Music Machine Learning

XIII – Pytorch

Master ATIAM - Informatique

Philippe Esling (esling@ircam.fr)

Maître de conférences – UPMC

Equipe représentations musicales (IRCAM, Paris)

Deep learning frameworks

- Modern tools make it easy to implement neural networks
- Often used components
  - Linear, convolution, recurrent layers etc.
- Many frameworks available: Torch (2002), Theano (2011), Caffe (2014), TensorFlow (2015), PyTorch (2016)

Which framework to choose ?

- Research, flexibility and being happy in life ?

- Deployment (mobile, web)

Pytorch

- Fast tensor computation (like numpy) with strong GPU support
- Deep learning research platform that provides maximum flexibility and speed
- Dynamic graphs and automatic differentiation

Developers


Facebook
Open Source


NVIDIA


salesforce


ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIE
PARIS INSTITUTE OF TECHNOLOGY


Carnegie
Mellon
University


UNIVERSITE
PIERRE & MARIE CURIE
LA SCIENCE A PARIS


UBER


Stanford
University


UNIVERSITY OF
OXFORD


NYU


Inria


ENS
ÉCOLE NORMALE
SUPÉRIEURE


VisionLabs
MACHINES CAN SEE


Digital
Reasoning

Outlines

- Basic concepts
- Write a model
- Classification of MNIST dataset
- Convolutional model

Basic Concepts

torch.Tensor
similar to numpy.array

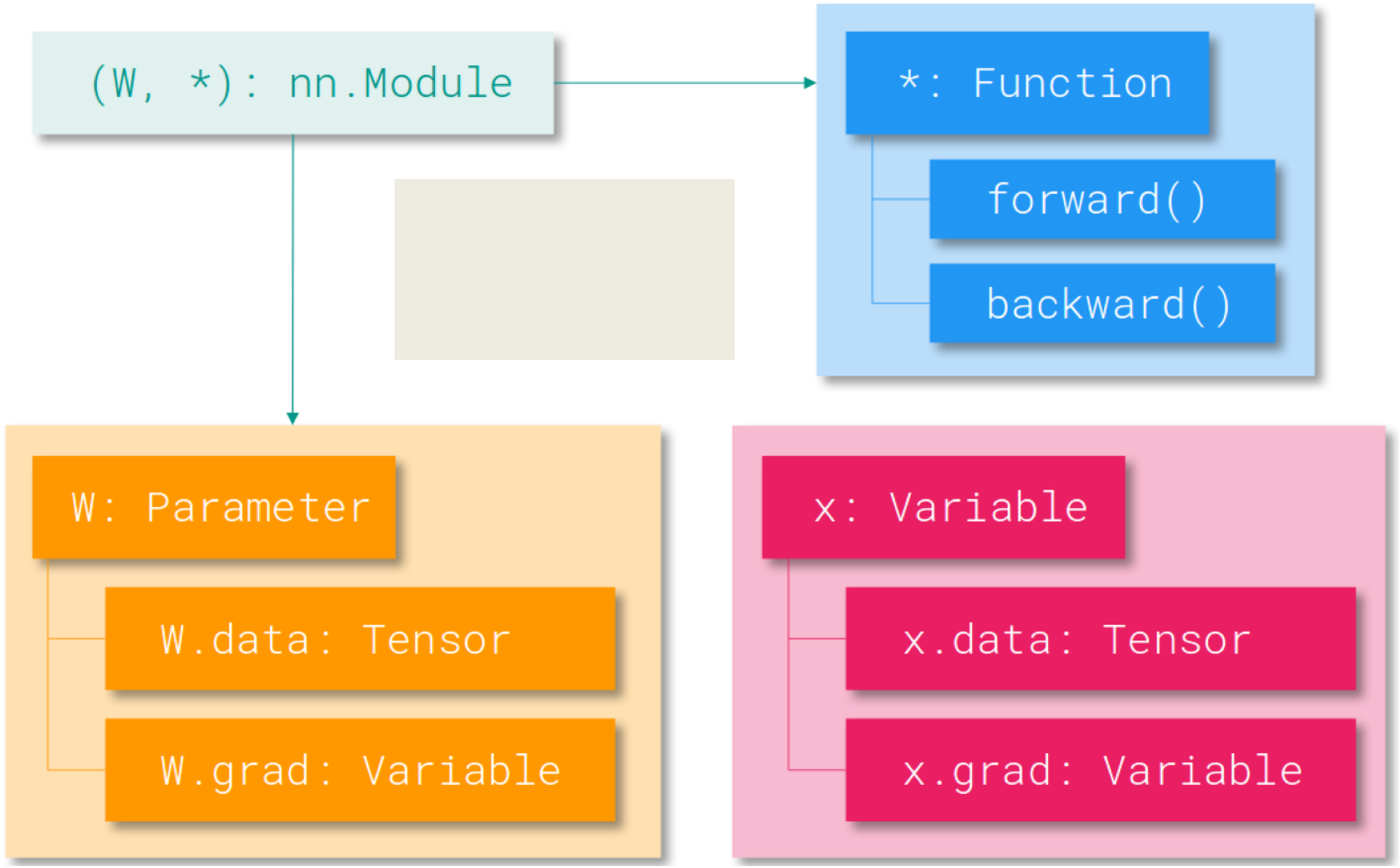
nn.Parameter
contain *Parameters* and define functions on input *Variables*

