256.00	10.000	0.0013479	0.0015497	87.502	86.080
runs/Jun13_19-...	☐	0.10000	64.000	10.000	0.0039495	0.0053355	90.593	87.720
runs/Jun13_19-...	☐	0.010000	64.000	10.000	0.0064962	0.0072452	85.473	83.250
runs/Jun13_19-...	☐	0.0010000	64.000	10.000	0.012328	0.012364	71.242	70.640

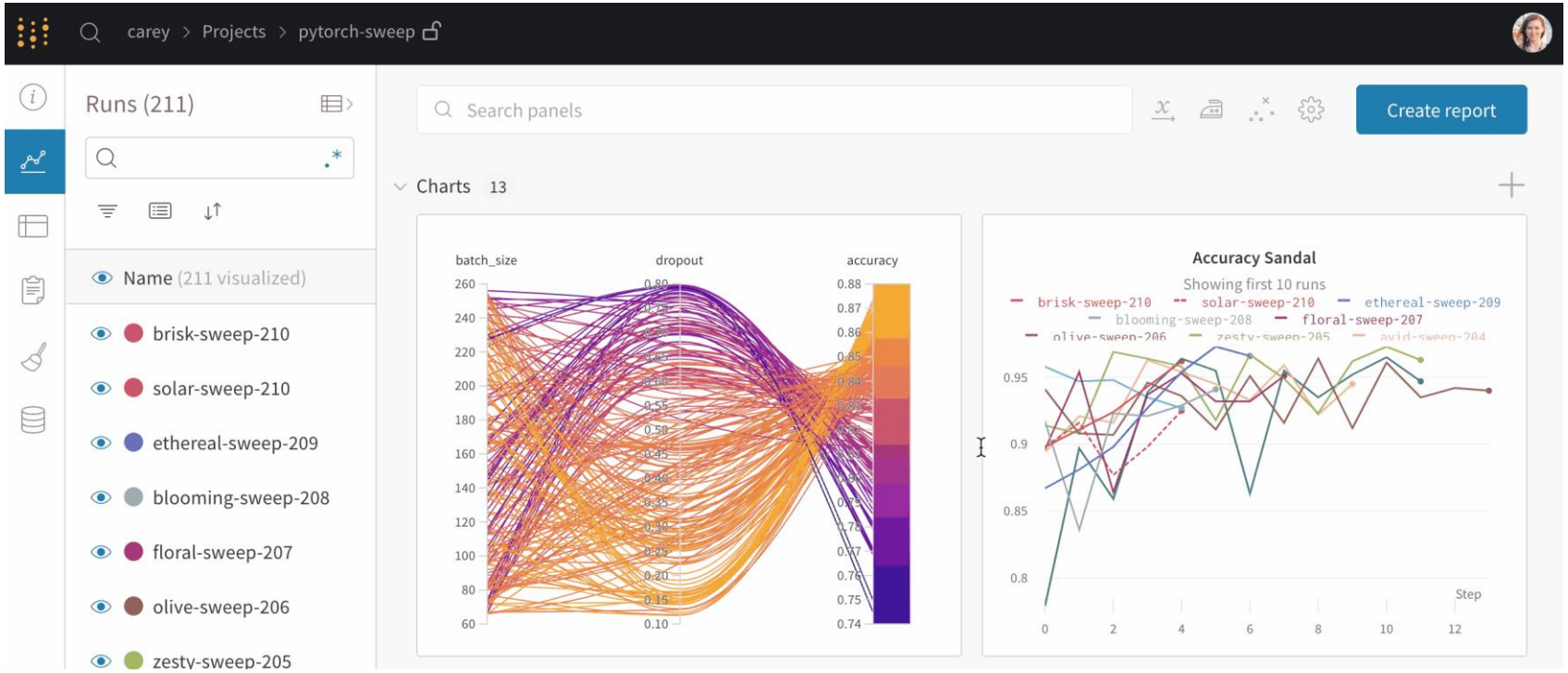
TensorBoard

Hyper-parameters



Weights & Biases

- More features (**sweeps**, external access, working in groups, etc.)
- Requires registration, limited memory



