PyTorch CHEAT SHEET

1 Load data 2 Define model 3 Train model 4 Evaluate model

General

PyTorch is a open source machine learning framework. It uses **torch.Tensor** – multi-dimensional matrices – to process. A core feature of neural networks in PyTorch is the autograd package, which provides automatic derivative calculations for all operations on tensors.

import torch	Root package
import torch.nn as nn	Neural networks
from torchvision import datasets, models, transforms	Popular image datasets, architectures & transforms
import torch.nn.functional as F	Collection of layers, activations & more

torch.randn(*size)	Create random tensor
torch.Tensor(L)	Create tensor from list
tnsr.view(a,b, ...)	Reshape tensor to size (a, b, ...)
requires_grad=True	tracks computation history for derivative calculations

Layers

nn.Linear(m, n): Fully Connected layer (or dense layer) from m to n neurons

nn.ConvXd(m, n, s): X-dimensional convolutional layer from m to n channels with kernel size s; X ∊ {1, 2, 3}

nn.Flatten(): Flattens a contiguous range of dimensions into a tensor

nn.MaxPoolXd(s): X-dimensional pooling layer with kernel size s; X ∊ {1, 2, 3}

nn.Dropout(p=0.5): Randomly sets input elements to zero during training to prevent overfitting

nn.BatchNormXd(n): Normalizes a X-dimensional input batch with n features; X ∊ {1, 2, 3}

nn.Embedding(m, n): Lookup table to map dictionary of size m to embedding vector of size n

nn.RNN/LSTM/GRU: Recurrent networks connect neurons of one layer with neurons of the same or a previous layer

torch.nn offers a bunch of other building blocks.
A list of state-of-the-art architectures can be found at https://paperswithcode.com/sota.

Load data

A dataset is represented by a class that inherits from **Dataset** (resembles a list of tuples of the form (features, label)).

DataLoader allows to load a dataset without caring about its structure.

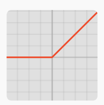
Usually the dataset is split into training (e.g. 80%) and test data (e.g. 20%).

```
from torch.utils.data
 import Dataset, TensorDataset,
        DataLoader, random_split

train_data, test_data =
 random_split(
    TensorDataset(inps, tgts),
    [train_size,test_size]
 )

train_loader =
       DataLoader(
          dataset=train_data,
          batch_size=16,
          shuffle=True)
```

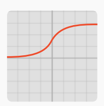
Activation functions

Common activation functions include **ReLU**, **Sigmoid** and **Tanh**, but there are other activation functions as well.

nn.ReLU() *creates a* nn.Module *for example to be used in Sequential models.* **F.relu()** *ist just a call of the ReLU function e.g. to be used in the* forward *method.*



nn.ReLU() or **F.relu()**
Output between 0 and ∞, most frequently used activation function



nn.Sigmoid() or **F.sigmoid()**
Output between 0 and 1, often used for predicting probabilities

nn.Tanh() or **F.tanh()**
Output between -1 and 1, often used for classification with two classes

Define model

There are several ways to define a neural network in PyTorch, e.g. with **nn.Sequential** (a), as a class (b) or using a combination of both.

```
model = nn.Sequential(
  nn.Conv2D(■,■,■)
  nn.ReLU()
  nn.MaxPool2D(■)
  nn.Flatten()
  nn.Linear(■,■)
)
```
a

```
class Net(nn.Module):
  def __init__():
    super(Net, self).__init__()

    self.conv
      = nn.Conv2D(■,■,■)
    self.pool
      = nn.MaxPool2D(■)
    self.fc = nn.Linear(■,■)

  def forward(self, x):
    x = self.pool(
        F.relu(self.conv(x))
    )
    x = x.view(-1,■)
    x = self.fc(x)
    return x
model = Net()
```
b

Save/Load model

model = torch.load('PATH')	Load model
torch.save(model, 'PATH')	Save model

It is common practice to save only the model parameters, not the whole model using model.state_dict()

```
torch.save(model.state_dict(), 'params.ckpt')
model.load_state_dict(
               torch.load('params.ckpt'))
```

GPU Training

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

If a GPU with CUDA support is available, computations are sent to the GPU with ID 0 using **model.to(device)** or **inputs, labels = data[0].to(device), data[1].to(device).**

