ÖSTERREICHISCHE AKADEMIE DER WISSENSCHAFTEN

# A super short introduction to PYTORCH

Wolfgang Waltenberger, ÖAW AI Summer, August 2019

With material mostly stolen from: https://cise.ufl.edu/~xiaoyong/materials/pytorch_tutorial.pdf

# PYTORCH

A pythonic open source deep learning platform that is very well suited for experimental approach and prototyping as well as production.

- Easy to build big computational graphs
- Automatic computation of gradients for learning
- Smoothly switch between CPU and GPU
- Pythonic: computational graphs are very dynamic
- High execution efficiency, written in C and CUDA

## Pytorch can be thought of as consisting of three levels of abstraction

- **Tensors** represent tensors of any order (e.g. a scalar is a zeroth-order tensor), technically equivalent to numpy arrays, with some additional features like the ability to put it on the GPU. Does not keep track of a “provenance”, or gradients.

- **Variables** represent nodes in a computational graph; stores data and gradient

- **Models and modules** eg. modules for neural networks (layers); allows for composition (a module can be composed of other modules). May store state, learnable weights

# Tensors

- Pytorch Tensors behave like numpy arrays, but are designed to run also on GPUs.
- Drop-in replacement for numpy arrays: API often exactly the same.
- Pytorch can handle tensors of any order (e.g. $0^{th}$ order tensor: scalar)
- No notion of computational graph, gradients, not necessarily tied to deep learning.

```
In [1]: import torch

In [2]: # define the dimensions of the network
        N, D_in, H, D_out = 64, 1000, 100, 10

In [3]: # create a random input tensor
        x = torch.randn ( N, D_in )

In [4]: # print it
        x

Out[4]: tensor([[ 0.2105,  0.9769,  0.1199,  ...,  1.1567,  1.3709,  0.9568],
                [ 2.1662,  0.3327,  0.3670,  ..., -1.9648, -0.2396, -0.4409],
                [-0.7170,  1.2136,  0.3757,  ...,  0.1793,  2.0523, -1.4737],
                ...,
                [-0.2276,  1.1723,  0.9590,  ..., -1.2858, -0.1605, -0.9088],
                [-1.9359, -1.9885, -0.2386,  ..., -1.7247,  0.0203,  0.0447],
                [ 0.8701, -0.1943, -0.4239,  ...,  0.9498,  0.5786, -0.7548]])

In [5]: # the shape of your tensor
        x.shape

Out[5]: torch.Size([64, 1000])

In [6]: # the type of your tensor
        x.type()

Out[6]: 'torch.FloatTensor'

In [7]: # on what device does your tensor live?
        x.device

Out[7]: device(type='cpu')

In [8]: # send the object to the GPU / CPU
        x = x.to("cpu")

In [9]: # reinterpret the second order tensor as one large first order tensor
        x.view(64000)

Out[9]: tensor([ 0.2105,  0.9769,  0.1199,  ...,  0.9498,  0.5786, -0.7548])
```

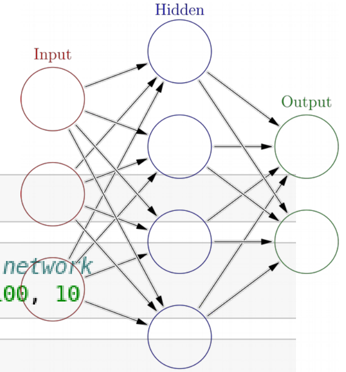
the dimensionality of the object

where does the object live? Cpu versus any of the GPUs

Move object to another device

pytorch_tensors1.ipynb

# Tensors




- Example: a fully-connected network with a single hidden layer, computing the gradient “manually”, not automatically.

“clamping” = cutting off the range of tensors. In this case all values < 0. are set to 0.