Topics

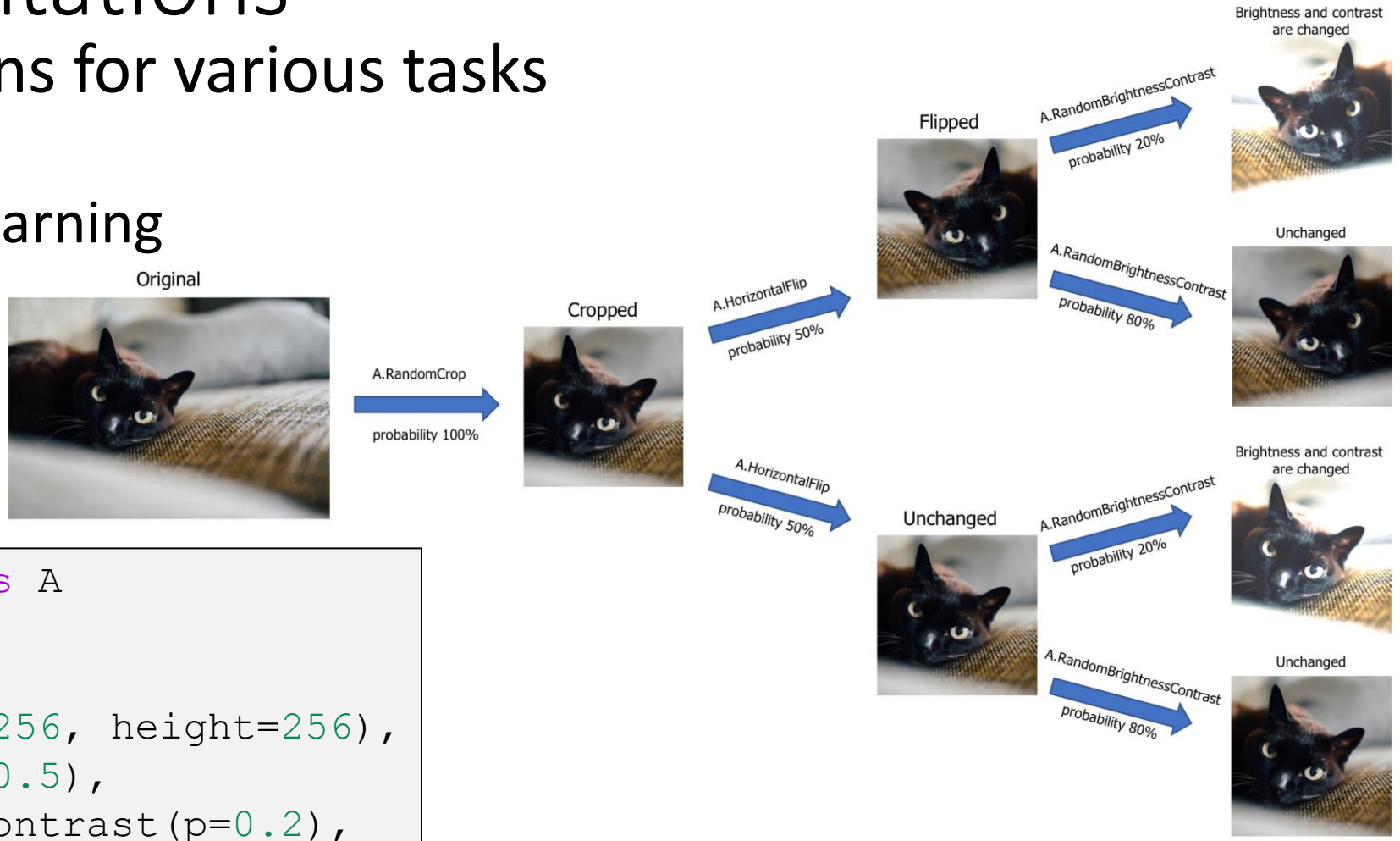
- Convolution-like operations
- Tensors in memory
- Data loading
- Hooks
- Training vs Inference
- Reproducibility
- Saving & Loading models
- **External Tools and libraries**
  - Using pre-trained models
  - Monitoring
  - **Data augmentations**
  - PyTorch Lightning

Don’t Reinvent the Wheel!