Train model

LOSS FUNCTIONS

PyTorch already offers a bunch of different loss fuctions, e.g.:

nn.L1Loss	Mean absolute error
nn.MSELoss	Mean squared error (L2Loss)
nn.CrossEntropyLoss	Cross entropy, e.g. for single-label classification or unbalanced training set
nn.BCELoss	Binary cross entropy, e.g. for multi-label classification or autoencoders

OPTIMIZATION (torch.optim)

Optimization algorithms are used to update weights and dynamically adapt the learning rate with gradient descent, e.g.:

optim.SGD	Stochastic gradient descent
optim.Adam	Adaptive moment estimation
optim.Adagrad	Adaptive gradient
optim.RMSProp	Root mean square prop

```
import torch.optim as optim

# Define loss function
loss_fn = nn.CrossEntropyLoss()

# Choose optimization method
optimizer = optim.SGD(model.parameters(),
                      lr=0.001, momentum=0.9)

# Loop over dataset multiple times (epochs)
for epoch in range(2):
    model.train() # activate training mode
    for i, data in enumerate(train_loader, 0):
        # data is a batch of [inputs, labels]
        inputs, labels = data

        # zero gradients
        optimizer.zero_grad()

        # calculate outputs
        outputs = model(inputs)
        # calculate loss & backpropagate error
        loss = loss_fn(outputs, labels)
        loss.backward()
        # update weights & learning rate
        optimizer.step()
```

Evaluate model

The evaluation examines whether the model provides satisfactory results on previously withheld data. Depending on the objective, different metrics are used, such as acurracy, precision, recall, F1, or BLEU.

model.eval()	Activates evaluation mode, some layers behave differently
torch.no_grad()	Prevents tracking history, reduces memory usage, speeds up calculations

```
correct = 0 # correctly classified
total   = 0 # classified in total

model.eval()
with torch.no_grad():
  for data in test_loader:
    inputs, labels = data
    outputs = model(inputs)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0) # batch size
    correct += (predicted==labels)
                   .sum().item()

print('Accuracy: %s' % (correct/total))
```

Stefan Seegerer, hi@stefanseegerer.de • Matthias Zürl, matthias.zuerl@fau.de • Last updated: 10/2021

Deep Learning with {torch} CHEAT SHEET

torch luz torchaudio torchvision

Intro

{torch} is based on Pytorch, a framework popular among deep learning researchers.

{torch}'s GPU acceleration allows to implement fast machine learning algorithms using its convenient interface, as well as a vast range of use cases, not only for deep learning, according to Its flexibility and its low level API.

It is part of an ecosystem of packages to interface with specific dataset like {torchaudio} for timeseries-like, {torchvision} for image-like, and {tabnet} for tabular data. It is complemented by {lux} for a higher-level programming interface

Define
- NN modules
- Instantiate a network

→ **Model params**
- Optimiser
- Loss
- Metrics

→ **Training loop**
model$train()
- Batch size
- Epochs
- Validation split

→ **Evaluation**
model$eval()
- Error on validation
- Plot

→ **Predict**
- classes
- probability

https://torch.mlverse.org/

https://mlverse.shinyapps.io/torch-tour/

INSTALLATION

The torch R package uses the C++ libtorch library. You can install the prerequisites directly from R.

https://torch.mlverse.org/docs/articles/installation.html

```
install.packages("torch")
library(torch)
install_torch()
```

See ?install_torch for GPU instructions

Working with torch models

DEFINE A NN MODULE

```
dense <- nn_module(
  "no_biais_dense_layer",
  initialize = function(in_f, out_f) {
    self$w <- nn_parameter(torch_randn(in_f, out_f))
  },
  forward = function(x) {
    torch_mm(x, self$w)
  }
)
```

Create a nn module names no_biais_dense_layer

ASSEMBLE MODULES INTO NETWORK

model <- dense(4, 3)
Instantiate a network from a single module

```
model <- nn_sequential(
    dense(4,3), nn_relu(), nn_dropout(0.4),
    dense(3,1), nn_sigmoid())
```

Instantiate a sequential network with multiple layers

MODEL FIT

model$train()
Turns on gradient update

```
with_enable_grad({
   y_pred <- model(trainset)
   loss <- (y_pred - y)$pow(2)$mean()
   loss$backward()
})
```

Detailed training loop step (alternative)

EVALUATE A MODEL

```
model$eval()
or
with_no_grad({
   model(validationset)
})
```