.mm() = matrix multiplication

pytorch_tensors2.ipynb

```
In [1]: import torch

In [2]: # define the dimensions of the network
        N, D_in, H, D_out = 64, 1000, 100, 10

In [3]: # create a random input tensor
        x = torch.randn ( N, D_in )
        y = torch.randn ( N, D_out )
        w1 = torch.randn ( D_in, H )
        w2 = torch.randn ( H, D_out )

In [4]: # perform forward step in a neural network manually
        h = x.mm(w1)

In [5]: h.shape

Out[5]: torch.Size([64, 100])

In [6]: # implement a "RELU" activation function manually
        h_relu = h.clamp ( min=0. )

In [7]: # second linear layer
        y_pred = h_relu.mm ( w2 )

In [8]: # compute the "loss" of the network
        loss = (y_pred - y ).pow(2).sum()

In [9]: loss

Out[9]: tensor(31799870.)

In [10]: # manually compute the gradient of the loss
         grad_y_pred = 2* ( 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)
         learning_rate = 1e-6

In [11]: # perform a learning step
         w1 -= learning_rate * grad_w1
         w2 += learning_rate * grad_w2
```

# Tensors

- Tensors come with lots of functionality, like “squeezing” (removing dimensions with only one element) “unsqueezing”, “reshaping”, “viewing”, “concatenating”, etc

```
In [1]:  import torch

In [2]:  # create a 2nd order tensor, fill with zeroes
         torch.FloatTensor(3,3).zero_()

Out[2]:  tensor([[0., 0., 0.],
                 [0., 0., 0.],
                 [0., 0., 0.]])

In [3]:  # create a tensor from a list
         t=torch.Tensor([[2.],[1.]])

In [4]:  # squeeze tensor, i.e turn a tensor with shape
         # of Ax1 into a tensor with shape A.
         ts = t.squeeze()

In [5]:  # dot product (special case of matrix multiplication)
         ts.dot(ts)

Out[5]:  tensor(5.)

In [6]:  # transpose tensor
         t.t()

Out[6]:  tensor([[2., 1.]])

In [7]:  ## concatenate tensors, but show only first 5 elements
         torch.cat([t]*5)[:5]

Out[7]:  tensor([[2.],
                 [1.],
                 [2.],
                 [1.],
                 [2.]])
```

pytorch_tensors3.ipynb

# Variables

- Variables are nodes in computational graphs that track their provenance, i.e. they know how they are constructed.
- The computation of the gradient comes therefore for free: automatic computation of the gradient → “autograd”.
- That way, one writes “differentiable” computer programs. ”differentiable programming” as a new programming paradigm.
- Variables implement the same API as tensors.

```
In [1]:  import torch
         from torch.autograd import Variable
         N, D_in, H, D_out = 64, 1000, 100, 10

In [2]:  # produce random input data, but we dont need gradients on them
         x = Variable ( torch.randn ( N, D_in ), requires_grad = False )

In [3]:  # the data, as a tensor are accessible via .data (here retrieving only
         # first ten entries of second column)
         x.data[:10:,1]

Out[3]:  tensor([ 0.9064,  0.5827, -0.2104, -0.6049, -0.4801, -0.3057, -0.7067, -0.2720,
                 -0.5093,  2.3095])

In [4]:  ## variables have all the convenience methods of tensors
         x.clamp(min=0.)[:10:,1]

Out[4]:  tensor([0.9064, 0.5827, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
                 2.3095])

In [5]:  # produce random labels, and no gradients needed on them, either
         y = Variable ( torch.randn ( N, D_out ), requires_grad = False )

In [6]:  # the weights however, should remember their gradients
         w1 = Variable ( torch.randn ( D_in, H ), requires_grad = True )
         w2 = Variable ( torch.randn ( H, D_out ), requires_grad = True )

In [7]:  # the gradient is also a Variable, and accessible via .grad
         #(but only after backprop)
         print ( w1.grad )

         None

In [8]:  y_pred = x.mm(w1).clamp(min=0.).mm(w2)
         loss = ( y_pred - y ).pow(2).sum()

In [9]:  # all variables track their provenance
         loss.backward()

In [10]: # now we have a gradient
         w1.grad[:10:,1]

Out[10]: tensor([  1344.9938,  -2858.6677,  19225.0352, -16002.1924,   8366.1250,
                  7764.0747,  -6582.6934,   6242.3301, -18540.0879,   -685.3781])
```