Use Existing Tools!

Albumentations

- Data Augmentations for various tasks
  - Classification / Representation Learning

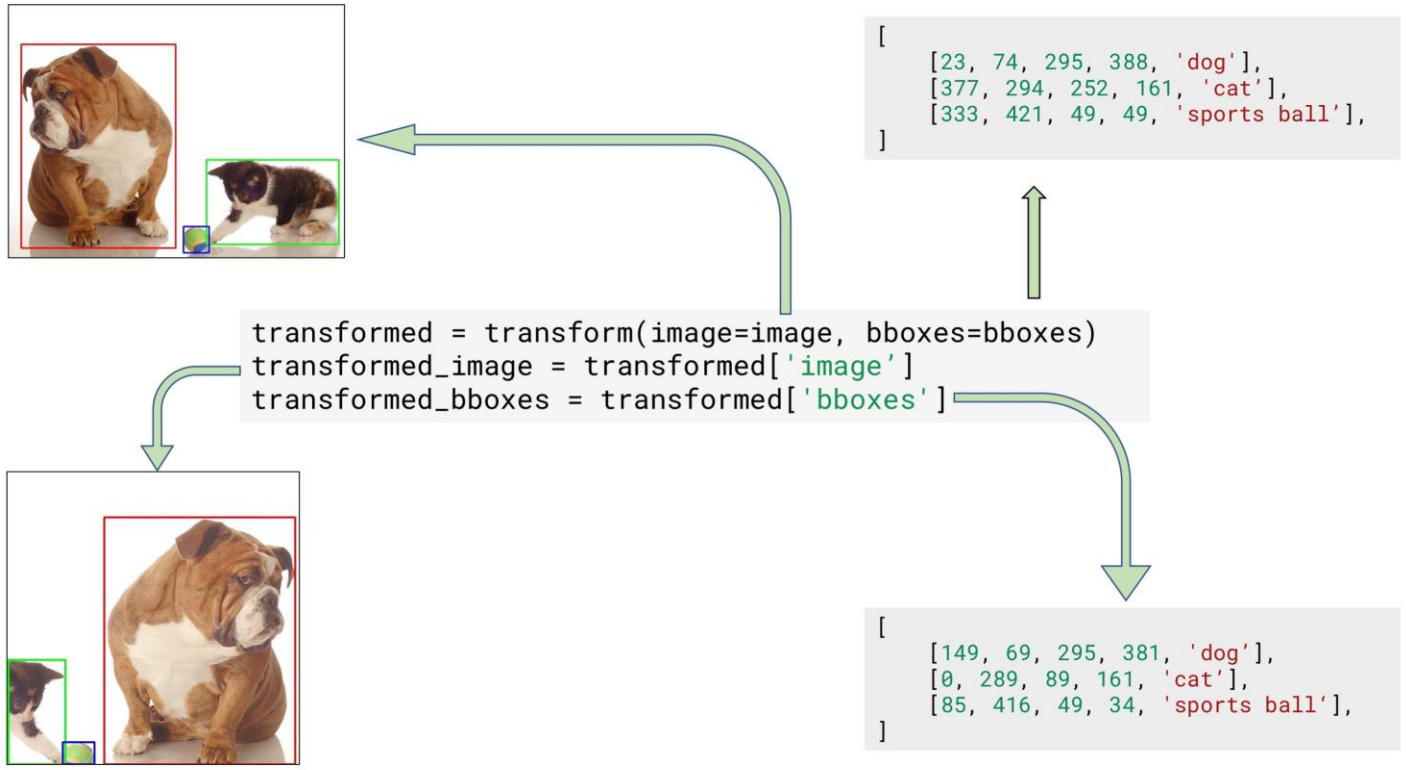



```python
import albumentations as A

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