Perform forward operation with no gradient update

OPTIMIZATION

optim_sgd()
Stochastic gradient descent optimiser

optim_adam()
ADAM optimiser

CLASSIFICATION LOSS FUNCTION

nn_cross_entropy_loss()
nn_bce_loss()
nn_bce_with_logits_loss()
(Binary) cross-entropy losses
nn_nll_loss()
Negative log-likelihood loss
nn_margin_ranking_loss()
nn_hinge_embedding_loss()
nn_multi_margin_loss()
nn_multilabel_margin_loss()
(Multiclass) (multi label) hinge losses

REGRESSION LOSS FUNCTION

nn_l1_loss()
L1 loss
nn_mse_loss()
MSE loss nn_ctc_loss()
Connectionist Temporal Classification loss
nn_cosine_embedding_loss()
Cosine embedding loss
nn_kl_div_loss()
Kullback-Leibler divergence loss
nn_poisson_nll_loss()
Poisson NLL loss

OTHER MODEL OPERATIONS

summary() Print a summary of a torch model

torch_save(); torch_load() Save/Load models to files

load_state_dict()
Load a model saved in python

Neural-network layers

CORE LAYERS

nn_linear()
Add a linear transformation NN layer to an input

nn_bilinear() to two inputs

nn_sigmoid(), nn_relu()
Apply an activation function to an output

nn_dropout()
nn_dropout2d()
nn_dropout3d()
Applies Dropout to the input

nn_batch_norm1d()
nn_batch_norm2d()
nn_batch_norm3d()
Applies batch normalisation to the weights

CONVOLUTIONAL LAYERS

nn_conv1d() 1D, e.g. temporal convolution

nn_conv_transpose2d()
Transposed 2D (deconvolution)

nn_conv2d() 2D, e.g. spatial convolution over images

nn_conv_transpose3d()
Transposed 3D (deconvolution)
nn_conv3d() 3D, e.g. spatial convolution over volumes

nnf_pad()
Zero-padding layer

ACTIVATION LAYERS

nn_leaky_relu()
Leaky version of a rectified linear unit

nn_relu6()
rectified linear unit clamped by 6

nn_rrelu()
Randomized leaky rectified linear unit

nn_elu(), nn_selu()
Exponential linear unit, Scaled Exp lineal unit

POOLING LAYERS

nn_max_pool1d()
nn_max_pool2d()
nn_max_pool3d()
Maximum pooling for 1D to 3D

nn_avg_pool1d()
nn_avg_pool2d()
nn_avg_pool3d()
Average pooling for 1D to 3D

nn_adaptive_max_pool1d()
nn_adaptive_max_pool2d()
nn_adaptive_max_pool3d()
Adaptive maximum pooling

nn_adaptive_avg_pool1d()
nn_adaptive_avg_pool2d()
nn_adaptive_avg_pool3d()
Adaptive average pooling

RECURRENT LAYERS

nn_rnn()
Fully-connected RNN where the output is to be fed back to input

nn_gru()
Gated recurrent unit - Cho et al

nn_lstm()
Long-Short Term Memory unit - Hochreiter 1997

Tensor manipulation

torch · luz · torchaudio · torchvision

TENSOR CREATION

tt <- torch_rand(4,3,2) uniform distrib.
tt <- torch_randn(4,3,2) unit normal distrib.
tt <- torch_randint(1,7,c(4,3,2)) uniform integers within [1,7)
Create a random values tensor with shape

tt <- torch_ones(4,3,2)
torch_ones_like(a)
Create a tensor full of 1 with given shape, or with the same shape as 'a'. Also **torch_zeros, torch_full, torch_arange,...**

tt$shape [1] 4 3 2 — **tt$ndim** [1] 3 — **tt$dtype** torch_Float
tt$requires_grad [1] FALSE — **tt$device** torch_device(type='cpu')
Get 't' tensor shape and attributes

tt$stride() [1] 6 2 1 — jump needed to go from one element to the next in each dimension

tt <- torch_tensor(a, dtype=torch_float(), device= "cuda")
Copy the R array 'a' into a tensor of float on the GPU
a <- as.matrix(tt$to(device="cpu")

TENSOR SLICING

tt[1:2, -2:-1,]
Slice a 3D tensor
tt[5:N, -2:-1, ..]
Slice a 3D or more tensor, N for last

tt[1:2, -2:-1, 1:1]
tt[1:2, -2:-1, 1, keep=TRUE]
Slice a 3D and keep the unitary dim.

tt[1:2, -2:-1, 1]
Slice by default remove unitary dim.