pytorch_variables1.ipynb

# Variables

In [1]:
```
import torch
from torch.autograd import Variable
N, D_in, H, D_out = 64, 1000, 100, 10
```

In [2]:
```
# produce random input data, but we dont need gradients on them
x = Variable ( torch.randn ( N, D_in ), requires_grad = False )
```

In [3]:
```
# the data, as a tensor are accessible via .data (here retrieving only
# first ten entries of second column)
x.data[:10:,1]
```

Out[3]:
```
tensor([ 0.9064,  0.5827, -0.2104, -0.6049, -0.4801, -0.3057, -0.7067, -0.2720,
        -0.5093,  2.3095])
```

In [4]:
```
## variables have all the convenience methods of tensors
x.clamp(min=0.)[:10:,1]
```

Out[4]:
```
tensor([0.9064, 0.5827, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        2.3095])
```

In [5]:
```
# produce random labels, and no gradients needed on them, either
y = Variable ( torch.randn ( N, D_out ), requires_grad = False )
```

In [6]:
```
# the weights however, should remember their gradients
w1 = Variable ( torch.randn ( D_in, H ), requires_grad = True )
w2 = Variable ( torch.randn ( H, D_out ), requires_grad = True )
```

In [7]:
```
# the gradient is also a Variable, and accessible via .grad
#(but only after backprop)
print ( w1.grad )
```

```
None
```

In [8]:
```
y_pred = x.mm(w1).clamp(min=0.).mm(w2)
loss = ( y_pred - y ).pow(2).sum()
```

In [9]:
```
# all variables track their provenance
loss.backward()
```

Show the gradient,
But only first 10
items, second (“,1”)
column

In [10]:
```
# now we have a gradient
w1.grad[:10:,1]
```

Out[10]:
```
tensor([  1344.9938,  -2858.6677,  19225.0352, -16002.1924,   8366.1250,
          7764.0747,  -6582.6934,   6242.3301, -18540.0879,   -685.3781])
```

pytorch_variables1.ipynb

# Variables

For “self-made” functions, one needs to implement the “backward” step oneself, i.e. specify the derivative oneself.

In [1]:
```
import torch.autograd
```

In [2]:
```
## this class implements a ReLU function, the backward pass implements
# the partial derivative (1 for x > 0., 0 for x < 0.).
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
```

In [3]:
```
f=ReLU()
```

In [4]:
```
# "calling" the object invokes the .forward member function
f(torch.autograd.Variable(torch.Tensor([3.,-3])))
```

Out[4]:
```
tensor([3., 0.])
```

pytorch_variables2.ipynb

# Module: nn

In PyTorch you usually define your neural network (“model”) as a sequence of layers

```
In [1]: import torch
        from torch.autograd import Variable
        learning_rate = 1e-6
        N, D_in, H, D_out = 64, 1000, 100, 10
```

```
In [2]: x = Variable ( torch.randn(N, D_in))
        y = Variable ( torch.randn(N, D_out), requires_grad = False)
```

```
In [3]: # define our network := model as a sequence of layers
        model = torch.nn.Sequential ( torch.nn.Linear(D_in, H),
                                      torch.nn.ReLU(),
                                      torch.nn.Linear(H, D_out))
```

```
In [4]: # define our loss function (MSE = Mean Squared Error)
        loss_fn = torch.nn.MSELoss( reduction = "sum" )
```

```
In [5]: for epoch in range(500): ## 500 epochs
            y_pred = model(x)
            ## forward pass: feed data to model, compute loss
            loss = loss_fn(y_pred, y)
            model.zero_grad()
            # backward pass: compute all gradients
            loss.backward()

            ## perform a step in direction of gradient
            for param in model.parameters():
                param.data -= learning_rate * param.grad.data
```

