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	**torch.randn(*size)**	Create random tensor
import torch.nn as nn	Neural networks	**torch.Tensor(L)**	Create tensor from list
from torchvision import datasets, models, transforms	Popular image datasets, architectures & transforms	**tnsr.view(a,b, ...)**	Reshape tensor to size (a, b, ...)
import torch.nn.functional as F	Collection of layers, activations & more	**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.

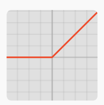
Usually the dataset is split into training (e.g. 80%) and test data (e.g. 20%).

```
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

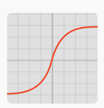
Common activation functions include **ReLU**, **Sigmoid** and **Tanh**, but there are other activation functions as well.

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