Deep Learning with torch:: CHEAT SHEET

### Intro
(torch) is based on PyTorch, a framework popular among deep learning researchers.

[torch](https://torch.mlverse.org/) is part of an ecosystem of packages to interface with specific datasets like [torchaudio] for timeseries-like, [torchvision] for image-like, and [tabnet] for tabular data.

[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.

### Working with torch models

#### DEFINE A NN MODULE

```r
define <- function(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

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

model <- sequential(dense(4,3), dropout(0.4), sigmoid()) Instantiate a sequential network with multiple layers
```

#### MODEL FIT

```r
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

```r
model$eval() Turns off gradient update

model(validationset)
```

### OPTIMIZATION

```r
optim_sgd() Stochastic gradient descent optimiser

optim_adam() ADAM optimiser
```

### CLASSIFICATION LOSS FUNCTION

```r
nn_cross_entropy_loss() Cross-entropy loss

nn_bce_loss() Binary cross-entropy loss

nn_bce_with_logits_loss() (Binary) cross-entropy losses

nn_l1_loss() L1 loss

nn_l2_loss() MSE loss

nn_log_loss() Log-likelihood loss

nn_margin_ranking_loss() Margin ranking loss

nn_margin_loss() Margin loss

nn_multilabel_margin_loss() Multiclass (multi label) hinge losses
```

### REGRESSION LOSS FUNCTION

```r
nn_l1_loss() L1 loss

nn_mse_loss() MSE loss

nn_reg_loss() Regression loss

nn_cosine_embedding_loss() Cosine embedding loss

nn_kl_div_loss() Kullback-Leibler divergence loss

nn_poisson_loss() Poisson NLL loss

nn_l1_loss() L1 loss

nn_l2_loss() MSE loss

nn_loss() Connectionist Temporal Classification loss

nn_cosine_embedding_loss() Cosine embedding loss

nn_kl_div_loss() Kullback-Leibler divergence loss

nn_poisson_loss() Poisson NLL loss
```

### CORE LAYERS

#### Convolutional layers

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

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

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

nn_transpose_conv1d() Transposed 1D (deconvolution)

nn_transpose_conv2d() Transposed 2D (deconvolution)

nn_transpose_conv3d() Transposed 3D (deconvolution)
```

#### Recurrent layers

```r
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
```

#### Activation layers

```r
nn_leaky_relu() Leaky version of a rectified linear unit

nn_relu() Rectified linear unit

nn_sigmoid() Sigmoid function

nn_rrelu() Randomized leaky rectified linear unit

nn_elu() Exponential linear unit
```

#### Pooling layers

```r
nn_max_pool1d() Maximum pooling for 1D to 3D

nn_avg_pool1d() Average pooling for 1D to 3D

nn_depthwise_conv1d() Depthwise separable 1D convolution

nn_depthwise_conv2d() Depthwise separable 2D convolution
```

#### Other model operations

```r
summary() Print a summary of a torch model

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

nnf.pad() Zero-padding layer
```

### Evaluation loop step (alternative)

```r
loss$backward()

loss$backward()
```

### Print a summary of a torch model

```r
print(model)$summary()
```

### Cheatsheet

<table>
<thead>
<tr>
<th>CORE LAYERS</th>
<th>ACTIVATION LAYERS</th>
<th>POOLING LAYERS</th>
<th>CONVOLUTIONAL LAYERS</th>
<th>RECURRENT LAYERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>nn_linear()</td>
<td>nn_leaky_relu()</td>
<td>nn_max_pool1d()</td>
<td>nn_conv1d()</td>
<td>nn_rnn()</td>
</tr>
<tr>
<td>nn_bilinear()</td>
<td>nn_relu6()</td>
<td>nn_max_pool2d()</td>
<td>nn_conv2d()</td>
<td>nn_gru()</td>
</tr>
<tr>
<td>nn_sigmoid()</td>
<td>nn_relu()</td>
<td>nn_max_pool3d()</td>
<td>nn_conv3d()</td>
<td>nn_lstm()</td>
</tr>
<tr>
<td>nn_dropout()</td>
<td>nn_sigmoid()</td>
<td>nn_avg_pool1d()</td>
<td>nn_transpose_conv1d()</td>
<td></td>
</tr>
<tr>
<td>nn_dropout2d()</td>
<td></td>
<td>nn_avg_pool2d()</td>
<td>nn_transpose_conv2d()</td>
<td></td>
</tr>
<tr>
<td>nn_dropout3d()</td>
<td></td>
<td>nn_avg_pool3d()</td>
<td>nn_transpose_conv3d()</td>
<td></td>
</tr>
</tbody>
</table>