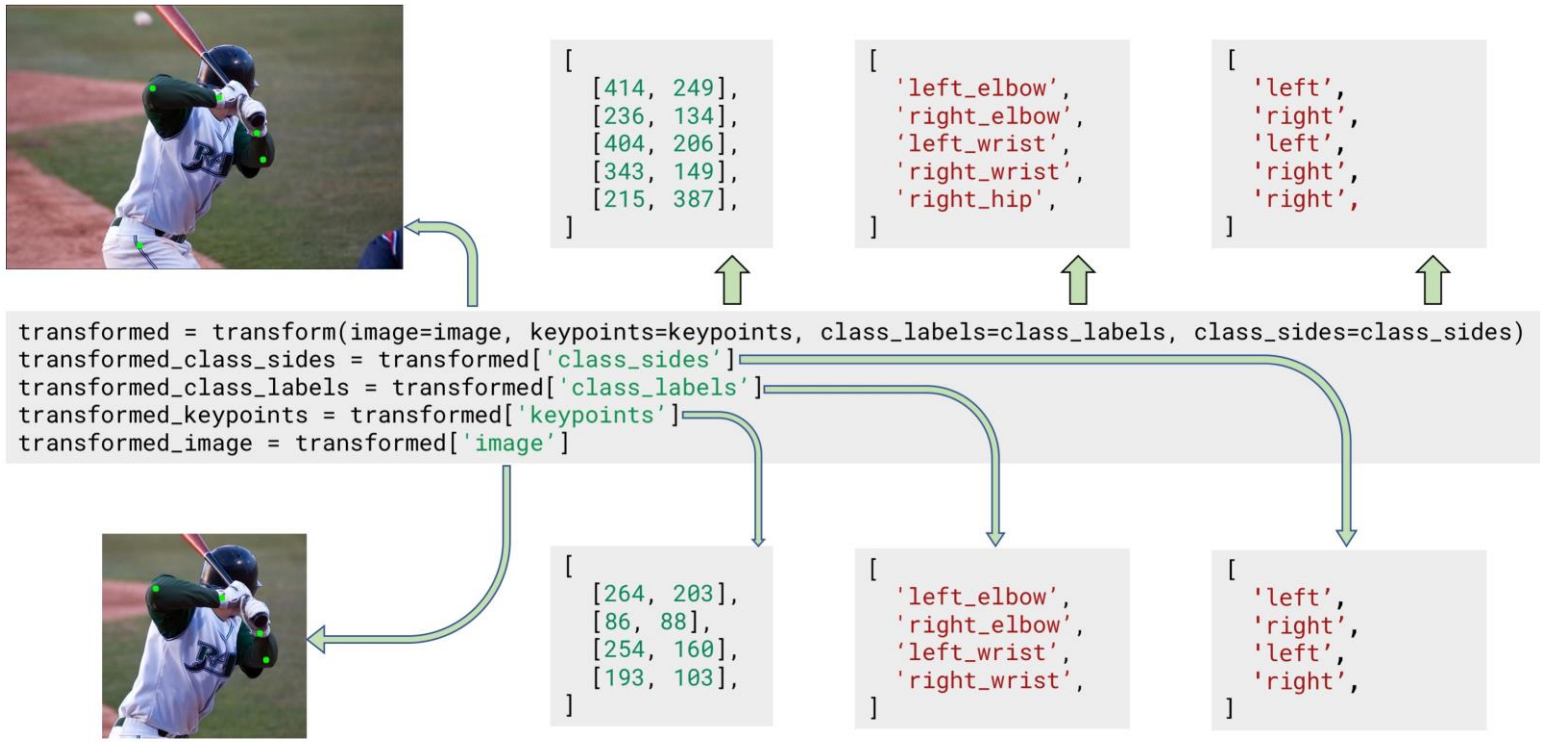
- Data Augmentations for various tasks
  - Classification / Representation Learning
  - Object Detection



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

