Deep Learning with torch:: CHEAT SHEET

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 and torchvision for image-like data.

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

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

-2 -1
1
2

tt[1:2, -2:-1, 1:1]
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 VALUES OPERATIONS

+, - , *
Operations with two tensors

$pow(2), $log(), $exp(), $abs(), $floor(), $round(), $cos(), $fmod(3), $fmax(1), $fmin(3)
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()

n

torch_repeat_interleave()
Repeats the input n times

TENSOR SHAPE OPERATIONS

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

tt$squeeze(1)
torch_squeeze(t,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 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

torch torchaudio torchvision

Pre-trained models

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.

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

5 0 4 1

The "Hello, World!" of deep learning

```
# input layer: use MNIST images
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 loss and 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()
 }
}
```