nn.Module
contain *Parameters* and define functions on input *Variables*

autograd.Variable
wraps a *Tensor* and enables auto differentiation

autograd.Function
operate on *Variables*, implements forward and backward

Basic Concepts



Basic Concepts

```
y = torch.rand(5, 3)
print(x + y)
```

```
print(torch.add(x, y))
```

```
# IN PLACE ADD
y.add_(x)
print(y)
```

```
import torch
x = torch.rand(10, 5)
y = torch.rand(10,5)
z = x * y
```

```
import torch
x = torch.ones(5,5)
b = x.numpy()
```

```
import torch
import numpy as np
a = np.ones(5)
x =
torch.from_numpy(a)
```

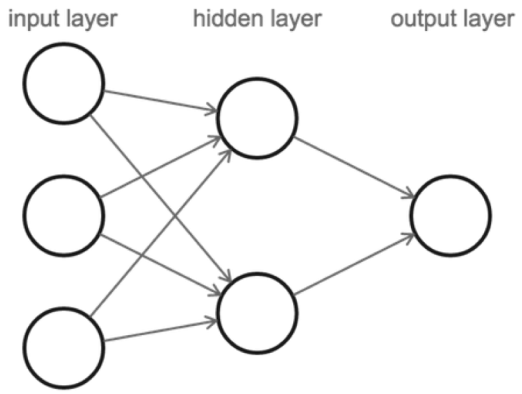
```
import torch
x= torch.rand(100,100)
x = x.cuda()
```

One single operation to go to GPU computing ☺

Creating a network (gran'ma way)

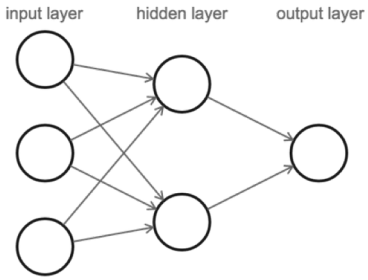
- PyTorch allows to write models in one line

```
import torch.nn as nn
model = nn.Sequential
(
        nn.Linear(input_dim, hidden_dim),
        nn.ReLU(True),
        nn.Linear(hidden_dim, output_dim),
        nn.Softmax()
)

x = torch.rand(16, input_dim)
model(x)
```



Low amount of control but easily implemented

Network definition



More flexible and controllable definition

Object-oriented through nn.Module

```
class TwoLayerNet(torch.nn.Module):
  def __init__(self, D_in, H, D_out):
    super(TwoLayerNet, self).__init__()
    self.linear1 = nn.Linear(D_in, H)
    self.linear2 = nn.Linear(H, D_out)
    self.relu = nn.ReLU(inplace=True)

  def forward(self, x):
    h_relu = self.linear1(x)
    h_relu = self.relu(h_relu)
    y_pred = self.linear2(h_relu)
    ypred = nn.functional.softmax(y_pred)
return y_pred
```

Operations can be layers

Or directly functions to apply (when no parameters involved)

Example of a residual network

```
class ResBlock(nn.Module):
  def __init__(self, dim, dim_res=32):
    super().__init__()
    self.block = nn.Sequential(
          nn.Conv2d(dim, dim_res, 3, 1, 1),
          nn.ReLU(True),
          nn.Conv2d(dim_res, dim, 1),
          nn.ReLU(True)
      )

  def forward(self, x):
    return x + self.block(x)
```

Mixing modules and direct operations !