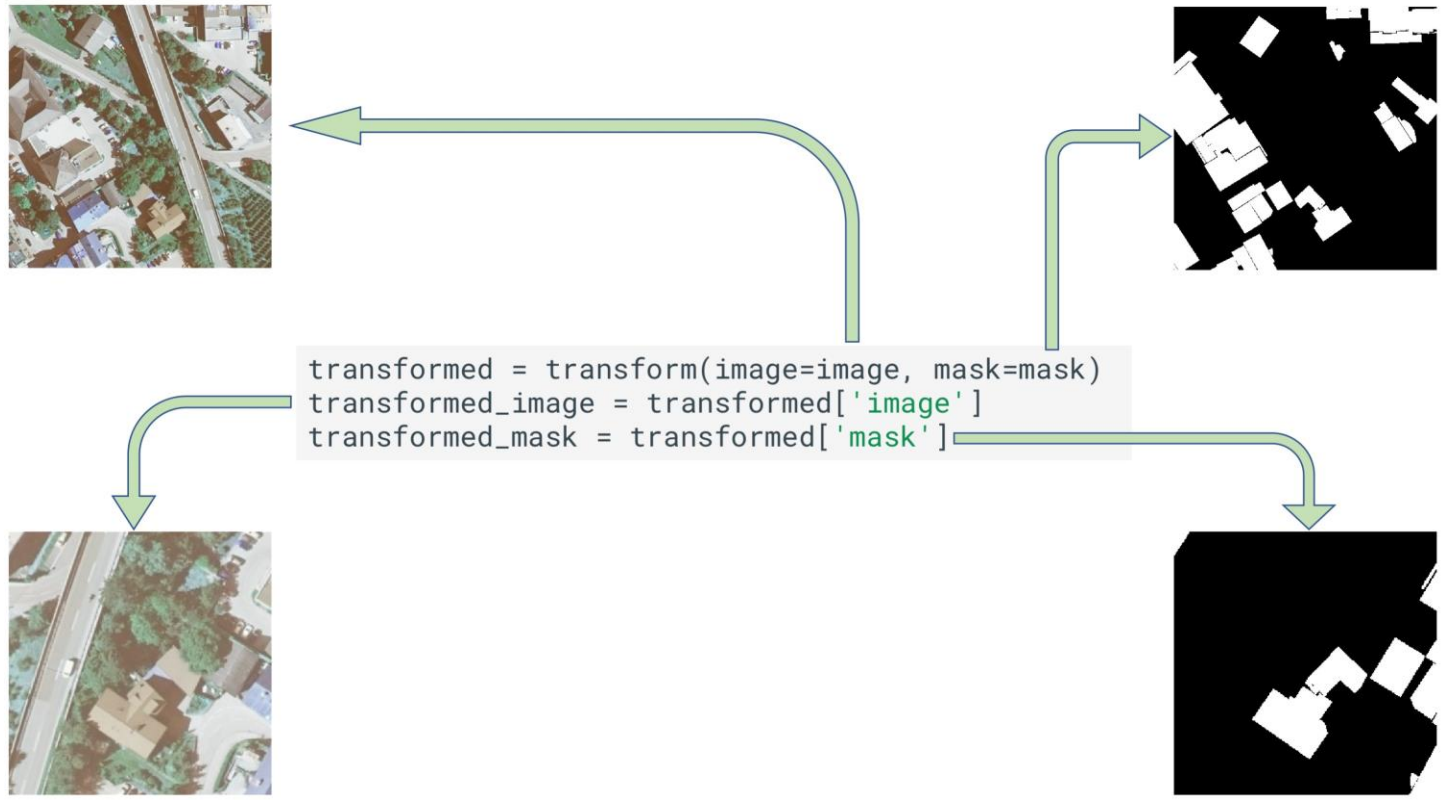


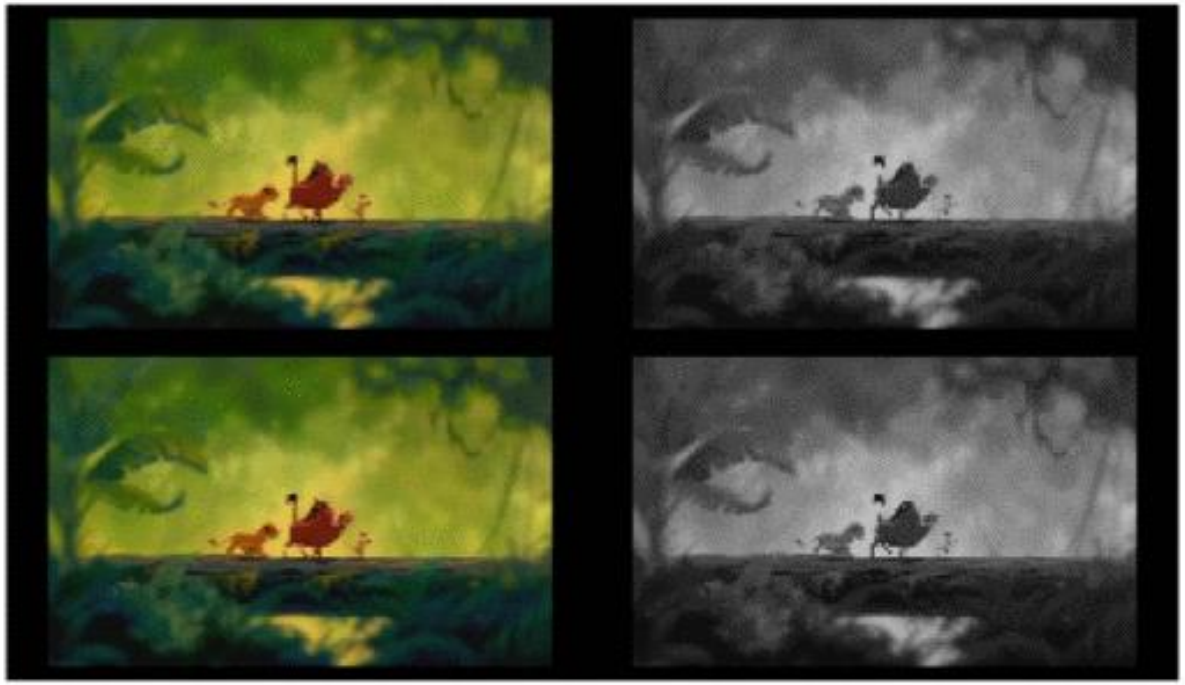
Image credit to albumentations tutorial:
https://albumentations.ai/docs/getting_started/mask_augmentation/

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))
```