pytorch_module1.ipynb

# Module: nn

Here is a more involved example of a “differentiable program”

```
In [1]: import torch.nn as nn
```

```
In [2]: # A more involved example of convolutional layers
        class ConvNet(nn.Module):
            def __init__(self, num_classes=10):
                super(ConvNet, self).__init__()
                self.layer1 = nn.Sequential(
                    nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
                    nn.BatchNorm2d(16),
                    nn.ReLU(),
                    nn.MaxPool2d(kernel_size=2, stride=2))
                self.layer2 = nn.Sequential(
                    nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
                    nn.BatchNorm2d(32),
                    nn.ReLU(),
                    nn.MaxPool2d(kernel_size=2, stride=2))
                self.fc = nn.Linear(7*7*32, num_classes)

            def forward(self, x):
                out = self.layer1(x)
                out = self.layer2(out)
                out = out.reshape(out.size(0), -1)
                out = self.fc(out)
                return out
```

pytorch_module2.ipynb

# Module: optim

Pytorch optimizers live in the torch.optim module.

```
In [1]:  import torch
         from torch.autograd import Variable
         learning_rate = 1e-6
         N, D_in, H, D_out = 64, 1000, 100, 10
```

```
In [2]:  x = Variable ( torch.randn(N, D_in))
         y = Variable ( torch.randn(N, D_out), requires_grad = False)
```

```
In [3]:  # define our network := model as a sequence of layers
         model = torch.nn.Sequential ( torch.nn.Linear(D_in, H),
                                       torch.nn.ReLU(),
                                       torch.nn.Linear(H, D_out))
```

```
In [4]:  # define our loss function (MSE = Mean Squared Error)
         loss_fn = torch.nn.MSELoss( reduction = "sum" )
```

```
In [5]:  ## now define the optimizer to use. In this case "adam".
         optimizer = torch.optim.Adam ( model.parameters(), lr=learning_rate )
```

```
In [6]:  for epoch in range(500): ## 500 epochs
             y_pred = model(x)
             ## forward pass: feed data to model, compute loss
             loss = loss_fn(y_pred, y)
             model.zero_grad()
             # backward pass: compute all gradients
             loss.backward()

             ## perform a step in direction of gradient
             optimizer.step()
```

pytorch_module3.ipynb

# Torchvision

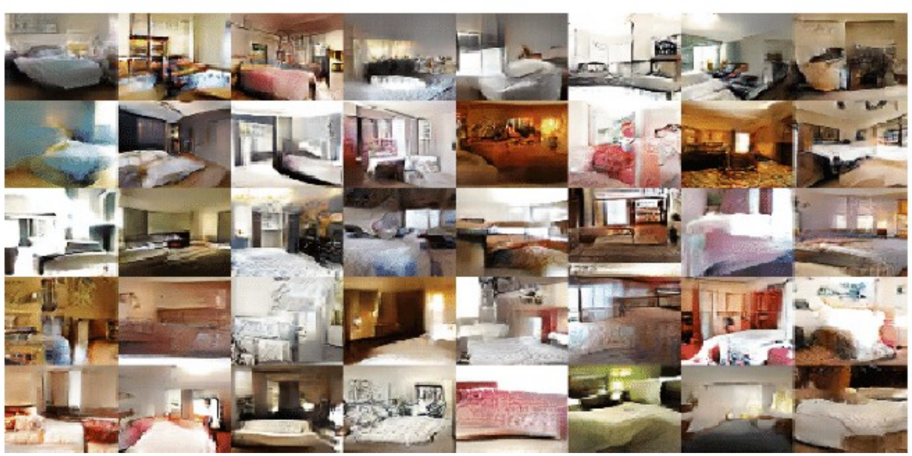
- Torchvision is a separate python package:
  `pip3 install -user torchvision`