tt[tt > 3.1]
Boolean filtering (flattened result)

TENSOR CONCATENATION

torch_stack()
Stack of tensors

torch_cat()
Assemble tensors

torch_split(2)
split tensor in sections of size 2

torch_split(c(1,3,1))
split tensor into explicit sizes

TENSOR SHAPE OPERATIONS

tt$unsqueeze(1)
torch_unsqueeze(tt,1)
Add a unitary dimension to tensor "tt" as first dimension

tt$squeeze(1)
torch_squeeze(tt,1)
Remove first unitary dimension to tensor "tt"

torch_reshape() — **$view()**
Change the tensor shape, with copy or (tentatively) without

torch_flatten()
Flattens an input

torch_transpose()

torch_movedim(c(1,2))
switch dimension 1 with 2

torch_movedim(c(1,2,3), c(3,1,2))
move dim 1 to dim 3, dim 2 to 1, dim 3 to 2
torch_permute(c(3,1,2))
Only provide the target dimension order

torch_flip(1) flip values along dim 1
torch_flip(2) 2
torch_flip(c(1,2)) both dims

TENSOR VALUES OPERATIONS

+, -, *
Operations with two tensors

$pow(2), $log(), $exp(), $abs(), $floor(), $round(), $cos(), $fmod(3), $fmax(1), $fmin(3)
torch_clamp(tt, min=0.1, max=0.7)
Element-wise operations on a tensor

$eq(), $ge(), $le()
Element-wise comparison

$to(dtype = torch_long())
Mutate values type

$sum(dim=1), $mean(), $max()
Aggregation functions on a single tensor
$amax()

torch_repeat_interleave()
Repeats the input n times

Pre-trained models

The "Hello, World!" of deep learning

Torch applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning.

NATIVE R MODELS

library(torchvision)
resnet34 <- model_resnet34(pretrained=TRUE)
Resnet image classification model

resnet34_headless <- nn_prune_head(resnet34, 1)
Remove top layer of a model

IMPORTING FROM PYTORCH

{torchvisionlib} allows you to import a pytorch model without recoding its nn modules in R.
This is done in two steps

1- instantiate the model in Python, script it, and save it:

```
import torch
import torchvision

model = torchvision.models.segmentation.\
    fcn_resnet50(pretrained = True)
model.eval()

scripted_model = torch.jit.script(model)
torch.jit.save(scripted_model, "fcn_resnet50.pt")
```

2- load and use the model in R:

```
library(torchvisionlib)
model <- torch::jit_load("fcn_resnet50.pt")
```

Troubleshooting

HELPERS

with_detect_anomaly()
Provides insight of a nn_module() behaviour

Callbacks

A callback is a set of functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training.

TRAINING AN IMAGE RECOGNIZER ON MNIST DATA

```
# load MNIST images through a data loader
library(torchvision)
train_ds <- mnist_dataset( root = " ~/.cache" ,
    download = TRUE,
    transform = torchvision::transform_to_tensor
)
test_ds <- mnist_dataset( root = " ~/.cache",
    train = FALSE,
    transform = torchvision::transform_to_tensor
)
train_dl <- dataloader(train_ds, batch_size = 32,
    shuffle = TRUE)
test_dl <- dataloader(test_ds, batch_size = 32)

# defining the model and layers
net <- nn_module(
  "Net",
  initialize = function() {
   self$fc1 <- nn_linear(784, 128)
   self$fc2 <- nn_linear(128, 10)
  },
  forward = function(x) {
   x %>% torch_flatten(start_dim = 2) %>%
    self$fc1() %>% nnf_relu() %>%
    self$fc2() %>% nnf_log_softmax(dim = 1)
  }
)
model <- net()
# define optimizer
optimizer <- optim_sgd(model$parameters, lr = 0.01)
# train (fit)
for (epoch in 1:10) {
  train_losses <- c()
  test_losses <- c()
  for (b in enumerate(train_dl)) {
   optimizer$zero_grad()
   output <- model(b[[1]]$to(device = device))
   loss <- nnf_nll_loss(output, b[[2]]$to(device = device))
   loss$backward()
   optimizer$step()
   train_losses <- c(train_losses, loss$item())
  }
  for (b in enumerate(test_dl)) {
   model$eval()
   output <- model(b[[1]]$to(device = device))
   loss <- nnf_nll_loss(output, b[[2]]$to(device = device))
   test_losses <- c(test_losses, loss$item())
   model$train()
  }
}
```

PyTorch Cheat Sheet

Using PyTorch 1.2, torchaudio 0.3, torchtext 0.4, and torchvision 0.4.