Modules are themselves usable in more complex models !

```
nn.Sequential(
          ResBlock(64, 32),
          ResBlock(64, 32),
)
```

Optimizing the Network

```
# Have some data
N, D_in, H, D_out = 64, 1000, 100, 10
# Create a network
model = TwoLayerNet(D_in, H, D_out)
# Select an adequate loss
loss_fn = torch.nn.MSELoss(size_average=False)
# Create an optimizer (here stochastic gradient descent)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

for epoch in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x); loss = loss_fn(y, y_pred)
    # Zero gradients (cumulative).
    optimizer.zero_grad()
    # Backward pass:
    loss.backward()
    # Apply your optimization (gradient descent)
    optimizer.step()
```

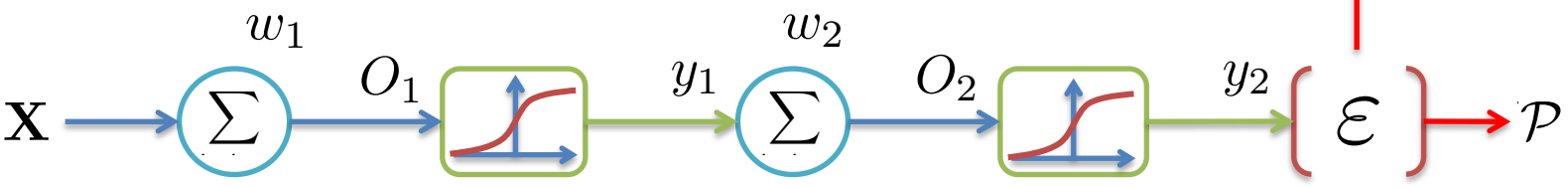
Details on losses

- A loss function takes a (output, target) pair of inputs,
- Computes the prediction error made by the network
- A simple loss: nn.MSELoss (mean-squared)

```
output = net(input)
target = torch.randn(10) # a dummy target, for example
criterion = nn.MSELoss()
loss = criterion(output, target)
print(loss)
```

