UF Herbert Wertheim College of Engineering
UNIVERSITY of FLORIDA

ECE CISE

POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE

Large-scale Intelligent Systems Laboratory (Li Lab)
National Science Foundation Center for Big Learning (CBL)
Department of Electrical and Computer Engineering (ECE)
Department of Computer & Information Science & Engineering (CISE)

Pytorch Tutorial

Xiaoyong Yuan, Xiyao Ma
2018/01

some slides copied from cs231n.stanford.edu

Introduction

- Why we need deep learning frameworks

(1) Easily build **big** computational graphs

(2) Easily compute **gradients** in computational graphs

(3) Run it all efficiently on **GPU**

- Why we need Pytorch?

(1) Easy to implement, code, and debug

(2) More flexible due to its **dynamic** computational graph.

(3) High execution efficiency, since it developed from C.

PyTorch: Three Levels of Abstraction

Tensor: Like array in Numpy, but runs on GPU

Variable: Node in a computational graph; stores data and gradient

Module: A neural network layer; may store state or learnable weights

PyTorch: Tensors

- PyTorch Tensors are just like numpy arrays, but they can run on GPU.
- No built-in notion of computational graph, or gradients, or deep learning.
- Here we fit a two-layer net using PyTorch Tensors

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

To run on GPU, just cast tensors to a cuda data type!
(E,g   torch.cuda.FloatTensor)

Create random tensors for data and weights.

Forward pass:
compute predictions and loss

Backward pass:
manually compute gradients

Gradient descent step on weights

```
import torch

dtype = torch.FloatTensor

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Tensor: Operations

http://pytorch.org/docs/master/index.html

Create a tensor:

An empty tensor can be constructed by specifying its size:

```
>>> torch.IntTensor(2, 4).zero_()
0  0  0  0
0  0  0  0
[torch.IntTensor of size 2x4]
```

Math Operation:

`torch.dot(tensor1, tensor2) → float`

Computes the dot product (inner product) of two tensors.

Other operations:

`torch.squeeze(input, dim=None, out=None)`

Returns a tensor with all the dimensions of `input` of size 1 removed.

For example, if *input* is of shape: $(A \times 1 \times B \times C \times 1 \times D)$ then the *out* tensor will be of shape: $(A \times B \times C \times D)$.

`torch.unsqueeze(input, dim, out=None)`

Returns a new tensor with a dimension of size one inserted at the specified position.

`torch.cat(seq, dim=0, out=None) → Tensor`

Concatenates the given sequence of `seq` Tensors in the given dimension.

PyTorch: Autograd

- A PyTorch Variable is a node in a computational graph
- x.data is a Tensor
- x.grad is a Variable of gradients (same shape as x.data)
- x.grad.data is a Tensor of gradients
- PyTorch Tensors and Variables have the same API!
- **Variables remember how they were created (for backprop)**

```python
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: Autograd

We will not want gradients (of loss) with respect to data

Do want gradients with respect to weights

Forward pass looks exactly the same as the Tensor version, but everything is a variable now

Compute gradient of loss with respect to w1 and w2 (zero out grads first)

Make gradient step on weights

```python
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: New Autograd Functions

```
class ReLU(torch.autograd.Function):
    def forward(self, x):
        self.save_for_backward(x)
        return x.clamp(min=0)

    def backward(self, grad_y):
        x, = self.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

Can use our new autograd function in the forward pass

```
N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    relu = ReLU()
    y_pred = relu(x.mm(w1)).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

PyTorch: New Autograd Functions

Define your own autograd functions by writing forward and backward for Tensors