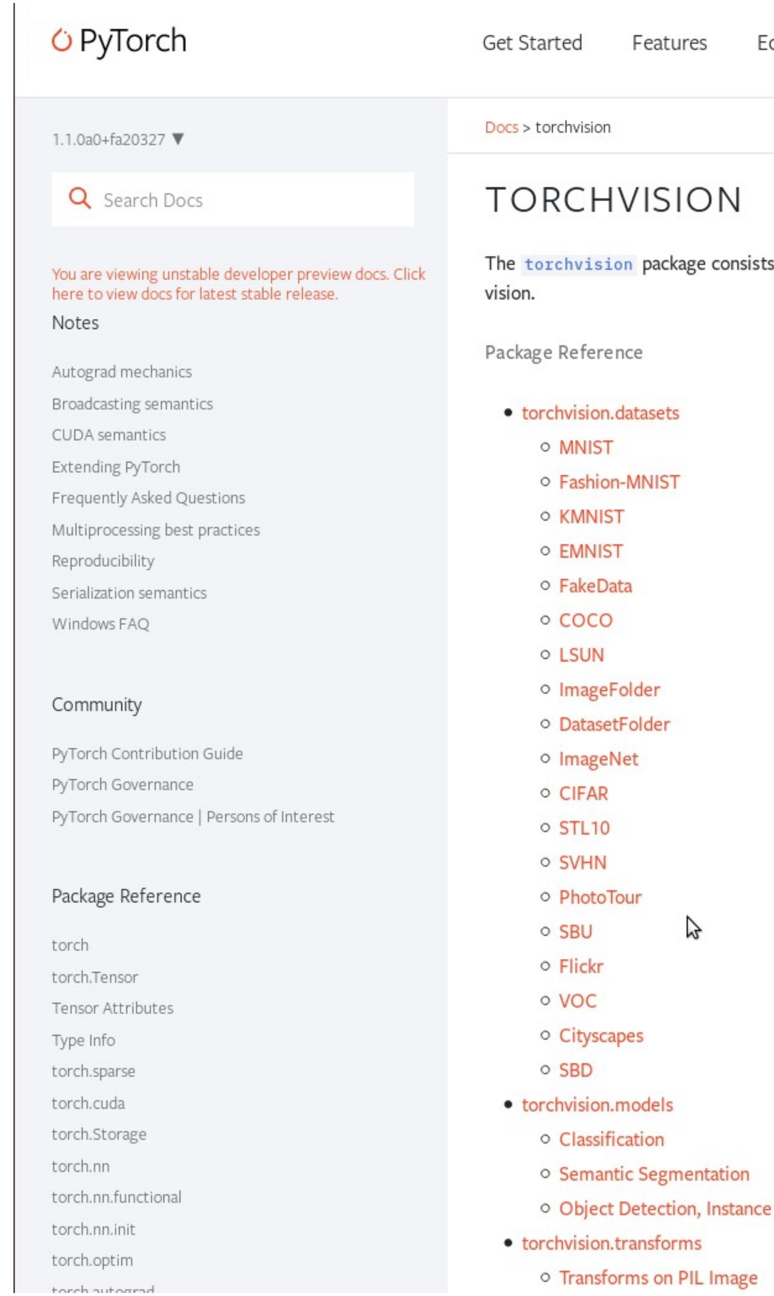
- It contains datasets, models and tools related to computer vision:

- Popular benchmarking datasets:
  cifar, coco, lsun, mnist, etc

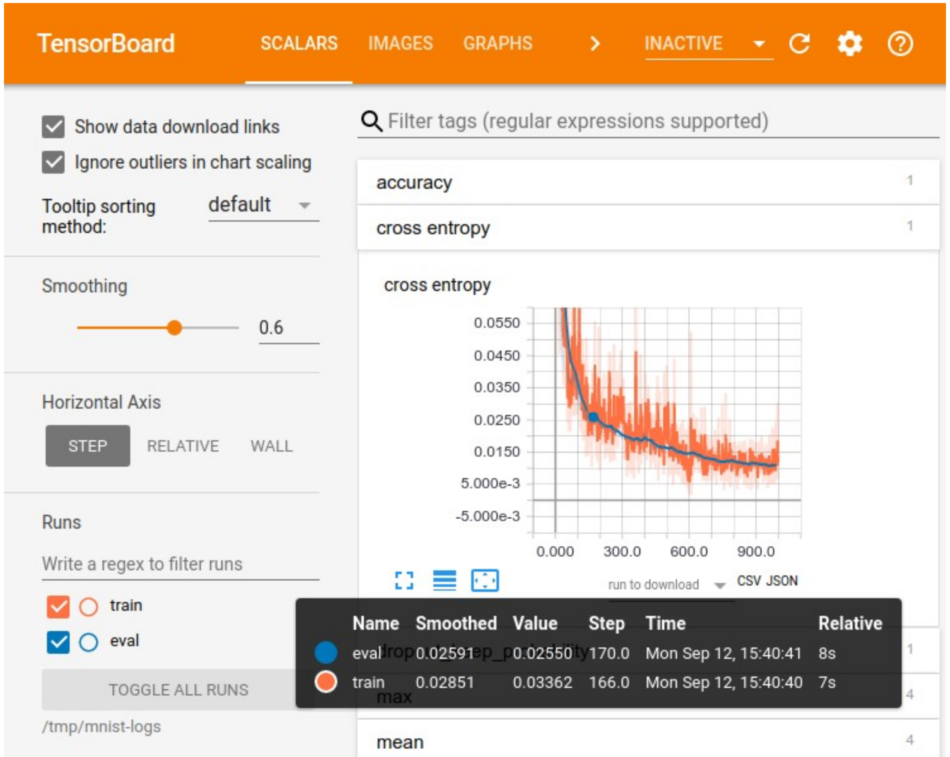
- Default (pre-trained) networks:
  alexnet, inception, resnet, vgg, etc

- A few tools that are useful mostly for computer vision:
  normalize, scale, pad, ….





# Visualization




TensorboardX
pip3 install –user tensorflow tensorboard tensorboardX

After you ran this code, do:
tensorboard –logdir runs/

```
In [1]: import torchvision.models as models
        from torchvision import datasets
        from tensorboardX import SummaryWriter

In [2]: ## instantiate the tensorboard writer
        writer = SummaryWriter()

In [3]: ## get the mnist dataset
        dataset = datasets.MNIST("mnist", train=False, download=True)

In [4]: # get the first 10 images
        images = dataset.data[:10].float()

In [5]: # and the first 10 labels
        label = dataset.targets[:10]

In [6]: # flatten the images
        features = images.view(10,784)

In [7]: images.shape, features.shape

Out[7]: (torch.Size([10, 28, 28]), torch.Size([10, 784]))

In [8]: writer.add_embedding ( features, metadata=label,
                               label_img=images.unsqueeze(1))

In [9]: ## write out data.
        writer.close()
```