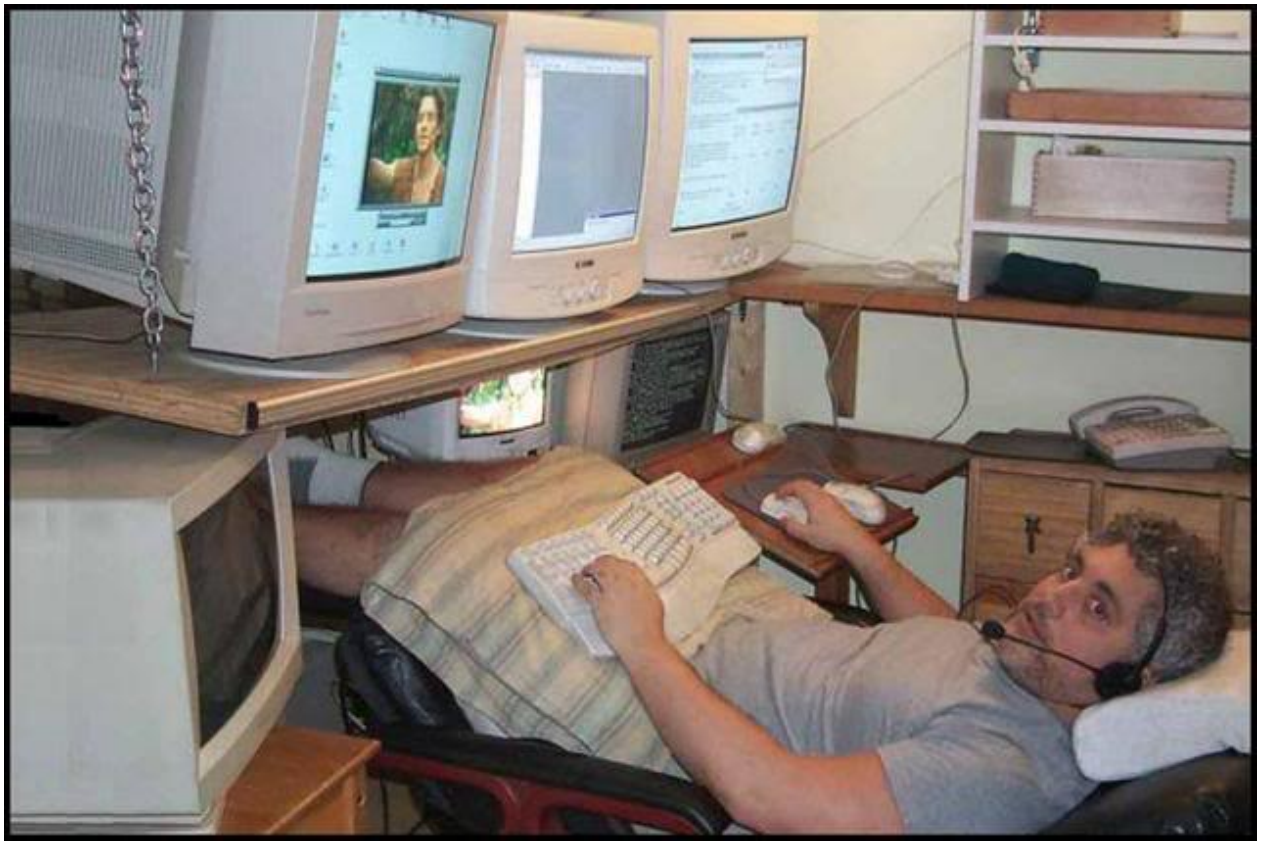
Flashback time

$$\Delta\bar{w} = r\left(\frac{\delta\mathcal{P}}{\delta w_1}, \frac{\delta\mathcal{P}}{\delta w_2}\right) \qquad \varepsilon = \sum_i (d_i - x_i)^2$$



Network definition #3

THE REAL DEAL

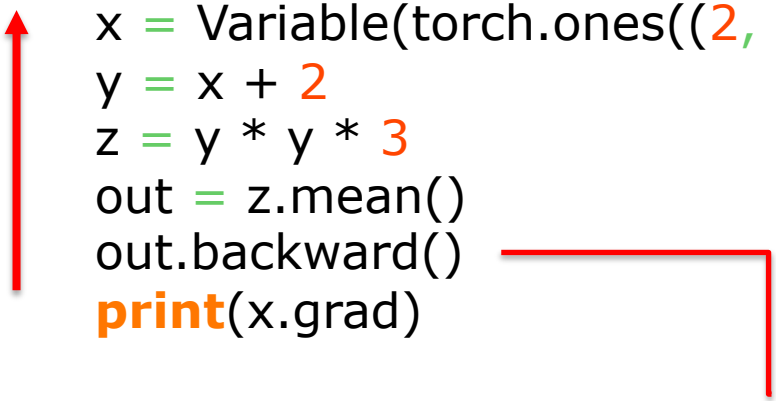


Gradients

NOTE = Since Pytorch 0.4
All tensors are Variables

- **Variables** wrap a tensor and saves a history
- Used for **automatic differentiation**

```
import torch
from torch.autograd.variable import Variable
x = Variable(torch.ones((2, 2)), requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean()
out.backward()
print(x.grad)
```



This operation goes through the whole differentiation graph

Network definition

Implementing [Yang et. al 2016]