General PyTorch and model I/O

```
# loading PyTorch
import torch

# cuda
import torch.cuda as tCuda # various functions and settings
torch.backends.cudnn.deterministic = True # deterministic ML?
torch.backends.cudnn.benchmark = False  # deterministic ML?
torch.cuda.is_available # check if cuda is is_available
tensor.cuda() # moving tensor to gpu
tensor.cpu() # moving tensor to cpu
tensor.to(device) # copy densor to device xyz
torch.device('cuda') # or 'cuda0', 'cuda1' if multiple devices
torch.device('cpu') # default

# static computation graph/C++ export preparation
torch.jit.trace()
from torch.jit import script, trace
@script

# load and save a model
torch.save(model, 'model_file')
model = torch.load('model_file')
model.eval() # set to inference
torch.save(model.state_dict(), 'model_file') # only state dict
model = ModelCalss()
model.load_state_dict(torch.load('model_file'))

# save to onnx
torch.onnx.export
torch.onnx.export_to_pretty_string

# load onnx model
import onnx
model = onnx.load('model.onnx')
# check model
onnx.checker.check_model(model)
```

Pre-trained models and domain-specific utils

Audio

```
import torchaudio
# load and save audio
stream, sample_rate = torchaudio.load('file')
torchaudio.save('file', stream, sample_rate)
# 16 bit wav files only
stream, sample_rate=torchaudio.load_wav('file')

# datasets (can be used with torch.utils.data.DataLoader)
import torchaudio.datasets as aDatasets
aDatasets.YESNO('folder_for_storage', download=True)
aDatasets.VCTK('folder_for_storage', download=True)


# transforms
import torchaudio.transforms as aTransforms
aTransforms.AmplitudeToDB
aTransforms.MelScale
aTransforms.MelSpectrogram
aTransforms.MFCC
aTransforms.MuLawEncoding
aTransforms.MuLawDecoding
aTransforms.Resample
aTransforms.Spectogram

# kaldi support
import torchaudio.compliance.kaldi as aKaldi
import torchaudio.kaldi_io as aKaldiIO
aKaldi.spectogram
aKaldi.fbank
aKaldi.mfcc
aKaldi.resample_waveform
```

```python
aKaldiIO.read_vec_int_ark
aKaldiIO.read_vec_flt_scp
aKaldiIO.read_vec_flt_ark
aKaldiIO.read_mat_scp
aKaldiIO.read_mat_ark

# functional/direct function access
import torchaudio.functional as aFunctional

# sox effects/passing data between Python and C++
import torchaudio.sox_effects as aSox_effects
```

Text

```python
import torchtext
# various data-related function and classes
import torchtext.data as tData
tData.Batch
tData.Dataset
tData.Example
tData.TabularDataset
tData.RawField
tData.Field
tData.ReversibleField
tData.SubwordField
tData.NestedField
tData.Iterator
tData.BucketIterator
tData.BPTTIterator
tData.Pipeline # similar to vTransform and sklearn's pipeline
tData.batch # function
tData.pool # function


# datasets
import torchtext.datasets as tDatasets
# sentiment analysis
tDatasets.SST
tDatasets.IMDb
tDatasets.TextClassificationDataset # subclass of all datasets below
tDatasets.AG_NEWS
tDatasets.SogouNews
tDatasets.DBpedia
tDatasets.YelpReviewPolarity
tDatasets.YelpReviewFull
tDatasets.YahooAnswers
tDatasets.AmazonReviewPolarity
tDatasets.AmazonReviewFull

# question classification
tDatasets.TREC

# entailment
tDatasets.SNLI
tDatasets.MultiNLI

# language modeling
tDatasets.WikiText2
tDatasets.WikiText103
tDatasets.PennTreebank

# machine translation
tDatasets.TranslationDataset # subclass
tDatasets.Multi30k
tDatasets.IWSLT
tDatasets.WMT14

# sequence tagging
tDatasets.SequenceTaggingDataset # subclass
tDatasets.UDPOS
tDatasets.CoNLL2000Chunking

# question answering
tDatasets.BABI20

# vocabulary and pre-trained embeddings
import torchtext.vocab as tVocab
tVocab.Vocab # create a vocabulary
tVocab.SubwordVocab # create subvocabulary
tVocab.Vectors # word vectors
tVocab.GloVe # GloVe embeddings
tVocab.FastText # FastText embeddings
tVocab.CharNGram # character n-gram
```