```
class ReLU(torch.autograd.Function):
    def forward(self, x):
        self.save_for_backward(x)
        return x.clamp(min=0)

    def backward(self, grad_y):
        x, = self.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x < 0] = 0
        return grad_input
```

PyTorch: nn

Higher-level wrapper for working with neural nets

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

PyTorch: nn

Define our model as a sequence of layers

nn also defines loss functions

Forward pass: feed data to model, and prediction to loss function

Backward pass: compute all gradients

Make gradient step on each model parameter

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

Example: Define your own Module

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

Applies a 2D convolution over an input signal composed of several input planes.

Parameters:

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = floor((H_{in} + 2 * padding[0] - dilation[0] * (kernel\_size[0] - 1) - 1)/stride[0] + 1)$$
$$W_{out} = floor((W_{in} + 2 * padding[1] - dilation[1] * (kernel\_size[1] - 1) - 1)/stride[1] + 1)$$

Variables:

- **weight** (*Tensor*) – the learnable weights of the module of shape (out_channels, in_channels, kernel_size[0], kernel_size[1])
- **bias** (*Tensor*) – the learnable bias of the module of shape (out_channels)

```python
import torch.nn as nn
import torch.nn.functional as F


class discriminator(nn.Module):
    def __init__(self, d=128):
        super(discriminator, self).__init__()
        self.conv1 = nn.Conv2d(1, d, 4, 2, 1)
        self.conv2 = nn.Conv2d(d, d*2, 4, 2, 1)
        self.conv2_bn = nn.BatchNorm2d(d*2)
        self.conv3 = nn.Conv2d(d*2, d*4, 4, 2, 1)
        self.conv3_bn = nn.BatchNorm2d(d*4)
        self.conv4 = nn.Conv2d(d*4, d*8, 4, 2, 1)
        self.conv4_bn = nn.BatchNorm2d(d*8)
        self.conv5 = nn.Conv2d(d*8, 1, 4, 1, 0)

    # forward method
    def forward(self, input):
        x = F.leaky_relu(self.conv1(input), 0.2)
        x = F.leaky_relu(self.conv2_bn(self.conv2(x)), 0.2)
        x = F.leaky_relu(self.conv3_bn(self.conv3(x)), 0.2)
        x = F.leaky_relu(self.conv4_bn(self.conv4(x)), 0.2)
        x = F.sigmoid(self.conv5(x))
        x = x.squeeze()
        return x
```

You can find more details about API in the following link:

http://pytorch.org/docs/master/index.html

PyTorch: optim

Use an optimizer for different update rules

Update all parameters after computing gradients

```
import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    optimizer.zero_grad()
    loss.backward()

    optimizer.step()
```

PyTorch: nn Define new Modules

Define our whole model as a single Module

Initializer sets up two children (Modules can contain modules)
Note: No need to define backward - autograd will handle it

Define forward pass using child modules and autograd ops on Variables

Construct and train an instance of our model

```
import torch
from torch.autograd import Variable

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

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

MNIST Example

https://github.com/pytorch/examples/blob/master/mnist/main.py

Extending PyTorch

- torch.autograd
  - customize *forward()* and *backward()*
  - check gradient
    - from torch.autograd import gradcheck

```
# Inherit from Function
class LinearFunction(Function):

    # Note that both forward and backward are @staticmethods
    @staticmethod
    # bias is an optional argument
    def forward(ctx, input, weight, bias=None):
        ctx.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    # This function has only a single output, so it gets only one gradient
    @staticmethod
    def backward(ctx, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing Nones are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = ctx.saved_variables
        grad_input = grad_weight = grad_bias = None

        # These needs_input_grad checks are optional and there only to
        # improve efficiency. If you want to make your code simpler, you can
        # skip them. Returning gradients for inputs that don't require it is
        # not an error.
        if ctx.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0).squeeze(0)

        return grad_input, grad_weight, grad_bias
```

Extending PyTorch

- torch.autograd
  - Adding function to a Module

```
class Linear(nn.Module):
    def __init__(self, input_features, output_features, bias=True):
        super(Linear, self).__init__()
        self.input_features = input_features
        self.output_features = output_features

        # nn.Parameter is a special kind of Variable, that will get
        # automatically registered as Module's parameter once it's assigned
        # as an attribute. Parameters and buffers need to be registered, or
        # they won't appear in .parameters() (doesn't apply to buffers), and
        # won't be converted when e.g. .cuda() is called. You can use
        # .register_buffer() to register buffers.
        # nn.Parameters can never be volatile and, different than Variables,
        # they require gradients by default.
        self.weight = nn.Parameter(torch.Tensor(output_features, input_features))
        if bias:
            self.bias = nn.Parameter(torch.Tensor(output_features))
        else:
            # You should always register all possible parameters, but the
            # optional ones can be None if you want.
            self.register_parameter('bias', None)

        # Not a very smart way to initialize weights
        self.weight.data.uniform_(-0.1, 0.1)
        if bias is not None:
            self.bias.data.uniform_(-0.1, 0.1)

    def forward(self, input):
        # See the autograd section for explanation of what happens here.
        return LinearFunction.apply(input, self.weight, self.bias)
```