Attention layer on h_{it}

Affine transform

1 $u_{it} = \tanh(W_w h_{it} + b_w)$

Softmax

$\alpha_{it} = \frac{\exp(u_{it}^T u_w)}{\sum_t \exp(u_{it}^T u_w)}$ **2**

3 $s_i = \sum_t \alpha_{it} h_{it}$

Trainable matrix

```
# Chord-level attention layer
class ChordLevelAttention(nn.Module):
  def __init__(self, n_hidden):
    super(ChordLevelAttention, self).__init__()
    self.mlp = nn.Linear(n_hidden, n_hidden)
    self.u_w = nn.Parameter(torch.rand(n_hidden))

  def forward(self, X):
    # get the hidden representation of the sequence
1   u_it = F.tanh(self.mlp(X))
    # get attention weights for each timestep
2   alpha = F.softmax(torch.matmul(u_it, self.u_w), dim=1)
    # get the weighted sum of the sequence
3   out = torch.sum(torch.matmul(alpha, X), dim=1)
    return out, alpha
```

And Pytorch handles all derivations and gradients !

So what about data ?

Using **torchvision** extremely easy to load known datasets.

Example on MNIST (that we will use)

```
import torchvision
import torchvision.transforms as transforms
# Use transform to Tensors of normalized range [-1, 1].
transform = transforms.Compose(
[transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
trainset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
shuffle=True, num_workers=2)

for i, data in enumerate(trainloader, 0):
    # get the inputs
    inputs, labels = data
```

Writing your own dataset

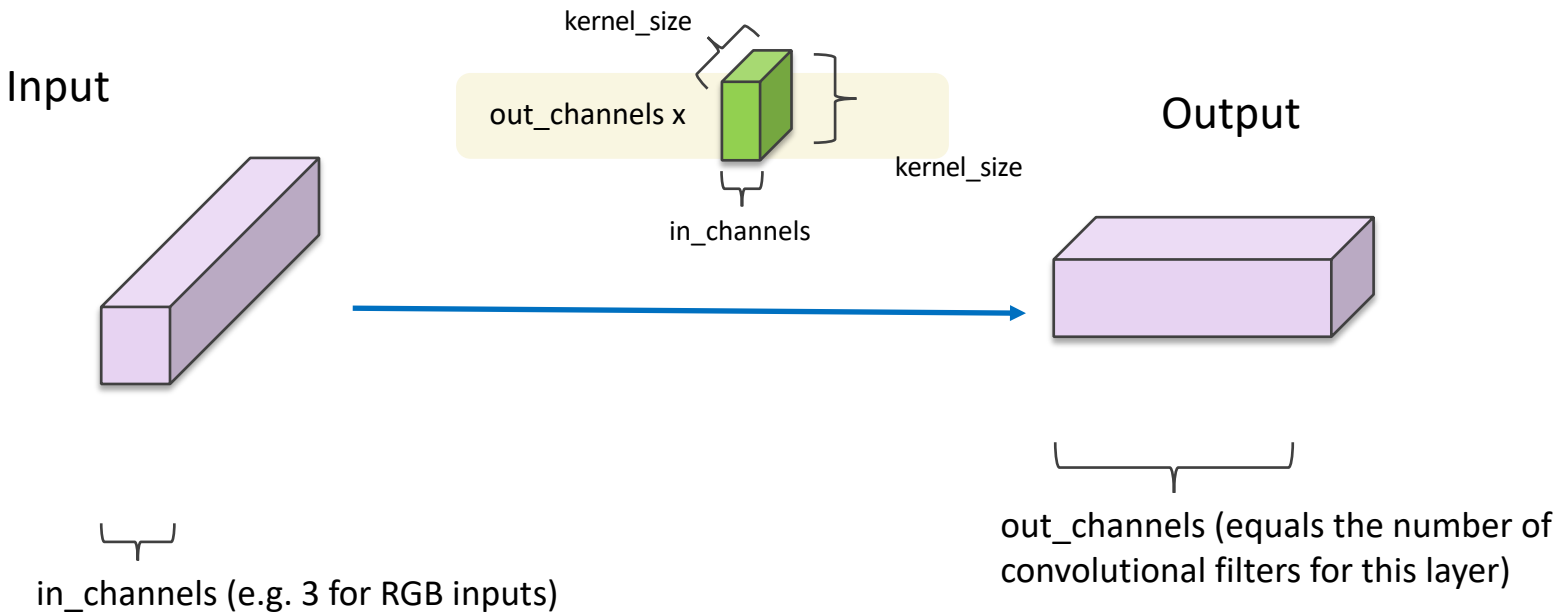
```
# Simplest Pytorch Audio dataset object ever
class AudioDataset(Dataset):
    def __init__(self, datadir):
        self.data_files = sorted(glob.glob(datadir + '/*'))

    def __getitem__(self, idx):
        try :
            audio_file, sr = librosa.core.load(self.data_files[idx])
            loaded_file = librosa.feature.melspectrogram(audio_file, sr)
            loaded_file = torch.from_numpy(np.log1p(np.flipud(loaded_file)))
            return loaded_file, 0
        except :
            return torch.Tensor(1), 0

    def __len__(self):
        return len(self.data_files)
```