Vision

```
import torchvision
# datasets
import torchvision.datasets as vDatasets
vDatasets.MNIST
vDatasets.FashionMNIST
vDatasets.KMNIST
vDatasets.EMNIST
vDatasets.QMNIST
vDatasets.FakeData # randomly generated images
vDatasets.COCOCaptions
vDatasets.COCODetection
vDatasets.LSUN
vDatasets.ImageFolder # data loader for a certain image folder structure
vDatasets.DatasetFolder # data loader for a certain folder structure
vDatasets.ImageNet
vDatasets.CIFAR
vDatasets.STL10
vDatasets.SVHN
vDatasets.PhotoTour
vDatasets.SBU
vDatasets.Flickr
vDatasets.VOC
vDatasets.Cityscapes
vDatasets.SBD
vDatasets.USPS
vDatasets.Kinetics400
vDatasets.HMDB51
vDatasets.UCF101

# video IO
import torchvision.io as vIO
vIO.read_video('file', start_pts, end_pts)
vIO.write_video('file', video, fps, video_codec)
torchvision.utils.save_image(image,'file')

# pretrained models/model architectures
import torchvision.models as vModels
# models can be constructed with random weights ()
# or pretrained (pretrained=True)
```

```
# classification
vModels.alexnet(pretrained=True)
vModels.densenet121()
vModels.densenet161()
vModels.densenet169()
vModels.densenet201()
vModels.googlenet()
vModels.inception_v3()
vModels.mnasnet0_5()
vModels.mnasnet0_75()
vModels.mnasnet1_0()
vModels.mnasnet1_3()
vModels.mobilenet_v2()
vModels.resnet18()
vModels.resnet34()
vModels.resnet50()
vModels.resnet50_32x4d()
vModels.resnet101()
vModels.resnet101_32x8d()
vModels.resnet152()
vModels.wide_resnet50_2()
vModels.wide_resnet101_2()
vModels.shufflenet_v2_x0_5()
vModels.shufflenet_v2_x1_0()
vModels.shufflenet_v2_x1_5()
vModels.shufflenet_v2_x2_0()
vModels.squeezenet1_0()
vModels.squeezenet1_1()
vModels.vgg11()
vModels.vgg11_bn()
vModels.vgg13()
vModels.vgg13_bn()
vModels.vgg16()
vModels.vgg16_bn()
vModels.vgg19()
vModels.vgg19_bn()

# semantic segmentation
vModels.segmentation.fcn_resnet50()
vModels.segmentation.fcn_resnet101()
vModels.segmentation.deeplabv3_resnet50()
```

```python
vModels.segmentation.deeplabv3_resnet101()

# object and/or keypoint detection, instance segmentation
vModels.detection.fasterrcnn_resnet50_fpn()
vModels.detection.maskrcnn_resnet50_fpn()
vModels.detection.keypointrcnn_resnet50_fpn()

# video classification
vModels.video.r3d_18()
vModels.video.mc3_18()
vModels.video.r2plus1d_18()

# transforms
import torchvision.transforms as vTransforms
vTransforms.Compose(transforms) # chaining transforms
vTransforms.Lambda(someLambdaFunction)

# transforms on PIL images
vTransforms.CenterCrop(height,width)
vTransforms.ColorJitter(brightness=0, contrast=0,
                        saturation=0, hue=0)
vTransforms.FiveCrop
vTransforms.Grayscale
vTransforms.Pad
vTransforms.RandomAffine(degrees, translate=None,
                         scale=None, shear=None,
                         resample=False, fillcolor=0)
vTransforms.RandomApply(transforms, p=0.5)
vTransforms.RandomChoice(transforms)
vTransforms.RandomCrop
vTransforms.RandomGrayscale
vTransforms.RandomHorizontalFlip
vTransforms.RandomOrder
vTransforms.RandomPerspective
vTransforms.RandomResizedCrop
vTransforms.RandomRotation
vTransforms.RandomSizedCrop
vTransforms.RandomVerticalFlip
vTransforms.Resize
vTransforms.Scale
vTransforms.TenCrop
```

```python
# transforms on torch tensors
vTransforms.LinearTransformation
vTransforms.Normalize
vTransforms.RandomErasing

# conversion
vTransforms.ToPILImage
vTransforms.ToTensor

# direct access to transform functions
import torchvision.transforms.functional as vTransformsF

# operators for computer vision
# (not supported by TorchScript yet)
import torchvision.ops as vOps
vOps.nms # non-maximum suppression (NMS)
vOps.roi_align # <=> vOps.ROIALIGN
vOps.roi_pool # <=> vOps.ROIPOOL
```