Extending PyTorch

- C-extension
  a. prepare your C code

```c
/* src/my_lib.c */
#include <TH/TH.h>

int my_lib_add_forward(THFloatTensor *input1, THFloatTensor *input2,
THFloatTensor *output)
{
    if (!THFloatTensor_isSameSizeAs(input1, input2))
        return 0;
    THFloatTensor_resizeAs(output, input1);
    THFloatTensor_cadd(output, input1, 1.0, input2);
    return 1;
}

int my_lib_add_backward(THFloatTensor *grad_output, THFloatTensor *grad_input)
{
    THFloatTensor_resizeAs(grad_input, grad_output);
    THFloatTensor_fill(grad_input, 1);
    return 1;
}
```

c file

```c
/* src/my_lib.h */
int my_lib_add_forward(THFloatTensor *input1, THFloatTensor *input2, THFloatTensor *output);
int my_lib_add_backward(THFloatTensor *grad_output, THFloatTensor *grad_input);
```

header with all functions

Extending PyTorch

- C-extension
  b. build custom extension

```
# build.py
from torch.utils.ffi import create_extension
ffi = create_extension(
name='_ext.my_lib',
headers='src/my_lib.h',
sources=['src/my_lib.c'],
with_cuda=False
)
ffi.build()
```

run it and get a new folder with a .so file

  c. include it in your Python code

```
# functions/add.py
import torch
from torch.autograd import Function
from _ext import my_lib


class MyAddFunction(Function):
    def forward(self, input1, input2):
        output = torch.FloatTensor()
        my_lib.my_lib_add_forward(input1, input2, output)
        return output

    def backward(self, grad_output):
        grad_input = torch.FloatTensor()
        my_lib.my_lib_add_backward(grad_output, grad_input)
        return grad_input
```

Torchvision

- popular datasets
  - cifar10, coco, lsun, mnist, ...
- popular model architectures (pretrained)
  - alexnet, densenet, inception, resnet, squeezenet, vgg
- common image transformations
  - Normalize, Scale, CenterCrop, Pad, RandomCrop, RandomFlip, …
  - Compose

Visualization

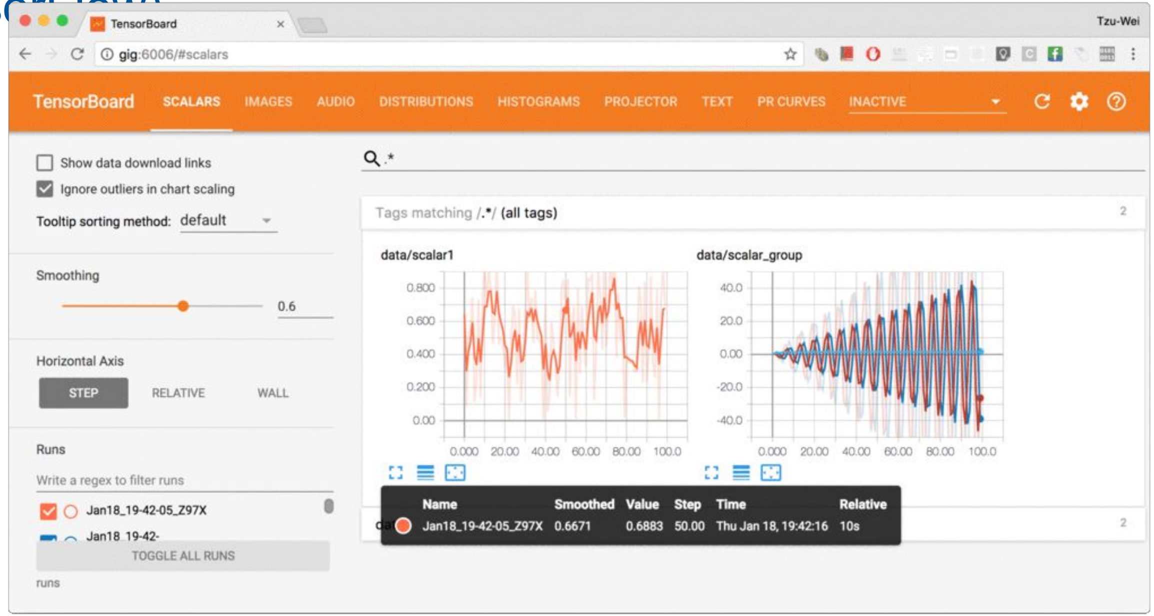
- TensorboardX
  - pip install tensorboardX
  - tensorboard for pytorch (and chainer, mxnet, numpy, ...)
  - Support scalar, image, histogram, audio, text, graph, onnx_graph, embedding and pr_curve
  - demo http://35.197.26.245:6006/