Things to remember

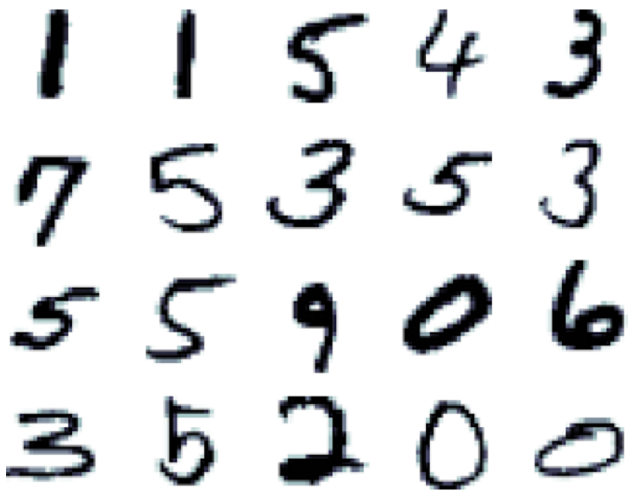
- Evaluate **generalization** on a **separate test set**
  - Beware of bad splits ! (cf. blue door / GTZAN)
  - *Is your neural network a horse ?* (Clever Hans)
- **Check your input** data thoroughly (normalization !)
- Address the **characteristics** of your data
- Select your **loss function** accordingly
- Always wonder about **regularization**
  - Dropout, BatchNorm, Augmentations
- Start by **solving simple toy** problems
  - Good test = *can you overfit on a small set ?*

Convolutional layers

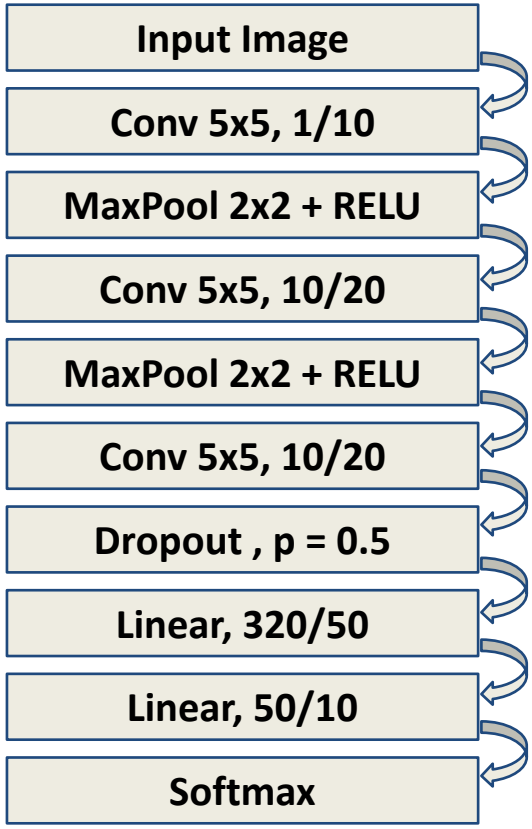
```
class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1,
groups=1, bias=True)  [source]
```



Classifying Handwritten Digits



MNIST Dataset



If you are bored ?

- **Perform some funky neural transfer with Pytorch**

https://pytorch.org/tutorials/advanced/neural_style_tutorial.html