Data loader

```python
# classes and functions to represent datasets
from torch.utils.data import Dataset, Dataloader
```

Neural network

```python
import torch.nn as nn
```

Activation functions

```python
nn.AdaptiveLogSoftmaxWithLoss
nn.CELU
nn.EL
nn.Hardshrink
nn.Hardtanh
nn.LeakyReLU
nn.LogSigmoid
nn.LogSoftmax
nn.MultiheadAttention
```

```
nn.PReLU
nn.ReLU
nn.ReLU6
nn.RReLU(lower,upper) # sampled from uniform distribution
nn.SELU
nn.Sigmoid
nn.Softmax
nn.Softmax2d
nn.Softmin
nn.Softplus
nn.Softshrink
nn.Softsign
nn.Tanh
nn.Tanhshrink
nn.Threshols
```

Loss function

```
nn.BCELoss
nn.BCEWithLogitsLoss
nn.CosineEmbeddingLoss
nn.CrossEntropyLoss
nn.CTCLoss
nn.HingeEmbeddingLoss
nn.KLDivLoss
nn.L1Loss
nn.MarginRankingLoss
nn.MSELoss
nn.MultiLabelSoftMarginLoss
nn.MultiMarginLoss
nn.NLLLoss
nn.PoissonNLLLoss
nn.SmoothL1Loss
nn.SoftMarginLoss
nn.TripletMarginLoss
```

Optimizer

```
import torch.optim as optim
# general useage
scheduler = optim.Optimizer(....)
scheduler.step() # step-wise
optim.lr_scheduler.Scheduler

# optimizers
optim.Optimizer # general optimizer classes
optim.Adadelta
optim.Adagrad
optim.Adam
optim.AdamW # adam with decoupled weight decay regularization
optim.Adamax
optim.ASGD # averged stochastic gradient descent
optim.LBFGS
optim.RMSprop
optim.Rprop
optim.SGD
optim.SparseAdam # for sparse tensors

# learning rate
optim.lr_scheduler
optim.lr_scheduler.LambdaLR
optim.lr_scheduler.StepLR
optim.lr_scheduler.MultiStepLR
optim.lr_scheduler.ExponentialLR
optim.lr_scheduler.CosineAnnealingLR
optim.lr_scheduler.ReduceLROnPlateau
optim.lr_scheduler.CyclicLR
```

Pre-defined layers/deep learning

```
# containers
nn.Module{ ,List,Dict}
nn.Parameter{List,Dict}
nn.Sequential

# linear layers
nn.Linear
nn.Bilinear
nn.Indentity

# dropout layers
nn.AlphaDropout
nn.Dropout{ ,2d,3d}
```

```
# convolutional layers
nn.Conv{1,2,3}d
nn.ConvTranspose{1,2,3}d
nn.Fold
nn.Unfold

# pooling
nn.AdaptiveAvgPool{1,2,3}d
nn.AdaptiveMaxPool{1,2,3}d
nn.AvgPool{1,2,3}d
nn.MaxPool{1,2,3}d
nn.MaxUnpool{1,2,3}d

# recurrent layers
nn.GRU
nn.LSTM
nn.RNN

# padding layers
nn.ReflectionPad{1,2}d
nn.ReplicationPad{1,2,3}d
nn.ConstantPad{1,2,3}d

# normalization layers
nn.BatchNorm{1,2,3}d
nn.InstanceNorm{1,2,3}d

# transformer layers
nn.Transformer
nn.TransformerEncoder
nn.TransformerDecoder
nn.TransformerEncoderLayer
nn.TransformerDecoderLayer
```

Computational graph

```
# various functions and classes to use and manipulate
# automaic differentiation and the computational graph
import torch.autograd as autograd
```

Functional

```
import torch.nn.functional as F
# direct function access and not via classes (torch.nn) ???
```