class `tensorboardX.SummaryWriter`(*log_dir=None, comment=''*)

```python
import torch
import torchvision.utils as vutils
import numpy as np
import torchvision.models as models
from torchvision import datasets
from tensorboardX import SummaryWriter

resnet18 = models.resnet18(False)
writer = SummaryWriter()
sample_rate = 44100
freqs = [262, 294, 330, 349, 392, 440, 440, 440, 440, 440, 440]

for n_iter in range(100):

    dummy_s1 = torch.rand(1)
    dummy_s2 = torch.rand(1)
    # data grouping by `slash`
    writer.add_scalar('data/scalar1', dummy_s1[0], n_iter)
    writer.add_scalar('data/scalar2', dummy_s2[0], n_iter)

    writer.add_scalars('data/scalar_group', {'xsinx': n_iter * np.sin(n_iter),
                                             'xcosx': n_iter * np.cos(n_iter),
                                             'arctanx': np.arctan(n_iter)}, n_iter)

    dummy_img = torch.rand(32, 3, 64, 64)  # output from network
    if n_iter % 10 == 0:
        x = vutils.make_grid(dummy_img, normalize=True, scale_each=True)
        writer.add_image('Image', x, n_iter)

        dummy_audio = torch.zeros(sample_rate * 2)
        for i in range(x.size(0)):
            # amplitude of sound should in [-1, 1]
            dummy_audio[i] = np.cos(freqs[n_iter // 10] * np.pi * float(i) / float(sample_rate))
        writer.add_audio('myAudio', dummy_audio, n_iter, sample_rate=sample_rate)

        writer.add_text('Text', 'text logged at step:' + str(n_iter), n_iter)

        for name, param in resnet18.named_parameters():
            writer.add_histogram(name, param.clone().cpu().data.numpy(), n_iter)

        # needs tensorboard 0.4RC or later
        writer.add_pr_curve('xoxo', np.random.randint(2, size=100), np.random.rand(100), n_iter)

dataset = datasets.MNIST('mnist', train=False, download=True)
images = dataset.test_data[:100].float()
label = dataset.test_labels[:100]

features = images.view(100, 784)
writer.add_embedding(features, metadata=label, label_img=images.unsqueeze(1))

# export scalar data to JSON for external processing
writer.export_scalars_to_json("./all_scalars.json")
writer.close()
```