### Installation

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

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

See ?install_torch for GPU instructions
Tensor manipulation

Tensor creation
- `t = torch.randn(4,3,2)` uniform distribution.
- `tt = torch.ones(4,3,2)` unit normal distribution.
- `tt = torch.randnint(1,7, c(4,3,2))` uniform integers within `[1,7]`.
- Create a random values tensor with shape `[1,2]`.
- `tt = torch.ones(4,3,2)` Create a tensor full of 1 with given shape, or with the same shape as `a`. Also `torch_zeros`, `torch_full`, `torch_around`, ... for doing the same with zeros.
- `tt[5:N,  -2:-1,  ..]` Tensor slicing.
- `tt$shape` Tuple with dimensions.
- `tt$ndim` Number of dimensions.
- `tt$dtype` Data type.
- `tt$to(device = "cuda")` Move `tt` to GPU.
- `torch_rand(4,3,2)` Create a random values tensor with shape `[1,2]`.
- `torch_randint(1,7,c(4,3,2))` Create a random values tensor with shape `[1,2]`.

Tensor shape operations
- `ttunsqueeze(1)` Add a unitary dimension to tensor `"tt"` as first dimension.
- `torch_squeeze(1)` Remove first unitary dimension to tensor `"tt"`.
- `torch_reshape()` View tensor.
- `torch_transpose()` Transpose the dimensions of a tensor.
- `torch_flatten()` Flatten an input into a tensor with a single dimension.
- `torch_permute()` Permute the dimensions of a tensor.
- `torch_moveaxis()` Move an axis of a tensor.
- `torch_tensor(a, dtype=torch_float(), device="cuda")` Convert `a` to tensor.

Tensor values operations
- `torch.flip(2)` Flip a tensor along the second dimension.
- `torch.flip(2)` Flip a tensor along the second dimension.
- `torch.flip(c(1,2), 2)` Flip a tensor along the second and third dimensions.
- `torch.transpose()` Transpose the specified dimensions of a tensor.
- Boolean filtering (flattened result)
  - `tt[ tt > 3.1]` Select elements greater than 3.1.
- Slice a 3D tensor
  - `tt[1:2, -2:-1, ..]` Slice a 3D tensor along the second and third dimensions.
  - `tt[1:2, -2:-1, 1]` Slice a 3D tensor along the second, third, and fourth dimensions.
  - `tt[1:2, -2:-1, 1]` Slice a 3D tensor along the second, third, and fourth dimensions.
- Utilities for strings
  - `torch_flatten()` Flatten an input into a tensor with a single dimension.
  - `torch_permute()` Permute the dimensions of a tensor.
  - `torch_moveaxis()` Move an axis of a tensor.
- Split tensor into explicit sizes
  - `torch_split(2)` Split tensor in sections of size 2.
  - `torch_split(c(1,3,1), 2)` Split tensor into explicit sizes.
  - `torch_repeat_interleave()` Repeat the input n times.

Troubleshooting

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.

Pre-trained models

NATIVE R MODELS
- `library(torchvision)`
- `resnet34 <- model_resnet34(pretrained=T)` Resnet image classification model.
- `resnet34$headless <- nn_prune_head(resnet34, 1)` Remove top layer of a model.

Native R models

Importing from PyTorch
- `from torchvision.models import pretrained_models` Imports torchvision pre-trained models.
- `resnet34 <- torchvision.models.resnet34(pretrained=T)` Resnet image classification model.

Importing from PyTorch

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.

Callbacks

Define loss and optimizer
- `optimizer <- optim_sgd(model$parameters, lr = 0.01)` Set up the optimizer.
- `loss <- nll_loss(output, target) + 0.5 * l2_loss(self$fc2)` Define the loss function.
- `model$fc2$weight$grad <- optimizer$step()` Update the weights.

Define loss and optimizer

Training an Image Recognizer On MNIST Data

- `library(torchvision)` Imports torchvision.
- `model <- nn_module( "Net" , ...)` Define the model.
- `self$fc1()` Forward pass.
- `self$fc2()` Forward pass.
- `model$eval()` Set model to evaluation mode.
- `model$train()` Set model to training mode.
- `model(b[[1]]$to(device = device))` Forward pass.
- `nnf_nll_loss(output, label)` Define the loss function.

Training an Image Recognizer On MNIST Data

CC BY-SA 3.0
ChistopheRegouby • torch 0.7.0 • Updated: 2022-05