NumPy-like functions

Loading PyTorch and tensor basics

```
# loading PyTorch
import torch

# defining a tensor
torch.tensor((values))

# define data type
torch.tensor((values), dtype=torch.int16)

# converting a NumPy array to a PyTorch tensor
torch.from_numpy(numpyArray)

# create a tensor of zeros
torch.zeros((shape))
torch.zeros_like(other_tensor)

# create a tensor of ones
torch.ones((shape))
torch.ones_like(other_tensor)

# create an idenity matrix
torch.eye(numberOfRows)

# create tensor with same values
torch.full((shape), value)
torch.full_like(other_tensor,value)

# create an empty tensor
torch.empty((shape))
torch.empty_like(other_tensor)

# create sequences
torch.arange(startNumber, endNumber, stepSize)
```

```
torch.linspace(startNumber, endNumber, stepSize)
torch.logspace(startNUmber, endNumber, stepSize)

# concatenate tensors
torch.cat((tensors), axis)

# split tensors into sub-tensors
torch.split(tensor, splitSize)

# (un)squeeze tensor
torch.squeeze(tensor, dimension)
torch.unsqueeze(tensor, dim)

# reshape tensor
torch.reshape(tensor, shape)

# transpose tensor
torch.t(tensor) # 1D and 2D tensors
torch.transpose(tensor, dim0, dim1)
```

Random numbers

```
# set seed
torch.manual_seed(seed)

# generate a tensor with random numbers
# of interval [0,1)
torch.rand(size)
torch.rand_like(other_tensor)

# generate a tensor with random integer numbers
# of interval [lowerInt,higherInt]
torch.randint(lowerInt,
              higherInt,
              (tensor_shape))
torch.randint_like(other_tensor,
                   lowerInt,
                   higherInt)

# generate a tensor of random numbers drawn
# from a normal distribution (mean=0, var=1)
torch.randn((size))
torch.randn_like(other_tensor)

# random permuation of integers
# range [0,n-1)
torch.randperm()
```

Math (element-wise)

```
# basic operations
torch.abs(tensor)
torch.add(tensor, tensor2) # or tensor+scalar
torch.div(tensor, tensor2) # or tensor/scalar
torch.mult(tensor,tensor2) # or tensor*scalar
torch.sub(tensor, tensor2) # or tensor-scalar
torch.ceil(tensor)
torch.floor(tensor)
torch.remainder(tensor, devisor) #or torch.fmod()
torch.sqrt(tensor)

# trigonometric functions
torch.acos(tensor)
torch.asin(tensor)
torch.atan(tensor)
torch.atan2(tensor)
torch.cos(tensor)
torch.cosh(tensor)
torch.sin(tensor)
torch.sinh(tensor)
torch.tan(tensor)
torch.tanh(tensor)

# exponentials and logarithms
torch.exp(tensor)
torch.expm1(tensor) # exp(input-1)
torch.log(tensor)
torch.log10(tensor)
torch.log1p(tensor) # log(1+input)
torch.log2(tensor)

# other
torch.erfc(tensor) # error function
torch.erfinv(tensor) # inverse error function
```

```
torch.round(tensor) # round to full integer
torch.power(tensor, power)
```

Math (not element-wise)

```
torch.argmax(tensor)
torch.argmin(tensor)
torch.max(tensor)
torch.min(tensor)
torch.mean(tensor)
torch.median(tensor)
torch.norm(tensor, norm)
torch.prod(tensor) # product of all elements
torch.std(tensor)
torch.sum(tensor)
torch.unique(tensor)
torch.var(tensor)
torch.cross(tensor1,tensor2)
torch.cartesian_prod(tensor1, tensor2, ...)
torch.einsum(equation,tensor)
torch.tensordot(tensor1,tensor2)
torch.cholesky(tensor)
torch.cholesky_torch(tensor)
torch.dot(tensor1, tensor2)
torch.eig(tensor)
torch.inverse(tensor)
torch.det(tensor)
```



```
torch.pinverse(tensor) # pseudo-inverse
```

Other

```
torch.isinf(tensor)
torch.sort(tensor)
torch.fft(tensor, signal_dim)
torch.ifft(tensor, signal_dim)
torch.rfft(tensor, signal_dim)
torch.rifft(tensor, signal_dim)
torch.stft(tensor, n_fft)
torch.bincount(tensor)
torch.diagonal(tensor)
torch.flatten(tensor, start_dim)
torch.rot90(tensor)
torch.histc(tensor)
torch.trace(tensor)
torch.svd(tensor)
```

PyTorch C++

(aka libtorch)

```
// PyTorch header file(s)
#import <torch/script.h>

torch::jit::script::Module module;
```