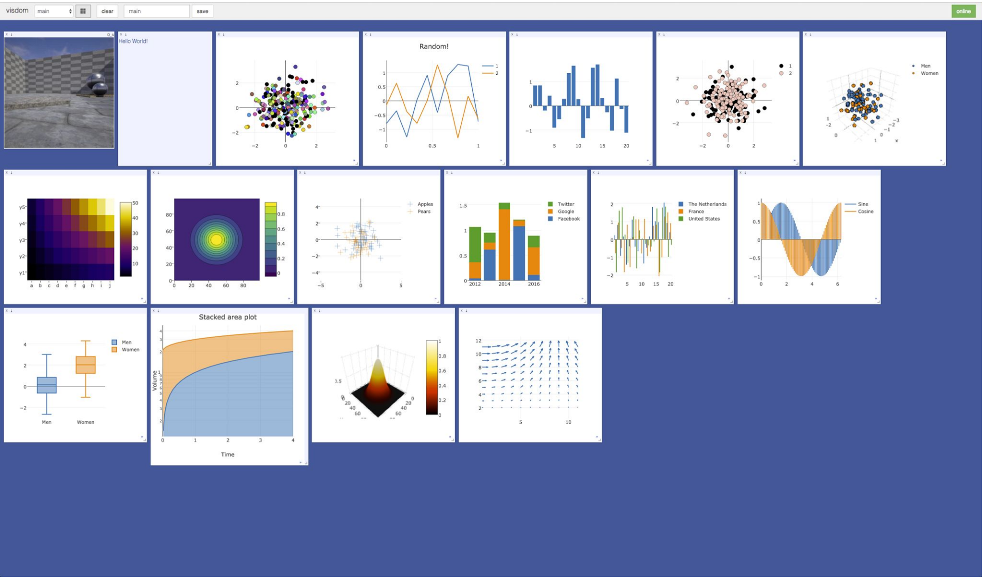
Visualization

- TensorboardX
  a. Run the demo script: python demo.py
  b. Use TensorBoard with tensorboard --logdir runs (needs to install TensorFlow)

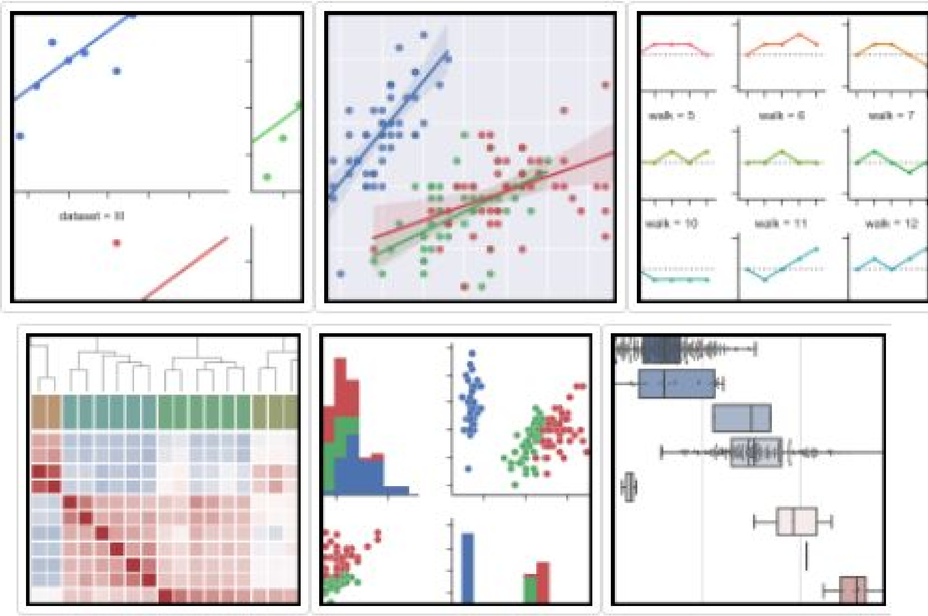


Visualization

- Other choices



Visdom



Seaborn

Onnx

- Open Neural Network Exchange (ONNX)
- an open source format to move models between tools
- defines an extensible computation graph model, built-in operators and standard data types
- Hardware Optimizations
- Supported Tools

Frameworks

Caffe2 Chainer Cognitive Toolkit mxnet PYTORCH

Converters

Runtimes

NVIDIA

References

1. http://pytorch.org/
2. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture8.pdf
3. https://github.com/pytorch/pytorch

Thank You!

NSF

NIH
National Institutes of Health

U.S. Department of Homeland Security

Department of Energy · United States of America ·