Image from kornia tutorial:
https://kornia-tutorials.readthedocs.io/en/latest/canny.html

kornia

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

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

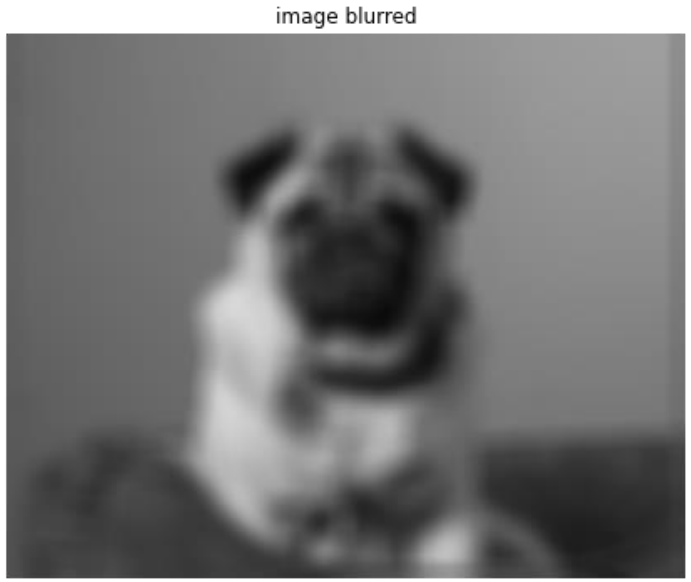
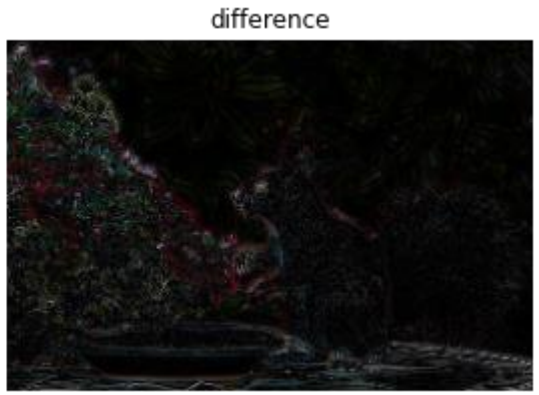