pytorch_tensorboard.ipynb

ONNX

# Open Neural Network eXchange

Binary open source file format to exchange models between tools.
Supported by all major players.

```
# These are the inputs and parameters to the network, which have taken on
# the names we specified earlier.
graph(%actual_input_1 : Float(10, 3, 224, 224)
      %learned_0 : Float(64, 3, 11, 11)
      %learned_1 : Float(64)
      %learned_2 : Float(192, 64, 5, 5)
      %learned_3 : Float(192)
      # ---- omitted for brevity ----
      %learned_14 : Float(1000, 4096)
      %learned_15 : Float(1000)) {
  # Every statement consists of some output tensors (and their types),
  # the operator to be run (with its attributes, e.g., kernels, strides,
  # etc.), its input tensors (%actual_input_1, %learned_0, %learned_1)
  %17 : Float(10, 64, 55, 55) = onnx::Conv[dilations=[1, 1], group=1, kernel_shape=[11, 11],
pads=[2, 2, 2, 2], strides=[4, 4]](%actual_input_1, %learned_0, %learned_1), scope:
AlexNet/Sequential[features]/Conv2d[0]
  %18 : Float(10, 64, 55, 55) = onnx::Relu(%17), scope: AlexNet/Sequential[features]/ReLU[1]
  %19 : Float(10, 64, 27, 27) = onnx::MaxPool[kernel_shape=[3, 3], pads=[0, 0, 0, 0], strides=[2,
2]](%18), scope: AlexNet/Sequential[features]/MaxPool2d[2]
  # ---- omitted for brevity ----
  %29 : Float(10, 256, 6, 6) = onnx::MaxPool[kernel_shape=[3, 3], pads=[0, 0, 0, 0], strides=[2,
2]](%28), scope: AlexNet/Sequential[features]/MaxPool2d[12]
  # Dynamic means that the shape is not known. This may be because of a
  # limitation of our implementation (which we would like to fix in a
  # future release) or shapes which are truly dynamic.
  %30 : Dynamic = onnx::Shape(%29), scope: AlexNet
  %31 : Dynamic = onnx::Slice[axes=[0], ends=[1], starts=[0]](%30), scope: AlexNet
  %32 : Long() = onnx::Squeeze[axes=[0]](%31), scope: AlexNet
  %33 : Long() = onnx::Constant[value={9216}](), scope: AlexNet
  # ---- omitted for brevity ----
  %output1 : Float(10, 1000) = onnx::Gemm[alpha=1, beta=1, broadcast=1, transB=1](%45, %learned_14,
%learned_15), scope: AlexNet/Sequential[classifier]/Linear[6]
  return (%output1);
}
```

TensorFlow

Microsoft Cognitive Toolkit

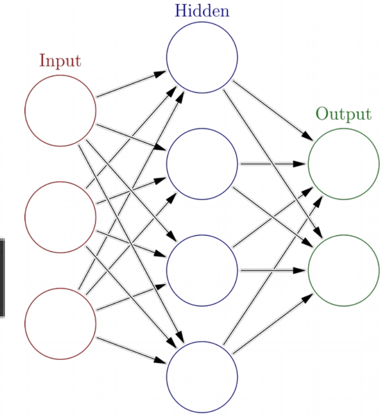
mxnet

Caffe2

Chainer

PYTORCH

PYTORCH




```
In [1]: import torch

In [2]: # define the dimensions of the network
        N, D_in, H, D_out = 64, 1000, 100, 10

In [3]: # create a random input tensor
        x = torch.randn ( N, D_in )
        y = torch.randn ( N, D_out )
        w1 = torch.randn ( D_in, H )
        w2 = torch.randn ( H, D_out )

In [4]: # perform forward step in a neural network manually
        h = x.mm(w1)

In [5]: h.shape

Out[5]: torch.Size([64, 100])

In [6]: # implement a "RELU" activation function manually
        h_relu = h.clamp ( min=0. )

In [7]: # second linear layer
        y_pred = h_relu.mm ( w2 )

In [8]: # compute the "loss" of the network
        loss = (y_pred - y ).pow(2).sum()

In [9]: loss

Out[9]: tensor(31799870.)

In [10]: # manually compute the gradient of the loss
         grad_y_pred = 2* ( 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)
         learning_rate = 1e-6

In [11]: # perform a learning step
         w1 -= learning_rate * grad_w1
         w2 += learning_rate * grad_w2
```

# Happy hacking!