Image from kornia tutorial:
https://kornia-tutorials.readthedocs.io/en/latest/gaussian_blur.html

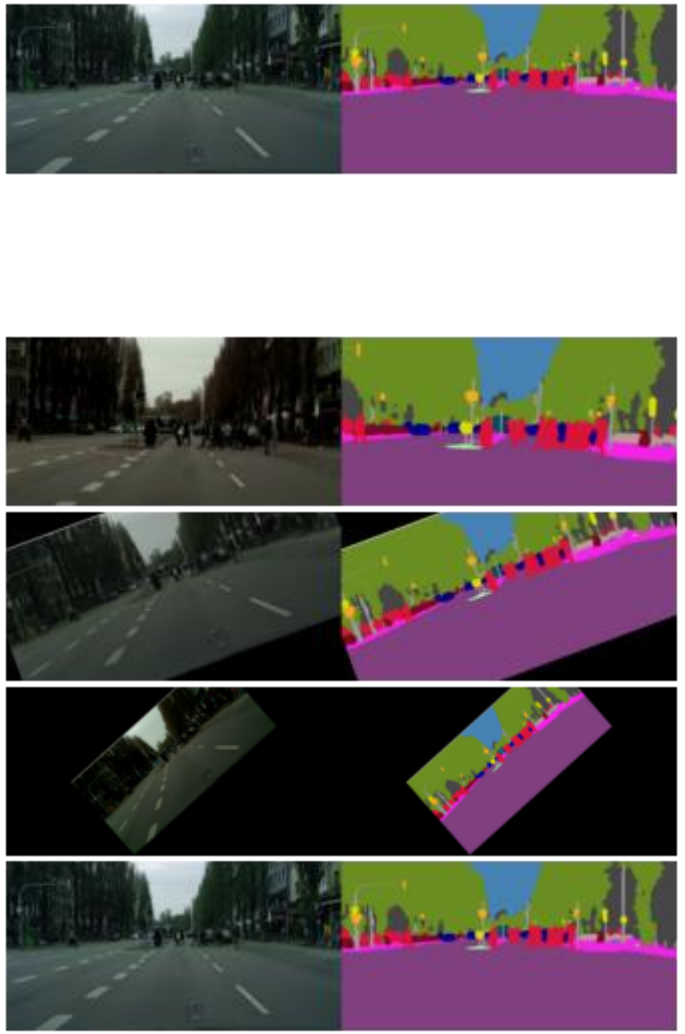
kornia

```
# 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()
```



kornia

```
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
```



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

Simple training

```
# Define logger and trainer
tb_logger = pl.logger.TensorBoardLogger()
trainer = pl.Trainer(max_epochs=30, gpus=8,
                     logger=tb_logger)

# Train model
trainer.fit(model, dataloader) # No training boilerplate code
```

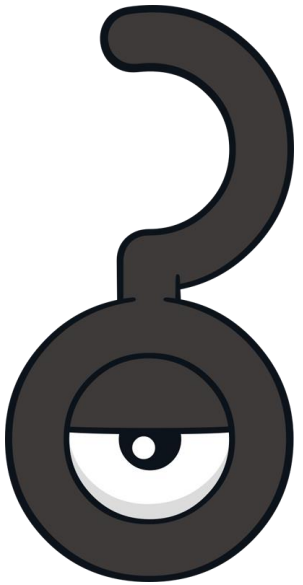
PyTorch Lightning

- Many additional features
  - Learning rate scheduling
  - Test and Validation split
  - Automatic (and heavily customizable) checkpoints
  - Simple Multi GPU usage
  - And many more

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